RICE UNIVERSITY

Theseus: Rethinking Operating Systems Structure and State Management

By

Kevin Boos

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

Lin Zhong
Lin Zhong (Jun 30, 2020 13:42 EDT)

Lin Zhong
Committee Chair,
Professor of Electrical and Computer Engineering and Computer Science

Eduardo Cuervo
Eduardo Cuervo (Jun 30, 2020 10:38 PDT)

Eduardo Cuervo
Software Engineer, Facebook/Oculus VR

Ang Chen
Ang Chen (Jun 30, 2020 11:42 CDT)

Ang Chen
Assistant Professor of Computer Science and Electrical and Computer Engineering

Dan Wallach
Dan Wallach (Aug 11, 2020 16:34 CDT)

Dan Wallach
Professor of Computer Science

HOUSTON, TEXAS

June 2020

ABSTRACT

Theseus: Rethinking Operating Systems Structure and State Management

by

Kevin Boos

State management has become an intractable problem in modern operating systems due to their sheer size and complexity. Despite efforts to cleanly modularize OSes, the propagation and mismanagement of states remains a significant obstacle to many computing goals, e.g., system evolution and fault tolerance. We identify the root cause of such obstacles to be *state spill*, the phenomenon in which a software entity's state undergoes a lasting change as a result of handling an interaction with another entity. We systematically study the existence and manifestation of state spill in existing OSes and find that it is deeply ingrained in both low-level OS kernels and framework-level components like Android system services.

To this end, we introduce Theseus, an experimental OS written from scratch in Rust that rethinks overall OS structure and treats state management as a first-class design concern. Theseus makes two primary contributions. First, its OS structure consists of many tiny cell-like entities with clear, runtime-persistent bounds that are all loaded and linked dynamically, and interact without holding states for one another. Second, its intralingual design and implementation realizes OS functionality using existing language-level mechanisms, empowering the compiler to enforce invariants about OS semantics and enabling us to shift the responsibility of resource bookkeeping from the OS into the compiler, vastly reducing the set of states the OS must necessarily maintain. Together, Theseus's structure, intralingual design, and state management principles facilitate easy and arbitrary live evolution, system flexibility, and availability through fault recovery, even for core OS components.

Acknowledgments

My first set of thanks goes out to my advisor, Lin, who encouraged me to think boldly and to fearlessly dive into the deepest depths of every software system I encountered. I also thank him for not balking at my wild and unnecessary idea of writing a new OS from scratch in a language neither of us knew, and for having the understanding and patience to support me for more years than most. Lin has helped me develop my palate for what constitutes good research and has given me new respect for and a lasting appreciation of what it takes to realize good research.

My wonderful wife, Trang, deserves more credit than one can possibly express in words. I cannot thank her enough for being a steadfast source of emotional (and let's be honest, financial) support, and for occasionally even digging into the minutia of helping me tabulate results and proofread papers.

Of course I must also thank those who worked with me to design, implement, and solve technical challenges within Theseus, in particular Namitha Liyanage and Ramla Ijaz. Namitha contributed many fault recovery mechanisms and led the evaluation experiments therein, while Ramla realized Theseus's unique heap design, developed network drivers, and various performance experiments. I also express my appreciation to Min Hong Yun for helping with compiler plugins and providing a sounding board for my wacky design ideas, and Wenqiu Yu for helping implement support for graphical displays. My thanks to Emilio Del Vecchio for assisting with the study of state spill. I also want to recognize the outstanding members of the Rice University ECE and CS departments, many of who represent lifelong friends of mine, and who offered not only advice in research and career advancement but also a respite from the toils of graduate work. Finally, the National Science Foundation provides financial support to my research, for which I am grateful.

Contents

Illustrations

Tables

Chapter 1

Introduction

Modern operating systems are large and complex, resembling an entangled web of components that are difficult to decouple. Existing efforts to modularize OSes and decouple their entities from one another have overlooked the problem of *state spill*, and in general treated state management as an afterthought; they prioritize traditional design principles like encapsulation and hardware-based isolation like privilege level separation. State spill is the phenomenon in which the state of a software entity undergoes a lasting change as a result of handling an interaction with another entity. The lack of understanding of and focus on state management in existing OSes motivated us to dig deeper, leading us to identify state spill as a root problem and core obstacle to many desirable computing goals, such as system evolution, flexibility, and availability.

> **Key Research Hypothesis***: Fundamentally redesigning an OS to avoid state spill will make it easier to evolve and recover from faults.*
> *Leveraging the power of modern languages and compilers can further help.*

This thesis consists of two related parts. The first part presents a thorough study of state spill in existing systems, including both qualitative and quantitative analysis of state spill and how it manifests in software entities to hinder the realization of said computing goals. This comprises Chapter 2 and Chapter 3, the latter of which presents the StateSpy tool to automate the detection of state spill in Android system services. We intend for this to increase awareness of state spill and its harmful effects, hopefully encouraging careful thought of how states are managed in large software systems.

The second part presents the design and implementation of Theseus*, a new operating system written from scratch in Rust to rethink OS structure and implementation philosophy. Theseus treats state management as a first-class concern, prioritizing the avoidance of state spill between its entities above all other concerns, such as performance or ergonomics/convenience. Our experience designing and implementing Theseus constitutes the majority of this thesis, comprising Chapter 4 through Chapter 7.

The Theseus Operating System

During several years of experimenting with Theseus's design and implementation, we realized that in order to minimize state spill and to achieve better state management in general, we must rethink OS structure altogether; in retrospect, this is obvious because state spill by definition depends on how the OS is modularized. We also realized that modern safe languages like Rust can be used not just to write safe OS code but also to statically ensure certain correctness invariants for OS behaviors. Thus, it became our goal to fully maximize the power of Rust's language features and its compiler, for example by shifting as many OS responsibilities into the compiler as possible, such as resource management.

Theseus is the primary contribution of this thesis. The overall design of Theseus specifies a system architecture consisting of many small distinct entities that can be composed and interchanged flexibly at runtime. All entities exist in a single address space (SAS) and execute at a single privilege level (SPL), building upon language-provided type and memory safety to realize isolation rather than hardware protection, a choice inspired by Singularity [1] and other safe-language OSes. Theseus establishes three key design principles:

P1: Require clear, *runtime-persistent* bounds for *all* entities.

*The *Ship of Theseus* is a paradoxical thought experiment that asks if every piece of a wooden ship is replaced over time, is it still the same ship?

P2: Maximize the power of the language and compiler.

P3: Make entities free from *state spill*.

Consisting of many tiny entities with clearly-defined, runtime-persistent bounds, the Theseus OS structure facilitates evolution, flexibility, and availability. It enables the system to maintain robust metadata about which entities currently exist and to track their bounds and the interdependencies between them; this offers a well-bounded "atomic" unit for change that is identifiable even at runtime and extends to all entities, from core kernel components to libraries and applications. Entities in Theseus are represented as *cells*[†], which have a consistent form from implementation time to compile time to run time. Their consistent form offers a unified view of the OS to developers and eases human understanding of the complex system. Thus, the logic for interchanging cell entities becomes safer and simpler, a key primitive in Theseus. Interchanging cells realizes live evolution and flexibility by replacing old entities with new or differently-configured ones, and fault recovery by replacing corrupted or failed entities with fresh instances.

More importantly, Theseus contributes the *intralingual* OS design approach, which entails (*i*) matching the OS environment to the runtime model of its implementation language, and (*ii*) implementing the OS itself (e.g., resource semantics) within the strong, static type system offered by modern languages. Through intralingual design, Theseus maximally empowers the compiler to apply its safety checks with no gaps in its understanding of code behavior. As a result, Theseus approaches end-to-end safety from applications to core kernel entities, and shifts semantic errors from runtime failures into compile-time errors, both to a greater degree than existing OSes.

The practical benefits of language safety are well-known; large-scale studies from Microsoft [2], Chromium [3], and Mozilla [4] have reported that ~70% of serious bugs

[†]Theseus's cell has no relation to Rust's `cell` type.

are memory safety bugs and have identified Rust as a potential cure. Notably, intralingual design in Theseus goes *beyond* safety, enabling the compiler to statically check OS semantic invariants (e.g., proper resource release) and assume resource bookkeeping duties; this reduces the states the OS must maintain, in turn reducing state spill and strengthening isolation.

Realizing Computing Goals and Evaluating Theseus

Chapter 8 demonstrates the utility of Theseus's novel structure and intralingual, spill-free design by showing how we realize *live evolution*, *fault recovery*, and flexible *OS personalities* within it. We then evaluate how well Theseus achieves these goals in Chapter 9. Through a set of case studies, we show that Theseus can easily and arbitrarily live evolve any entity within the OS, even core system components, in ways beyond that of prior live update works, e.g., joint application-kernel evolution or evolution of microkernel-level entities. We also demonstrate Theseus's ability to tolerate transient hardware faults that manifest in core system components; by design, Theseus can recover from all language-level faults as well. To this end, we present a large study of fault manifestation and recovery in Theseus, and a carefully designed comparison with MINIX 3 of fault recovery for entities that necessarily exist inside the microkernel, e.g., the inter-entity communication (IPC) layer.

Although performance is not a primary goal of Theseus, we find that its intralingual and spill-free designs do not impose a glaring performance penalty, but that the impact varies across subsystems. For example, microbenchmark stress tests show that intralingual memory mapping is slightly more performant, while task and heap management suffer mild to moderate overhead.

As a research prototype, Theseus is far less complete than commercial systems or certain experimental ones such as Singularity [1] and Barrelfish [5] that have undergone

substantially more development. For example, Theseus currently lacks POSIX support and has an incomplete standard library. We discuss the limitations of Theseus in Chapter 11. We have not yet evaluated some important OS requirements, namely security and efficiency; we focus on the structure and intralingual design of Theseus and its ensuing benefits for live evolvability and availability. Theseus is currently implemented on x86_64 with support for most hardware features, such as multicore processing, preemptive multitasking, SIMD extensions, basic networking and disk I/O, and graphical displays. It represents roughly four person-years of effort and comprises ~38000 lines of from-scratch Rust code, 900 lines of bootstrap assembly code, 246 crates of which 176 are first-party, and 72 unsafe code blocks or statements across 21 crates, most of which are for port I/O or special register access.

Theseus is open-source and is available at http://www.theseusos.org.

Chapter 2

State Spill Overview

We now define the concept of *state spill* and describe the conditions under which it occurs. For details about Theseus OS itself, please see Chapter 4. State spill is the phenomenon in which the state of one entity `B` undergoes a *lasting change* after handling an interaction with another entity `A` [6]. State spill causes entities to be irrevocably and tightly coupled together, preventing the entity `B` ("server") from being swapped or modified because its changed state must be preserved to satisfy the expectations of entity `A` ("client"). We focus on such scenarios where the client `A` (e.g., an application or system task) has a future dependency on prior state it spilled into the server `B`, primarily server entities that provide core OS functionality. Interchanging an entity that harbors state spill will cause hard crashes or undefined behavior due to missing or inconsistent states; thus, state spill poses a significant obstacle to many desirable computing goals: process migration, fault isolation, fault tolerance, live update and hot-swapping, virtualization of software entities, maintainability, data privacy and security, and more. In this work, we are primarily concerned with evolvability and availability, with a secondary focus on flexibility.

2.1 System Model of Interacting Software Entities

In order to properly define state spill and the conditions under which it occurs, we first outline a system model that assumes a nested hierarchy of abstraction (Figure 2.1). Each layer of abstraction contains one or more *software entities*, a generic term that covers

common programming and runtime abstractions, e.g., processes (distinct address spaces), threads, modules, classes, functions. The various granularities of software entities and how they pertain to state spill are described further in §2.2.

2.1.1 Transactions: Inter-Entity Communication

We model all communication between entities using the notion of *transactions*, which have IPC-like semantics.A *transaction* is the flow of control (and optionally data) from entity `A` to `B`, in which `A` initiates the transaction and `B` receives and handles it, returning control to `A` upon completion.

Transactions can represent procedure calls, system calls, interrupts, signals, and more. Asynchronous interactions like message passing that do not require blocking control flow transfers can be modeled as two separate transactions. The exact incarnation of a transaction is defined by which entities are involved; for example, a function call is a transaction from one function to another function in the same thread or between modules that contain said functions; system calls are transactions from a userspace entity to a kernel one; IPC is a transaction from one process to another. Transactions have the following explicitly-defined characteristics:

- If a transaction is interrupted (e.g., preempted), it is simply treated as incomplete until it resumes and returns control.
- If a transaction is divergent and never completes, such as a call to an infinitely-looping function that listens for messages, no future actions from different source entities can be affected by its changes; thus, it is irrelevant to state spill.
- If a transaction handler fails, e.g., by returning an error, it is treated the same as a completed successful transaction.

We are primarily interested in short-lived transactions that leave a lasting change on the destination entity `B`, such that the future behavior of entity `A` will be affected by and depend on that state change.

2.1.2 Active and Inactive Entity Conditions

In assessing state spill, we are interested in the effect of a *single* transaction on an entity; thus an entity's state only matters at certain points: before and after a transaction, not during. We use the term *inactive* to identify the stable condition of an entity when it has no in-progress, unfinished transactions from external entities. When a transaction is in progress, the entity is considered *active*. Given how communication in our system model is entirely transaction-based, the inactiveness of a software entity is determined solely by its transaction handling.

This differs from traditional definitions of quiescence, which typically specify that a given entity must be completely suspended, sleeping, or absent from all CPU runqueues [7, 8]. Other definitions of quiescence go even further by saying that an entity cannot be considered quiescent if any of its functions are on the call stack of any other process [9, 10, 11, 12]. Epoch-based quiescence utilizes an interposition layer to mediate access to a given entity, ensuring quiescence is reached once all other entities are finished interacting with the mediated entity [13, 14].

These interpretations of quiescence are far too strict for the needs of state spill analysis, as they would cause nearly every entity to *never* reach quiescence, especially those that execute in the background or never terminate, e.g., system services and the kernel. While such definitions cause many event types to prevent quiescence, e.g., scheduler preemption, our definition simplifies it to occur *only* upon the completion of inter-entity transactions. This is necessary because, although an event like scheduler preemption may cause the entity

to be non-runnable, that entity may still be in the midst of handling a lengthy transaction when it is blocked or otherwise taken off the runqueue. Note that Theseus still abides by existing forms of quiescence for realizing live evolution, in particular the call stack-based form.

2.1.3 State of a Software Entity

The state of a software entity consists of the set of program variables within the entity's scope, and the values of those variables. This definition confines entity state to include only the information within the entity's scope, i.e., everything visible within the entity's logical bounds. For example, the scope of a *module* includes the local variables in the functions within that module and the global variables that they access. The scope of a process entity is identical to the scope defined by the program executing within it, which includes all program variables but excludes the process control block and other OS state external to the process. For class object entities, all class member fields are considered to be in-scope throughout the entire duration of that class object, in addition to any local variables and public static members accessible from the current execution point.

When an entity is inactive, its state becomes a refinement of general entity state, exclusive of temporary states. A *temporary state* is a program variable whose underlying lifetime [15] (duration in memory) does not persist beyond the transaction currently being handled by the entity. For example, local variables with automatic lifetimes are temporary states, whereas class member fields or global variables in a process are non-temporary. Only lasting changes that stem from a transaction can present complex challenges to the aforementioned computing goals. Changes to temporary states are unimportant and irrelevant to state spill because they are entity-local and cannot affect the complex interdependencies and coupling between entities.Recall that our definition of active and inactive isolates the specific state

changes due to a single transaction. This, combined with the elimination of temporary changes, leads to more accurate analyses with meaningful results: state spill that actually matters.

Defining software entity state in this way necessarily enables us to treat varied OS components in a consistent way amenable to static and dynamic (runtime) analysis. Another benefit of this definition is its Markovianness, i.e., the exclusion of prior input, events, or control flow from an entity's state. This allows easy analysis of an entity: only current contents are considered, prior condition is disregarded.

To determine whether state spill has occurred as a result of a given transaction, one must compare the non-temporary variables in entity `B` before and after that transaction. Because transaction-based entity state ignores the changes due to internal interactions within the entity, one must also ignore all changes from those internal actions and compare *only* states that can be modified by entity `B`'s handling of that transaction. We accomplish this by determining the *modification reachability* of the variables in entity `B`, i.e., the set of variables that can be modified by the transaction handler.

In some systems, state equivalence testing may be impractical due to the size or highly dynamic nature of an entity's state. Although we have yet to encounter such an entity, an approximation of state equivalence testing may be more valuable than a direct comparison of entity state. For example, a good litmus test for state spill may be whether an entity `A` continues operating successfully after random changes are injected into entity `B`'s state without informing `A` of the change.

2.2 Relative Granularity of State Spill

Based on the above definitions, state spill is relative to both the spatial granularity of entities and the temporal grouping of transactions between those entities.

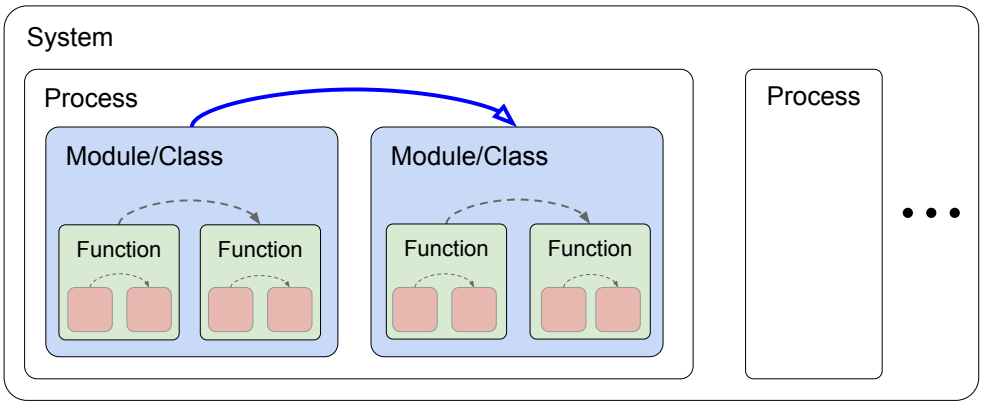


Figure 2.1 : Entity granularities have a nested hierarchy; state spill is by definition relative to the chosen granularity. The module/class-based entity choice shown here means that state spill can occur only in transactions between those entities (⇨); internal interactions between finer-grained entities (⇢) are irrelevant. In other words, an interaction must cross an entity boundary to qualify as being eligible to cause state spill.

2.2.1 The Choice of (Spatial) Entity Granularity

Because transactions are by definition between two entities, state spill depends greatly upon the spatial granularity of the involved entities, i.e., how their bounds are defined. Entity granularities range from the coarsest level of considering the entire computer as a single entity to the finest level of considering each individual function or even each basic block as an entity. At the highest granularity level, state spill occurs from one computer to another via network transactions; any interaction between applications within a computer is irrelevant. Moving down a level, if an entire multi-process application was considered a single entity, then state spill happens from one application to another due to inter-application transactions; any interactions within that application's processes or threads are irrelevant. At the lowest level, in which each function is considered its own fine-grained entity, transactions between functions (function calls) across the entire system are eligible to cause state spill, making state spill ubiquitous, less meaningful, and not actionable. Thus, both very high and very low levels of entity granularity are neither useful nor appropriate for analysis, not to mention

impractical.

The choice of entity granularity for state spill analysis is both platform-specific and goal-specific. For example, if one wishes to determine the barriers to process migration in a Unix-like system, each process should be its own entity such that state spill would be detected in transactions beyond a process's address space. If one wishes to live update a single function in isolation without having to worry about its dependencies on other functions, entity granularity should be set to a single function in order to track inter-function state spill. For analysis of Android system services (§3.3), a class-based entity granularity is more appropriate because each service is implemented as an OOP class.

In our study of state spill, we take the middle ground by choosing a moderately coarse entity granularity, effectively that of a module, which is a group of related functions and data. In Theseus, this takes the form of a cell, described in Chapter 5. The coarsest granularity of an entire computer as one entity has already been studied and addressed in the form of RESTful software architectures [16] used across web services [17]. In theory, RESTful design principles ensure that transactions between client and server systems are self-contained and require no pre-existing state to be held on the server, preventing state spill from the outset. We choose a finer granularity than whole-computer entities because state spill in mid-level entities is more relevant to the systems research goals we target.

The choice of module-like entity granularity manifests as the following condition: only interactions destined for externally-visible entry points in the destination entity are considered transactions capable of causing state spill, whereas interactions between internal functions within a single entity are beneath the cutoff and cannot cause state spill. This distinction classifying whether state spill can occur is illustrated by the solid vs. dashed arrows in Figure 2.1.

2.2.2 Temporal Granularity of Transactions

In addition to the spatial dimension of entity granularity, state spill also has a temporal dimension: the granularity of a transaction. Temporal transaction granularity specifies how many consecutive interactions between two given entities are grouped together into a single logical transaction. The exact form of a transaction is orthogonal to its temporal granularity and dictated solely by entity granularity, e.g., a transaction between two function entities *must* be a function call.

While this work focuses on analyzing one transaction at a time, it can be useful to collapse multiple transactions into one. When analyzing state spill in a common procedure of three transactions in series, e.g., connecting to and configuring updates from a sensor service, there is no point in determining which states are spilled during the first two intermediary transactions. In fact, doing so during a tool-based analysis would create unnecessary overhead, especially considering that state spill only *needs* to be analyzed with respect to two points of entity inactivity, not an individual transaction. Thus, a transaction's temporal granularity is another dimension we consider when understanding and detecting state spill.

Chapter 3

Analysis of State Spill in Existing Systems

We now take a broad look at various real-world operating systems to demonstrate the ubiquity and harmful effects (**boldfaced**) of state spill therein.

3.1 Qualitative Analysis and Classification of State Spill

We contribute a classification of state spill based on various entity design patterns: *indirection layers*, *multiplexers*, *dispatchers*, and *inter-entity collaboration*. Although other forms exist, these concisely represent the vast majority of state spill in OS entities and their roles in causing it.

3.1.1 State Spill in Indirection Layers

The first form of state spill, depicted in Figure 3.1(a), occurs when a client interacts with an *indirection layer* to access a lower-level resource. Indirection Layers can be any horizontal slice in a multi-layered software stack: for example, an application (client) can use APIs provided by a library (indirection layer); a system service (client) can access I/O devices through the kernel driver (indirection layer). The indirection layer translates client requests expressed at a high level of abstraction into lower-level commands that the underlying resource can understand; in doing so, the indirection layer harbors spilled state from the client entity, e.g., information about how the client is utilizing the resource. Developers of indirection layer entities necessarily choose to store client-specific information internally

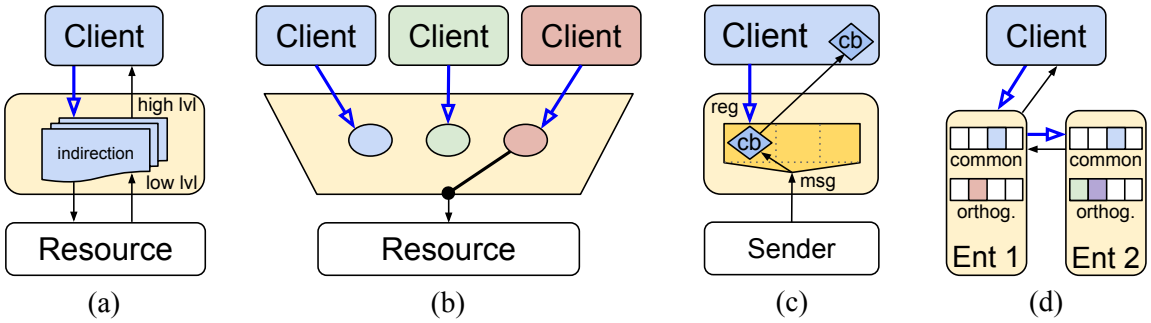


Figure 3.1 : Common design patterns that cause state spill (⇾). (a) Indirection Layers cause state spill when converting between two representations of data/commands; (b) Multiplexers harbor state spill when serving multiple clients; (c) Dispatchers harbor state spill by holding registered callbacks (`cb`) for message/event delivery. (d) Inter-entity collaboration causes state spill when non-orthogonal states must be synchronized.

to (*i*) ensure that access privileges are upheld, and (*ii*) maintain a notion of client progress during a series of client-resource translations.

Common Abstractions: Processes and VFS

The process abstraction is an indirection layer that allows unprivileged user programs to safely access the CPU without understanding the underlying hardware. Most monolithic OSes implement the process abstraction via an indirection layer in the kernel and maintain per-process metadata as a single list therein, e.g., Linux kernel's `task_struct` list. The user program running atop and utilizing the process abstraction will cause state spill beyond its bounds into `task_struct`'s member variables and others within the kernel's process management subsystem.

This state spill stems from a design choice to have all process-related information in a convenient centralized place, and also from the need to protect such information from malicious userspace tampering. However, it renders **process migration** infeasible because one must track and retrieve states spilled from a process into other OS entities (the kernel and

system daemons), a complex and impossible task. This is a recognized problem previously termed "residual dependencies" [18, 19], but is a symptom of state spill. As discussed in the next section, microkernel OSes also harbor similar state spill, but within userspace servers that implement indirection layers rather than the kernel.

The Virtual File System (VFS) indirection layer, present in monolithic and microkernel OSes alike, provides a simple file abstraction for user processes (clients) to access low-level device drivers (resources). We take Linux's VFS implementation and I^2C driver as an example. The VFS entity is itself an indirection layer, whose state includes a `struct file` that manages and holds references to the underlying I^2C device and driver as part of its `private_data` element. When a client process issues an `ioctl` transaction on that VFS entity's *device file*, the kernel routes the operation from the VFS layer to the corresponding driver's `ioctl` implementation, `i2cdev_ioctl` in the case of I^2C. Because it was initiated by VFS, this driver holds a reference to the VFS's I^2C `file` structure and its functions can directly modify VFS states, e.g., file mode, read-ahead, mutex, ownership permissions; other operations like `read` and `write` behave similarly and modify other VFS entity states, e.g., its `file_pos` offset.

This is a deceptive form of implicit state spill: the VFS layer *appears* to not modify its own state or keep data passed in from userspace, but rather its state is modified transparently by the driver entity hidden beneath the abstraction provider itself. This design choice was made for convenience and efficiency reasons: since the VFS and driver entities share a single kernel address space, it is easier for developers and faster to directly update the elements in the VFS's `file` structure using shared references rather than explicit message passing. Similar state spill exists in the userspace VFS servers of microkernel OSes like MINIX 3, Genode, and others.

Microkernel Userspace Servers

Microkernel-based OSes like MINIX and seL4 move the vast majority of OS functionality into userspace servers in favor of a very small kernel core. These servers are often indirection layers that act as middlemen between applications and the microkernel, e.g., converting POSIX API calls into MINIX-specific functions that their microkernel can handle. This software architecture results in state spill that directly impedes **live update** and **hot-swapping** of microkernel servers, as evidenced by the MINIX authors' tedious, ad-hoc undertaking to enable live updates in MINIX 3 [11].

Besides abstraction levels, userspace servers also bridge privilege levels: userspace servers have more privileges than applications but less than the kernel, e.g., they cannot do context switches or top-half interrupt handlers. The introduction of different privilege and abstraction levels is a design choice that prioritizes modularity and the minimization of kernel size at the cost of high susceptibility to state spill. For example, MINIX 3's userspace scheduler SCHED [20] is an indirection layer that sits between user processes and the microkernel's context switch *mechanism* to control the system's scheduling *policy*. SCHED lacks context switch privileges, so it simply chooses the next target process to be run and relies upon the kernel's context switching mechanism (`sys_schedctl`). The `sys_schedctl` system call copies process-relevant parameters into the kernel's list of `struct procs` — a prime example of state spill — before actually triggering the context switch. Spilled states include the target process's scheduling flags, endpoint, parent endpoint, priority, timeslice quantum, CPU affinity mask, schedule-enqueue bit, and more.

The same userspace-policy kernelspace-mechanism structure is used for many other MINIX 3 servers, such as the process manager (§3.1.2), virtual file system, memory manager, and reincarnation server, all of which cause similar forms of state spill. We omit their details for brevity's sake, but such state spill is a direct obstacle to realizing **live update**, **hot-**

swapping, and **virtualization** of said userspace servers.

3.1.2 State Spill when Multiplexing

The second form of state spill commonly occurs when an entity acts as a *multiplexer* that allows multiple clients to access a single underlying resource by means of sharing it temporally or spatially. As depicted in Figure 3.1(b), a multiplexer entity causes state spill by harboring states that correspond to each client's individual interactions with the resource. This form of state spill typically occurs out of necessity, in that the multiplexing entity must maintain contextual data about its users in order to properly partition and manage its resource. For example, a device driver holds a representation of each process's usage of its hardware peripheral; the virtual memory subsystem contains mappings and other allocation details to multiplex applications' accesses to physical memory.

Process Management

Process Management (PM) entities temporally multiplex access to the underlying CPU (resource) among multiple user processes (clients). They maintain metadata about every client process in order to monitor and control them, e.g., deciding when and what to run. This process-specific metadata is a prime example of state spill that exists across many OS designs and negatively impacts the computing goals below.

For example, when one process creates another via `fork`, the PM multiplexer creates a set of data structures to represent that new process's state, a form of implicit state spill. In monolithic OSes like Linux, this spill occurs from a process to the kernel's PM subsystem; in microkernel OSes like MINIX 3, this spill occurs from a process to the userspace PM server. In addition to process creation, other PM actions cause state spill; as processes execute and interact with peripherals and other entities, the PM entities in userspace servers

or in the kernel must update their *process tables* accordingly to reflect the process's new condition. For example, when a process accesses an I/O device by invoking the driver's `read()` system call, the device driver may block that process by setting a flag in its process table entry while fetching the requested data, causing convoluted state spill from the client process to the PM multiplexer via the driver.

Not only does state spill in multiplexers induce the aforementioned residual dependency problem that hampers process migration (§3.1.1), but it also breaks **fault isolation** guarantees by inextricably tying the states of its client entities together in a single entity, causing fate sharing. That is, if one process causes corruption or faults in the PM server, other processes will also be affected. On a related note, **fault tolerance** in multiplexers is impossible in the face of state spill: restoring a failed server instance on behalf of one client process may be successful, but it will fatally disrupt any other clients using that server due to the unexpected absence or change of server-side states [21]. In fact, this is true for most indirection layers and multiplexers, not just PM entities.

The architectural philosophy of monolithic and microkernel OSes may necessitate such state spill, but some experimental OSes reduce it with unique approaches. Genode [22] uses a hierarchical PM technique in which an entity that creates a process maintains metadata and control over that new process. This does reduce the extent of state spill into PM entities, but does not fully decouple the creator or its created process from said PM entities.

Window Management

While the above PM multiplexers are *temporal*, window management (WM) systems are *spatial* multiplexers of graphical resources like framebuffers. Client applications create and manipulate their view contents by submitting requests that contain application-specific state to the WM multiplexer, which assigns a region of the screen/framebuffer resource to

that application. The state stored in the WM entity when handling an application's request constitutes state spill from application to WM entity. WM entities also harbor explicit state spill when receiving configuration data and window content from the applications, which they store directly instead of allowing applications control over their own data.

The X window system [23], though designed as loosely-coupled modules, harbors such state spill among various multiplexer components. One example is the X server, an entity that multiplexes the local keyboard, mouse, and display resources between many client applications, storing client-specific information in the server entity when a given client requests window creation or content display. Other actions handled by the X server include centralized reparenting of windows and caching of offscreen graphics (for context menus and transient pop-up windows) to avoid client-server round trips; these actions require states from application requests to be kept in the X server entity's local storage, necessarily causing state spill from applications into the X server multiplexer. Another X component that causes state spill is its window manager, e.g., Compiz, KWin, a multiplexer that maintains Z-depth and other layout data on behalf of each application window in order to properly position windows relative to one another.

Nitpicker, the atypical WM multiplexer in Genode [24], forces each client to take ownership of its private views and buffers in an effort to improve inter-client security. Nitpicker then accesses these client buffers via shared memory mappings, only upon an explicit client request. This reduces state spill by allowing clients to allocate and manage their own view buffers, but still harbors some spilled states in the form per-client metadata: depth layering information, thumbnails of each view, keyboard shortcuts, etc.

In WM multiplexers, as with PM multiplexers above, state spill violates **fault isolation**, which makes **hot-swapping** and **live update** exceedingly difficult because the separate entities (e.g., client and X server) cannot be updated in isolation or swapped independently.

Updating multiple separate entities atomically is a requirement for consistency — many live update works [12, 11, 14, 7, 9] devote the bulk of their efforts to determining quiescence and accommodating states distributed across multiple entities — but is practically infeasible for windowing systems with many tightly-coupled components. State spill in WMs does not significantly complicate process migration because a WM entity will likely need to reconstruct an application's graphical window and displayed content on the new target machine post-migration. For **security**, state spill in WM multiplexers is particularly harmful because private content may be revealed to other client windows via the shared multiplexer medium. Due to spilled states stored communally in the multiplexer entity, malicious X client applications may be capable of tampering with other windows, injecting false keystrokes or commandeering them for keylogging purposes, or stealing the contents of shared buffers (e.g., clipboard data), and more [25]. A crafted request from an X client can even cause the X server to overwrite arbitrary memory of window buffers spilled into its multiplexer entity [25], causing arbitrary code execution or a denial of service for clients reliant upon those buffers.

3.1.3 State Spill in in Dispatchers

The third form of state spill arises in *dispatchers* that allow client entities to register callbacks in order to receive events or messages from a sending entity, as depicted in Figure 3.1(c). For example, an application that needs to wait for a particular event may register a callback with the daemon or kernel that receives the event; this is very common in event-driven programming models. Dispatchers necessarily cause state spill by holding client-provided callback references in order to properly route communication between entities. In fact, this frequently appears as a subcomponent of other patterns; for example, the X server (client) registers callbacks in the kernel's I/O driver (multiplexer) to receive HID events.

In many IPC implementations, dispatchers cause state spill; for example, System V IPC dispatcher entities in the kernel maintain `ipc_perm` keys mapped to external reference IDs and synchronize `msgque` vectors and `task_struct` IPC status bytes. Unix domain sockets, inotify, and d-bus subsystems are similar. Although signals do not necessarily require callback registration, the Linux kernel's signal dispatcher can cause state spill when, for example, the signal `blocked` mask or `sigaction` array is changed in a given process's task descriptor, or a signaled process is blocked. Finally, mutexes and locking features can also contribute to state spill; for example, the semaphore implementation in MINIX 3 is a dispatcher entity that implements mutex by adding references to the waiting processes in the PM multiplexer's process table, such that it can dispatch mutex release events to those waiting processes. These references, although necessary, do constitute state spill when a process utilizes semaphores.

In the Swift system [26], a lower-layer entity (sender) is able to invoke a procedure in a higher-layer entity (client) using the *upcall* mechanism offered by an intermediate entity (dispatcher). State spill occurs because the higher-layer entity must *downcall* into the dispatcher to register itself such that future upcalls can deliver the correct execution context for the higher-layer entity. Furthermore, because key procedures in the lower layers of the Swift system rely on the correct implementation of higher-layer procedures, à la polymorphism, those procedures tend to access common data across a "vertical stripe" spanning several layers.

The author of Swift recognized this issue and attempted to rectify it by mandating that all client-facing common data be unlocked and consistent before an upcall; however, this guarantee was difficult to implement. Any data in this vertical stripe *not* protected by mutexes contributes to state spill, resulting in interdependent coupling between layered entities, a direct challenge to **fault isolation**, **fault tolerance**, and **maintainability**. In fact,

this problem and other symptoms of state spill in upcall mechanisms, such as unpredictable control flow hazards, are barriers to proper **security**; hence, dispatchers supporting arbitrary upcalls to userspace were never accepted into the mainline Linux kernel [27, 28].

3.1.4 State Spill due to Inter-Entity Collaboration

Finally, the fourth form of state spill occurs when multiple entities communicate with each other to ensure correctness and consistency in their view of the OS, as depicted in Figure 3.1(d). This stems from a desire to accomplish separation of concerns, in which a complex OS feature like process creation is broken up into a series of smaller duties that are each assigned to specific entities (e.g., system services, daemon processes). On paper, each entity's responsibilities and states are orthogonal, leading to a more modular software architecture; however, in reality, each entity has *common* states that are either shared with or dependent upon other entities. The common information — not necessarily identical replicas — must therefore be synchronized, causing state spill that harms **maintainability**, **live update**, and more.

For example, MINIX 3 divides such complex OS tasks among multiple userspace servers, each of which maintains its own version of key data structures, e.g., process tables, to fulfill its duties. When an application on MINIX 3 wishes to set its `uid` or `gid`, the PM server is the first to receive and handle that call. However, the PM's duties do not extend to the filesystem, so it must ask the VFS server to complete the file-related aspects of the call, and then proceed to collaborating with the memory management (MM) server, SCHED, and so on; each server has its own version of common data structures that must be synchronized. This pattern occurs in many other POSIX calls, e.g., exec, fork, virtual memory functions, and also in other microkernel OSes and Android. This form of state spill is particularly insidious because while the user may anticipate state spill in the transaction target (the PM

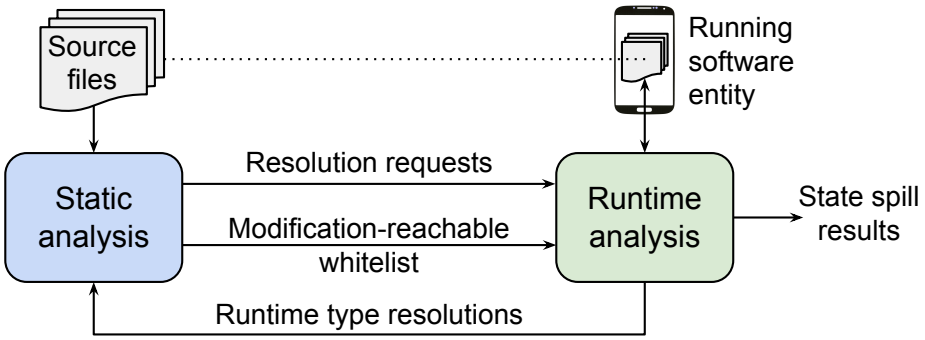


Figure 3.2 : StateSpy employs both runtime and static analysis in a cooperative feedback loop to accurately detect state spill.

server), the synchronization-induced state spill in other entities (VFS, scheduler, MM) is completely obscured.

3.2 Automating State Spill Analysis with StateSpy

In order to detect and analyze state spill in real-world systems, we design and implement StateSpy, a tool suite that automates the discovery of state spill and assists developers in understanding the conditions under which state spill can occur in their entities. Developers simply connect StateSpy to an existing entity running in a real OS and allow it to execute as normal (ideally with a variety of inputs), after which results are automatically outputted containing all detected state spill occurrences and the actions that caused them. StateSpy also accepts as input the source code of an entity to help limit the scope of analysis and prune false results.

As shown in Figure 3.2, StateSpy's runtime analysis component works cooperatively with the static analysis component to generate state spill results. Our approach is related to concolic testing [29] in that it blends symbolic execution-like static analysis with runtime analysis, but inverts concolic testing because our results come directly from the runtime analysis component. As concolic testing begins from a single real execution trace, it may

inadvertently eliminate some potential execution paths based on those initial runtime seed values, causing poor coverage. However, because STATESPY's static analysis conservatively eliminates irrelevant states and not paths, its runtime results cannot miss state spill instances in any Java code path.

The following sections describe the challenges in designing said joint analysis and the realization of STATESPY itself. We implemented STATESPY's runtime analysis and static analysis in 1643 and 2648 lines of Java code, respectively. We strive to keep our design OS-agnostic wherever possible; however, as STATESPY currently targets Android system services (§3.3), some aspects are Java-specific. A tool for native, non-managed systems languages (e.g., C/C++) is easily realizable using techniques similar to those described below.

3.2.1 Challenges of State Spill Analysis

In designing a state spill analysis tool, one must address four main challenges: detecting inactiveness in an entity, capturing an entity's states *with meaningful context*, differencing those states, and filtering out irrelevant results. The first three challenges are addressed by the runtime analysis component of STATESPY, the last by static analysis. An additional challenge stems from the shortcomings inherent in runtime analysis and static analysis when trying to jointly use both techniques; this is addressed in §3.2.4.

Detecting Active or Inactive Entities

In the general case, it is difficult to determine when an entity is active or inactive because they require environment-specific knowledge. However, our *transaction-oriented* definition simplifies this: we only need to detect transactions incoming to an entity and when those handlers have completed. In Android, this is relatively straightforward because transactions

between entities have a clear entry and exit point, due to the strict nature of the Binder IPC protocol. Thus, StateSpy can simply monitor the entity's execution to pause threads at the beginning of a specified Binder method (e.g., `onTransact`), creating an arbitrary period of inactivity. In other systems, StateSpy can accept developer-defined entry and exit points to determine what constitutes a transaction, e.g., publicly-exposed functions or other pieces of an API.

Capturing a Software Entity's State

Capturing the state of an arbitrary software entity remains a hard problem that revolves around a tradeoff between genericness and accuracy. State capture approaches fall under two broad headings: (*i*) capturing an entity's underlying memory contents beneath the runtime environment, or (*ii*) capturing its state contents at the language level.

The former approach (*i*) is generic — no language-level support, understanding, or modification is necessary — but requires bridging the gap to recover the semantic knowledge of those captured memory contents, which remains an open problem in the forensic science domain. For example, if the kernel transparently captured a user process's address space, it would not know which memory contents represented which states. Thus, although this approach could support arbitrary software entities, classifying entity states (or their underlying memory) as temporary, modification-reachable, or any other trait would be infeasible, preventing us from determining whether they contribute to state spill.

The latter approach (*ii*) is less generic — it only works for a given language or runtime environment — but preserves contextual metadata like variable descriptors and type information. Meaningful context is vital to understanding and classifying states in an entity to accurately detect state spill. Therefore, we adopt a language-level approach that avoids the gap between state values and their semantic meanings, allowing StateSpy to preserve

states in their original form for analysis. The challenges in designing language-level state capture approaches are best evidenced by the following shortcomings of existing approaches.

- **Static analysis** cannot always deterministically guarantee whether a given state *will* change, only that it *may* potentially change. In addition, its inability to accurately resolve abstract types or methods prevents the full traversal of all possible execution paths.

- **Record and replay** is generally easier to implement than checkpoint-based state capture, but cannot inspect the *actual contents* of changed states, just the actions that evoked those changes.

- **Serialization** requires special support from the language on a per-type basis, which most legacy systems do not offer, and is infeasible to implement generically.

- **Runtime instrumentation** requires modifying the runtime to expose state information, an incredibly complex approach that risks disrupting the entity's behavior or even violating its correctness. Also, any executable code that is compiled into native or machine binaries will bypass the runtime, making it impossible to interpose upon that code even with the proper hooks available. Although runtime plug-ins allow for this unreliable introspection, many runtime implementations lack support for standardized features and hooks that these plug-ins rely on, e.g., Android's ART/Dalvik runtime.

To overcome these challenges, our key insight is to leverage *debugging frameworks* that already exist in the runtime or execution environment in order to non-invasively capture entity state (§3.2.2). Exploiting debugging extensions is a flexible technique generalizable to nearly all other platforms and systems, providing access to entity states with the full contextual information necessary for state spill analysis.

Differencing Captured States

In order to determine whether a transaction has resulted in state spill, one must difference an entity's state before and after that transaction when it is inactive. This is difficult because each state must be compared according to its underlying details, e.g., variable type and size. Essentially, this requires a full semantic understanding of each state, because comparing a list of integers is different than comparing two custom `structs`. Fortunately, the correct differencing of states goes hand-in-hand with the proper capture of states above; that is, utilizing debugging frameworks also provides sufficient state metadata for StateSpy to conduct proper state comparison.

In addition to the challenge of correct semantic state comparison, one must address the challenge of representing captured states in a way that supports arbitrary structure, circular references, and hierarchical relationships among states. For this, StateSpy builds a tree-like cyclical digraph that mirrors the structure of object states in the running entity (§3.2.2). With a proper structural representation and semantic understanding of entity states, we can apply existing tree comparison algorithms to identify which states changed during a transaction, i.e., occurrences of state spill.

Filtering Results

Delivering only relevant state spill occurrences is difficult because there is no ground truth for what actually constitutes state spill, aside from an expert developer's determination. Runtime analysis cannot differentiate between states that were changed as a direct result of the transaction and states that happened to change during the transaction (e.g., by background threads), a condition we term *modification reachability*. This leads to a potential abundance of false positive results, i.e., when a state changed during a transaction but not due to that transaction. To remedy this, we rely on static analysis to assess the modification reachability

of all states in an entity, described in §3.2.3.

3.2.2 Runtime Analysis Design

StateSpy's runtime analysis component detects state spill according to the procedure in §2.1.3: an entity's state is captured twice, at inactive points before and after a transaction, and then differenced to identify spilled states. To address the first three challenges above, we leverage language-level debugging frameworks that can determine if an entity is active, non-intrusively access its state, and obtain full contextual metadata to gain a semantic understanding of all entity states. Despite inherent language-specificness, we avoid environment-specific features — watchpoints are unsupported on many JVMs like ART/Dalvik — and strive to keep our key design concepts language-agnostic, such that the core ideas are portable even though the implementation is not. Utilizing debugging frameworks does indeed have many advantages:

- All managed language runtimes offer robust debugging extensions, even those that do not support instrumentation or specialized plug-ins;
- Introspection via debugging hooks is risk-free and cannot compromise the correctness of the entity or runtime;
- Debugging support is also available in execution environments without an underlying runtime or virtualization layer, such as native C/C++ processes and Rust programs.

The runtime analysis component of StateSpy is designed as a standalone application that runs in a separate *host* runtime from the *target* runtime, which contains the entity under analysis. It builds upon standard debugger hooks, e.g., breakpoints, variable inspection, expression evaluation, in order to pause the target runtime and induce inactiveness before accessing the state contents within. In our Java-specific implementation of StateSpy's

runtime analysis, the software entity of interest runs in the target JVM runtime and consists of a Java *object*, an instance of a Java class. That class contains transaction-handling methods that modify the object's states, i.e., its member fields.

The full procedure undertaken by the runtime tool is:

1. Attach to the target JVM process, and wait for inactiveness by establishing breakpoints at the entry of every transaction handler method in the entity object.
2. When an entry breakpoint is hit, induce inactiveness by suspending threads in the target JVM. Then, capture the full state of the entity object into the host JVM as the *initial* state.
3. Set breakpoints at the exit points of the current transaction handler method and resume threads in the target JVM.
4. When the exit breakpoint is hit, induce inactiveness yet again by suspending the target JVM's threads. Capture the full state of the entity object as the *final* state.
5. Disable the exit breakpoints, re-enable all entry breakpoints from Step 1, and resume threads in the target JVM.
6. Finally, in the host JVM, compute the difference in pre- and post-transaction entity state (*final* $-$ *initial*), which represents the state spill caused by that transaction.

However, even with debugging frameworks available, capturing the full state of an entity class object is surprisingly non-trivial. We cannot simply create a shallow copy or duplicate the references to the object's state inside the target JVM, for two reasons: (*i*) it would cause the pre-transaction state values to be overwritten by the post-transaction values, as the two objects are one in the same and cannot co-exist; (*ii*) it violates our guarantee that we will not intrusively modify the target JVM's internal states.

Instead, STATESPY must fully explore the entity's complex object graph, starting at the top-level class definition and recursively following all of its field references to other subclass instances. This depth-first exploration continues until a bottom-level primitive object is reached, at which point a custom *tree-based representation* of the object graph is generated in the tool's memory (host JVM) that mirrors both the object references and primitive values of the entity in the target JVM. This tree representation bears several advantages: it matches developers' intuition and is amenable to existing differencing algorithms and visualization tools.

To reduce execution time and storage space, STATESPY cherry picks which object references or primitive values to include or exclude. This is accomplished through the modification reachability whitelist from the static analysis component, but can also be manually adjusted by developers. In addition, STATESPY is aware of an object's type, semantic value, and any references to it; this enables construction of trees with uniquely-identifiable nodes, supporting hierarchical containment and circular references while avoiding the redundant capture of already-encountered nodes.

3.2.3 Static Analysis Design

As previously mentioned, the primary shortcoming of runtime analysis above is its inability to distinguish between states that changed during a transaction and states that changed *as a direct result* of that transaction, leading to potential false positives. Therefore, the sole objective of our static analysis is to address that ambiguity, i.e., to solve the *modification reachability* problem. STATESPY's static analysis produces a conservative whitelist including only the states that are or may be modification reachable. This problem is reminiscent of taint tracking and data-flow analysis (see Chapter 10), but handles the additional complex case of implicit state spill that is undetectable by information flow analysis and better

addressed with techniques like symbolic execution. However, data-flow's requirements for source/sink identification and symbolic execution's term-based representation of object values are both inappropriate for determining modification reachability, so we develop our own algorithm.

STATESPY's static analysis algorithm recursively explores every instruction of every method reachable from the entry point of each transaction handler method in the target entity. Starting with an initial set of *variables of interest* (VOI), containing the target entity's non-constant member fields, the algorithm propagates VOI and variables that have been modified or aliased to and from every method invocation. We use the Soot framework [30] to analyze Jimple, a simple intermediate representation of Java, which means that variables can only be modified when on the left side of an assignment statement, and VOI can only be aliased when on the left side of an assignment statement as well.

Effectively, this technique is a form of forward program slicing [31] that selects and analyzes only the instructions in that slice, i.e., those capable of modifying any VOI. However, forward slicing suffers from search space explosion; to mitigate this, we only track and propagate the modification or aliasing of VOI, not the propagation of *all* variables. To further reduce analysis time, we cache which variables are modified and aliased by each method; for the sake of reusability, these are expressed as string literals (e.g., `this`, `return`, `param#`) instead of actual variable identifiers. Then, the next time a cached method is encountered, we instantly know whether the base object, return value, or parameter values will be modified or aliased by that method.

The full algorithm, given in [6], outputs a per-transaction list of modification-reachable fields in the target entity, which is fed into the runtime analysis tool to reduce false positives.

3.2.4 Bridging the Gap with a Feedback Loop

As previously mentioned, we establish a feedback loop (Figure 3.2) between the runtime and static analysis components of StateSpy to address the shortcomings of each. However, this introduces a circular dependency: static analysis alone is unable to fully explore all methods due to ambiguous declared types or multiple candidate implementations for a given method invocation, so it needs type resolution information from the runtime component to accurate resolve these ambiguities; runtime analysis needs resolution requests and whitelists from the static component. However, runtime analysis can only resolve types it encounters in real execution, but does not know which execution paths to explore until the static component requests mappings for missing types.

To address this catch 22, we break the circular dependency by synthesizing artificial test cases that force the runtime tool to traverse an execution path containing the runtime type resolution of the abstract declared type in question, which is then fed back into the static analysis component. We effectively bootstrap the static analysis tool with a few type resolution mappings and the runtime tool with a simple all-included whitelist, and then iteratively increase the type mappings set while reducing the whitelist according to modification reachability. Since all feedback data passed between the components does not change across analysis runs, we save this information persistently for future usage.

3.2.5 Validating StateSpy's Limitations

Currently, our StateSpy implementation strives for completeness over soundness, but guarantees neither. For clarity, StateSpy is considered *complete* if it does not omit any results that are actually state spill (i.e., no false negatives), and *sound* if all results it returns are indeed true state spill instances (i.e., no false positives). We aim to eliminate false negatives while still minimizing false positives. In this case, a false positive is a changed

state identified as state spill by our tool that is actually not true state spill, whereas a false negative is a real case of state spill that our tool missed.

We applied StateSpy to a random sampling of 60 transactions in Android system services and manually verified the results. We found that StateSpy achieves a false positive rate under 11% across these 60 transactions, most of which stems from our immature support for native methods. Without static analysis, the runtime analysis alone results in a false positive rate well over 60% for some services, highlighting the necessity of the modification reachability constraint. Due to the sheer magnitude of transactions in Android and the lack of ground truth, we are unable to accurately assess StateSpy's false negative rate; however, a case study in §3.3.4 shows false negatives can occasionally occur.

As mentioned above, we conservatively include a state in the whitelist if static analysis cannot determine whether it is modification reachable, due to reasons like dynamic dispatch, unfollowable native code, or opaque external type references, Based on our experience, we believe that from a developer's perspective, determining the legitimacy of a state spill result is relatively easy, whereas discovering state spill results that were mistakenly omitted by StateSpy (false negatives) is practically impossible. Thus, false positives are less harmful than false negatives, so we prioritize completeness.

3.3 Analysis of State Spill in Android System Services

While §3.1 studied and classified state spill in a broad variety of OS entities, this section takes a deep dive into Android system services guided by StateSpy. Our results show that state spill is a complex, widespread problem that permeates the vast majority of Android system services and is detrimental to the aforementioned computing goals.

3.3.1 Experimental Methodology

To obtain a large, representative body of transactions for StateSpy to analyze, we downloaded 150 Android applications from various top categories in the Google Play store and utilized the `monkey` [32] UI automation tool to run each application with hundreds of random yet meaningful inputs. It is unimportant which applications are run, only that they invoke a large set of transactions across many system services, which StateSpy can automatically monitor for state spill analysis. We believe that analyzing a varied, real-world set of transactions is preferable to a generated set of contrived test cases that may be unrealistic.

StateSpy is very easy to use with any Java-based system and supports Android system services out of the box; it can automatically detect and create activeness and inactiveness in system service entities, their transaction entry points, and from which application a given transaction originates. Over 96% of services in Android follow the same design structure: a main service class implements one or more Binder IPC stub interfaces, which are auto-generated by Android's build system from Interface Description Language specifications. A stub is an abstract class that most services either directly inherit from or implement as an anonymous inner class; both variants are detectable by StateSpy. Once a stub class is located, we simply assign the Binder `onTransact()` IPC handler as its entry point and monitor it for incoming transactions. All data was collected from stock AOSP running Android Marshmallow 6.0.1 on a Nexus 5 smartphone.

3.3.2 State Spill is Prevalent in Android

Our holistic analysis of Android system services reveals the ubiquity and extent of state spill, as depicted in Figure 3.3. State spill is so deeply embedded in Android services that our tool detected no state spill in less than 6% of the 100+ transaction-handling stubs (AIDL interfaces) we analyzed. We present results on a per-stub basis instead of per-service

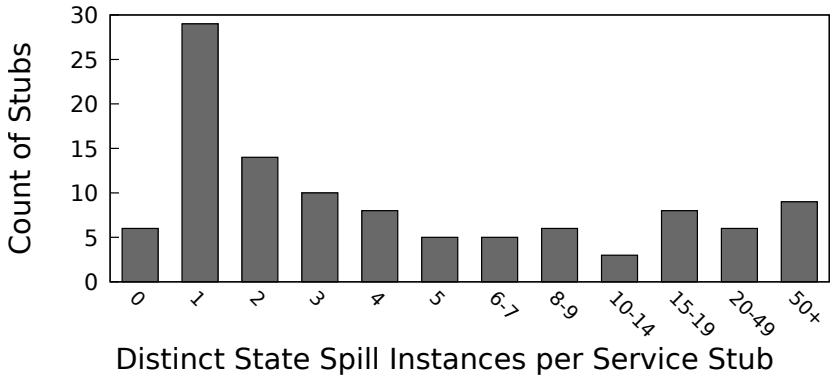


Figure 3.3 : A histogram of how many distinct state spill instances occur within a given Android system service stub. Multiple stubs can exist within a single system service, but a typical service has only one or two.

for granularity and precision reasons, though most services only implement a single stub. Though 76% of service stubs have fewer than 10 *distinct spilled states*, some of the larger stubs have far more, which typically scales with the complexity of the implementing service. A distinct spilled state refers to a single member field in the service entity that was changed by a transaction handler.

Some state spill is relatively straightforward but still hinders various computing goals. For example, callbacks and configuration settings render **process migration** and **virtualization** of those services infeasible. StateSpy detected state spill in Android's `ClipboardService` and `AlarmManagerService` (among others) that jeopardizes **fault tolerance**. When these services restart after a crash, they no longer function properly — clipboard copy/paste and alarms cease to work — and cause user applications to fail mysteriously. If hardened against state spill, these services can be properly restored post-failure to preserve application correctness, as we demonstrate in [33].

Other state spill instances are more unexpected and have complicated implications difficult to observe with manual inspection alone. For example, a secure application can prompt the user to input his/her password by issuing the `verifyUnlock` transaction to

the `KeyguardService`. At first glance, one would think that such a simple transaction is free from state spill; however, StateSpy reveals that the `KeyguardService` causes 3 instances of state spill — in the form of boolean status variables — while handling the `verifyUnlock` transaction. This may represent a potential **security** issue: after a password prompt sets these boolean values, a precise timing attack could allow a malicious application to bypass its own prompt if the original prompt crashes, because those spilled values remain.

Another adverse effect of state spill is the hidden breakage of functionality that applications rely on to ensure security. For example, many banking applications interact with and spill states into the `AlarmManagerService` in order to impose a timeout-based automatic logout, which protects the user's identity and private information in the event of a stolen or misplaced device. A **security** problem arises if that service fails: the timeout will never occur and the application will not automatically log out, despite the application being unaware of the failure and still expecting the timeout to trigger.

To dig deeper into state spill in Android services, we randomly selected 60 service transactions from 21 services (the same set from §3.2.5) for manual classification and analysis. We classified each state spill instance into one or more of the four categories from §3.1, based on the semantics of the service code that causes the spill. These categories are not mutually exclusive; for example, all callbacks spilled into a dispatcher service can be considered communication-related, but only a subset of those are for multiplexing purposes. First, state spill in indirection layers is the most common (39%) because many Android services exist *solely* to provide a simpler high-level abstraction of a lower-level feature. Second, state spill in multiplexers is the least common (21%) because most I/O device multiplexing is implemented in Android's hardware abstraction layer (beneath the service layer). Third, state spill in dispatchers is quite common (36%) because Android offers

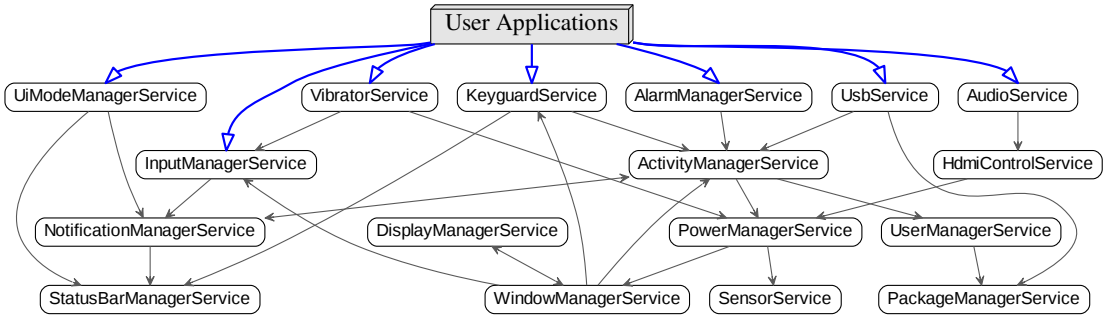


Figure 3.4 : A selected subgraph of complex primary and secondary state spill between applications and system services in Android. The open-ended blue arrows represent primary state spill, while the black arrows represent secondary state spill.

many avenues of communication between applications and services, many of which follow a dispatcher-heavy event-driven programming model. Finally, inter-entity collaboration is a fairly common cause of state spill (23%) because many Android services work together to achieve shared goals, necessitating the tight synchronization described below.

3.3.3 Primary and *Secondary* State Spill in Android

Our analysis thus far has focused on understanding state spill between a pair of entities, from one source to one destination. Interestingly, we discovered that in addition to such *primary* state spill, we also observed *secondary* state spill in Android from the original destination service to another service. This inter-service spill occurs when one service incites change(s) in both its state and in the state of another service while handling a transaction. Secondary state spill occurs when the handling of one transaction in entity `A` results in state spill from `A` $\rightarrowtriangle$ `B` and also triggers another transaction that spills state from `B` $\rightarrowtriangle$ `C`. The states spilled in the latter transaction may originate from either `A` or `B`.

We found 52 transactions across 27 system services that cause secondary state spill,

a lower bound. Figure 3.4 shows a select subset of these services and depicts only state spill, not dependence or usage relationships. This graph highlights the "hot" services that harbor the most state spill, indicating which ones stand to benefit the most from a software redesign.

As an example of secondary spill, applications that interact with the `UiModeManagerService` to change the UI will spill view mode configuration state into that service. That service then causes secondary state spill by inciting a change in both the visibility state of the `StatusBarManagerService` and the notification content state of the `NotificationManagerService`. Similar events transpire when an application displays a notification via the `NotificationManagerService`, which spills state into the `VibratorService` and `AudioService`. These forms of state spill are serious obstacles to **live updating** or **hot-swapping** these services, not to mention a reduction in **maintainability**.

3.3.4 Flux Case Study

To further demonstrate the utility of StateSpy, we compare its results against Flux's manual augmentation of Android system services to support application migration. Flux [19] requires *decorator methods* that selectively record and replay the side effects of a service method (transaction handler), effectively addressing the problem of state spill in those methods, as shown below. The authors of Flux shared with us their full list of decorated methods, which we manually analyzed to determine the ground truth of state spill in each method. We developed an instrumentation tool to trace all Binder transactions invoked by a sample set of applications and ran StateSpy on each transaction. To ensure a fair comparison, we conducted this study using the same version of Android 4.4.2 and the same set of applications used in Flux's evaluation, we ignored all graphics-related services

because Flux circumvents the need for migrating graphics service states, and we ignored the `ActivityManagerService` because Flux heavily customizes its behavior for migration purposes.

Our evaluation captured 113 unique Binder transactions across 39 unique service stubs. We manually analyzed all 113 transactions to evaluate StateSpy's accuracy and found 1 false positive and 3 false negatives, caused by incomplete handling of native methods. Of the 113 transaction methods, 26 were supported by Flux decorators, while 87 were not. Interestingly, 24 of those 26 decorated methods harbor state spill, highlighting the strong correlation between state spill in service methods and the need to understand and augment those service methods (i.e., handle residual dependencies) to ensure correct post-migration behavior.

In addition, StateSpy detected true state spill in 18 of these unsupported 87 methods, indicating potentially incorrect behavior in a Flux-migrated application. For example, state spill in the `AppWidgetService` can cause application widgets on the homescreen to no longer receive and display updates from the migrated application. State spill in two `TextServicesManagerService` methods could cause (*i*) predictive text suggestions to fail when typing, and (*ii*) a loss of Binder death notifications, meaning that applications would not know if the service had failed. The `PackageManagerService` harbors state spill that may result in application components (e.g., services, content providers, activities) that were enabled/disabled on the source device not being properly enabled or disabled on the target device post-migration. Likewise, state spill in the `NfcService`'s `setAppCallback()` method can cause NFC-reliant migrated applications to lose communication with the target device's NFC radio. While it is of course possible for Flux to add support for these 18 methods, this case study demonstrates the utility of the StateSpy tool in identifying transaction methods and spilled states that hinder application migration.

Chapter 4

Theseus: Overview and Design Principles

In the quest to address state spill and its harmful effects in software systems, we design and implement Theseus, a new operating system written from scratch in Rust. The overall design of Theseus specifies a system architecture consisting of many small distinct entities that can be composed and interchanged at runtime. All entities exist in a single address space (SAS) and execute at a single privilege level (SPL), building upon language-provided type and memory safety to realize isolation rather than hardware protection.

Theseus follows three key design principles:

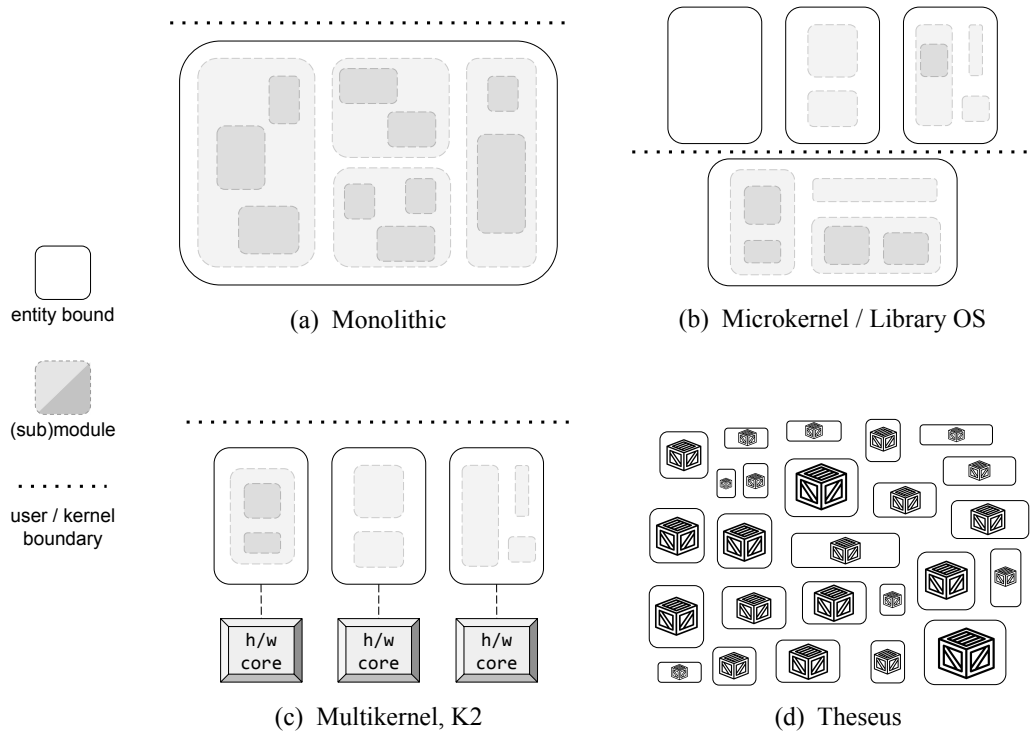
P1. Require *runtime-persistent* bounds for *all* entities.

P2. Maximize the power of the language and compiler.

P3. Make entities free from *state spill*.

Chapter 5 describes how Theseus's structure realizes the first principle of runtime-persistent entity bounds, while Chapter 6 and Chapter 7 discuss the second and third principles, respectively.

As depicted in Figure 4.1, Theseus is neither a monolithic kernel, microkernel, exokernel, nor multikernel, though it does incorporate elements from and benefits of each. Traditional monolithic OSes like Linux have vague boundaries between system entities, such as file-bound or subsystem-bound entities, but beyond entity bounds being ambiguous, they are also completely lost at runtime because the build process statically combines all entities into a large monolithic binary. Like a monolithic kernel, Theseus achieves performance



(a) Monolithic (b) Microkernel / Library OS

(c) Multikernel, K2 (d) Theseus

Figure 4.1 : The structure of entities that provide kernel functionality with clear bounds known at runtime in *(a)* monolithic OSes, *(b)* microkernel and library OSes, *(c)* OSes for manycore/heterogeneous hardware, and *(d)* Theseus. Unlike other OSes, all Theseus entities are elementary in scope (contain no submodules), consistent in form and uniformly representable as cells based on Rust `crates`, and structured independently of hardware layout or protection domains.

efficiency by running kernel code within one address space, and goes further to run *all* code within that single address space. However, monolithic kernels are not trivially decomposable into clearly-defined entities because their entity bounds, if present at all, disappear at runtime. This lack of runtime bounds results from the static linkage of kernel entities, which complicates evolutionary logic.

Microkernels offer clearer entity bounds and stronger isolation than monolithic kernels, but rely on hardware-defined processes for entity bounds and isolation, restricting all future system incarnations to be process-based. Although microkernels export functionality out of the kernel, the microkernel itself is still built into an indivisible code chunk that erases all

notion of distinct entities (**P1**). This prevents the microkernel itself from evolving or flexing at all; only userspace server processes can change, but are too coarse-grained (**P2**) to do so arbitrarily or efficiently. In contrast, Theseus does not move functionality out of the kernel, does not force entities to execute in different privilege domains or address spaces, and can consist of arbitrarily small entities. Importantly, the overall structure of Theseus and its entity form is not derived from hardware features in any way, unlike process-based systems.

Library OS works [34, 35, 36] offer flexibility in the form of *personalities*, in which each application uses its own version of a system component that meets its needs better than a mandatory single system-wide abstraction. However, the underlying exokernel or hypervisor is fixed and is not evolvable, flexible, nor recoverable from faults. Existing personalities lack arbitrarily flexibility because they only allow system functionality to differ on a per-application basis at the process boundary, and are inefficient due to the need to switch between process address spaces and hardware privilege modes. In contrast, Theseus can expose different personalities more efficiently in software without such overhead, and its personalities can inhabit any arbitrary form.

Like a multikernel, Theseus can contain multiple replicated entities (e.g., across personalities), but does so independently of the underlying hardware. This contrasts existing approaches that structure OSes based on hardware characteristics, such as Barrelfish [5], fos [37], and Helios [38] for manycore processors, K2 [39] for heterogeneous processors, and LegoOS [40] for resource-disaggregated systems. Although Theseus does not focus on scalability, its architecture is inspired by and resembles a distributed system of loosely-connected interchangeable components, especially in its state spill-free design (Chapter 7).

In summary, the Theseus design incorporates several beneficial aspects of existing designs: monolithic OS efficiency, library OS flexibility, and a microkernel's clear boundaries of isolation. The next chapter describe its structural design in greater detail.

4.1 Background on Rust

Theseus leverages myriad Rust features to realize an intralingual, safe OS design, as described in the following sections. The Rust programming language [41] is designed to provide strong type and memory safety guarantees at compile time, offering the power of a high-level managed language with the C-like efficiency of no underlying runtime layer or garbage collector. The *ownership* model is the key to Rust's compile-time safety. Each value has a single owner, and the scope of the value is the same as that of its owner; for example, a value on the stack (an object in stack memory) is owned by the local variable bound to it. When the owner's scope ends, the owned value is dropped and freed by virtue of the compiler inserting a call to its destructor. This is inspired by affine types, in which values can be used at most once; after a value is moved to another owner and its ownership thus transferred, the previous owner can no longer use it.

Values can also be borrowed to obtain references to them, and the lifetime of those references cannot outlast the lifetime of the owned value. Rust's compiler includes a borrow checker to enforce these lifetime rules, as well as the core tenet of *aliasing XOR mutability*, in which there can be multiple immutable references or a single mutable reference to a value, but not both at once. This allows it to statically ensure memory safety for values on the stack and heap, in both single-threaded and concurrent execution environments. Though we do not use its standard library, we do use other fundamental libraries (`core` and `alloc`) that contain convenient types like heap-allocated collections and automatic atomic reference counting (`Arc`).

Chapter 5

Structure of Runtime-Persistent Entities

Entities in Theseus must have bounds clearly defined before build time that persist into and throughout runtime, for the entire lifetime of that entity. This applies to *all* entities, not just a select subset such as loadable modules in monolithic systems (Linux), kernel extensions in safe-language systems (SPIN), or userspace servers in microkernels; there are no exemptions for entities in a "base kernel" image. Explicit entity bounds identifiable at runtime are the foundation for strong data and fault isolation, state management that avoids state spill, and system evolvability and flexibility along those bounds.

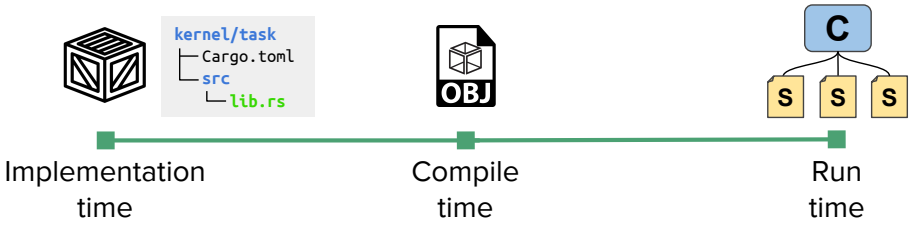


Figure 5.1 : Theseus's cell-based entities provide a consistent view of the OS's structure throughout all phases of an entity's existence, from implementation to compile to run time. Here, "C" is a cell loaded from a crate's object file, and "S" is a section within it, as depicted further in Figure 5.2.

5.1 The Cell is Theseus's Entity Choice

Every entity in Theseus is a *cell*[*], a software-defined unit of modularity that serves as the core building block of the OS, much like their namesake of biological cells in an organism.

[*]Theseus's cell has no relation to Rust's `cell` type.

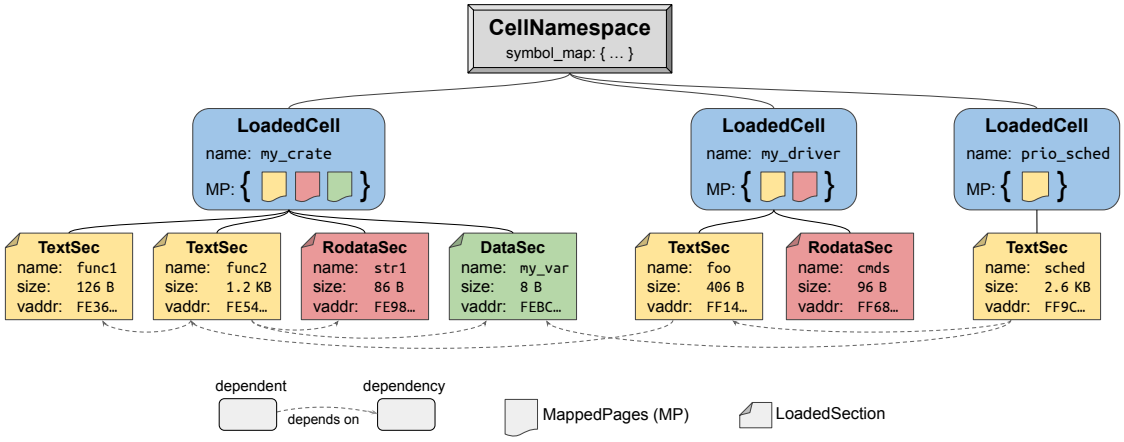


Figure 5.2 : Theseus constructs detailed metadata that tracks runtime cell bounds in memory and bidirectional, per-section dependencies in order to simplify cell swapping logic. The figure legend is in gray at the bottom.

As shown in Figure 5.1, a cell exists as a Rust *crate* at implementation time, a single object file at compiler time, and a set of loaded memory regions with per-section bounds and dependency metadata at runtime. The *crate* is Rust's project container and *translation unit*, an elementary unit of compilation that contains source code and a dependency manifest. We compile each crate into a single object file.

At runtime, Theseus dynamically loads and links *all* cells into the system on demand, e.g., upon first use. Briefly, this entails finding and parsing the cell object file, loading its sections into memory, resolving its dependencies to write linker relocation entries, recursively loading any missing cells as required, and adding new public symbols to a symbol map. In doing so, Theseus is able to construct detailed *cell metadata*, which represents runtime knowledge crucial for live evolution (§8.1), flexibility (§8.3), and fault recovery (§8.2).

Figure 5.2 depicts the structure and contents of Theseus's cell metadata. The main content is the set of all cells that have been loaded and linked into the running system. The node representing each cell tracks that cell's set of constituent sections, the memory regions (`MappedPages` objects, see §6.3.1) that hold those sections, and additional metadata for

cell swapping and other system functions. The sections in each cell correspond to those in its crate's object file, comprising its executable functions (`.text`), read-only data (`.rodata`), and read-write data (`.data` and `.bss`) sections. The metadata node for each section in a cell tracks the following details of that section: its size, its location in memory expressed as an offset into its containing `MappedPages`, its outgoing dependencies, and its incoming dependents, among other items. As such, dependencies are tracked bidirectionally on a per-section basis; a section's outgoing dependencies are the set of sections in foreign crates that it depends on, while its incoming dependents are the set of sections in foreign crates that depend upon it.

The set of loaded cells defines a CellNamespace, a true namespace comprised of all symbols publicly exported by those cells. Each symbol in the CellNamespace is a string that maps to a public section in one of its crates, effectively a system symbol map that is used to quickly resolve dependencies that one cell has on others. There is always a one-to-one relationship between a CellNamespace and a set of cells, and both are runtime constructs.

Persistence of Cell Bounds Reduces Complexity

Theseus's cell metadata maintain the knowledge that a cell at runtime was loaded from an object file which came from one Rust crate, as shown in Figure 5.1. This consistent view of the system reduces not only the complexity of a developer's mental model of the OS but also that of fault recovery and evolution logic, as Theseus can introspect upon and manage its own code from the same cell-oriented viewpoint at runtime.

The SAS-SPL architecture augments this consistent view with *completeness*, in that everything from top-level applications and libraries to core kernel components are observable as cells. In fact, the difference between application and kernel cells is merely a classification for purposes of gating access to privileged functionality. This enables Theseus to (*i*) imple-

ment a single mechanism, cell swapping, that is uniformly applicable to *any* cell, and (*ii*) jointly evolve cells across multiple system layers (e.g., applications and kernel components) in a safe manner due to system-wide completeness of cell dependency metadata.

5.2 Striking a Balance with Cell Granularity

Theseus cells are elementary in scope of functionality and cannot contain one another. We reject Rust's containment-based organizational hierarchy that divides functionality into source-level (sub)modules within one parent crate, as this structure loses submodule bounds when the crate is built into one object file. Instead, we elevate would-be submodules into distinct crates, which we then group into folders in the repository, one per subsystem. We follow separation of concerns to split functionality into multiple fine-grained crates whenever possible, erring on the side of making crates very small, but letting the presence of unavoidable circular dependencies between too-small crates halt further decomposition.

This offers the best of both worlds: a source-level view of a hierarchical subsystem as a single crate that re-exports symbols from its constituent crates to offer a simple and familiar programming interface, and a system-level view of all cells as a flattened tree of distinct object files, one per crate. It also strikes a balance between the impracticality of safely replacing a huge cell monolith, the inefficiency and redundancy of swapping a large cell when only a small part needs to change, and the complexity of needing to swap myriad tiny cells to change one feature.

In Theseus, the notion of an *entity* is decoupled from that of an *execution context*, which contrasts microkernel and hybrid monolithic OSes that modularize entities along userspace process bounds. Thus, Theseus can manage cells orthogonally from managing tasks, e.g., allowing a cell to be evolved and replaced without killing a task that depends upon it.

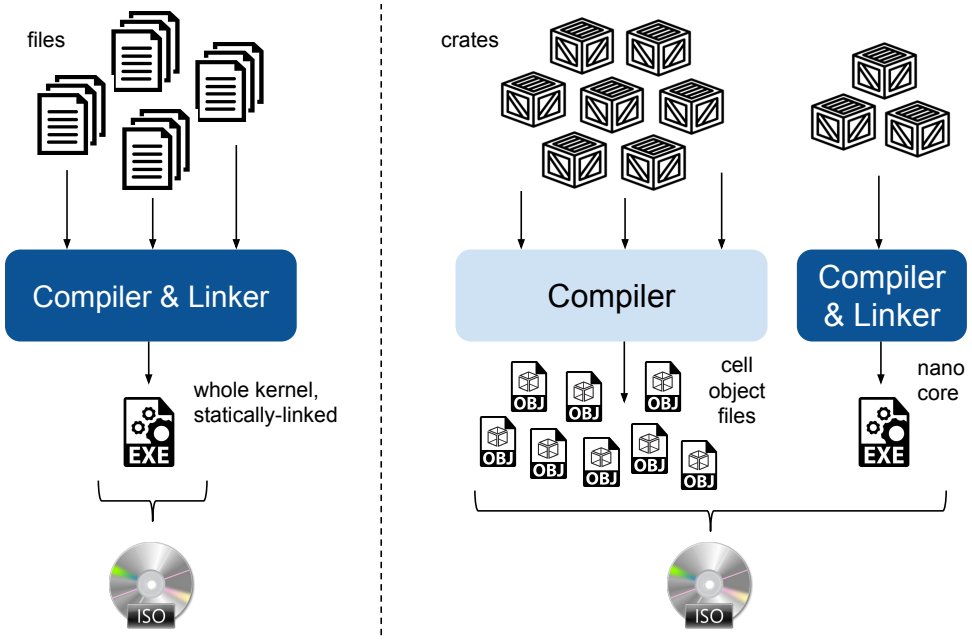


Figure 5.3 : (Left) Traditional monolithic kernel and microkernels build the entire kernel into a single statically-linked binary, losing modularity details at runtime. (Right) Theseus builds only the `nano_core` into a statically-linked binary, all other crates are compiled into distinct object files and packaged into the OS image separately.

5.3 Bootstrapping Theseus with the `nano_core`

Theseus splits the compilation process at the linker stage, placing raw cell object files directly into the OS image such that linkage is deferred to runtime, as depicted in Figure 5.3. From a practical standpoint, unlinked object files cannot run, so we must jump-start Theseus with the `nano_core`. The `nano_core` is a set of normal cells statically linked together into a tiny, executable "base kernel" image. As shown in Figure 5.4, the `nano_core` is as close to a theoretical minimum as possible, with a smaller scope than existing microkernels and KeyKOS's nanokernel [42], which adds capabilities, disk journaling, and checkpointing. It comprises only components needed to bootstrap a bare-minimum environment in which all other cells can be loaded dynamically. Thus, the `nano_core` better satisfies Liedtke's *minimality principle* [43] because it lacks hardware abstractions or stubs and makes neither

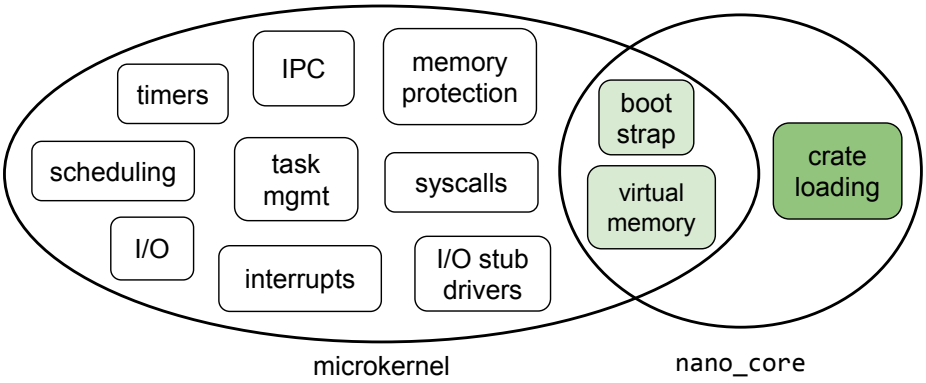


Figure 5.4 : Theseus's "base kernel," the `nano_core`, is truly minimal compared to existing small kernels, consisting of only the functionality necessary to support crate loading.

policy nor mechanism decisions for the rest of Theseus.

Because statically linking cells into the `nano_core` loses their bounds and dependencies, Theseus fully replaces the `nano_core` by dynamically loading its constituent cells one by one, meeting the requirement of runtime-persistent bounds for *all* cells. Theseus recovers metadata for all symbols and sections in the `nano_core` by reading augmented symbol tables and linker script constants encoded into its OS image, and uses the `nano_core` to break circular dependencies when loading fundamental language crates like the Rust core library and compiler built-ins. After bootstrap, the `nano_core` is safely unloaded, as all future cells will be linked against and thus depend on its newly-loaded constituent cells.

Chapter 6

Power to the Language

The second design principle Theseus follows is to maximally leverage the power of the language by enabling the compiler to check safety and correctness invariants to the fullest extent possible. We term this approach *intralingual*, within the language, as it involves matching Theseus's execution environment to that of the language's runtime model, and implementing OS semantics within the strong, static type system offered by modern safe systems programming languages like Rust. By doing so, we extend compiler-checked invariants to *all* types of resources, not just those built into the language. For example, Rust prevents dangling references to objects in stack and heap memory, while intralingual design extends this to allow Rust's compiler to prevent dangling references to all resources.

Intralingual design offers three primary benefits. First, it empowers the compiler to take over resource management duties, reducing the states the OS must maintain which in turn reduces state spill and strengthens isolation. Second, it shifts semantic errors from runtime failures into compile-time errors, accelerating the development process. Third, it enables the compiler to apply safety checks with no gaps in the compiler's understanding of code behavior, approaching end-to-end safety from applications to core kernel entities.

In contrast, traditional OSes take an *extralingual* approach, relying on hardware-provided features (e.g., MMU-protected address spaces, privilege levels) and runtime checks (not integrated with the compiler) to uphold invariants for safety, isolation, and correctness. These features are transparent to the compiler and require unsafe code. Even existing safe-language OSes [1, 44, 45, 46] have a gap between language-level safe code and the

underlying unsafe core that implements the language's required abstractions as a black box. The following sections describe how Theseus closes this gap and opens up such black boxes.

6.1 Matching the Language's Runtime Model

The compiler for many languages, including Rust, expects that its output will become (part of) an executable program that runs within one address space and privilege level, e.g., a single userspace process. Thus, the compiler cannot holistically observe or check the behavior of independently-compiled entities that co-execute in different address spaces or privilege levels.

To address this shortcoming, we tailor Theseus's OS execution environment to match Rust's runtime model: (*i*) only a single address space (SAS) exists and thus a single set of addresses is visible, for which Theseus guarantees a one-to-one virtual to physical mapping; (*ii*) all code executes within a single privilege level (SPL), thus there is no other world or mode of execution; (*iii*) only a single allocator instance exists, matching the compiler's expectation that a global heap serves all allocation requests. Note that Theseus does support multiple arbitrary heaps by implementing them intralingually within a single global allocator instance (§9.3).

6.2 Intralingual OS Design

Matching the language's runtime model only allows the compiler to *view* all Theseus entities. For the compiler to *understand* those entities and apply its safety checks to them, we must implement them in a manner that exposes their safety requirements, invariants, and semantics to the compiler. As an aside, Theseus uses safe code to the fullest extent possible at all layers of the system, prioritizing safety over all else (e.g., convenience, performance). It

only descends into unsafety when fundamentally unavoidable: instructions that are *directly* above hardware, and functions within Rust's foundational libraries (`core` and `alloc`).

Theseus goes beyond language-level safety to further empower the compiler, effectively "tricking" it into checking our custom OS invariants as if they were built in. First, for each OS resource, Theseus identifies the set of invariants that prevent unsafe behavior and incorrect usage. As the Rust compiler already checks myriad invariants for the usage of language-provided types and mechanisms, Theseus maximally employs these language-level mechanisms to allow its resource-specific invariants to be *subsumed* into those compiler invariants. For example, Theseus uses Rust's `Arc` type to share resources across multiple tasks with automatic reference counting, e.g., a memory region (§6.3.1) or channel endpoint (§6.3.3). This eliminates the potential for errors due to use-after-free mismanagement of these resources by subsuming them into the common invariant that an object shall not be freed until all of its users (owners) have finished.

Second, Theseus employs *lossless* interfaces for both external functions that export a resource's semantics and internal functions that implement those semantics. An interface is lossy if crossing it results in discarding language-level context, e.g., the type information, lifetime, or ownership/borrowed status of an object. Furthermore, the *provenance* of that language-level context must be statically determinable, such that the compiler can authenticate that there was no broken link in the chain of calls and interface crossings when using a given resource. In other words, language-level knowledge must not be lost and then reconstituted extralingually. For example, invoking a system call loses type information, object ownership, and lifetimes of its arguments because they must be reduced to raw integer values to cross the user-kernel boundary.

Ensuring Resource Cleanup via Unwinding

One significant invariant that we enforce beyond default Rust safety, and towards correctness, is to prevent resource leakage. Resource leakage refers to not freeing an acquired resource even after there remain no outstanding references to that acquired resource. It is important to note that leaking resources like memory does *not* violate safety, so Rust need not guarantee it by default, but it is generally incorrect behavior.

Theseus realizes the prevention of resource leakage in two parts. First, Theseus employs *stack unwinding*, which together with compiler-emitted unwinding information, ensures acquired resources are always released not only in normal execution but also in exceptional execution. Second, Theseus implements all cleanup semantics in drop handlers, a lossless, language-level approach that allows the compiler to solely determine *when* it is safe to trigger a given resource cleanup action without defining *what* that cleanup action is. When acquiring resources in Theseus, each task directly owns objects representing those resources on its stack (§7.1). The Rust compiler tracks ownership of those objects to statically determine when a resource is dropped (i.e., no longer in use), and thus, where to insert its cleanup routine. Implementing all resource cleanup in only drop handlers frees developers from the burden of correctly ordering release operations and considering myriad corner cases, e.g., exceptional control flow jumps that skip cleanup routines. As this also applies to acquired locks, Theseus can statically prevent many cases of deadlock by releasing lock guards during unwinding, and through domain-specific locks that automatically disable/re-enable preemption or interrupts (see §6.3.2).

We implement Theseus's *unwinder* from scratch in Rust, with custom unwinding logic independent of existing compiler- or debugger-based unwind libraries; thus, it works in core OS contexts without a standard library or allocation. At runtime, Theseus starts the unwinder after a task crashes or upon request to kill a task. This prevents failed or uncooperative

tasks from jeopardizing resource release and reclamation and as a result, strengthens fault isolation. The unwinder uses compiler-emitted information along with cell metadata, such as in-memory locations of `.text` sections, to locate previous frames in the call stack based on their instruction pointer, calculate and restore the register values that were present during that frame, and discover and invoke all necessary cleanup routines or exception-catching blocks. The unwinder further leverages cell metadata to traverse through nonstandard stack frames for hardware-entered asynchronous calling contexts, e.g., interrupts or CPU exception handlers.

On a related note, Theseus must allow resources to be revoked in scenarios where an entity should no longer have access to a resource, e.g., due to malicious abuse, buggy behavior, or a request to re-balance resource distribution. To bypass the potential for unsoundness, Theseus *forbids* the OS from revoking an individual resource out from underneath the language. This does not preclude the OS from forcibly reclaiming resources; instead, Theseus reclaims resources at task granularity by killing and unwinding a task. Though this appears drastic, executing a task that assumes its resource accesses are safe will result in undefined behavior if those assumptions suddenly do not hold, breaking isolation. Killing a task represents the worst case; reclaimable resources (e.g, in-memory caches, thread pools) must express the possibility of resource absence within their type definition, e.g., by using `Option` or weak references. By unifying system-level and language-level resource acquisition and release actions, resources will be freed exactly once, not by both the language and the underlying OS (double free) or by the OS while in use by the unaware language (use after free).

The following section demonstrates how language-level implementation and expression of OS semantics leads to compiler-enforceable invariants in key Theseus subsystems.

6.3 Examples of Intralingual Subsystems

We next describe how Theseus intralingually implements foundational OS resources, namely memory management, task management, and inter-task communication.

6.3.1 Memory Management

Theseus intralingually implements virtual memory through the `MappedPages` type, which represents a region of virtually-contiguous pages statically guaranteed to be mapped to real physical frames. `MappedPages` is the fundamental, sole way to map and access memory in Theseus, and serves as the backing representation for stacks, heaps, and arbitrary memory regions like MMIO and loaded cells.

As briefly depicted in Listing 6.1, the design of `MappedPages` empowers the compiler's type system to enforce key invariants that uphold memory safety, described below. These invariants extend Rust's memory safety checks to *all* OS memory regions, not just the compiler-known stack and heap. Supported memory safety invariants include preventing multiple invalid aliases (aliasing XOR mutability), out-of-bounds access, use after free and double free, and forbidden mutable access.

M.1: Bijective Virtual to Physical Mapping

The mapping between virtual pages to physical frames must be one-to-one, or bijective. This prevents aliasing (sharing) from occurring beneath the language, forcing all shared memory access in Theseus to use *only* language-level mechanisms, e.g., primitive references (`&MappedPages`). In Theseus's SAS environment (§6.1), this is both possible and non-restrictive. In contrast, both conventional and existing safe-language OS designs allow different virtual pages to map the same physical frame; this extralingual approach renders sharing transparent to the compiler and thus uncheckable for safety.

```
pub fn map(pages: AllocatedPages, frames: AllocatedFrames,
       flags: EntryFlags, ...) -> Result<MappedPages, Error> {
   for (page, frame) in pages.iter().zip(frames.iter()) {
      let mut pg_tbl_entry = pg_tbl.walk_to(page, flags)?
         .get_pte_mut(page.pte_offset());
      pg_tbl_entry.set(frame.start_address(), flags)?;
   }
   Ok(MappedPages { pages, frames, flags })
}
pub struct MappedPages {
   pages:  AllocatedPages,
   frames: AllocatedFrames,
   flags:  EntryFlags,
}
impl Drop for MappedPages {
   fn drop(&mut self) {
      // unmap here: clear page table entry, invalidate TLB.
      // AllocatedPages/Frames are auto-dropped and deallocated.
   }
}
impl MappedPages {
   pub fn as_type<'m, T>(&'m self, offset: usize) -> Result<&'m T, Error> {
      if offset + size_of::<T>() > self.size_in_bytes() {
         return Error::OutOfBounds;
      }
      let t: &'m T = unsafe {
         &*((self.pages.start_address() + offset) as *const T)
      };
      Ok(t)
   }
   pub fn as_slice<'m, T>(&'m self, offset: usize, count: usize)
         -> Result<&'m [T], Error> {
      if offset + (count * size_of::<T>()) > self.size_in_bytes() {
         return Error::OutOfBounds;
      }
      let s: &'m [T] = unsafe {
         slice::from_raw_parts(self.pages.start_address() + offset, count)
      };
      Ok(s)
   }
}
```

Listing 6.1: The `MappedPages` type and its interface. The `map()` interface for creating virtual memory regions and the `MappedPages` struct definition with fields and member methods for accessing those memory regions. Sanity checks and other details are omitted for brevity.

```
struct HpetRegisters {
    pub capabilities_and_id: ReadOnly<u64>,
    _padding0:               u64,
    pub general_config:      Volatile<u64>,
    _padding1:               [u64, ...],
    pub main_counter:        Volatile<u64>,
    ...
}
fn hpet_usage_example() -> Result<(), Error> {
    let frames = get_hpet_frames()?;
    let pages = allocate_pages(frames.count())?;
    let mp_pgs = pg_tbl.map(pages, frames, flags)?;
    let hpet: &HpetRegisters = mp_pgs.as_type(0)?;
    let ticks = hpet_regs.main_counter.read();
    // `mp_pgs` auto-dropped here
}

```

Listing 6.2: Example of interacting with the HPET device using `MappedPages` to safely access its MMIO registers. The `hpet_usage_example()` is representative of a function invoked from the main function of an application or system task, thus the `frames`, `pages`, `mp_pgs`, and borrowed `hpet` reference are states that exist solely on that task's stack.

We realize the M.1 invariant via the `map()` interface (**L1**), which accepts only exclusively-owned `AllocatedPages` and `AllocatedFrames` to create a new mapping. In this way, we leverage Rust's ownership model and type safety to force `map()` to *consume* (take ownership of) the allocated pages and frames before it can return a new `MappedPages` object. That `MappedPages` then owns those allocated pages (**L11**) and frames (**L12**), representing the fact that the memory region was created from those pages and frames. `AllocatedPages`, `AllocatedFrames`, and `MappedPages` are opaque, affine types that cannot be cloned or forged in safe code.

Thus, the `map()` function is a lossless interface, as it statically ensures the provenance of this relationship between `AllocatedPages`, `AllocatedFrames`, and `MappedPages`. The `map()` interface also prevents extralingually creating a new mapping from

raw virtual/physical addresses or from borrowed references to allocated pages or frames, which would bypass their affine type guarantees. This ensures they cannot be used more than once to create duplicate mappings that violate the bijective mapping invariant. In addition, `map()` ensures the provenance of the progression from allocating pages and frames to obtaining a `MappedPages` instance, there is no way to break that chain of type safety. This progression is reminiscent of session types [47] used to statically verify communication protocols, such as in Singularity [1].

M.2: In-bounds Memory Access

Memory must not be accessible beyond the mapped region's bounds. The sole way to access a memory region is through `MappedPages` methods like `as_type()` (**L23**) that overlay a statically-sized struct type `T` atop it. The in-bounds invariant (**L25**) is checked dynamically unless the size and the offset are known statically, which does occur in some MMIO device cases. A similar function `as_slice()` (**L31**) supports overlaying dynamically-sized types atop the memory region, returning a borrowed slice of statically-sized elements (`&'m [T]`). These access functions are lossless because they preserve the relationship between the lifetime of the underlying memory region and the lifetime of the re-typed borrow of that memory.

M.3: Unmap Exactly Once when Unused

A memory region must be unmapped exactly once, only after there remain no outstanding references to it. `MappedPages` realizes its release and cleanup semantics only within a drop handler, a custom implementation of Rust's `Drop` trait (**L15-20**) that is automatically invoked when all references to the `MappedPages` object have ended. The compiler together with Theseus's unwinder ensures that a `MappedPages` region, e.g., `mp_pgs`

created on **L12** of Listing 6.2, will be dropped and unmapped in both normal execution (**L15**) and exceptional execution (not shown).

Correspondingly, memory must not be accessible after it has been unmapped. The access methods like `as_type()` only allow *borrowing* a `MappedPages` instance to obtain a reference `&'m T` to an overlaid struct. This guarantees memory safety by tying the overlaid struct's lifetime `'m` to the duration of its backing `MappedPages` instance, allowing the compiler's lifetime checks to statically prevent use-after-free. As such, obtaining ownership of an overlaid struct is impossible by design, as that would be a lossy interface that discards the above lifetime relationship.

M.4: Access Matches Protection

A memory region must not be mutable or executable if not mapped as writable or executable, respectively. We transform this into a static type check via dedicated types, e.g., `MappedPagesMut` and `MappedPagesExec`, which can only be obtained if mapped as such. Only the `MappedPagesMut` type exports lossless interfaces for obtaining mutable references to the memory, `as_type_mut()` and `as_slice_mut()`; similarly, only `MappedPagesExec` exports the `as_function()` interface. Leveraging the type system in this way statically prevents page faults due to protection violations, e.g., write or noexec.

Lines **12-14** in Listing 6.2 show how to use `MappedPages` interfaces to overlay and access MMIO registers for reading the tick value of the a hardware timer, here the High-Performance Event Timer (HPET) found in modern x86 chipsets. The necessary unsafety in **L29** in Listing 6.1 is innocuous as it merely indicates that the compiler cannot guarantee that the overlaid struct type has been constructed with valid contents; this is unavoidably left to the developer, e.g., to ensure the layout of `HpetRegisters` (**L1**) in Listing 6.2 is correct.

Most importantly, no matter how incorrect the developer is, the other `MappedPages` invariants ensure that such innocuous unsafety cannot violate fault or data isolation as it cannot access anything safe code cannot access.

All other memory management is safe, down to the lowest level where `MappedPages` internally manages page tables and TLB entries, but does so only upon language-level actions like invoking `map()`, `remap()`, or being dropped and auto-unmapping itself. Walking page tables necessitates extralingual functionality to accommodate the hardware-defined (MMU) format of page table entries (PTEs). Although Theseus ensures PTEs are validly written, it cannot empower the compiler to statically check that PTE indices and offsets refer to the correct PTE or frame, beyond basic sanity checks.

Altogether, `MappedPages` bridges the semantic gap between the compiler's and OS's knowledge of memory, gaining myriad Rust safety checks for free. Theseus can thus guarantee at compile time that invalid, unexpected page faults cannot occur.

6.3.2 Task Management

Theseus collapses the concept of a language-level thread and a native OS thread into one, a task, which represents an execution context. Theseus implements task management, which handles each stage of the task lifecycle: spawning and entering new tasks, modifying task runstates as they run, and exiting and cleaning up tasks. While `MappedPages` is the center of intralingual memory management, the `Task` struct is minimized in Theseus, both in content and significance as evidenced by Listing 6.3. Rather, Theseus's task management centers around leveraging generic type parameters that remain consistent across all task lifecycle functions.

Theseus enforces the invariants described in the following paragraphs to empower the compiler to uphold safety throughout the task lifecycle.

```
pub struct Task {
    name: String,
    runstate: RunState,
    saved_stack_ptr: RegisterContext,
    task_stack: Stack,
    entry_point_cell: Option<LoadedCell>,
    namespace: CellNamespace,
}
```

Listing 6.3: The contents of Theseus's task struct, consisting of only the items required for safely executing and switching between tasks. This is possible only because the compiler performs bookkeeping of a task's acquired resources, so it need not contain states to track the resource usage of each task for purposes of cleanup. In contrast, Linux's monolithic task struct contains roughly 200-275 items depending on configuration (as of kernel v5.8); granted, Linux is much more featureful than Theseus, but this still illustrates the difference in state management philosophy.

T.1: Spawning a Task must be Safe

Spawning a new task must not violate memory safety. Rust already ensures memory safety among multiple concurrent userspace threads, so long as they were created using its standard library thread type. Theseus cannot use Rust's standard library in its core OS entities; instead, it provides its own task abstraction. This allows Theseus to overcome the standard library's need to extralingually accommodate unsafe, platform-specific thread interfaces, e.g., `fork()`.

Theseus's task abstraction must also preserve safety in a similar fashion as standard Rust threads. To do so, Theseus's `spawn` interface requires the caller to statically know and provide the exact type of the entry function's signature `F`, argument `A`, and return type `R`. The constraints imposed upon these type parameters are an extension of Rust's threads: (*i*) the entry function must be runnable only once (`FnOnce`), (*ii*) the argument and return type must be safe to transfer between threads (`Send` in Rust), and (*iii*) the lifetime of said types (function, argument, return) must outlast the duration of the task itself. As a result, the

compiler will naturally extend its safety guarantees to concurrent execution across multiple Theseus tasks. All functions within the task lifecycle are fully type-safe and parameterized with the same generic type parameters, `<F,A,R>`, ensuring losslessness and the provenance of type information. This enables the compiler to statically prevent spawning a new task with an invalid argument passed to its entry function.

T.2: Safe, Deadlock-free Access to Task State

Accessing task states must always be safe and atomic without inducing deadlock. As with shared access to built-in Rust types, access to shared task states must be safe; however, accessing task states is particularly prone to deadlock because it may prevent other tasks from running. First, Theseus identifies tasks through direct references to them via the `TaskRef` type, rather than raw pointers or task IDs of which the compiler cannot ensure correct usage. Allocating a unique task ID is therefore unnecessary. Second, `TaskRef` is effectively a wrapper around an `Arc<MutexPreempt<Task>>`, which allows the compiler to ensure that references to a task cannot outlive the task itself. The `MutexPreempt` type is a custom preemption-safe `Mutex` derivative that forcibly disables preemption while the lock is held. Thus, when accessing a task (e.g., modifying its runstate) in a concurrent OS environment, Theseus empowers the compiler to statically prevent both data races and deadlock due to preemptive task switching while a lock is held. A similar approach statically prevents deadlock by wrapping data accessed within other interrupt handlers with an interrupt-safe mutex, e.g., task scheduling states or device state accessed in both regular execution and interrupt contexts.

T.3: Task States must be Released

All task states must be fully released in all possible execution paths. This is a task-specific application of the more generic invariant that resources must not be leaked, introduced in §6.2. However, releasing task states requires special consideration beyond simply dropping a `Task` object during unwinding, for three reasons. First, task states themselves, e.g., the stack, are used during and must persist throughout the procedure of that task being unwound, and only cleaned up once unwinding is complete. Second, there are multiple stages in the task cleanup procedure, e.g., a task can be exited, killed, or reaped, which each reflect varying levels of resource release. For example, we can release a task's stack and saved register context when it is killed or exited, but cannot do so for a task's runstate or exit value until it has been reaped. Third, there are multiple potential paths in the final parts of the task lifecycle that each require different actions to be taken, as described below.

When a task runs to completion, its entry function naturally returns execution to the `task_wrapper`, which can then safely mark the task as exited and return its exit value to the "parent" task that spawned it. When a task crashes, the exception (panic or machine fault) handler starts the unwinding procedure, iteratively releasing all task-held resources until the stack is fully unwound. The end of the unwinding routine then invokes a task failure function that marks the task as crashed. Both normal and exceptional execution paths invoke a final task cleanup function that removes the task from runqueues and deschedules it. All of these functions are parameterized with `<F,A,R>` types, allowing them to statically know and losslessly preserve the types in the entry function signature of the crashed task; this is a key part of intralingual fault recovery mechanisms like restartable tasks (§8.2).

T.4: Reachable Memory must Outlive a Task

All memory transitively reachable from a task's entry function must outlive that task. This invariant is a form of preventing use-after-free errors, but requires special consideration beyond the invariants provided by `MappedPages`. Specifically, Theseus must ensure that any data or text sections that a task may access or execute will persist in memory until the task is completely exited, reaped, and dropped. This includes both static data sections (`.data`, `.bss`) as well as the stack and heap.

It is difficult to use Rust lifetimes to sufficiently express the relationship between tasks and arbitrary memory regions upon which they rely, even though all memory regions in Theseus are represented by `MappedPages`. This is because a Rust program running as a task cannot specify in its code that its variables (objects in memory) are tied to the lifetime of an underlying `MappedPages` instances. Those underlying `MappedPages` instances are hidden beneath the abstraction of the stack, heap, executable text sections, and static data sections. Even if possible, this would be highly unergonomic and inconvenient because literally all program variables would need to be gated by a lifetime, rendering ownership useless. For example, all local stack variables would need to be defined as borrowed references with lifetimes derived from the lifetime of the `MappedPages` representing the stack memory itself.

Thus, to uphold this invariant, we establish a chain of ownership: each task owns the cell that contains its entry function, and that cell owns any cells it depends on, given by the per-section dependencies in the cell metadata (Chapter 5). As such, the `MappedPages` containing all functions and data reachable from a task's entry function are guaranteed to outlive that task itself. This removes the need to litter lifetime constraints across the program in every single defined variable, and allows Rust code to be written normally with the standard assumption that the stack, heap, data, and text sections will always exist.

Only low-level context switch routines are not intralingual, as one cannot represent switching stack pointer registers in inline assembly to the language. On a related note, Theseus does not offer `fork` because it is an extralingual operation known to be unsafe and unsuitable for SAS systems [48], as it duplicates task context, states, and underlying memory regions without any reflection of that duplication at the language level.

In contrast, extralingual approaches in conventional systems leave the enforcement of these invariants to the programmer. For the first invariant, the systems programmer and (depending on language) application developer are responsible for ensuring that spawning a new thread does not pass invalid arguments across thread boundaries, and that multiple threads do not cause concurrency violations like race conditions when accessing shared data. The remaining invariants are similar. The second invariant requires programmers to ensure that (*i*) the right locks are manually acquired and released in the proper order before and after a task state is accessed. The third invariant necessitates that programmers be cognizant of all execution paths, and manually determine at what stage of cleanup a task is in in order to release its states correctly. Finally, the fourth invariant requires the backing memory used by a task to be manually freed once that task (and all other tasks) have fully exited; that usage information must be tracked explicitly with out-of-band data at runtime.

6.3.3 Inter-Task Communication Channels

In a SAS/SPL OS like Theseus, inter-task communication (ITC) can occur via a simple shared memory location, such as a reference to a mutex-protected heap object. Thus, ITC is less important in Theseus than in microkernel or monolithic OSes because system components interact with each other using direct function calls rather than inter-process communication (IPC) mechanisms. Nevertheless, in order to facilitate ITC with stronger semantic guarantees than simple shared memory, Theseus intralingually implements ITC

channels of two varieties: a synchronous rendezvous-based bufferless channel and an asynchronous buffered channel. Theseus channels expose sender and receiver interfaces inspired by Rust's standard library channels, but ours differ in key ways and enforce additional invariants, as described below.

C.1: Statically-typed, Safe Channels

A channel is statically typed and fully safe. A channel can only transmit messages of one type, which must be thread-safe, statically sized, and specified at channel creation time. Dynamically-sized types can be sent using a layer of indirection, such as a heap-allocated `Box`. Singularity [1] imposes fewer restrictive conditions on message transmission than Theseus's channels, by using a dedicated contract checker in the Sing# compiler to enforce transitions between session types that represent a channel's state. We enforce Theseus's channel conditions strictly using existing compiler checks.

Both the `Channel` and its `Sender` and `Receiver` endpoint types are parameterized with the message type bound `M: Send`, as with a task's entry function argument and return type. This allows the compiler to statically ensure that only legally-typed channels can be constructed and that message data is never in an invalid state of ownership, even temporarily. For example, this prevents primitive references (`&M`) from being sent across channels.

Unlike Rust's built-in channel types, we implement Theseus's channels with wholly type- and memory-safe code; all internal functions never use unsafe code and are fully lossless, preserving type, ownership, and lifetime information of message objects and channel state. All channel functions statically know the type and fixed size of messages, and provenance is ensurable by the compiler because no channel characteristics are determined dynamically. Furthermore, we express the semantics of transferring a message using ownership: the sender must relinquish ownership of a message object so that the receiver can take ownership.

In addition, Theseus's channels enable arbitrary sharing of endpoints, which effortlessly realizes multi-sender multi-receiver channels. In the absence of unsafe code, the compiler can ensure that end-to-end usage of channels, including cloning and sharing of endpoints across multiple tasks, cannot cause data races. This differs from the extralingual mechanisms in Rust's built-in channels that use unsafe code to internally reconcile and handle multiple endpoints and channel flavors in a manner hidden from the language, necessitating unsafely marking the sender or receiver type as `Send`able.

C.2: Endpoints Heed Channel Status

All channel endpoints must observe and heed the channel's connection status, e.g., sending must not succeed if no receivers exist. The `Sender` and `Receiver` types implement separate drop handlers that update an atomic connection status enum in the `Channel` and notify endpoints on the other side of the disconnection. Unlike Rust's built-in channels, Theseus differentiates between sender- and receiver-side disconnection. Treating only one side as disconnected enables critical system tasks to recover from disconnection by reconnecting their endpoints to an existing partially-disconnected channel; this is useful for fault recovery via restartable tasks (§8.2). For example, after a sender task fails, is unwound, and restarted, it can reconnect to the same channel and continue sending messages if a receiver still exists (see §9.2). In this case, the receiver task would not lose any progress state, and an additional protocol could be layered atop this to enable the sender to "catch up" to the receiver by synchronizing their state over the channel when needed; generally, stateless communication principles would handle this automatically and render such synchronization messages unnecessary.

The compiler and Theseus's unwinder ensures that when all endpoints are dropped, the channel is also automatically dropped to release its internal state and buffers. Both channel

varieties offer a blocking `send()` and `receive()` and a non-blocking `try_send()` and `try_receive()`. We realize these interfaces using dedicated, distinct error return types that express the possibility of timeout, disconnection, lack of message or buffer space, or failure to block or wake a task endpoint. This allows the compiler to check that channel users properly handle and respond to every possible error condition, preventing undefined behavior in tasks using an endpoint.

Chapter 7

State Management in Theseus

The third design principle Theseus follows is to minimize and ideally eliminate state spill in its cells. Theseus's cell entity choice stipulates that state spill can only occur in interactions (e.g., function calls) that cross a cell boundary.

7.1 Opaque Exportation through Intralinguality

Theseus employs *opaque exportation* to avoid state spill in client-server interactions, as shown in Figure 7.1. Each client is responsible for owning the state that represents its progress with the server, hence *exportation*, but cannot arbitrarily introspect into or modify that server-private state due to type safety, hence *opaque*. Opaque exportation is only possible because Theseus's safe, intralingual design enables shifting the burden of resource/progress bookkeeping from the OS into the compiler. This allows bookkeeping states to be *distributed*, or offloaded to each client, e.g., held only on a task's stack. Theseus's unwinder finds and invokes cleanup routines without needing OS knowledge (states) about which resources a client has acquired, thus the server entity and OS at large need not maintain bookkeeping states for each client. Thanks to intralingual revocation powered by Theseus's unwinder, opaquely-exported states owned by a client can still be revoked and reclaimed on behalf of the server that granted them.

Conversely, Theseus eschews traditional state encapsulation common in monolithic kernels and system service architectures, in which a server entity holds all states representing

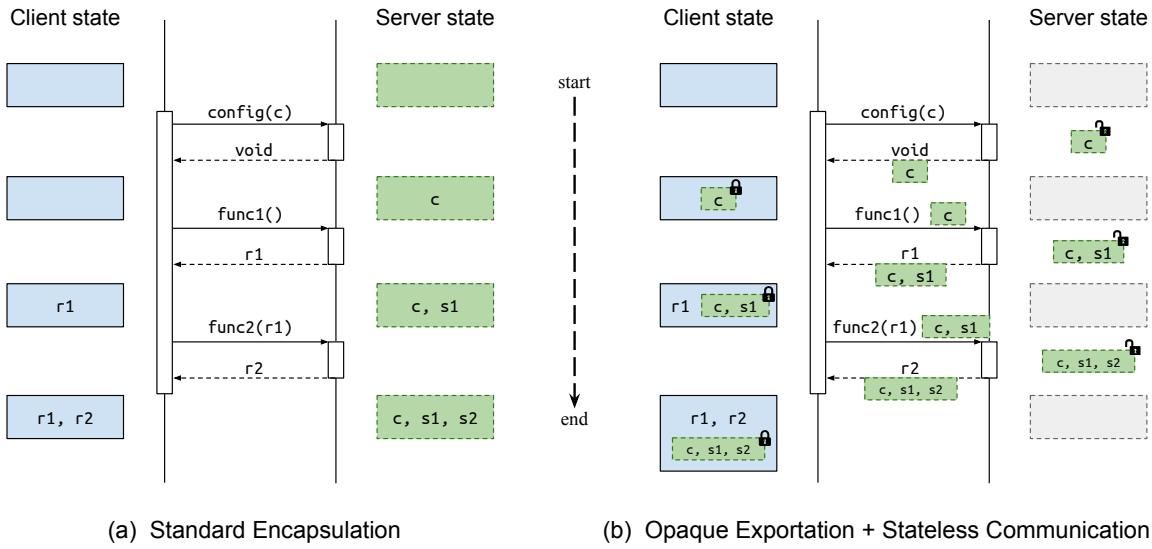


Figure 7.1 : (a) Traditional encapsulation-based modularization causes client-server entanglement due to state spill in the server after an interaction; state spill takes the form of `c`, `s1`, and `s2` in (a). (b) In Theseus, server entities opaquely export progress states to their client(s), which avoids state spill in the server, enabling stateless communication and disentangling the client and server.

its clients' progress and resource usage, depicted in Figure 7.1(a). Such encapsulation constitutes state spill and causes fate sharing that breaks isolation: when a server crashes and loses its state, its clients will also fail. Opaque exportation still preserves *information hiding* [49], a primary benefit of encapsulation.

A corollary of opaque exportation is *stateless communication* (à la RESTful web architectures [50]), which dictates that everything necessary for a given request to be handled should be included in that request. Server entities that employ stateless communication need not store intermediary states between successive client interactions, as future interactions will be self-sufficient, containing previously-exported states, as depicted in Figure 7.1(b).

7.1.1 Avoiding Common Spillful Abstractions

Opaque exportation enables Theseus to avoid common spillful abstractions such as *handles* and *signals*. Client-side handles to server-owned data forces the server to maintain a global table mapping each client's handle to its underlying resource object, a form of state spill. Theseus rejects handles in favor of a client (or multiple clients jointly) owning the underlying resource object directly, relieving the server from the burden of maintaining a handle table. Unix-like signals lack sufficient information in the signal message itself to fully handle the signal, forcing the signal receiver to maintain and rely on existing (previously spilled) states to handle it with proper context. Instead, Theseus employs stateless communication media to convey events with full context, e.g., through function arguments or channel messages.

7.1.2 Accommodating Multi-Client States

Because a server-defined resource may pertain to more than one client, Theseus extends opaque exportation to enable all pertinent clients to jointly own that resource state, i.e., *multi-client states*. As joint ownership is realized via heap-allocated objects with automatic reference counting (e.g., `Arc`), the existence and sharing of multi-client states can technically be viewed as state spill into the heap.

However, considering state spill into the heap itself is not useful, for the following two reasons. First, heap allocations are represented by owned objects elsewhere that point back to the heap, e.g., types like `Box` or `Arc`. Therefore, it suffices to consider only the propagation of those objects when determining where state spill occurred, rather than observing the internal state of the heap itself. Second, state spill into the heap is unavoidable; if we considered the heap and the stack as regular software entities, every basic action from a function call to creating a new local string variable would constitute state spill, rendering it a useless metric. Note that we view other allocators similarly, e.g., page and frame

allocators, whose allocations are tracked via propagation of representative owned states, `AllocatedPages` and `AllocatedFrames`.

7.2 Management of Special States in Theseus

Theseus cells often hold **soft states**, those that can be lost or discarded without causing permanent failure [51, 52]. Soft states exist for the sake of convenience or performance, e.g., in-memory caches of values read from hardware like a clock source's period. Although soft states technically constitute state spill, they can be idempotently re-obtained or recalculated when lost with no impact on correctness. Therefore, Theseus permits soft states as harmless state spill with no adverse effects on evolution or availability. Theseus even provides a generic type abstraction for declaring soft states with enforced idempotent regeneration.

We identify **unavoidable states** in two general forms: (*i*) *clientless* states, those that hardware requires the OS to maintain on its behalf, and (*ii*) states needed to handle asynchronous, hardware-invoked entry points that do not provide sufficient context. The former renders opaque exportation impossible and the latter violates stateless communication. In the first case, we cannot modify the behavior or capacity of underlying hardware to accommodate exported states. Thus, Theseus must hold these states to ensure they persist throughout all execution. Examples include low-level x86 structures like the Global Descriptor Table (GDT), Task State Segment (TSS), Interrupt Descriptor Table (IDT). In the second case, Theseus must store necessary contextual states with a static lifetime and scope that exceeds that of the asynchronous hardware event's entry function (e.g., an interrupt handler).

To preserve the interchangeability of server entities in both such cases, Theseus assigns their states a well-defined owner and static lifetime by moving them into the `state_db`, a state storage facility with minimal semantics akin to key value databases. Any singleton cell can move its static state into the `state_db` and then hold a weak reference to it, reducing

it to harmless soft state and imposing negligible performance impact (upgrading to a strong reference). The `state_db` retains interchangeability despite containing states spilled from other cells, as it uniquely must cooperate in its own swapping process by hardening itself via serialization to nonvolatile storage. The only other entity in Theseus that requires self-participatory swapping is the cell manager, which similarly serializes its cell metadata. This design decouples a hardware state's lifetime from that of the server cell interacting with it, enabling said cell to be evolved without losing unavoidable system-wide states.

7.2.1 Examples of Intralinguality Leading to State Spill Freedom

We further illustrate the relationship between intralingual design and state spill freedom with two example subsystems: memory and task management.

Memory Management: First, Theseus's `MappedPages` type (§6.3) eliminates state spill through opaque exportation: it is owned by the client requesting the mapping rather than the server (`mm` cell) that created it. In contrast, `mm` entities in existing OSes harbor state spill in the form of metadata representing each memory mapping, e.g., virtual memory area (VMA) objects; clients must blindly assume that the underlying mapping and VMA persist throughout the usage of its virtual address handle. Importantly, we consider page tables as hardware-required MMU states, much like x86's Global Descriptor Table, Task State Segment, etc. Page table entries are not language-level objects with lasting variable name bindings in Theseus; thus, writing to a page table is a side effect externalized to hardware and does not constitute state spill. Crucially, the state representing this side effect — the transition from "unmapped" to "mapped" — is not lost, but reflected in the client-side `MappedPages` object rather than a hidden server-side `mm` state change.

Task Management: Second, Theseus's intralingual design and its ensuing opaque exportation *significantly* reduce the scope and size of its `Task` struct, thus avoiding most

instances of state spill from other subsystems into its task management cells. Theseus also moves task-related states specific to other OS entities, e.g., runqueue and scheduler information, out of the task struct and into those entities themselves. This better follows separation of concerns than conventional OSes that hoard a huge list of OS states needed for manual resource bookkeeping and task cleanup into a centralized, all-encompassing task struct. Such a task struct design causes myriad OS operations to spill state into the task management entities and results in cross-cutting dependencies that closely entangles entities together, hindering their evolution or recovery. Thus, as shown in Listing 6.3, Theseus's task struct can contain only the bare necessities, e.g., the task's runstate and saved execution context (register state). Correspondingly, it excludes lists of open files, open sockets, memory mappings, wait queues, etc.

Microkernel designs are more distributed like Theseus, but require frequent IPC exchange to synchronize task list instances replicated in each userspace server process, a form of state spill. To associate subsystem-specific states with a relevant task (e.g., the scheduler context for a task), subsystems in Theseus use direct references to a task instead of independent partial task lists indexed by task ID. These references, realized via wrapper types like `Arc` or `Weak` in Rust, require no explicit synchronization because they refer to a single shared task object in memory. Weak references to a task that no longer exists are harmless and can simply be lazily removed when convenient. We use such techniques to wholly remove runqueue and scheduler states from the task struct, which decouples the implementation of scheduling and runqueue management from task management, with minor performance overhead (§9.3).

Chapter 8

Realizing Evolvability, Availability, and Flexibility

To demonstrate the utility of Theseus's structure, intralingual design, and novel state management strategy, we implement mechanisms inside it to support three challenging computing goals: live evolution, availability, and flexibility. We strive to realize all three goals in an easy and arbitrary manner beyond that of related work.

8.1 Live Evolution via Cell Swapping

The fundamental evolutionary mechanism in Theseus is *cell swapping*, a multi-stage procedure that replaces O existing "old" cells with N "new" ones; O and N need not be equal. (*i*) First, Theseus loads all new cells into a new empty CellNamespace (Chapter 4), an isolated linking environment. Doing so ensures that new cells are linked only against each other at first,preventing them from accidentally linking against and depending on old cells that export the same symbols. (*ii*) Theseus then verifies dependencies bidirectionally: new cells must satisfy all existing dependencies on the old cells, and the dependencies of all new cells must be satisfied by existing cells. Isolated loading allows this to occur before making invasive changes to the running system. (*iii*) Theseus redirects all cells that depend on the old cells to depend on their new counterpart. This involves rewriting their relocation entries and the corresponding dependency metadata, updating on-stack references to the old cells, and transferring states if necessary. (*iv*) Finally, Theseus atomically moves the new cells into the existing CellNamespace whilst removing the old cells and their symbols. This

guarantees that cells loaded in the future cannot depend on old cells.

Evolving a running instance of Theseus can be as easy as committing to a repository. We realize live evolution, i.e., live OS continuous delivery, in the following manner. A developer pushes a new commit to the Theseus repository, which triggers our build server to re-compile Theseus. Our custom build tool generates an evolution manifest file, an augmented diff that specifies which newly-built cells need to replace which old running ones, and then notifies the evolution client running in Theseus that new cells are available to download and apply. Alternatively, users or maintainers can select individual cells to evolve, and all others that must be changed alongside them will be brought in automatically to ensure that the evolution manifest is well-formed and sensible.

Theseus facilitates cell swapping and simplifies known live update techniques like quiescence and state transfer. First, *runtime persistence* of cell bounds greatly simplifies cell swapping. In stage (*i*), Theseus's dynamic loader guarantees that a given cell's runtime bounds in memory cannot overlap those of another, preventing sections in different cells from being interleaved throughout the address space or crossing a region (page) boundary, which commonly occurs in statically linked kernel binaries. Thus, each cell can claim sole ownership of its memory regions and be cleanly removable in stage (*iv*). Moreover, dynamic loading produces precise dependency information with no false positives/negatives, which is preserved throughout runtime for use in stages (*ii*) and (*iii*). New cells can replace old cells of any size at any address even without position-independent code, removing the barrier to techniques like ASLR and facilitating reloading corrupted cells in memory for fault recovery. Finally, dynamic loading supports evolving the very modularity of Theseus itself, i.e., replacing M old cells with N new ones in order to accommodate major refactoring (merging or splitting) of cells.

Spill-free design of cells in Theseus simplifies state transfer. As previously mentioned,

opaque exportation allows a server cell to be more easily swapped because it need not maintain state between successive interactions with clients, increasing its quiescent periods. Stateless communication reduces the dependencies a given function has on other cells because it receives necessary states and function callbacks or closures via its arguments. Overall, these points lead to faster dependency rewriting and state transfer steps in stage (*iii*).

The *cell metadata* accelerates cell swapping. In stage (*ii*), dependency verification reduces down to a quick search for a fully-qualified symbol in the CellNamespace's symbol map. In stage (*iii*), Theseus need not scan every task's stack, rather only a limited subset for which the old cells' public functions or data are reachable from the task's entry function; reachability is trivially determined by following dependency links in the metadata. If types (e.g., function signatures) have not changed, Theseus can apply stack changes immediately; otherwise it must wait for quiescence until the changed types are absent from task stacks.

To further accelerate state transfer in stage (*iii*), each cell's metadata contains a set of only the data sections in its crate object file, an optimization that avoids needing to slowly iterate over all sections in the cell, of which there may be hundreds. Each data section can then be transferred into the new cell's corresponding section via a language-level `Clone` operation, which is typically a shallow copy of a reference to that data. This is a rare stop-gap need, as state spill-free designs naturally avoid static data. On a related note, ownership semantics allow Theseus's cell manager to blithely remove old cells and their symbols in stage (*iv*) without checking for their usage elsewhere. This avoids a computationally-complex graph traversal through all metadata across the system. Moreover, doing so is safe because the compiler has already ensured that the removed cells will not be unloaded (dropped) until they are no longer in use, i.e., depended upon by other cells' sections.

Theseus's intralingual design extends to *transfer functions* needed for evolving a data

structure in stage (*iii*). We allow and require such functions to be implemented safely and intralingually using Rust's built-in traits for type conversions (e.g., `Into`). Although transfer functions are not provably valid in the general case [53], we can support practical cases by auto-generating implementations of conversion traits using Rust's procedural macro system, i.e., trait derivation. Generating the Rust syntax for these functions occurs within the compiler, shifting would-be runtime errors for a failed conversion into compiler errors for failed generation. Generation of transfer functions is ongoing work, thus the results reported in this thesis use manually-implemented transfer functions.

There are two limitations to this approach. First, it cannot support extralingual usage of changed data structures, e.g., accessing data structure fields in blocks of assembly code; we rely on developers to change that assembly code together with the Rust code that uses and describes the data structure. The current prototype of Theseus has only two such instances: context switching, which solves this problem via static assertions about the layout of a saved task's Context structure, and VESA graphics hardware detection, which must be implemented in x86's 16-bit real mode but only occurs once upon system boot. Second, we restrict automated evolution to occur between adjacent versions of the code (commits in the repository) in order to avoid the explosion in code size that would stem from generating myriad versions of data structure transfer functions. Note that this is more of a practicality choice than a feasibility matter; it is still possible to use the build system and the above mechanisms to formulate an evolutionary progression from version N to versions beyond $N + 1$, but they are not generated by default.

8.2 Availability via Fault Recovery

We next describe how Theseus recovers entities and execution contexts from faults, including language-level exceptions (Rust `panic`s) and machine faults (CPU exceptions) due to

hardware failures or memory corruption. The following work on fault recovery represents a collaboration with Namitha Liyanage; additional details can be found in [54].

8.2.1 Fault Isolation

Building on safety and intralinguality, Theseus strengthens fault isolation via unwinding: task that encounters a fault and crashes will release its acquired resources, enabling other tasks to progress even if they depend on those shared resources. The alternative extralingual approach of simply freeing a failed task's stack or heap memory does not undo resource actions to ensure isolation or safety; this leads to minor failures, e.g., leaking memory by not decrementing a reference count, and critical failures, e.g., not releasing a lock on a core system state. Even if a shared resource enters an invalid state post-fault, e.g., a poisoned mutex, progress can still be made because the resource state can be examined, as opposed to all involved tasks (and likely the whole system) being locked up permanently.

In addition, fault isolation in Theseus is finer-grained and more widely applicable than isolation in monolithic or microkernel OSes. It supports the ability to isolate tasks from one another, equivalent to isolating threads within a process. Theseus can also automatically unwind and clean up *application-provided* resources, not just OS-provided resources, in order to isolate dependent applications from each other's faults. Even a monolithic kernel that maintains a full set of states to track application usage of kernel resources cannot properly release application-provided resources because it would have no knowledge of them.

8.2.2 Fault Recovery

Theseus's design and its operating environment both present unique challenges to fault recovery. First, state spill freedom eliminates the existence of in-kernel states that track

resources acquired by a given task (as found in conventional OS designs), meaning that the kernel code itself does not know which resources need to be released nor how to release them after a fault occurs. Second, to offer availability in the absence of hardware redundancy, we restrict fault recovery to rely upon only software techniques; the single address space and single privilege level OS environment renders hardware protection facilities useless anyway. Even in redundant systems, software mechanisms can recover from a fault in a less destructive manner by preserving more contextual state than a hard reset, a complementary approach to redundancy.

The majority of possible faults are prevented statically by virtue of using a safe language to ensure type and memory safety, such as unhandleable page faults, invalid opcodes, or general protection faults, all of which Theseus prevents at compile time for normal execution. We focus primarily on recovering from transient faults that do not reoccur during normal execution, e.g., random memory corruption and other temporary hardware failures; this also covers soundness bugs leading to faults that originate from software but manifest like said hardware problems. Handling permanent hardware failures in systems software is beyond the scope of this work.

Theseus follows a multi-stage, cascading approach towards fault recovery, taking increasingly drastic measures until normal execution is recovered. To track progression through recovery stages, Theseus maintains a system-wide fault log that records fault context (e.g., instruction pointer, current task) and the recovery action taken. This enables us to detect recurring faults in order to proceed to the next stage.

The first recovery stage is to simply tolerate the fault by taking the default action of fully cleaning up a failed task via unwinding. This allows other tasks that share resources with it but are otherwise unaffected by its failure to continue running. The alternative extralingual approach of simply freeing a failed task's stack or heap memory does not undo resource

actions, leading to failures like not decrementing a reference count or releasing a critical lock on a core system state.

The second recovery stage is to respawn a new instance of the failed task. Theseus extends its existing task spawning infrastructure to provide a fully intralingual implementation of *restartable tasks*. The `spawn_restartable()` interface further constrains the `<F,A,R>` type parameters (§6.3) to enable the compiler to enforce that all restartable tasks have (*i*) an entry function `F` and argument `A` that can be safely duplicated (`Clone`), and (*ii*) an entry function that can be safely executed multiple times (`F: Fn`, not `FnOnce`).

The third stage is to reinitialize a cell's static data regions, which is only taken if the fault occurs when accessing static data. cells that expose access to static data typically do so in ways that can tolerate data returning to an initial state, due to Rust's RAII semantics and type safety requirements. For example, the `Once` and `lazy_static` wrapper types inherently represent whether a given data item has been initialized, requiring its accessors to accommodate its uninitialized state. Although returning said data to its initial state will likely result in a subsequent error or panic, those are language-level problems much easier to recover from. Note that we do not evaluate this stage in this thesis.

The most significant recovery stage reuses the cell swapping mechanism (§8.1) to replace corrupted cells with freshly-loaded instances at a different memory location. This approach addresses faults that occur on invalid accesses of cell data or text sections, indicating they have been corrupted (e.g., due to a hardware memory failure). This operation represents the simplest possible case of cell swapping, with no possibility of missing dependencies or changes to code or data types. This is followed by the second stage of restarting the failed task, which allows that task to successfully execute atop the new cell instance(s).

If all prior recovery attempts fail, Theseus has no alternative but to perform a full reboot, which does not preserve any volatile states. Currently, we only trigger recovery actions

within panic/exception handlers; further techniques like using hardware watchdog timers to detect a lack of progress are complementary to our approach. One limitation of our approach is that we do not attempt to detect errors beyond those that materialize into faults and resulting in halting of code progression. Fault handling logic in our approach is only triggered by panics and hardware exceptions.

Notably, Theseus's fault recovery mechanisms operate with few dependencies, allowing Theseus to tolerate faults in the lowest layers of the system in the face of multiple failed subsystems. Only a basic execution environment needs to work, such that Theseus can access the stack and invoke functions for unwinding; more complex recovery stages need task spawning and cell swapping abilities. All of this can safely execute within the context of a CPU exception handler in Theseus. In comparison, fault recovery approaches in reliable microkernels like MINIX 3 [55] require support for context switches, interrupts, IPC, and userspace to work properly.

8.3 Flexibility via Multiple CellNamespace-based Personalities

Theseus's structure and execution environment makes it easy to realize OS personalities in an arbitrary manner, in which each personality contains a subset of features that differ from the rest of the OS. As described in Chapter 5 and Figure 5.2, Theseus maintains metadata to track the in-memory bounds and bidirectional dependencies of dynamically-loaded cells and their sections. In actuality, Theseus permits *multiple* sets of metadata, i.e., multiple CellNamespaces, to simultaneously exist. As depicted in Figure 8.1, each CellNamespace can represent a distinct OS personality that offers custom differences in functionality, i.e., system flexibility. This works by loading and linking a new set of differing cell (s) into just one CellNamespace, similar to the evolutionary procedure in §8.1 but without the destructive changes to "old" cell (s) in other existing CellNamespaces.

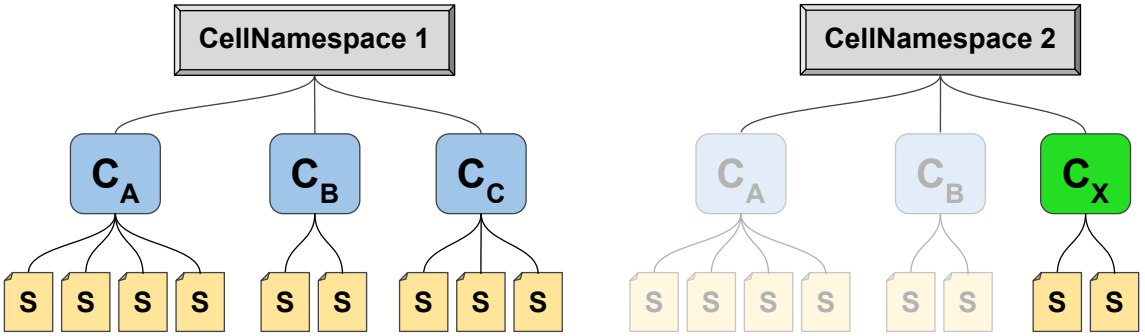


Figure 8.1 : Theseus realizes system flexibility by allowing multiple CellNamespaces to exist side-by-side as siblings and sharing any number of arbitrary cells. This figure depicts a second CellNamespace (right) that shares all crates with an existing primary CellNamespace (left), differing only in a single cellC_X that replaced C_C. Thus, when a task is spawned with an entry function in CellNamespace 1, it will execute through sections in C_A, C_B, and C_C. When a similar task is spawned in CellNamespace 2, it will execute through the same sections in C_A and C_B, but will transparently use C_X instead of C_C.

Theseus's CellNamespace-based personalities are significantly more arbitrary and efficient than library OS personalities, which are realized along hardware-based bounds, i.e., within separate process address spaces. First, personalities can exist along practically any dimension, e.g., per task, per task group, per core, per kernel subsystem, etc. Second, personalities are realized efficiently in software without the overhead of switching between hardware privilege modes or address spaces, a well-known advantage of SAS/SPL safe-language OSes [56]. Cells are shared across CellNamespaces by virtue of joint ownership through a custom clone-on-write shared reference wrapper type, achieving a high rate of fine-grained code deduplication. Cells containing states that must exist as system-wide singletons cannot be duplicated across personalities; it is currently left to the developer to properly implement `Clone` on such states to permit only shallow copies. Also, there is no overhead due to layers of indirection, such as function redirection or mediator objects commonly used in live update works (Chapter 10). Instead, cells are directly linked to other cells both within the same CellNamespace and those shared from others.

Note that tasks in all CellNamespaces in Theseus still execute in a SAS/SPL environment. A CellNamespace is simply a collection of crates linked together and bears no resemblance to process virtualization containers like Linux's `cgroups`.

The power of arbitrary personalities in Theseus is exemplified by an ongoing project that uses Theseus as the OS for a 5G basestation. This 5G basestation implements the computationally-heavy physical layer, including baseband processing, in pure software. Baseband processing crates are compiled with SIMD support and run in a SIMD-enabled personality, executing directly alongside all other OS cells that run in a normal (non-SIMD) personality. This accommodates code that benefits from or requires SIMD acceleration while allowing regular OS code to avoid the overhead of saving/restoring SIMD registers on every interrupt and context switch. To the best of our knowledge, Theseus is the first OS to support multiple personalities with heterogeneous compiler targets running on the same processor in the same privilege mode and address space.

In the future, we will develop a *git*-style interface for easy management of CellNamespace personalities and easy exploration of historical evolution of crates, as there are strong parallels between CellNamespaces and version control systems. This abstraction lends familiarity to the notion of flexibility and evolution in Theseus: a git branch corresponds to a CellNamespace personality, a tracked file to a cell, a commit to a change in one or more cells, checking out a different commit corresponds to swapping cells, and forking a repository corresponds to creating a new CellNamespace from an existing one.

Chapter 9

Evaluation

We evaluate Theseus to show that it achieves easy and arbitrary evolution and increases system availability through fault recovery. We also develop a series of microbenchmarks to assess the impact of intralingual, state spill-free designs on memory and task management performance, and to compare Theseus's base performance with that of Linux. This thesis does not evaluate system flexibility, an ongoing work for which we have yet to find a killer application in which Theseus's arbitrary CellNamespace-based personalities can shine. All experiments were conducted on an Intel NUC 6i7KYK [57] with 4 (8 SMT) 2.6 GHz cores and 32 GB memory, unless otherwise stated.

9.1 Live Evolution

Theseus's evolutionary mechanisms are implemented in-band, that is, within the OS core and using its own features, which differs from existing works that implement live update functionality out-of-band or on a mature OS. As such, it is difficult to conduct a statistical analysis showing which historical commits can be supported by Theseus's live evolution. For example, it is impossible to test live evolution between commits that do not support fundamental features like virtual memory or cell management, which is quite common throughout the change history of a from-scratch system like Theseus.

Thus, we instead use case studies (as in [58]) to demonstrate that Theseus is able to evolve core system components in unique manners beyond prior live update works. We select

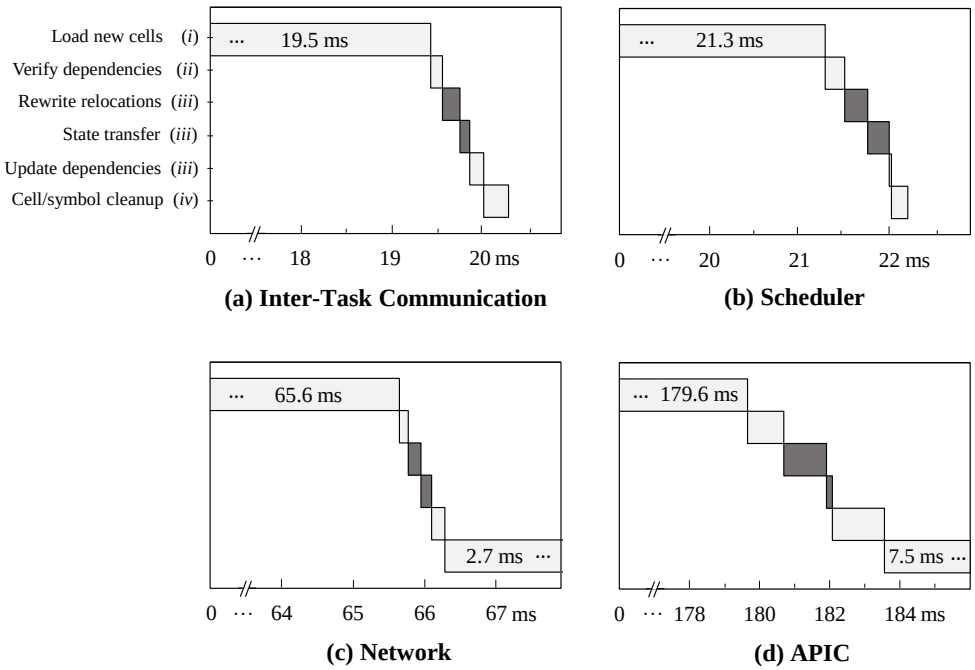


Figure 9.1 : The time taken for each step in Theseus's live evolution procedure, with cell swapping stages marked (*i-iv*) (§8.1). The first two steps are performed in a new, isolated CellNamespace and do not affect the running system. Only the middle two steps are critical (shaded) and may impact execution by requiring atomicity, i.e., a system pause, but this can be avoided when the evolved components robustly handle state unavailability errors, as in the scheduler **(b)** case. The last two steps must lock the CellNamespace being changed to prevent overlapping evolution, but do not affect execution.

four case studies that exhibit easy, arbitrary live evolution of core OS entities: Figure 9.1(a), evolving "IPC" channels from synchronous to asynchronous, to reflect a change that would require evolution of a microkernel core, Figure 9.1(b), evolving the scheduling and runqueue subsystems with no pre-existing infrastructure and no execution downtime, Figure 9.1(c), evolving the Ethernet NIC driver and the evolution client itself over that very same network interface without packet loss, and Figure 9.1(d), evolving system support for the Advanced Programmable Interrupt Controller (APIC) to demonstrate changing the modularity itself of a core OS entity at the center of the dependency spiderweb to significantly augment its functionality. Although Figure 9.1 shows the time scale of evolutionary stages in each case,

these are more suited to qualitative discussion, as follows.

9.1.1 Inter-Task Communication Channels (à la IPC)

In this experiment, we analyze how Theseus can evolve its ITC channel layer, Theseus's equivalent of IPC because it eschews the process abstraction. This case study demonstrates that Theseus advances the state of the art by *live* evolving (*i*) a fundamental OS primitive that must be implemented within a microkernel, (*ii*) a kernel API that necessitates joint evolution of dependent userspace and kernel entities, and (*iii*) a widely-used component, whilst preserving the execution context of entities that depend on it.

Specifically, the experiment evolves Theseus's existing synchronous channel that uses an unbuffered rendezvous protocol into a new asynchronous buffered channel. The rationale behind this choice stems from our in-depth survey of microkernel change histories, which refutes the argument that microkernels need not evolve because they are sufficiently small and simple, such that all required evolutionary changes can occur in their userspace servers. Both literature about and the history of evolutionary changes to MINIX 3, seL4, and QNX Neutrino microkernels reveal significant, necessary reworks to their microkernel cores. One of the most notable changes they underwent was transitioning the IPC layer from a synchronous rendezvous to an asynchronous implementation [59, 60]; all require a standard reboot to apply the change.

During this experiment, we spawn multiple applications that exchange messages with each other over multiple synchronous channels, in addition to system tasks (e.g., input event manager) that already use said channel, and then issue a live evolution command at a random point while messages are in flight. Because Theseus can evolve cells independently from execution contexts, it can swap the channel implementation out from underneath a running application without having to kill it. As shown in Figure 9.1(a), this improves

availability by reducing median downtime to 385 μs because it preserves the application's runtime progress, avoiding the domino effect of needing to restart multiple other dependent tasks transitively. This is especially significant for long running tasks, compared with the conventional approach of restarting the task from scratch.

Theseus can leverage unwinding to roll back a task's call stack to the execution point right before calling the channel cell's functions (`sync_chan::send/recv`), effectively popping off all `sync_chan`'s content from the stack in order to engender a quiescent point. This technique is also useful for evolving a task that may be deadlocked within a mutex loop due to a bug, as unwinding the stack will release said locks and swap out the buggy cell before reattempting that code path. After the `async_chan` replaces the `sync_chan` cell, the application tasks proceed as normal with the new execution path that will invoke the corresponding `async_chan::send/recv` functions. This requires special consideration to ensure correctness, in that it only works for idempotent functions or those with side effects that can be cleanly undone (e.g., acquiring a lock, waiting to send/receive). As §9.2 evaluates unwinding-based recovery from faults, we do not present it here.

Given these design considerations, Theseus can automate the evolution from a synchronous to asynchronous IPC channel without losing state, because a synchronous channel must rendezvous to exchange data. However, when evolving from asynchronous to synchronous, the data already in the channel's buffer will be lost because the synchronous rendezvous channel has no queue into which that data can be placed. Custom logic could re-send each queued message, but that is difficult to automate in the general case.

In real usage of channels in Theseus, Theseus actually need not swap any kernel cells to evolve a channel implementation in use by applications, as it uses generically-typed channels that will be monomorphized into the cells of the applications that use the channel.

This enables us to evolve one group of applications using one channel cell *completely independently* of other groups using that same channel cell. Thus, in order to realize a kernel interface and structure analogous to the singleton IPC provider entity found in microkernel and monolithic OSes, we create a singleton concrete instance of a channel that exposes a non-generic channel API for specific types like strings and byte arrays. This forces Theseus to swap this singleton channel cell along with the applications that use it, just as what would be necessary to realize such evolution within a microkernel or monolithic OS's IPC primitive.

9.1.2 Scheduling and Runqueue Subsystems

In this experiment, we replace the existing round-robin scheduler with a new priority scheduler, and the existing dequeue-based runqueue with a binary heap-based priority queue that holds task priorities and timeslice tokens. All the while, Theseus runs multiple tasks of varying priorities that print messages in a loop, illustrating the visible difference in task execution order and frequency before and after evolution.

This showcases Theseus's ability to evolve at runtime: (*i*) the OS's modularity itself (entity bounds), (*ii*) a core entity used incessantly and depended upon by many other entities, and (*iii*) a low-level entity that could jeopardize system stability.

At first glance, this appears trivial because existing OSes can already switch between multiple schedulers at runtime. The key distinction is that Theseus booted as an OS that did not originally contain *a priori* knowledge of or in-band support for multiple schedulers, whereas existing OSes do so by requiring a scheduler infrastructure with a pre-defined common interface that accommodates multiple scheduler policies. This illustrates a significant benefit: subsystems in Theseus need not incorporate a special design or interface to support multiple versions of a given component; in this case, task management functions

like `task_switch()` need not be aware of multiple schedulers. Instead, Theseus components can rely upon an arbitrary, out-of-band generic cell swapping mechanism to evolve or flexibly switch between multiple alternate implementations. This results in a simpler design, e.g., the task management subsystem need not be aware of system schedulers need not be aware of any other schedulers, have no required interface to follow, and are not involved in the evolutionary process, and the task switch function has no knowledge of multiple schedulers. In addition to temporally evolving a system-wide scheduler, Theseus can also easily realize arbitrary flexibility to simultaneously use different schedulers alongside one another (e.g., one per core) by loading them into different CellNamespaces, again without any accommodation from the scheduler or runqueue cells themselves.

9.1.3 Network Update Client and Ethernet Driver

In this experiment, we evolve Theseus to fix unreliable network downloads, comprising two cells that must be evolved simultaneously: (*i*) the core Ethernet driver underneath the network stack, and (*ii*) Theseus's evolution client application that sits atop the network stack to communicate with the build update server. This demonstrates Theseus's capacity for coordinated, multi-part evolution (as does the above scheduler case) versus small-scope live updates that only patch one driver function. The new Ethernet driver fixes an insidious bug that caused inconsistent connections due to incorrectly setting head and tail registers for the ring buffer of received packets, while the new evolution client fixes its HTTP layer usage to properly recover from unexpected remote socket disconnections. We achieve this without losing any NIC configuration settings or packet data progress, tested by downloading files during the evolution and verifying them with checksums.

This case shows that Theseus can provide improved *availability without redundancy*, e.g., for solitary embedded systems in the field, and even higher availability despite hardware

redundancy, e.g., for datacenter network switches that must be brought down during driver updates. Moreover, it shows that Theseus can perform "meta-evolution," i.e., loading a new NIC driver over that very NIC itself and a new evolution client using the client's own evolutionary features. This procedure is facilitated by state spill freedom that results in network states being owned by the application task, except for minimal states necessary to coordinate asynchronous receive buffer handling. In our experience, this makes it easier to cleanly replace a running driver in Theseus than in conventional designs that scatter states across many entities in the network stack.

9.1.4 APIC Subsystem for Multicore and Interrupts

We begin with a running instance of Theseus that has a single APIC cell with a rudimentary feature set: all interrupts are blindly routed to the first-booted CPU, LAPIC timer interrupt periods are hardcoded, interrupts can only be configured minimally (e.g., edge-triggered only), etc. We then evolve it to expand its functionality and also split that one cell into several elementary cells: a TLB shootdown cell, an IOAPIC cell for device interrupt redirection, an IPI cell, and a standard APIC cell for handling local APIC timers more robustly. The evolved cells support arbitrary interrupt routing, custom interrupt periods and configuration, and a fuller interface that exposes all of the available LAPIC interrupts and IOAPIC flags. Theseus's full metadata and loose structural requirements enable it to not only accommodate a change in modularity (one cell to many cells), but also to safely pre-verify that the replacement cells will not violate any dependency needs of other cells that depended on the initial single APIC cell.

Unlike the priority scheduler case study, replacing the APIC cell does require a system pause when rewriting relocations and transferring states because it is a critical entity subject to transient state inconsistencies, and is depended upon by many (25+) other cells for

sensitive operations like preemption and timer interrupts. In other words, it is challenging to isolate and recover from transient, mid-evolution software errors that occur in the APIC cells because of its low-level nature. However, the system pause duration is under 1.3ms (Figure 9.1(d) shaded bars), which could be decreased by incrementally changing the APIC cell's modularity rather than all at once. In this case study, the workloads run are immaterial, as practically every system function will end up relying on and utilizing the critical functionality of the APIC cells; they are used for everything from timers to device interrupts to preemption and context switching. All other conclusions are similar to the priority scheduler case study, but the APIC component is even lower-level and more widely used than the scheduler.

9.2 Fault Recovery

We demonstrate Theseus's novel ability to tolerate faults within low-level core components, e.g., those that necessarily exist inside a microkernel. We focus on stress-testing whether Theseus can recover from unexpected hardware-induced faults beneath the language. By design, Theseus can recover from all language-level faults because the compiler understands and can account for them, guaranteeing that unwinding will work. The following fault recovery experiments represent a collaboration with Namitha Liyanage; additional details and results can be found in [54].

Our **fault injection method** is to run Theseus atop the QEMU emulator [61] to enable us to automate arbitrary changes to hardware state, including randomly flipping one bit or overwriting full quadwords in memory, and randomly incrementing the instruction pointer register to skip instructions. This follows approaches in the literature [62, 63, 55] that show myriad hardware faults manifest to the OS in this manner. We inject faults while running a workload including graphical rendering, task spawning, FS access, and ITC channel usage.

We monitor the workload/OS behavior to determine whether the fault manifests into an error, again following practices in the literature [55, 63]. As found in other fault injection works [55], very few randomly injected faults (<0.5%) manifest into observable failures; thus, we augment our fault injector to target specific regions in memory where faults are likely to manifest, namely a given task's working set of stack, heap, and cell memory (text, data, and rodata sections).

9.2.1 Theseus Recovers from Microkernel-level Faults

Literature on fault-tolerant microkernels, e.g., MINIX 3 [64] and CuriOS [63], only evaluates recovery from faults injected into *userspace* system servers, not the microkernel itself. To show that Theseus supports recovery from faults in such low-level components, we manually inject faults into both MINIX 3's IPC layer and Theseus's ITC channels and evaluate their ability to recover. To ensure a fair comparison, we manually inspect all layers of MINIX 3's IPC implementation and Theseus's ITC channel implementation to discover 13 faults that cause reproducible, deterministic failures in both systems. The specifics of each fault induced in MINIX 3 and Theseus and their responses to said faults are given in [54].

Out of the 13, Theseus recovers correctly in all but two cases. MINIX 3 fails to recover correctly in all 13: it crashes in 11 cases and loses a message in the other two. For example, corrupting the pointer to a passed message within the IPC receive routine manifests as an unhandleable page fault (an invalid memory access) in both Theseus and MINIX 3; this resulted in MINIX 3's kernel crashing and rebooting, whereas it resulted in Theseus unwinding and properly restarting the ITC receiver task, allowing the sender to progress. In the two cases that Theseus fails to recover correctly, the receiver and sender tasks are hung, respectively, but Theseus does not crash. This problem can be solved through timeouts or resetting the channel, which can be achieved without losing a message (but losing endpoint

progress).

9.2.2 General Fault Recovery Measurements

To comprehensively assess Theseus's ability to recover from faults, we injected 800,000 faults into subsystems actively in use by the above workloads, of which 0.083% manifested as observable failures. Table 9.1 shows that Theseus successfully recovered from 69% of total manifested faults. Restarting the failed task was sufficient in 11% of cases, indicating corrupted stack or heap values; in the remaining 89%, Theseus needed to reload one or more cells, indicating corruption of cell text or data sections.

Theseus failed to recover from 31% of manifested faults, primarily due to a lack of asynchronous unwinding in Rust/LLVM (see Chapter 11). Briefly, this forces Theseus's unwinder to search for the closest entry in the unwinding table that corresponds to the instruction pointer where the fault manifested. Because that unwinding information is not an exact match for the instruction pointer, the cleanup routine(s) it invokes may not completely release all resources acquired at the point of failure. In 30 such cases, the fault caused the system or workload task to hang, preventing our software-only recovery mechanisms from being initiated; these can be addressed by complementary hardware mechanisms like a watchdog timer that triggers fault recovery logic. In another 18 cases, Theseus failed to reload a new cell instance or failed to replace the corrupted cell with the new one. For the 62 faults that manifested in the code path of the unwinding routine itself, the unwinder will fail repeatedly as expected. Collectively, these represent limitations of Theseus's fault tolerance.

Downtime Measurements: Table 9.2 shows the downtime imposed by Theseus's fault recovery mechanisms for lightweight (ITC) and expensive (rendering) tasks. The downtime of an ITC task is the interval from when a channel endpoint is disconnected until it comes back up, reconnects, and transfers another message. The downtime of a rendering task is the

Successful Recovery	**461**
Restart task	50
Reload cell	411
Failed Recovery	**204**
Incomplete unwinding	94
Hung task	30
Failed cell replacement	18
Unwinder failure	62
Total manifested faults	**665**

Table 9.1 : Theseus recovers from 69% of manifested faults in our fault injection trials, which emulate hardware failures by corrupting memory arbitrarily.

Workload	Restart Task	Reload Cell
ITC (rendezvous)	0.475 ± 0.053 ms	8.214 ± 0.353 ms
ITC (async)	0.587 ± 0.040 ms	6.072 ± 0.432 ms
Display rendering	3.815 ± 0.003 s	3.825 ± 0.007 s

Table 9.2 : The downtime imposed by fault recovery in Theseus.

interval from when a rendering operation fails until it resumes operation and fully displays the same content. This downtime is high due to the overhead of Theseus's software-based display renderer, which takes 0.88 s to clean up the old windows and 2.9 s to re-initialize the entire display stack and re-draw all window content; recovery operations impose less than 45 ms of overhead.

These measurements are complementary to the downtime values in Figure 9.1, which represent the upper bound of downtime observable by users of the scheduler or networking subsystems. The only overlapping test case is that of replacing the rendezvous ITC channel; the difference here stems from only needing to replace the corrupted rendezvous channel cell (8.2 ms) rather than both the evolved channel cell and application cells that use it (20.6 ms).

The closest existing fault recovery approach to Theseus is that of CuriOS [63], which

does not provide downtime figures, but suspends *all* system execution during recovery; we thus suspect its downtime is significantly higher than that of Theseus. FGFT [65] reports downtimes ranging from 40 μs to 410 ms when performing driver-specific fault recovery, while MINIX 3 reports an average downtime of 500 ms for its userspace network driver. Theseus is not necessarily faster than these works, but can recover from faults within the core kernel and without subsystem-specific logic.

9.3 Cost of Intralinguality & State Spill Freedom

Although performance is not a primary goal of Theseus, its intralingual and spill-free designs naturally raise performance questions. Due to Theseus's experimental nature, specifically its lack of POSIX support, we cannot make an apples-to-apples comparison with Linux or other OSes. Instead, we compare versions of Theseus with controlled differences to tease out their performance impact; we compare against Linux by porting LMBench microbenchmarks to Rust and running them atop both Linux and Theseus.

Therefore, results reported here should be regarded as *informative* rather than conclusive. We focus on memory and task management because they are core subsystems that affect most other OS entities, and because they comprise the most radical redesigns of all subsystems within Theseus. We find the performance impact varies on a per-subsystem basis: memory mapping performance improves, but task and heap management suffer moderate overhead in microbenchmark stress tests. Overall, we do not observe glaring performance penalties from Theseus's intralingual and spill-free designs.

9.3.1 `MappedPages`: Better Performance and Scalability

We compare Theseus's intralingual, spill-free `MappedPages` implementation against a conventional spillful implementation that encapsulates a red-black tree of VMAs, à la

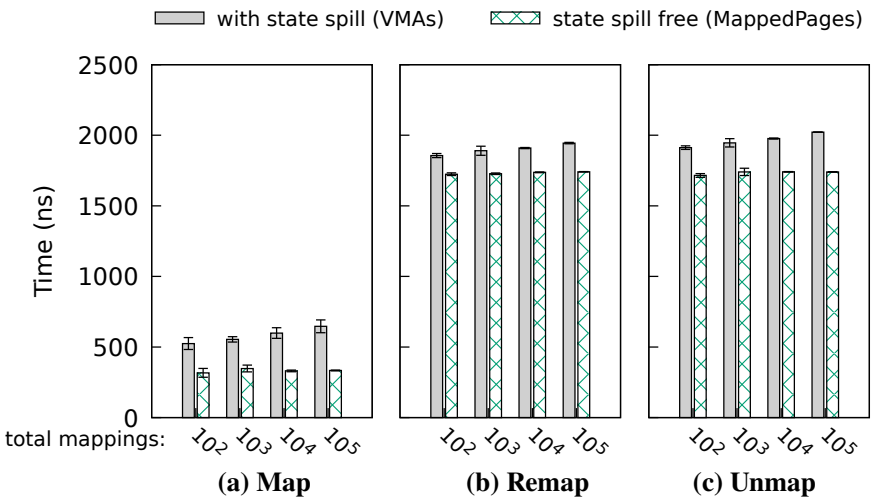
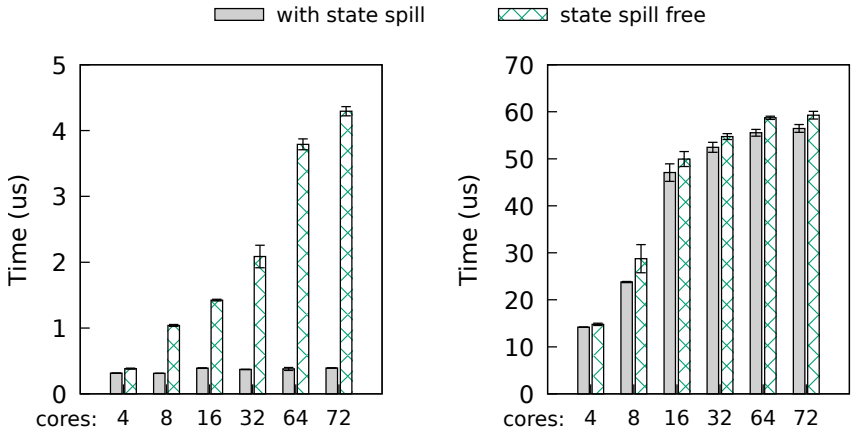


Figure 9.2 : The time to map, remap, and unmap one 4 KiB page is constant for Theseus's state spill-free `MappedPages` approach, which is slightly more performant and scalable to many mappings than a traditional spillful approach based on a red-black VMA tree.

traditional OSes like Linux. The conventional implementation is in Rust, carefully modeled after Linux, and optimized over multiple iterations. Figure 9.2 shows that the `MappedPages` design performs slightly better despite its invariant checks to uphold memory safety. This is primarily due to tasks having direct ownership over `MappedPages` objects, a form of distributed bookkeeping that obviates the need to search through the VMA tree to find the memory region for a given virtual address. Overall, this minor performance difference is unlikely to significantly impact real system workloads.

9.3.2 Removing Task Struct States has Negligible Overhead

As described in §7.2.1, Theseus removes runqueue- and scheduler-related states from the task struct to avoid state spill from the runqueue and scheduler entities, subverting the conventional design of an all-inclusive task struct. This causes overhead because an exited task must iterate through and remove itself from *all* runqueues rather than just the



(a) Remove task from runqueue **(b) Spawn empty task**

Figure 9.3 : (a) The time to remove a task from the runqueue(s) it is on increases when eliminating runqueue states from the task struct, but is minor in the worst realistic case of (b) spawning an empty task.

runqueue(s) it is known to be on. We evaluate the worst-case scenario in which a task is only present on *one* runqueue; the more runqueues a task is on, the less relative overhead Theseus's approach has. These experiments are run on a 36-core (72 SMT) Supermicro 119u-7 server with one runqueue per hardware thread, to accurately reflect caching effects when searching through runqueues on other cores.

The first experiment of Figure 9.3(a) repeatedly adds and removes a non-running task to/from a random runqueue; while this is a contrived microbenchmark impossible in any OS workload, it shows that the overhead of Theseus's spill-free design increases with the number of total runqueues. In real (non-benchmark) system operation, this can be improved with heuristics that remove tasks from runqueues based on prior per-core runstate, especially in the presence of schedulers that do not migrate tasks across cores (or actively remove tasks from old runqueues during core migration). The second experiment of Figure 9.3(b) spawns and runs a dummy task that immediately exits, which represents the worst possible

Heap Designs	*threadtest*	*shbench*
unsafe	20.27 ± 0.009 s	3.99 ± 0.001 s
partially-safe	20.52 ± 0.010 s	4.54 ± 0.002 s
safe	24.82 ± 0.006 s	4.89 ± 0.002 s

Table 9.3 : Heap microbenchmark results for different design points (for regular-sized allocations).

realistic overhead of Theseus's runqueue-less task struct design. The impact is negligible because the prerequisite of spawning a task dominates the overhead of removing it from every runqueue. Furthermore, although our spill-free design may slightly complicate the relatively rare operation of exiting a task, it simplifies the much more common operation of task scheduling because runqueue information need not be updated on each context switch.

9.3.3 Intralingual Heap Bookkeeping causes Overhead

Table 9.3 shows that an intralingual, safe implementation of heap allocation can impose up to 22.5% overhead in allocation bookkeeping costs compared to an unsafe version. We evaluate each heap design point using two popular heap benchmarks ported to Theseus, threadtest [66] and shbench [67], which allocate/deallocate 100 million 8-byte objects and 20 million objects from 1 to 1000 bytes, respectively.

All heap designs are variants of Theseus's per-core heap allocator based on the slab allocation algorithm, which exposes one `alloc` and `dealloc` interface for all heap instances, in order to match Rust's language model of heap allocation (§6.1). Theseus's per-core heaps store memory as lists of pages, where each list can serve allocation requests for objects of a specific size. Each page uses its last few bytes to store metadata such as the heap ID and a bitmap to keep track of allocations. Any memory request smaller than 8KiB is satisfied through these page lists, and are referred to as regular allocations. Larger allocation requests are satisfied by directly requesting memory from the OS, received in

the form of `MappedPages` instances; thus, they are effectively a layer of indirection that abstracts access to `MappedPages`. This is analogous to heap behavior in Linux, in which the internal implementation of `malloc`) invokes `mmap()` if the requested allocation size is above a certain threshold. The heap can grow by adding pages to these lists, either by moving existing pages between the heaps or by requesting more memory from the `mm` cell.

In the unsafe design, a heap neither owns its backing `MappedPages` objects nor knows of their lifetimes at all, and maintains raw pointers to allocation block metadata. In the partially-safe design, a heap owns its backing `MappedPages` but embeds raw pointers to them within the allocation metadata, discarding lifetime information. In the safe design, a heap maintains a safe collection type (e.g., red-black tree) that maps a virtual address to its allocation metadata and the `MappedPages` objects that back them. This allows the compiler to ensure that the association between an allocation and its backing `MappedPages` is never lost, which is crucial for Theseus to safely implement multiple, per-core heaps that can exchange memory pages. However, it causes overhead during deallocation because it must lookup the allocation metadata for a given address. Note that our goal is not to present a new heap design, as the scalability and fragmentation improvements of per-core heaps are known [66].

We similarly evaluate large allocations (>8 KiB), in which the heap requests new memory regions from the OS, and observe a maximum of 1.8% overhead in the safe vs. unsafe design. This is because the time to map/unmap a memory region dwarfs that of allocation bookkeeping.

9.3.4 LMBench Microbenchmark Comparisons with Linux

We reimplement select LMBench benchmarks [68] in safe Rust and run them on both Linux and Theseus. We omit benchmarks that are irrelevant to core OS components or have no

LMBench Benchmark	*Linux (Rust)*	*Theseus*	*Theseus (static)*
null syscall	0.28 ± 0.01	0.02 ± 0.00	0.02 ± 0.00
context switch	0.61 ± 0.06	0.35 ± 0.00	0.34 ± 0.00
create process	567.78 ± 40.40	242.11 ± 0.88	244.35 ± 0.06
memory map	2.04 ± 0.15	1.02 ± 0.00	0.99 ± 0.00
IPC	3.65 ± 0.35	1.06 ± 0.00	1.03 ± 0.00

Table 9.4 : Microbenchmark results in microseconds (μs), smaller is better. *Linux (Rust)* is LMBench benchmarks reimplemented in safe Rust on Linux, *Theseus* is those benchmarks on Theseus, and *Theseus (static)* is those benchmarks on a statically-linked build of Theseus. Standard deviations of zero are much smaller than the HPET timer period of 42 ns, and cannot be accurately measured.

equivalent in Theseus (e.g., RNG latency, futexes), and those that test subsystems that are still rudimentary in Theseus (e.g., networking, filesystems).

Table 9.4 shows the results as the mean value across 100,000 iterations of each benchmark. We do not claim that Theseus generally outperforms existing OSes like Linux, but our results do not indicate significant performance drawbacks. The differences shown stem from eliminating the overhead of switching between hardware protection modes and address spaces; these are known benefits of safe-language, SAS/SPL OSes [1]. Whether such performance trends extend to large-scale application workloads is the subject of future work.

We briefly describe each microbenchmark below. First, the *null syscall* benchmark invokes the `getpid()` system call on Linux, for which the equivalent behavior in Theseus is to invoke the `curr_task()` function.

Second, the *context switch* benchmarks on Linux sets up a pair of threads within the same process on the same core, which then continuously yield to one another. This is a fairer comparison with Theseus, which does the same with two tasks, in contrast to the default LMBench behavior which uses pipes to pass a token between two separate processes. The performance difference herein is due to the lack of privilege mode switching in Theseus.

Third, the *create process* benchmark uses `fork` and `exec` on Linux to create a new "Hello, World!" process, which in Theseus is analogous to loading and linking a new `hello` application cell from scratch — without caching or sharing its `.rodata` or `.text` sections — and then spawning a task that enters its `main()` function.

Fourth, the *memory map* benchmark measures the time to map, write to the first byte, and unmap a single 4 KiB page in both systems. To ensure a fair comparison, we adjust the LMBench benchmark on Linux to create an anonymous mapping with the `MAP_POPULATE` `mmap` flag, such that page table entries are added at the time of creation, matching Theseus's default behavior. Without adjustment, Linux's default mapping flags are slightly worse due to a page fault occurring upon first write, which extends the benchmark time to 2.46 μs. These results show that Theseus avoids expected forms of overhead, including system calls (privilege mode switches) and the creation, insertion, and removal of VMAs.

Fifth, the *IPC* benchmark on Linux measures the RTT of a one-byte message sent over a non-blocking pipe between two threads within the same process, running on dedicated cores to eliminate the effects of context switching. The fairest comparison, i.e., the one in which Linux performs most favorably against Theseus, is to use Theseus's asynchronous ITC channels to pass a single byte (`u8` type) back and forth between two tasks on the different cores.

In addition, we compare against microkernel IPC fastpaths by implementing an ITC fastpath within Theseus that bypasses the disconnection semantics of Theseus's channels and their associated overhead. We realize this fastpath in fully safe code via shared references to an atomic type (e.g., `AtomicU64`) that holds a small-sized message. Theseus's fastpath has an RTT of 687 cycles, higher than seL4's *one-way* fastpath latency of 224 cycles [69]. On a related note, Theseus's asynchronous channel has an RTT of 1664 cycles, similar to Singularity's reported 1415-cycle RTT for its channels in a single virtual address space

configuration [56].

Finally, the *overhead of runtime-linked code* due to dynamic cell loading in Theseus is generally negligible, as shown in the rightmost column of Table 9.4. We obtain these figures by running the same set of benchmarks atop a build of Theseus in which all kernel cells are statically linked into a monolithic kernel binary. While the OS boot procedure takes longer, as it must dynamically load and link many cells during initialization, normal post-init execution is unaffected. Although static linkage may speed up code in future applications, it produces a monolithic object that cannot be decomposed into easily interchangeable independent entities with runtime-persistent bounds.

Chapter 10

Related Work

Theseus draws inspiration from and builds upon many prior works, which we discuss along each major dimension of contributions made: state spill as a concept, various similar OS structures, Theseus's live evolution vs. existing live update, and fault tolerance approaches. See Chapter 4 for a previous overview of Theseus's system architecture compared with existing major OS architectures, such as monolithic, microkernel, multikernel, and library OS systems.

10.1 State Spill

Although we are the first to identify and study the problem of state spill to the best of our knowledge, many prior works have recognized its challenges but failed to identify it specifically. Instead, they often go to great lengths to treat or circumvent the symptoms of state spill or to address related concepts, as described below.

Modularity: As state spill occurs between two entities or modules, it is clear that modularity is a *prerequisite*, not a *solution*, for state spill. Therefore, efforts toward improving OS modularity, e.g., Flux OSKit [70], Knit [71], THINK [72], and OpenCom [73], do not by themselves reduce state spill. Wong et. al [74] explore the concept of modularity violations, in which modules that are frequently updated together lack independence and should therefore always be updated together; this is similar to how state spill inhibits maintainability and evolvability (to a minor extent), but relies on the imprecise heuristic of version change

history.

Module Coupling: Software engineering works have studied and classified various forms of coupling and dependency interactions between modules [75, 76, 77, 78, 79]. Although state spill often falls into one of the six types of data and control coupling in [75], coupling is concerned with the transfer of data and control between entities, while state spill measures its lasting effects; thus, coupling is necessary but not sufficient for state spill to occur. For example, a parameter that only affects temporary variables causes coupling but not state spill; thus, it has no lasting effect on the entity and does not impact the aforementioned computing goals.

Information Flow: State spill is related to taint tracking and other information flow control techniques, well-studied topics in systems software [80, 81, 82] and more recently Android [83, 84]. While these works use similar detection techniques as StateSpy, they are primarily concerned with the security/privacy implications of data propagating through a system and leaking from a flawed entity, not whether that data induces an impactful change in the entities to which it propagates. Other works explore the bigger picture of how interactions between multiple entities (Android components) can cause security vulnerabilities [85, 86, 87]. These tools do not identify state spill, but are complementary to StateSpy and could improve its scope by revealing a finer-grained, more detailed chain of interactions between source, intermediary, and destination entities.

Other works have implicitly targeted symptoms of state spill, the closest of which is CuriOS [63] that shows how holding client-relevant states in server processes complicates fault recovery. CuriOS moves said states into each client's address space, temporarily mapping them into a server's address space during an interaction; this offers effective isolation but only for userspace servers in a microkernel OS, and incurs overhead from modifying page tables and TLB shootdowns. Theseus isolates client and server states within

the same protection domain using type and memory safety.

10.2 OS Structure and Overall Design

Language-based Protection in Systems: Numerous prior works leverage safe languages to provide isolation between OS entities, such as C# in Singularity [1], Modula 3 in SPIN [45], Java in JX [88], Go(lang) in Biscuit [89]Inferno [90], and MirageOS [36]. Recent works have utilized Rust to realize safer embedded and Unix-like OSes, e.g., Tock [44] and Redox [46], and have even proposed ways to implement classic kernel abstractions in Rust [91]. This also extends to domains beyond traditional OS design, such as realizing network functions [92], pushing client code into key-value stores [93] or trusted execution environments [94], and safer kernel extensions [95]. Although Theseus also uses Rust and reaps many of the same benefits, its intralingual design approach (§6.2) goes beyond merely using a safe language. By subsuming resource-specific invariants into existing Rust invariants, Theseus empowers the compiler to check safety and correctness invariants to an unprecedented extent. Theseus also removes gaps in the compiler's knowledge of OS semantics, shifting the burden of resource bookkeeping and cleanup into the compiler.

On a related note, Theseus's *Single Address Space* (SAS) and *Single Privilege Level* (SPL) execution environment was inspired by SPIN [45] and Singularity [1] but for a different purpose than performance: matching the OS's runtime model to that of the language (§6.1).

Runtime-Persistent Bounds: Theseus employs dynamic loading of cleanly-structured cells to achieve entities for *runtime-persistent bounds*, across all levels of the system. Dynamic loading is common in OSes to support kernel extensibility [96, 97, 98, 45]. Unlike Theseus, these systems only load new modules alongside (not in place of) existing running modules, and typically use module loading for device drivers or extensions only; the majority

of the system remains monolithic without clear runtime bounds. The authors of [99] embed a microkernel within the monolithic Linux kernel as an indirection layer to decompose Linux into lightweight capability domains; this helps to isolate kernel subsystems but not to evolve or recover them.

Componentized Systems: Component-based systems expand upon modularity to provide a library of reusable components or object frameworks that can be built into a fuller OS, , e.g., the Flux OSKit project [70], THINK [72], and Taligent [100]. These works offer build-time flexibility through configuration but lose module bounds after compile/link time, as with other monolithic kernel images. This hinders such systems from determining the ways in which states propagate through different components at runtime and cause entanglement. The Flux OSKit [70] introduces *separable* components that can be used individually without specific dependency on other components, allowing components to be substituted freely as long as they adhere to the same interface. The Flux team also provides a language for component definition and linking called Knit [71], intended for statically linking OSKit components at build time via smart linker transformations that adapt each interface based on a component's expected execution domain; this is potentially useful for dynamic loading as well. OSKit uses indirection layers to glue components together along with late binding based on dynamic dispatch; as with K42, requiring components or objects to have a common interface does not meet our standard of arbitrariness.

Microservices and Serverless Environments: With runtime-persistent bounds, Theseus's structure coincidentally resembles that of modern distributed applications, e.g., the microservice architecture, in which a large monolithic application is broken up into many small distributed components that run in isolated containers. The microservices architecture [101] has been widely lauded and adopted by Uber, Netflix, Verizon, Amazon, and many enterprise web services [102, 103]. Microservices have been shown to yield improvements

in fault isolation, generic code reuse and maintainability, and scalability [104, 105], and although Theseus was not inspired by microservices, we acknowledge the similarities in motivation and structure. However, the ad-hoc decoupling strategy and mechanisms for isolation and communication used in microservices rely on underlying OS infrastructure and vary widely across different business units, leaving little that can be applied to Theseus. A significant difference between Theseus and microservice applications is that microservices are each a separate execution context, e.g., a different process or even a different machine entirely, whereas each cell in Theseus is simply a separate set of code and data objects through which multiple execution contexts (tasks) can flow. In Theseus, there is no mandatory correlation between a cell entity and an execution context; a cell's code may spawn a new task or simply act as a passive library for other tasks to use; tasks are free to move throughout multiple cells by invoking public functions within them.

Another push towards finer granularity architectures in industry is *serverless environments*, in which cloud providers enable users to run a single function (function-as-a-service) instead of paying for an entire VM or container [106]. As with opaque exportation in Theseus, these functions can be stateless: each invocation is independent of previous ones. In this regard, Theseus's architecture excels over classic OS abstractions like Linux containers [107] due to very lightweight isolation and the ability to inject a lambda's dependencies into the OS by swapping in a new cell.

Microkernel OSes [64, 63, 69] also have persistent bounds: most OS services run in hardware-isolated userspace processes. Genode [108] is a similarly-modularized OS framework that leverages its hierarchical tree structure of components processes for strong access control. These OS structures make it easier to recover from service failures or update an OS service, e.g., by restarting the service process. However, microkernel OSes choose to modularize along coarse-grained entity bounds defined by hardware, i.e., a process, and in

general do not address state spill between services. This limits their ability to evolve and recover from faults in low-level core microkernel entities, which Theseus can support. Also, Theseus minimizes state spill between OS entities and prescribes a much finer granularity that makes hardware-enforced entity bounds uneconomical.

Library OSes [34, 35] also place the application and (pieces of) the OS in a single address space and at the same privilege level as a single appliance, similar to Theseus. As with Theseus, these systems are orthogonally decoupled to separate mechanism from policy, allowing for flexible configuration of the OS's personality. However, they restrict the customization and evolution of core kernel components, and each application personality still must exist in its own address space that is not shared with other applications. Importantly, this necessitates an underlying hypervisor or exokernel layer to isolate and multiplex between application appliances. In contrast, in Theseus, all tasks run in the same address space and at the same privilege level as the kernel itself; there is no higher authority and there is no restriction in personality form, e.g., along process bounds. See Chapter 4 and §8.3 for details and a diagram depicting the distinction.

Hardware-driven decoupling strategies leverage classic distributed systems principles and delineate entity bounds based on the underlying hardware structure, e.g., cores, coherence domains, etc. Single-node machines have now embraced loosely-coupled and heterogeneous processing units, inspiring recent OS works such as Barrelfish [5], Helios [38], fos [37], and K2 [39] to refactor the kernel to achieve highly-scalable performance through replicated software entities that run (mostly) independently on each core. While these approaches improve scalability, they do not address evolution, runtime flexibility, or fault isolation; Theseus's approach is almost opposite of these works, in the sense that we explicitly avoid reliance upon underlying hardware to define OS structure and functionality.

10.3 Live Update of OS Components

Live update of systems software has been extensively studied. Many works retrofit live update into legacy OSes like Linux [109, 110, 12, 111, 112, 113, 114]. Existing solutions need deep kernel expertise or tedious manual effort to generate or apply an update [113, 110, 109]; some impose overhead due to intermediary layers of indirection [14, 115, 109] or full-system checkpointing [114]; others are unable change kernel APIs, internal data structures, or non-function entities [110, 12, 111]. Most works target small, localized security patches, whereas Theseus targets sweeping evolutionary changes to core kernel entities, their modularity, and kernel APIs with a new OS structure and spill-free design.

A unique approach to live update is KUP [114], which switches out the entire Linux kernel with a new updated version after checkpointing all user applications. This approach is easy to apply, but is not arbitrary because the user-kernel interface cannot change, and applications and kernel components cannot be jointly updated. In addition, this coarse-grained approach incurs several seconds of overhead even for tiny kernel changes. In fact, many approaches that we encountered in the literature rely upon a similar assumption — that the user-kernel interface is stable and never changes — which is true only in Linux. This comes at great human cognitive cost, and significantly limits flexibility and innovation.

K42 [14, 115, 58, 116] is an object-oriented OS that deeply explores how to realize live update in an OS environment by hot swapping object instances, similar to Theseus's cell swapping. Like cells in Theseus, an object has clear runtime persistent boundaries, but objects are described by a virtual base class. In K42, each address space has an *object translation table* through which all object invocations must go; K42 leverages this layer of indirection to redirect object invocation to an updated one. This indirection layer incurs a constant overhead to *all* interactions on a given object [115]. Unlike Theseus, K42 requires a uniform indirection layer beneath all objects and can swap only objects, not anything

beneath the OOP language layer, such as low-level code for handling exceptions, interrupts, and interacting with hardware; this is due to its object tracking layer that depends upon per-thread references to objects.

Similarly, microkernel solutions like PROTEOS [11], based on MINIX 3, can accommodate complex system updates for userspace server processes. Theseus chooses a finer-grained entity bounds that is decoupled from hardware-based protection, allowing it to evolve and recover from faults in entities at the microkernel level. Theseus builds upon PROTEOS's techniques for state transfer but does not make significant contributions therein; we still consider Proteos [11] to be the state-of-the-art for easily defining live update state transfer functions.

10.4 Fault Tolerant OSes

The literature on tolerating faults within OS components spans a wide variety of approaches, including using software domains to isolate and recover from failures in drivers and select OS subsystems [117, 118, 119], hardware isolation between OS service processes in microkernels [64, 63], and checkpoint/restore of drivers [65] or OS services [55] for faster, stateful recovery. Theseus uses intralingual mechanisms like unwinding to ensure that language-level safety assumptions and compiler-provided isolation are not violated by recovery actions. Theseus also distinguishes between recovering an entity (cell) and an execution context (task), can recover and replace finer-grained entities than processes, and leverages novel state management techniques to simplify recovery logic.

Language-level Errors: A concept related to handling faults at the OS level is how errors are handled at the language level. There exist two primary categories of error handling mechanisms: *divergent* error handling that completely interrupts and bypasses normal control flow to execute a dedicated exception path, or *convergent* error handling that returns

an explicit error directly to the caller after checking for said conditions. In managed language runtimes (e.g., Java, Python), the former approach is often utilized because it allows error checking to be frequently hidden, resulting in code that may be simpler to follow at first but is more challenging to determine exactly which exceptions may propagate or be caught. In Rust, the former choice equates to throwing a `panic`, Rust's version of a untyped exception, while the latter equates to simply returning a specific error type.

It is our belief that convergent error handling is the superior choice as it makes error conditions and their handling explicit, which is especially valuable in an OS environment where reliability and clarity is paramount. Furthermore, handling an exception by unwinding the call stack is expensive, obscures control flow, and may leave objects in an inconsistent state; thus, exceptions should be avoided particularly when error conditions are likely, as they frequently are in an OS environment. Ideally, Theseus would be free from code that may panic, but this is currently impossible because even core Rust libraries (and many third-party crates) make copious use of `panic`s internally. This is one of the many reasons why we support the handling of divergent errors via Theseus's unwinder that activates upon exceptions; in fact, Theseus is a rarity in that freestanding language environments without an underlying OS and standard library (e.g., OS kernels) do not support exception handling. In the implementation of all Theseus entities, we always follow convergent error handling practices to explicitly check for all error conditions and return a specially-typed value to represent the potential of said errors, e.g., Rust's `Option` type or `Result` type parameterized with a domain-specific error enum.

Chapter 11

Discussion and Limitations

11.1 Unsafe Code: an Unfortunate Necessity

In a low-level kernel environment, unsafe code is necessary to interface with hardware because the compiler rightly lacks a model of hardware semantics. An interesting line of future work entails augmenting with specifications of machine, device, or ISA behaviors that detail the conditions for correct access to or interaction with them, then the compiler could check that those conditions are met and the behavioral semantics are not violated, essentially extending the coverage of instructions that can are considered *safe* beyond normal heap or stack memory accesses. Adding that knowledge into the Rust compiler would allow it to determine, for example, that "loading the GDT and TSS structures with the values XYZ is correct and cannot possibly cause a CPU exception" or that "writing value X to model-specific register Y is valid." Doing so would avoid the vast majority of unsafe code.

Not all unsafe code is equal; many unsafe statements do not break isolation, such as writing to model-specific registers (`wrmsr`) or an I/O port. Therefore, we distinguish between two types of unsafe code: *innocuous* and *infectious*, in which infectious code may violate data isolation but innocuous cannot. Unsafe code is *infectious* if it can access data inside another entity, thereby "infecting" it, e.g., dereferencing arbitrary pointers, accessing beyond the bounds of a sized type, but is *innocuous* if it merely accesses data reachable from safe code, e.g., a function that writes the address of its argument to an I/O port. Innocuous code can still cause incorrect behavior. As part of ongoing work, we develop a compiler

plugin to automate checks for the reachability and type safety of addresses accessed in unsafe blocks; this currently supports language-level unsafe blocks (e.g., within `MappedPages`) but requires manual whitelisting of inline assembly, e.g., context switch routines.

11.2 Incomplete Fault Recovery

As described in §9.2, Theseus's unwinder works perfectly for language-level exceptions (Rust `panics`), but not for all machine faults. This is because all potential panic locations – ranges of instruction pointers – are known statically to the compiler and can thus be accounted for when the compiler decides how and when to generate unwinding cleanup routines (drop handlers). In contrast, machine faults (e.g., CPU exceptions) may occur at any instruction pointer, so the compiler cannot know about them statically.

The known solution for this is to generate *asynchronous* unwind tables, in which cleanup routines cover *all* instruction pointer ranges, allowing unwinding to properly occur at any point. Unfortunately, LLVM, and thus Rust, does not support asynchronous unwind tables. As such, local variables may not be covered by synchronous unwinding tables. This is only problematic in the excepted stack frame, as all prior stack frames are reachable through standard call sites and thus can be fully cleaned up.

We are exploring possible OS-level solutions to augment unwinding information, such as leveraging DWARF debug information to discover drop routines, or manually recreating local variables from function arguments to obtain a droppable instance of non-unwound values. Other in-compiler approaches include modifying backend code generation, e.g., to insert additional safepoints or dummy call instructions that increase coverage of unwinding tables, but these are beyond the scope of this work and likely incomplete solutions. While these techniques are far from complete, any best-effort approach is an improvement over the default OS behavior of a kernel oops and shutdown.

11.3 Stack Corruption and Stack Overflow

Stack corruption is challenging to recover from, but in general cannot happen in a safe language. Nevertheless, we can make at least some progress towards recovery when a stack gets corrupted via two mechanisms. First, our unwinder maintains its progress in the call stack and its full context independently of the stack being unwound. Second, sanity checks are possible due to the availability of detailed cell metadata, for example, we can confirm that calculated register values and addresses are sensible, e.g., the instruction pointer register points to an address in a crate's text section, the landing pad address points to a region within a crate's LSDA section and is properly aligned, the calculated stack pointer register contains an address within the range of the current task's stack bounds, etc. This approach helps us proactively detect invalid stack values that lead to unwinding errors before they actually happen. Third, we can leverage a separate "emergency" stack that is automatically switched to upon multiple failures, e.g., by using the double fault stack specified by the TSS in x86_64. This enables Theseus to attempt to continue unwinding after an invalid address is accessed (if sanity checks were insufficient), or in the worst case scenario, jump to an arbitrary routine, e.g., to forcibly stop the corrupted task and restart it. This emergency stack is used to handle stack overflow, which we can properly recover from as well. Theseus can optionally reserve additional pages beyond the end of the stack that can be used to extend it, and map them upon an unexpected page fault or double fault due to stack overflow; this is a simple form of demand paging.

11.4 Limitations of Reliance on Safe Language

The most significant limitation is that all applications and system components must be implemented in safe Rust. Legacy code in other safe or managed languages could be

supported by implementing their VMs/runtimes in Rust, but unsafe languages would require hardware protection or dynamic interposition on memory accesses (à la SFI [117]) if isolation was desired. In addition, Theseus must trust that the Rust compiler and its core/alloc libraries uphold safety guarantees without soundness holes. Given the continued emergence of significant flaws in hardware protection mechanisms [120, 121], we prefer to trust an open-source compiler rather than closed hardware.

Fortunately, multiple works are ongoing to improve and verify the Rust compiler and its base libraries by checking unsafe usage [122, 123, 124], lowering the risk of trusting Rust. As we do not rely upon the Rust standard library in which most of the soundness holes have been found, our trusted computing base is still smaller and less risky than general Rust projects.

Fault isolation in a safe language requires us to trust the Rust compiler and its core library, but notably *not* the standard library because it is not used or built in Theseus's kernel. For example, the build system limits access to certain system resources by throwing a compiler error if, for example, an application crate tries to use a data type or invoke a function in a low-level kernel crate that is only accessible by other kernel entities. This is akin to a capability system, specifically the part that checks which capabilities a given entity should be allowed to request access to.

However, we do not actually need to trust that the compiler is doing our bidding, since we have another layer — the loader/linker in the cell manager — that is more trusted and can actually verify at runtime whether a given crate is requesting access to other crates' types and functions that it cannot access (per its manifest). This reduces the trust in the compiler's build dependency system to get things right, and also presents an opportunity for secondary verification, potentially allowing us to run pre-built crate executables, although we do not do so now.

11.5 Less Global Knowledge due to State Spill Freedom

Currently, Theseus possesses no global list of all memory allocations, which by default prevents a mechanism for obtaining statistics about all memory usage, or having a centralized way to free all memory upon. There are other benefits to this choice, beyond those those generally resulting from state spill freedom. Primarily, the owner of a `MappedPages` object is *guaranteed* that it has sole control over and sole knowledge of the memory mapping, meaning (*i*) that it can be assured that the underlying memory will never be unmapped out from underneath it and rendered inaccessible, and also (*ii*) that it is secure in using the memory region, in that no other entity (even system entities) is able to access it, mutably or immutably, without it expressly granting another entity access to that memory region.

To recover memory in use, e.g., under memory pressure, the default action is to kill the current task and then proceed to killing lower-priority tasks until an allocation succeeds. This does not require the system to have any knowledge of application's memory usage, since killing a task will fully unwind all application tasks' stacks and thus guarantee that their allocated and owned memory regions are freed and that all of their resources dropped/released. Additionally, several types that represent cached information, e.g., block device caches, implement functionality that offers up their underlying memory regions for reclamation by the OOM handler. To implement more convenient memory reclamation, this can be extended to track any arbitrary `MappedPages` region, e.g., by using weak references or indirect records of allocated regions. Though this does constitute state spill, spilling weak references rather than strong shared references to these cache regions into an OOM-handling crate limits the negative effects of said spill.

11.6 Limitations of Theseus's Prototypical Evolutionary Mechanisms

Not all evolutionary actions are automatically facilitated by Theseus. For example, evolving a cell whose code does not execute again, such as a device initialization routine that only runs once upon boot, will not affect that existing device's configuration. Theseus takes no default action upon loading a crate, such as invoking a crate initializer function, which we regard as best practice because it reduces side effects and removes the complexity of unexpected execution contexts. Instead, a new function should be explicitly invoked to reconfigure said device, similar to how other crates would independently make use of new APIC or priority scheduler features. Both lazy initialization and default type values can help with this, e.g., `lazy_static` or `Once` and the `Default` trait in Rust, respectively. Similarly, loading a crate with differing data structure definitions may not necessarily replace prior instances of the same structure throughout the system, as CellNamespaces permit multiple instances of the same crate or different crates with identical type definitions.

11.7 Potential Benefits for System Efficiency

Though we do not make claims of an overall improvement in efficiency, the measurements in §9.3 indicate that Theseus has the potential to increase system performance. Various works in the OS literature also claim that safe languages can increase performance [56] along several dimensions, primarily due to avoiding overhead due to switching between address spaces and privilege modes.

Currently, we are investigating whether modern languages like Rust can be even more efficient than those used in prior safe-language OSes (e.g., Java, Modula 3), as most have relied upon an underlying runtime layer and garbage collection for memory management. For this endeavor, we use Theseus as a testbed to experiment with various OS configurations

in which hardware features are toggled on or off, e.g., different privilege modes (protection rings), multiple address spaces, virtual addressing vs. physical addressing, and address translation caching. In addition, we posit that Theseus's intralingual design can further reduce runtime overhead beyond that of typical safe languages, due to shifting resource bookkeeping into the compiler and the distributed approach towards state management that avoids contention on globally-held resource states. Finally, we expect to realize efficiency gains by leveraging Rust's zero-cost abstractions and monomorphized generics, as well as reducing layers of indirection, especially in glue layers that connect a high-level standard library interface to the underlying Theseus-specific implementation.

11.8 Extending Design Principles to Applications

Although this thesis and our overall design efforts have focused on core OS subsystems, the design principles presented herein are also effective in higher layers of the system, such as applications and libraries. In Theseus, we strive to follow the same design principles for applications and libraries as those that we follow for core OS components.

As libraries and applications represent the very clients into which Theseus's opaque exportation principle may push states, the guidelines surrounding state spill obviously must be relaxed. Applications clearly must harbor sufficient state for them to operate. However, it is best to minimize states spilled across different application instances in order to simplify the process of updating or restarting a crashed application. Granted, it is unlikely that a user application's state will pose a significant obstacle to system evolution or flexibility, at least not to the degree that a core kernel component does. This is because application states can typically be externalized to persistent storage as part of a backup or checkpoint operation more easily than core OS states, as they have far fewer dependencies. Designing an application or library to have fewer stateful dependencies is possible through techniques

like stateless communication protocols, although these may be inefficient in systems with multiple address spaces and privilege levels.

Regarding intralinguality, the focus within application layers is less about representing hardware semantics in a compiler-understandable manner and more about preserving existing knowledge stemming from lower layers. Thus, intralinguality is important for losslessness of language context, but less important overall because incorrect application semantics that break compiler-assured invariants are unlikely to jeopardize system stability. Note that this does not hold for "system"-like applications, such as the window manager or a terminal emulator that is used to drop into a shell during recovery scenarios. Such applications or libraries that add significant complexity or layers of abstraction atop a kernel-provided resource should take great care to avoid interfering with or misrepresenting the safety guarantees of those underlying layers.

11.9 Wish List for Rust Features

Rust's built-in lifetimes are both powerful and expressive, but are insufficient for some cases in Theseus due to the limitations of the single-process program model. For example, there is no way to express that an object is valid for the lifetime of the current task (its containing task). It is possible to throw together a hacky solution in which the task's `Stack` object or `Task` struct object itself could be passed from the task's entry point to every function it invokes, but that would lead to highly unergonomic code and limit the actual utility and possibility of transferring ownership of objects.

Many built-in types do not even offer the ability to parameterize them with lifetimes, because they are assumed to have a static type lifetime. For example, the primitive function type `fn` assumes the function itself has a static lifetime and that all of the references to it will persist forever. With dynamic swapping of cells in Theseus, there is no guarantee that

the function's underlying `.text` section will exist for all eternity (as the static lifetime implies), but there is no way to express the lifetime of a function to be non-static. Though a function reference itself can have a lifetime, it is of no help here. Closures can have lifetimes, but only when borrowed, and the ability to place a closure into a Box as an owned object on the heap further complicates this, as Rust lacks the syntax to specify that said closure's text memory region is itself non-static and may not outlast that of the surrounding allocated Box object. We sidestep this limitation with a task-cell ownership relation, but an explicit lifetime-based expression of this relationship would be more clear.

Chapter 12

Conclusion

This thesis has introduced and explored the concept of state spill, including an in-depth study demonstrating its prevalence in existing systems and a discussion of how various forms of state spill hinder multiple desirable computing goals. On the quest to address state spill and avoid its harmful effects on system evolvability, availability, and flexibility, we presented Theseus, an OS written from scratch in Rust to rethink state management and the division of responsibilities shared between OS and compiler.

During our design process of experimenting with unique OS structure and the power of Rust, a modern systems programming language that provides compile-time type and memory safety, we came to the realization that an OS's usage of a safe language could go beyond mere safety. We thus established three design principles for Theseus to follow: (*i*) require all entities to have clear bounds that persist into runtime, (*ii*) empower the language and compiler to the fullest extent possible, and (*iii*) avoid state spill in OS entities. To satisfy these principles, Theseus adopts a novel OS structure of many tiny entities that are dynamically loaded and linked into a language-isolated environment with no hardware protection, and adopts an intralingual design that implements OS semantics using language-provided mechanisms that expose all OS functionality and resource behavior to the compiler. Its intralingual design enables the compiler to take over bookkeeping and management of OS resources, significantly reducing state spill and simplifying OS management of application and task states. This in turn facilitates the realization of live evolution, availability via fault recovery, and arbitrarily flexible OS personalities.

In this work and our future endeavors, we strive to revive and refocus the path of OS research towards modern safe languages, cleanly disentangled system design, unencumbered flexibility to accommodate runtime evolution and availability, and most importantly, proper state management that gives due thought to state spill across all OS components.

Bibliography

[1] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *ACM SIGOPS Operating Systems Review*, 2007.

[2] "A proactive approach to more secure code." https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/. Accessed: 2020-05-22.

[3] "The Chromium projects: Memory safety." https://www.chromium.org/Home/chromium-security/memory-safety. Accessed: 2020-05-22.

[4] "Implications of rewriting a browser component in Rust." https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/. Accessed: 2020-05-22.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A new OS architecture for scalable multicore systems," in *Proc. ACM SOSP*, 2009.

[6] K. Boos, E. D. Vecchio, and L. Zhong, "A characterization of state spill in modern operating systems," in *Proc. ACM EuroSys*, 2017.

[7] M. Siniavine and A. Goel, "Seamless kernel updates," in *Proc. IEEE/IFIP DSN*, 2013.

[8] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler, "User-level checkpointing through exportable kernel state," in *Proc. IEEE Workshop on Object-Orientation in Operating Systems*, 1996.

[9] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security," in *Proc. USENIX Security*, 2005.

[10] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," in *Proc. ACM EuroSys*, 2007.

[11] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Safe and automatic live update for operating systems," in *Proc. ACM ASPLOS*, 2013.

[12] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proc. ACM EuroSys*, 2009.

[13] C. A. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, "System support for online reconfiguration.," in *Proc. USENIX ATC*, 2003.

[14] K. Hui, J. Appavoo, R. Wisniewski, M. Auslander, D. Edelsohn, B. Gamsa, O. Krieger, B. Rosenburg, and M. Stumm, "Supporting hot-swappable components for system software," in *Proc. HotOS*, 2001.

[15] R. Sethi, *Programming languages: concepts and constructs*. Addison-Wesley Reading, 1996.

[16] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology (TOIT)*, 2002.

[17] C. Pautasso and E. Wilde, "Why is the web loosely coupled?: a multi-faceted metric for service design," in *Proc. WWW*, 2009.

[18] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, 2000.

[19] A. Van't Hof, H. Jamjoom, J. Nieh, and D. Williams, "Flux: Multi-surface computing in Android," in *Proc. ACM EuroSys*, 2015.

[20] B. P. Swift, "User mode scheduling in MINIX 3," tech. rep., Vrije University Amsterdam, 2010.

[21] K. Boos, A. Amiri Sani, and L. Zhong, "Eliminating state entanglement with checkpoint-based virtualization of mobile OS services," in *Proc. ACM APSys*, 2015.

[22] Genode, "Design of the Genode OS architecture: Interfaces and mechanisms." http://genode.org/documentation/architecture/interfaces. Accessed: 2016-02-22.

[23] R. W. Scheifler and J. Gettys, "The X window system," *ACM Transactions on Graphics (TOG)*, 1986.

[24] N. Feske and C. Helmuth, "A Nitpicker's guide to a minimal-complexity secure GUI," tech. rep., Technische Universität Dresden, 2005.

[25] "X.Org Security Advisory: Dec. 9, 2014." https://www.x.org/wiki/Development/Security/Advisory-2014-12-09/. Accessed: 2016-10-11.

[26] D. D. Clark, "The structuring of systems using upcalls," in *Proc. ACM SOSP*, 1985.

[27] P. Barton-Davis, "Linux kernel mailing list: upcalls." http://lkml.iu.edu/hypermail/linux/kernel/9809.3/0922.html. Accessed: 2016-10-13.

[28] "Invoking user-space applications from the kernel." https://www.ibm.com/developerworks/library/l-user-space-apps/. Accessed: 2016-10-13.

[29] K. Sen, "Concolic testing," in *Proc. IEEE/ACM ASE*, 2007.

[30] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proc. CASCON*, 1999.

[31] M. Weiser, "Program slicing," in *Proc. ACM/IEEE ICSE*, 1981.

[32] "UI/Application Exerciser Monkey." https://developer.android.com/studio/test/monkey.html. Accessed: 2016-10-10.

[33] P. Dong, "Reducing fate-sharing in software systems via fine-grained checkpoint and restore," Master's thesis, Rice University, 2016.

[34] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An operating system architecture for application-level resource management," in *Proc. ACM SOSP*, 1995.

[35] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in *Proc. ACM ASPLOS*, 2011.

[36] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proc. ACM ASPLOS*, 2013.

[37] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, "An operating system for multicore and clouds: Mechanisms and implementation," in *Proc. ACM SoCC*, 2010.

[38] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: Heterogeneous multiprocessing with satellite kernels," in *Proc. ACM SOSP*, 2009.

[39] F. X. Lin, Z. Wang, and L. Zhong, "K2: A mobile operating system for heterogeneous coherence domains," in *Proc. ACM ASPLOS*, 2014.

[40] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *Proc. USENIX OSDI*, 2018.

[41] S. Klabnik and C. Nichols, "The Rust programming language." https://doc.rust-lang.org/book/. Accessed: 2020-05-22.

[42] A. C. Bomberger, A. P. Frantz, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro, "The KeyKOS nanokernel architecture," in *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[43] J. Liedtke, "On micro-kernel construction," in *Proc. ACM SOSP*, 1995.

[44] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64KB computer safely and efficiently," in *Proc. ACM SOSP*, 2017.

[45] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, safety and performance in the SPIN operating system," in *Proc. ACM SOSP*, 1995.

[46] "Redox - your next(gen) os." https://www.redox-os.org/. Accessed: 2017-08-11.

[47] K. Takeuchi, K. Honda, and M. Kubo, "An interaction-based language and its typing system," in *Intl. Conf. on Parallel Architectures and Languages Europe*, 1994.

[48] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe, "A fork() in the road," in *Proc. HotOS*, pp. 14–22, 2019.

[49] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[50] R. Fielding, "Representational state transfer," *Architectural Styles and the Design of Network-based Software Architecture*, 2000.

[51] A. Schuett, S. Raman, Y. Chawathe, S. McCanne, and R. Katz, "A soft-state protocol for accessing multimedia archives," in *Proc. Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 1998.

[52] D. Clark, "The design philosophy of the DARPA internet protocols," in *Proc. ACM SIGCOMM*, 1988.

[53] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Transactions on Software Engineering*, 1996.

[54] N. Liyanage, "Fault recovery in the Theseus operating system," Master's thesis, Rice University, 2020.

[55] J. Herder, *Building a dependable operating system: fault tolerance in MINIX 3*. PhD thesis, Vrije Universiteit Amsterdam, 2010.

[56] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus, "Deconstructing process isolation," in *Proc. ACM Workshop on Memory System Performance and Correctness*, pp. 1–10, 2006.

[57] "Intel NUC Kit NUC6i7KYK technical specifications." https://www.intel.com/content/www/us/en/products/boards-kits/nuc/kits/nuc6i7kyk.html. Accessed: 2020-04-26.

[58] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr, "Providing dynamic update in an operating system.," in *Proc. USENIX ATC*, 2005.

[59] K. Elphinstone and G. Heiser, "From L3 to seL4: What have we learnt in 20 years of L4 microkernels?," in *Proc. ACM SOSP*, 2013.

[60] E. Lahav, "Implementing mutexes in the QNX Neutrino realtime OS." https://www.osnews.com/story/29090/implementing-mutexes-in-the-qnx-neutrino-realtime-os/. Accessed: 2020-05-22.

[61] F. Bellard, "QEMU: a fast and portable dynamic translator," in *Proc. USENIX ATC*, 2005.

[62] L. Wang, Z. Kalbarczyk, W. Gu, and R. K. Iyer, "An OS-level framework for providing application-aware reliability," in *proc. IEEE Pacific Rim Intl. Symposium on Dependable Computing (PRDC)*, 2006.

[63] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "CuriOS: Improving reliability through operating system structure," in *Proc. USENIX OSDI*, 2008.

[64] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "MINIX 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[65] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Fine-grained fault tolerance using device checkpoints," in *Proc. ACM ASPLOS*, 2013.

[66] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[67] M. Inc., "Microquill smartheap 4.0 benchmark." http://microquill.com/.

[68] L. W. McVoy, C. Staelin, *et al.*, "LMbench: Portable tools for performance analysis.," in *USENIX annual technical conference*, pp. 279–294, San Diego, CA, USA, 1996.

[69] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proc. ACM SOSP*, 2009.

[70] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The Flux OSKit: A substrate for kernel and language research," in *Proc. ACM SOSP*, 1997.

[71] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide, "Knit: Component composition for systems software," in *Proc. USENIX OSDI*, 2000.

[72] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller, "THINK: A software framework for component-based operating system kernels," in *Proc. USENIX ATC*, 2002.

[73] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Transactions on Computer Systems*, 2008.

[74] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *Proc. ACM/IEEE ICSE*, 2011.

[75] A. J. Offutt, M. J. Harrold, and P. Kolte, "A software metric system for module coupling," *Journal of Systems and Software*, 1993.

[76] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt, "Maintainability of the Linux kernel," *IEE Proceedings-Software*, 2002.

[77] L. Yu, S. R. Schach, K. Chen, G. Z. Heller, and J. Offutt, "Maintainability of the kernels of open-source operating systems: A comparison of Linux with FreeBSD,

NetBSD, and OpenBSD," *Journal of Systems and Software*, 2006.

[78] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors," in *Proc. ACM/IEEE ICSE*, 2000.

[79] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *Proc. ACM OOPLSA*, 2005.

[80] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proc. USENIX OSDI*, 2006.

[81] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," in *Proc. ACM SOSP*, 2007.

[82] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres, "Securing distributed systems with information flow control.," in *Proc. USENIX NSDI*, 2008.

[83] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proc. ACM PLDI*, 2014.

[84] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. USENIX OSDI*, 2010.

[85] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. ACM MobiSys*, 2011.

[86] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with EPICC: An essential step towards holistic security analysis," in *Proc. USENIX Security*, 2013.

[87] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM CCS*, 2014.

[88] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder, "The JX operating system," in *Proc. USENIX ATC*, pp. 45–58, 2002.

[89] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a POSIX kernel in a high-level language," in *Proc. USENIX OSDI*, 2018.

[90] S. M. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. W. Trickey, and P. Winterbottom, "The Inferno ™ operating system," *Bell Labs Technical Journal*, 1997.

[91] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, "The case for writing a kernel in Rust," in *Proc. ACM APSys*, 2017.

[92] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. USENIX OSDI*, 2016.

[93] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, "Splinter: bare-metal extensions for multi-tenant low-latency storage," in *Proc. USENIX OSDI*, 2018.

[94] M. Yun and L. Zhong, "Ginseng: Keeping secrets in registers when you distrust the operating system," in *Proc. NDSS*, June 2019.

[95] S. Miller, K. Zhang, D. Zhuo, S. Xu, A. Krishnamurthy, and T. Anderson, "Practical safe Linux kernel extensibility," in *Proc. HotOS*, 2019.

[96] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. ACM SOSP*, 1993.

[97] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, "Dealing with disaster: Surviving misbehaved kernel extensions," in *Proc. USENIX OSDI*, 1996.

[98] G. C. Necula and P. Lee, "Safe kernel extensions without run-time checking," in *Proc. USENIX OSDI*, 1996.

[99] C. Jacobsen, M. Khole, S. Spall, S. Bauer, and A. Burtsev, "Lightweight capability domains: Towards decomposing the Linux kernel," *SIGOPS Oper. Syst. Rev.*, 2016.

[100] G. Andert, "Object frameworks in the Taligent OS," in *Digest of Papers for Compcon Spring*, 1994.

[101] J. Lewis and M. Fowler, "Microservices." https://martinfowler.com/articles/microservices.html. Accessed: 2017-08-10.

[102] M. Ranney, "What I wish I had known before scaling Uber to 1000 services." Presentation at Int. Software Development Conf. (GOTO Chicago), https://youtu.be/kb-m2fasdDY, 2016.

[103] T. Mauro, "Adopting microservices at Netflix: lessons for architectural design." https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, Accessed: 2017-08-10.

[104] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Proc. HotOS*, 2017.

[105] S. Newman, *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.

[106] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *Proc. International Workshop on Mashups of Things and APIs*, 2016.

[107] R. Koller and D. Williams, "Will serverless end the dominance of Linux in the Cloud?," in *Proc. HotOS*, 2017.

[108] N. Feske, "Genode operating system framework." https://genode.org/documentation/genode-foundations-19-05.pdf, 2015. Accessed: 2017-08-19.

[109] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," in *Proc. ACM Int. Conf. Virtual Execution Environments*, pp. 35–44, 2006.

[110] RedHat, "Introducing kpatch: Dynamic kernel patching." https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching, 2014.

[111] SUSE, "SUSE releases kGraft for live patching of Linux kernel." https://www.suse.com/c/news/suse-releases-kgraft-for-live-patching-of-linux-kernel/, 2014.

[112] S. J. Vaughan-Nichols, "Kernelcare: New no-reboot Linux patching system." https://www.zdnet.com/article/kernelcare-new-no-reboot-linux-patching-system/, 2014.

[113] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," in *Proc. ACM EuroSys*, 2003.

[114] S. Kashyap, C. Min, B. Lee, T. Kim, and P. Emelyanov, "Instant OS updates via userspace checkpoint-and-restart," in *Proc. USENIX ATC*, 2016.

[115] C. A. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. A. Auslander, M. Ostrowski, *et al.*, "System support for online reconfiguration.," in *USENIX Annual Technical Conference, General Track*, pp. 141–154, 2003.

[116] A. Baumann, J. Kerr, J. Appavoo, D. Da Silva, O. Krieger, and R. W. Wisniewski, "Module hot-swapping for dynamic update and reconfiguration in K42," in *6th Linux. Conf. Au*, 2005.

[117] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proc. ACM SOSP*, 2003.

[118] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," in *Proc. USENIX OSDI*, 2004.

[119] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: an organizing principle for recoverable operating systems," in *Proc. ACM ASPLOS*, 2009.

[120] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proc. Usenix Security*, 2018.

[121] 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 *Proc. IEEE Symp. Security and Privacy (SP)*, 2019.

[122] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," in *Proc. ACM POPL*, 2017.

[123] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: An aliasing model for Rust," in *Proc. ACM POPL*, 2020.

[124] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world Rust programs," in *Proc. ACM PLDI*, 2020.