A SECURE OPERATING SYSTEM FOR THE INTERNET OF THINGS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Amit Aryeh Levy

November 2020

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(David Mazières) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Philip Levis)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Keith Winstein)

Approved for the Stanford University Committee on Graduate Studies

Abstract

The Internet of Things describes the phenomenon of connecting low-resource edge devices to the Internet. This phenomenon, while exciting, uncovers safety, performance, and flexibility challanges for the low-resource computers.

In particular, the microcontrollers that power these devices lack some of the hardware features and memory resources that enable multiprogrammable systems. Accordingly, microcontroller-based operating systems have not provided important features like fault isolation, dynamic memory allocation, and flexible concurrency. However, the Internet of Things changes these devices into software platforms, rather than single purpose devices, requiring multiprogramming features.

This dissertation describes Tock, a new operating system for low-power platforms, that takes advantage of the limited hardware-protection mechanisms of contemporary microcontrollers as well as the type-safe features of the Rust programming language to provide a multiprogramming environment for microcontrollers. Tock isolates software faults, provides memory protection, and efficiently manages memory for dynamic application workloads written in any language. It achieves this while retaining the dependability requirements of long-running applications.

The trade-off between extensibility, safety, and performance is long standing in operating systems. More specifically, prior systems in the research have attempted to use language type safety to replace, or augment, hardware-based isolation mechanisms in the kernel. However, these systems used garbage collected languages, generally viewed as too resource-heavy for kernel programming in production systems.

Tock's use of Rust shows that, given a type-safe language with sufficient control of memory and processor resources, language-based approaches can be *more* efficient than hardware. Moreover, Tock's use in academia and industry show that this approach is practical in real-world deployments.

Acknowledgments

I thank my advisors, "Professor" David Mazières and Professor Phil Levis, for the consistent encouragement, freedom, and resources to pursue the most ambitious work I could dream-up. I also thank them for helping shape my aesthetic for system design and research, each in their (very) unique ways. Thanks to Keith Winstein, Alejandro Russo, Dan Boneh, John Mitchell, Dawson Engler, and Prabal Dutta for providing informal mentorship, guidance, and wisdom, and for being a sounding board for ideas and frustrations. Before graduate school, when I was still planning on being able to afford rent, Roxana Geambasu, Hank Levy, Arvind Krishnamurthy, and Tadayoshi Kohno introduced me to research, and have continued to support and encourage me ever since. I am forever in your debt.

Next I would like to thank my other close research collaborators, who are responsible for much of this and other work: Pat Pannuto, Branden Ghena, Brad Campbell, Daniel Giffin, Deian Stefan Michael Andersen, Gabe Fierro, David Culler, Pablo Buiras, Henry Corrigan-Gibbs, James Hong, Laurynas Riliskis, and Thomas Bauer. My lab-mates at Stanford, who enriched, motivated, and inspired me: Andrea Bittau, Adam Belay, Ali Mashtizadeh, Edward Yang, and Sergio Benitez. Thanks to my friends at Columbia University for graciously hosting me: Vaggelis Atledakis, George Argyros, Mathias Lecuyer, Riley Spahn, and Theofilos Petsios.

Extra special thanks to my co-author-slash-lab-mate-slash-housemate-slash-co-founder-slash-confidant-slash-friend David Terei for all-of-the-above and more.

I thank my committee—Dan Boneh, Phil Levis, Keith Winstein, David Mazières, and Heather Hadlock—for their time and feedback on this work.

During my Ph.D. I was lucky to collaborate with folks in industry on Tock. In particular, I thank Dominic Rizzo, Marius Schilder, Bill Richardson, and Jay Kickliter for their trust, feedback, and support of both me and the work. I was also lucky to work with Ian Ross, Sascha Trifunovic, and Lindsey Jacks at MemCachier. Thanks for being amazing colleagues and accommodating my split attention.

Thanks to whoever stole David and my bikes from *inside* our apartment. Looking at smart door-locks got me started on this whole Internet of Things kick. So, in a way, this was all your idea.

Finally, and most importantly, thanks to Kaitlin, without whom there would be little point to any of this. You know what you did.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Using type safety to enforce isolation in low-level systems is practical and efficient.

This dissertation describes the design and implementation of Tock, a secure operating system for low-memory microcontrollers particularly targeted at the Internet of Things domain. The Tock kernel is written in Rust, a relatively new, but popular type-safe programming language that uses a linear type system to avoid garbage collection. Tock leverages Rust to get type safety without the overhead typically associated with type-safe language runtimes.

The Internet of Things presents substantially more adversarial deployment scenarios than most previous uses for microcontrollers. Similarly, microcontrollers are substantially more resource constrained than the computers that most secure operating systems are designed for. As a result, Tock's design radically departs in a number of ways from both traditional, so called, "embedded" operating systems for microcontrollers and previous secure operating systems for resource-rich general-purpose computers, such as servers, desktops, and smart phones.

In particular, Tock requires an extremely lightweight isolation mechanism. On one hand, while many previous uses of microcontrollers could get away with no isolation mechanisms whatsoever, many Internet of Things devices cannot since they require software updates, implement complex network protocols, run third-party applications, and serve security critical functions. On the other hand, resources, in particular volatile memory, are so constrained in microcontrollers that traditional isolation techniques alone cannot reasonably satisfy this requirement.

We found it most pragmatic to use the type system, rather than hardware mechanisms, to provide a lightweight isolation abstraction. Tock introduces two novel mechanisms for providing extremely fined-grained isolation in a reliable kernel, while supporting dynamic workloads from arbitrary processes. In the coming chapters, I'll describe the system design with a focus on both the challenges and benefits of using Rust's particular type system, and linear types in general, to build a practical system for deployment in real-world applications.

This dissertation argues that system designers can use type-safe languages to provide expressive and lightweight isolation primitives at compile time. While prior work has shown that doing so is *possible*, my goal is to show that it is practical—that type safety solves problems arising in real systems better than using hardware mechanisms alone.

I use the Tock kernel as an existence proof for my thesis. Tock's use of the type system to provide isolation is practical because it satisfies constraints driven by real-world requirements that could not have been solved with hardware mechanisms alone. Tock is used and deployed in both academic research and industrial settings. Finally, Tock targets a particularly resource constrained setting—microcontrollers with much less than 1 MB of RAM, slow CPUs, and applications that need to draw microwatts on average. Successfully building and deploying an operating system that leverages type safety as a key isolation mechanism suggests this technique is practical in less resource constrained settings as well.

This dissertation is organized as follows: Chapter 2 contexualizes my thesis and Tock within operating system literature, and describes emerging trends in Internet of Things class computers and applications that motivated Tock's design. Chapter 3 explains why the Rust programming language is a particularly well suited type-safe language for implementing an operating system kernel. Chapter 4 describes the Tock operating system and how it uses the type system to sandbox most extensibility in the system. Chapter 5 discusses related work and Chapter 6 concludes.

Chapter 2

Background

2.1 Isolation & the Extensible Operating System

Operating systems are software that manage a computer's resources for its users and their applications [6]. Commonly, operating systems attempt to provide applications with flexible, safe, and performant access to hardware and software services.

Each of safety, flexibility, and performance tend to draw a cost on the others. For example, ensuring the system is safe from buggy or malicious applications requires some sacrifice in flexibility (indeed, preventing malicious behavior tautologically reduces the flexibility to write malicious applications). Providing application with flexible access to hardware often means sacrificing performance that could be gained by optimizing drivers for narrower use cases. And safety mechanisms usually incur a performance penalty (e.g., a context switch or reserving some unused hardware resource, such as memory).

As a result, this trade-off appears in the operating systems research from the earliest days of the community [53] and has since motivated decades of research in design of extensible, performant, and secure operating systems [1, 14, 17, 27, 33, 38, 89, 93]

Ultimately, the particular trade-offs a system makes are determined by the choice of isolation mechanism to sandbox applications or other extensions, and how the system *uses* these mechanisms through abstractions and interfaces. Broadly, there have been three categories of isolation mechanisms that have been explored in both research and practical systems: hardware-based isolation, software-based fault isolation, and using type-safe languages to sandbox code at compile-time or using a language runtime.

2.1.1 Hardware Address Space Isolation

Hardware mechanism have been the predominant choice for isolation in both research and practical systems. In particular, operating systems use some form of virtual memory, such as paged virtual memory, segmentation, or nested paging, to provide applications or other extensions with unique address spaces. All memory accesses from unprivileged code are translated in hardware from virtual to physical addresses through a mapping only privileged code can modify.

Commonly, operating systems expose this mechanism through the abstraction of a memory-isolated execution thread. Processes in UNIX and other monolithic operating systems isolate applications. Services in microkernels isolate operating system components such as device drivers. Hypervisors use the same principle to isolate guest operating systems. For simplicity, I'll refer to this category of abstraction as a process.

Address translation itself is generally very fast, since it is implemented by the hardware. Moreover, the process abstraction encapsulates both memory and performance isolation, since each isolated process is executed separately. This makes process-like abstractions attractive in many common scenarios, particularly for running independent compute-bound applications.

There are, however, three main drawbacks to hardware address space isolation.

First, communication between processes and the operating system often requires a context switch. For example, accessing shared hardware resources, such as I/O busses, requires a context switch from the process to a multiplexer. This makes it expensive to layer distrustful components. Reducing context switch overhead has been the target of optimization, particularly in the microkernels [42]. Recent research in systems for low-latency I/O has, instead, addressed this by moving application logic to hardware peripherals, such as the network interface controller [13, 64].

Second, commodity hardware only provides a fixed number of protection levels—typically, one between the kernel and applications, and another between the hypervisor and guest operating systems. This limits their utility. For example, applications cannot use the page table to sandbox untrusted components within the application.

Finally, the process abstraction couples execution threads with access control. This coupling is a feature for isolating applications in a multi-user operating system, but it can be undesirable when breaking up layered kernel components with sequentially dependent executions, such as the network stack, into isolated components.

2.1.2 Software-Based Fault Isolation

Software-based fault isolation (SFI), originally proposed in [84], is an isolation mechanism that transforms object code such that it is constrained in the data and code it can access. Importantly, SFI does not rely on virtualizing the address space of each extension and, therefore, does not require specialized hardware.

Commonly, SFI wraps “unsafe instructions” in a security monitor: a procedure call or additional inline instructions that dynamically verify the operation. An unsafe instruction is any instruction that cannot be statically verified to be safe, such jumping to an address stored in a register. The security monitor checks, for example, that the address stored in the register is within the extension’s code segment.

This approach provides much lighter weight protection domains than hardware-based virtual address spaces, since switching protection domains only requires a few more instructions than a procedure call rather than a context switch. Conversely, “common-case” performance is worse, since the instrumentation much add security monitor checks to many instructions that do not cross protection domains.

Software-based fault isolation techniques have been used in a variety of deployed systems. Hypervisors used binary translation for isolating guest operating systems on systems without support for virtualization in hardware [83]. The Chrome web browser includes a sandboxing mechanism, Native Client [91], that uses SFI to run native code included in untrusted websites. However, in both cases, SFI was ultimately phased out in favor of hardware-based approaches once it became available on x86, or language based approaches [22].

2.1.3 Using Type-Safe Languages

Another branch of academic work from the operating systems, networking, and programming languages communities uses strictly enforced type and module systems as an isolation mechanism. The idea is simple: use the compiler or language runtime to verify that isolated components cannot access memory unless they are specifically allowed to by the type system. This is sufficient to encapsulate sensitive data and reserve functionality to certain components.

The Spin [14] operating system used the Modula-3 type system to isolate extensions to the kernel while using virtual address spaces to isolate applications. Pilot [67] and Singularity [38] did away with hardware protection entirely and used a type-safe language to isolate all components in the system.

Active Networks [88] used the Java virtual machine to allow untrusted end-hosts to extend network middleboxes.

JIF [61] and LIO [71] enforced very high-level security primitives using the type system. Safe Haskell [75] allowed programmers to explicitly separate between modules and libraries they trust to use unsafe features of the language and those that should only have access to a safe subset.

Generally speaking, using the type system for isolation is very flexible. Unlike hardware protection, type- and module-systems are not reserved for one or two trusted components. Isolated components can re-use the same abstractions to sandbox their own extensions. Similarly, it is non-hierarchical: components can be mutually distrustful. Language-based isolation is substantially more expressive than both hardware and software-based fault isolation. In particular, language-based isolation can express finer-grained access permissions than a memory address by encapsulating the same data in different types, each providing different semantics (e.g. constraining writes to only valid values). Finally, languages and compilers are much easier to change and extend than hardware and, thus, better suited to experimentation with higher-level security primitives, such as information flow control or capabilities.

Conversely, such systems typically sacrifice performance. While switching protection domains in a language isolated system can be done at no cost (or the cost of a procedure call), use of the language itself typically imposes a performance penalty. Type-safe languages are typically higher-level, providing the programmer with limited control over the execution and memory layout, and they are generally garbage collected, adding difficult to predict and manage overhead in terms of performance or memory consumption [55].

As a result, there has been limited adoption of language-based techniques for extensibility. A notable counter-example is the Web. Web browsers include the ability to run arbitrary third-party code written in JavaScript by simply typing in a URL. JavaScript in web pages is sandboxed from internal browser functionality because the runtime exposes a limited interface. JavaScript's success in the browser shows the power of language-based techniques. However, JavaScript's poor performance has lead to browsers implementing more and more performance sensitive functionality [40, 85] as well as myriad attempts to replace JavaScipt with alternatives [3, 16, 69, 91].

Research in designing extensible systems using the type system as an isolation primitive has successfully shown that these techniques are possible and have clear benefits. However, is it practical? This dissertation argues that it is. Specifically, that new languages with fundamentally better performance characteristics enable use of the type system for isolation in scenarios where hardware isolation is impractical.

2.2 Emerging Trends in Low-Power Microcontrollers

Historically, embedded applications have been designed to solve specific problems: collecting environmental data [23, 80, 87], localizing a sniper [47], recording fitness data [29], or detecting household fires [62]. In other words, they are single-purpose monoliths. The application requirements determine the hardware used, operating system configuration, and application software.

However, a new, emerging class of embedded applications breaks this monolithic model: they are software *platforms*, which support multiple, independent, dynamically-loadable applications. For examples, sports watches run applications that use the same hardware for different activities [30, 72]; USB authentication devices need to isolate multiple services from each other for security reasons (Figure 4.1); and city sensing infrastructure can run multiple applications written by different stakeholders [2].

Unfortunately, current operating systems cannot meet the requirements of these applications

	TelosB [65] (2004)	**Signpost [2] (2017)**
MCU	MSP430F1611 [60]	ATSAM4LC8CA [70]
Sleep Current	0.2 µA	1.6 µA
Word size	16-bit	32-bit
CPU Clock	8 MHz	12–48 MHz
Flash	48 kB	512 kB
RAM	10 kB	64 kB

Table 2.1: Embedded microcontroller RAM and flash have increased modestly over the past decade; a modern high-end platform such as Signpost uses a 32-bit Cortex-M4 microcontroller, with tens of kilobytes of RAM.

given the resource limitations of embedded microcontrollers.

2.2.1 Microcontrollers

Low-power microcontrollers (MCUs) have extremely limited resources compared to hardware platforms used for mobile, desktop or server computing. MCUs run at tens of megahertz, with tens of kilobytes of RAM and a megabyte or less of flash storage. Moreover, Moore's law will not obviate these limitations in the future since the limiting factor is energy. Improvements in MCU resources do not follow the same growth curves as CPUs. Table 2.1 shows the clock speed, RAM, and flash memory of two embedded research platforms, the TelosB mote (2004) [65] used in a decade of sensor network research, and Signpost (2017) [2], a recent platform for city-scale sensing which is representative of other recent platforms [5]. Although more than a decade has passed, RAM has only increased from 10 kB to 64 kB. The corresponding increase in sleep current to retain RAM contents has, and will, continue to limit growth.

While MCU resources have increased only modestly, 32-bit Cortex-Ms have a new feature absent in earlier microcontrollers: a memory protection unit (MPU). As they only have 10s of kilobytes of RAM, MCUs have neither virtual memory nor segmentation: every memory address is an absolute physical address. The MPU allows a kernel to protect regions of physical memory, providing memory isolation between applications as well as between applications and the kernel. Adapting the memory protection unit to existing embedded OS designs, however, has only limited benefit. FreeRTOS, for example, supports using a memory protection unit to prevent an application from writing to kernel memory. However, the FreeRTOS system call interface requires the kernel to trust any pointers passed through system calls, such that an application can make the kernel read or write arbitrary memory.

System	Concurrency	Memory Efficiency	Dependability	Fault Isolation	Loadable Applications
Arduino [10]		✓			
RIOT OS [9]		✓			
Contiki [25]	✓	✓			✓
FreeRTOS [12]	✓		✓		
TinyOS [50]	✓	✓	✓		
TOSThreads [44]	✓		✓		✓
SOS [37]	✓	✓			✓
Tock	✓	✓	✓	✓	✓

Table 2.2: Properties of embedded operating systems. Unlike prior designs, which trade off between different properties, Tock provides all five through grants, a memory-safe kernel, and a non-blocking API that supports memory protection units.

2.2.2 Embedded Operating Systems

Emerging embedded applications require that the embedded operating system support five key features: concurrency, dependability from resource exhaustion, fault isolation, memory efficiency, and application updates at runtime. While existing operating systems do not support all of these features, as Table 2.2 shows, each provides some of them.

Dependability Because embedded applications are often unattended or have limited user interfaces, they place a high premium on dependability—ensuring the system will continue running without intervention—often at the expense of other performance characteristics like speed or throughput. For example, a sensor network may be deployed in a remote or inaccessible location [23] and be unable to rely on human intervention to recover from faults. As a result, many embedded operating systems strive to increase dependability from memory exhaustion by ensuring that memory use is predictable at compile-time. They typically achieve this by either statically allocating memory for long-lasting values (TinyOS [50]) or restricting dynamic allocation to boot time (FreeRTOS [12]) for fail-fast behavior.

Concurrency Many embedded applications have tight energy budgets: a fitness tracker should maximize its runtime before recharging, and a city-sensing network might rely only on solar power. Increasing concurrency improves energy efficiency, since overlapping I/O operations allow the device to spend more time in a low-power sleep state [43]. As a result, most embedded operating systems allow many operations to occur in parallel. Systems such as TinyOS and SOS [37] provide concurrency through a cooperative run-to-completion model which simplifies stack management. Long running operations starve the CPU. To prevent this, some existing systems such as TOSThreads [44], FreeRTOS [12], and RIOT OS [9] use preemptive threads to run some or all code.

Efficiency As discussed in Section 2.2, RAM is a particularly valuable resource. Therefore, embedded OSs strive to be *memory efficient*, minimizing the amount of RAM allocated to exactly what an application needs. TinyOS, for example, statically counts callbacks to ensure it allocates just enough to service every callback [32], while Arduino [10] is just a thin wrapper for a monolithic C application. OSs that support dynamically loading new applications, such as SOS and TOSThreads, trade off memory efficiency and memory exhaustion. SOS can dynamically load and link new "modules," which can dynamically allocate memory from a shared global heap. This is efficient: applications do not allocate more than they need. However, this flexibility harms dependability, as an application's allocation can fail due to other applications. In contrast, TOSThreads is more dependable by statically allocating RAM in the kernel for every potential system call. However, this is inefficient: in typical use cases, 80% of this RAM is wasted (Section 4.4.3).

Fault isolation Isolating memory faults between system components is a common technique for supporting multiple applications running on the same system to ensure they cannot corrupt each other's state. However, until recently, microcontrollers provided no hardware memory protection mechanisms. As a result, embedded operating systems do not typically provide fault isolation and, instead, rely on careful API design or memory guard regions [37]. Some existing systems run applications in a bytecode interpreter [11, 49] which can provide software-based fault isolation.

Unlike existing systems, Tock simultaneously provides all five of these features by leveraging recent advances in microcontrollers and programming languages.

Chapter 3

Writing an OS Kernel in Rust: Benefits and Challanges

Most operating system kernels assume all kernel code is trusted. In part, this is because systems builders have traditionally relied on hardware enforced memory protection, and the process abstraction in particular, to provide isolation. The process, however, is a heavy-weight abstraction: it has stacks, file descriptors, and many kernel data structures. Furthermore, switching between virtual address spaces is costly. Systems such as Nooks [73], lightweight contexts [54] and seL4 [42] have all explored using address spaces for isolation at a finer grain than a process, showing ways to significantly reduce overhead.

An alternate approach is to entirely throw away hardware protection within the kernel and, instead, write the kernel in a memory-safe programming language. This approach has been tried before, with varying success: Spin [15] allows applications to extend and optimize kernel performance by downloading modules written in Modula-3 [20], while Singularity [38] is written in Sing# (a variant of C#) and provides a software isolated process (SIP) abstraction. Both Spin and Singularity, however, use garbage-collected languages, which pose many problems for kernels. Garbage collection complicates memory placement and layout, creates timing non-determinism from background locks and introduces stop-the-world intervals that pause the entire OS. Furthermore, Spin depends on a large, unsafe code base: the Spin kernel.

Rust is a type-safe language designed for systems software [56]. Instead of relying on garbage collection for safe memory management, Rust uses a memory management technique from the programming language literature that relies on affine types. In Rust, values are tracked by their lifetime and deallocated as they go out of scope. This provides an opportunity to write the entire kernel in a type-safe programming language that is *not* garbage collected. By not relying on a complex garbage collector or other runtime services, such a kernel is simultaneously memory safe *and* gives kernel programmers the degree of memory management control and deterministic behavior they need.

While writing any kernel requires some unsafe code, a primary design goal for secure kernels is to *minimize the amount of unsafe code that must be trusted.* The Tock kernel follows this design goal. Chapter 4 discusses how this goal is leverage to isolate kernel extensions, but broadly, unsafe code falls into two categories: Rust library code, written by language developers, and kernel code, written by kernel developers. The required Rust library code includes only a few low-level operations, such as floating point, bounds checks, and type casts. The kernel trusted code is also extremely small. It includes the standard abstractions of context switches, system call traps, interrupt handling, and memory-mapped I/O, as well as one new abstraction, `MapCell`, that stems from Rust's memory model. A `MapCell` allows kernel code to safely modify complex structures by using inline closures that statically compile with no overhead.

This chapter first describes Rust's memory model and the challenges kernel code introduces to this model [51] (Section 3.1). Then, it describe a minimal set of trusted abstractions that can be used to build a kernel (Section 3.2) and provides a few examples of how such a kernel uses these abstractions to provide common OS features (Section 3.3).

3.1 Rust

Rust was originally motivated by the challenges of writing Firefox's layout engine to be both fast and highly parallel. Since then, it has also been successfully used in large scale projects like Dropbox's back-end storage [59]. It is a particularly attractive language for low-level systems because it preserves type safety (e.g. no memory leaks or buffer overflows) while providing runtime characteristics similar to C's.

Rust offers two important features that make it attractive as a language for writing a secure operating system. First, similar to Modula-3 [19] and Haskell [76], Rust allows the programmer to explicitly separate code which is strictly bound to the type system from code which may circumvent it. Second, it preserves type safety without relying on a runtime garbage collector for memory management. Instead, Rust uses affine types [81] to determine when memory can be freed *at compile-time.*

3.1.1 The `unsafe` Keyword

An operating system must perform certain operations which cannot be modeled in a type system. For example, hardware is often configured through memory mapped I/O registers which must be cast into usable data structures from arbitrary pointers. Rust accommodates this by explicitly separating *trusted* code, which can circumvent the type system in certain ways, from *untrusted* code, which cannot. Specifically, trusted code can use `unsafe` blocks to perform operations that are not type safe (e.g. dereference a raw pointer) or call functions marked unsafe.

The `unsafe` keyword can be used in two ways. First, any block of code can be wrapped in an unsafe block to allow it to perform operations that might break the type system. For example, a hardware abstraction layer can use this feature to expose a memory-mapped I/O register as a normal Rust struct:

```
  let mydevice : &mut IORegs = unsafe {
    &mut *(0x200103F as *mut IORegs)
  }
```

Second, functions can be annotated with `unsafe`, preventing untrusted code from calling them. For example, the standard library's `transmute` function casts its input into any other type of the same size:

```
  pub unsafe fn transmute<T,U>(e: T) -> U
```

Packages compiled with `-F unsafe_code` cannot use the `unsafe` keyword, allowing system builders to isolate trusted code on a per package basis.

3.1.2 Only One Mutable Reference

Most safe languages achieve memory safety by using runtime checks to determine when memory can safely be freed. Rust, instead, avoids the runtime overhead by using the concept of *ownership*, generally referred to as affine types in the literature [81].

Each value in Rust has a unique *owner*—namely, the variable to which it is bound. When the owner of a value goes out of scope, the value is freed. For example:

```
{
  let x = 43;
}
```

Within the scope above, memory for the value `43` is allocated and bound to the variable `x`. When the scope exits and `x` is no longer accessible its memory is reclaimed.

Because there can only be a single owner, aliasing is disallowed. Instead, values are either copied (if the type of the value implements the `Copy` trait) or moved between variables. Once a value is moved, it is no longer accessible from the original variable binding. For example, the following code is not permissible:

```
{
  let x = Foo::new();
  let y = x;
  println("{}", x);
}
```

Because `Foo::new()` has been moved from `x` to `y`, `x` is no longer valid. Similarly, moving a value into a function by passing it as an argument invalidates the original variable. As a result, functions must explicitly hand ownership back to the caller:

```
fn bar(x: Foo) -> Foo {
  // do something useful with `x`
  x  // <- return x
}

let my_x = Foo;
let my_x = bar(my_x);
```

To simplify programming, Rust allows references to a value, called *borrows*, without invalidating the original variable. Borrows are created using an `&` and can be either mutable or immutable. There are two main restrictions on borrows:

1. Borrows cannot outlive the value they borrow. This prevents dangling-pointer bugs.
2. A value can only be mutably borrowed if there are no other borrows of the value.

The first restriction is important to allow the compiler to determine, statically, when to free both heap and stack allocated memory. This renders memory bugs such as double-free and use-after-free impossible. The second restriction is necessary because allowing mutable aliases would allow a program to circumvent the type system [35].

For example, consider Rust's `enum` types which allow multiple distinct types to share the same memory, similar to unions in C. In this example, the `enum` can be either a 32-bit unsigned number, or a mutable reference (pointer) to a 32-bit unsigned number:

```
// Rust
enum NumOrPointer {
  Num(u32),
  Pointer(&mut u32)
}
```

```
// Equivalent C
union NumOrPointer {
  uint32_t Num;
  uint32_t* Pointer;
};
```

Unlike unions in C, a Rust `enum` is type safe. The language ensures that it is impossible to access a `NumOrPointer` as a `Num` when the compiler thinks it is a `Pointer`, and vice-versa.

Having two mutable references to the same memory could violate `NumOrPointer`'s safety and would allow code to construct arbitrary pointers and access any memory. Suppose that the `NumOrPointer` is currently a `Pointer`. If one of the references is to a `Pointer` but the other can change it to a `Num`, then it can create an arbitrary pointer:

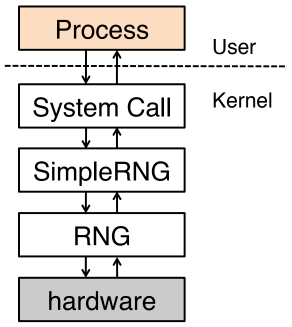

Figure 3.1: Software architecture for a system call interface to a hardware random number generator: both `RNG` and the system call interface need references to `SimpleRng`.

```
// Rust
// n.b. will not compile
let external : &mut NumOrPointer;
match external {
  Pointer(internal) => {
    // This would violate safety and
    // write to memory at 0xdeadbeef
    *external = Num(0xdeadbeef);
    *internal = 12345;  // Kaboom
  },
  ...
}

// Equivalent C
// compiles without warning
union NumOrPointer* external;
uint32_t* numptr = &external->Num;
*numptr = 0xdeadbeef;
*external->Pointer = 12345;
```

3.1.3 Kernels Need Multiple References

Operating system kernels often depend on callbacks and other event-driven programming mechanisms. Often, multiple components must both be able to mutate a shared data structure. Consider, for example, the random number generator software stack as shown in Figure 3.1. `RNG` provides an abstraction of an underlying hardware RNG, such as Intel's RDRAND/RDSEED [39] or a TRNG on an ARM processor [8].

`SimpleRng` sits between `RNG` and the system call layer. It translates between userspace system calls and the `RNG` interface. It calls into `RNG` when a process requests random numbers and calls back to the system call layer when random numbers are ready to deliver to the process. A natural way of structuring this stack is for both the system call layer and `RNG` to have a reference to `SimpleRng`:

```
pub struct SimpleRNG {
  busy: bool,
  ...
}

impl SimpleRng {
  fn command(&mut self) { self.busy = true; ... }
  fn deliver(&mut self, rand: u32) { self.busy = false; ... }
}

impl SysCallDispatcher {
  fn dispatch(&mut self, num: u32) {
    match num {
      // ...
      43 => self.simple_rng.command(),
    }
  }
}

impl RNG {
  fn done(&mut self, rand: u32) {
    self.simple_rng.deliver(rand);
  }
}
```

However, Rust's ownership model does not allow both structures to have a mutable reference to `SimpleRng`. [1] This would seem to imply that it is impossible to model a system this way using Rust. But, as the next section describes, that is not the case.

3.2 Towards a Rust Kernel

Writing an operating system in Rust does not require any changes to the language, but does require trusting two types of unsafe code. The first consists of Rust language mechanisms and libraries, written by the Rust language team. These provide a safe interface, but their underlying implementations use unsafe code. The second type of code which must be trusted are portions of kernel code, written by kernel developers, that use unsafe code in order to implement basic operating system functionality but again can provide safe interfaces.

The rest of this section describes what code falls into these two categories. Together, they constitute the complete set of unsafe code in the kernel and are surprisingly small, primarily consisting of mechanisms that any language or kernel needs to provide.

[1]the reference must be mutable because `command` marks `SimpleRNG` as `busy`, and `deliver_rand` marks it as not `busy`, mutating the internal state of the `SimpleRNG` object

However, an additional abstraction is required to resolve the tension between unique mutability and circular data structures posed by Section 3.1.3: interior mutability. Types with interior mutability are special types that allow a caller to mutate their data despite having only a shared, rather than mutable, reference. Such types are *implemented* using unsafe language features, but expose a safe API because they restrict *how* data can be mutated.

This section describes two such types: `Cell`, provided by the Rust core library, allows multiple references to a mutable object but exposes a limited API that requires copying the data out in order to access it; And *MapCell*, provided by the kernel allows kernel code to safely share complex structures with no runtime overhead (such as copies) and a very simple programming abstraction.

3.2.1 Trusted Rust Code

Rust has a large set of available libraries, including data structures, web browser engines, JavaScript compilers and I/O. A kernel requires only *libcore*, which supports primitive types such as arrays and an interface to compiler (LLVM [46]) intrinsic operations. A kernel wanting to use a generic, existing dynamic memory allocator would require *liballoc* as well.

Both of these libraries contain some trusted code either because they must subvert the type system (memory management requires type casts) or for performance optimizations. Specifically, a kernel relies on the following four Rust abstractions that use unsafe code:

Bounds checks: Arrays are bounds-checked, so unsafe code uses the length field to ensure accesses are safe.

Iterator optimizations: The canonical way to operate across Rust arrays is with iterators, which use unsafe code to avoid unnecessary intermediate checks.

Compiler intrinsics and primitive casts: Floating point, volatile loads/stores and casting between primitive types have architecture specific details that Rust relies on LLVM for.

Cell: An abstraction that encapsulates data such that interior references cannot escape and it can be operated on with an immutable reference.

`Cell` provides a partial solution to the problem of event driven code needing to hold multiple mutable references (Section 3.1.3). A Rust `Cell` is an opaque memory container that code can copy into and out of, but cannot internally reference. The key feature of `Cell` is that an immutable reference can copy into it. The unsafe type cast that arose with enums in Section 3.1.2 cannot happen with `Cell` since multiple referrers operate on separate copies of the shared data.

```rust
pub struct SimpleRng {
  busy: Cell<bool>,
  ...
}
```

```
impl Syscall for SimpleRng {
  fn command(&self) {
    ...
    self.busy.set(true);
  }
}
```

`Cell` can solve the problem in the random number generator example (Figure 3.1). Both the `RNG` and the system call dispatcher hold an immutable reference to the same `SimpleRng`. Normally, this would mean that calls from either would not be able to modify `SimpleRNG`'s internal state. However, as shown below, `Cell` allows the `command` method to set `busy` to be true even though `&self` is an *immutable* reference.

3.2.2 Trusted Kernel Code

Rust core libraries provide safe interfaces to the kernel programmer. Assuming these libraries are correctly implemented, they do not allow callers to violate type safety. However, kernels must do some fundamentally unsafe things (such as context switch), and must wrap these unsafe implementations in safe interfaces. Surprisingly, this requires very little Rust code. The following six pieces of unsafe kernel code need to be trusted:

Context switches: Switching between thread contexts requires saving and restoring the program counter and stack pointer as well as potentially changing process state between protected and unprotected mode.

Memory-mapped I/O and structures: Processors provide I/O through memory-mapped registers, so the kernel needs to be able transform raw memory addresses (e.g., 0x40008000) into typed registers and bit fields, while file systems require casting disk blocks into structures.

Memory allocator: Kernels define specialized memory allocators (e.g., slab [18]) which must type cast raw pointers.

Userspace buffers: Because user space could pass invalidly sized buffers to the kernel, unsafe code must check that buffers are valid.

Interrupt/exception handlers: Handlers can preempt the kernel at arbitrary points and need access to data structures shared with the kernel. Unsafe code must ensure that handlers access these data structures in a way that does not introduce race conditions.

MapCell: An abstraction that allows multiple references, like `Cell`, but without memory copies.

`MapCell` is unique among these in that it is a purely software abstraction designed to allow efficient, safe kernel code. `Cell` works well for primitive types and small values for which the copying semantics do not add any overhead. On these types, `Cell` optimizes down to just loading into a register and storing to memory. `MapCell` is for larger or more complex data objects. Rather than copy values out of a `MapCell`, a program passes code *in*, through a closure. For example, a system call interface which keeps caller/process state through a structure named `App` can access it this way:

```
struct App { /* many variables */ }
app: MapCell<App>

self.app.map(|app| {
  // code can read/write app's variables
});
```

Like Rust's `Cell`, a `MapCell` internally owns the shared data, e.g. a mutable reference, and a `MapCell` can be shared by multiple callers. However, rather than copy values out of a `MapCell`, its API consists of a single method, `map(f)`. Normally, `map(f)` will invoke the closure `f` with a reference to the `MapCell`'s internal data. However, in cases where there already exists a reference to the internal data—such as a recursive call—`map(f)` is a no-op and does not execute the closure.

As a result, `MapCell` is a form of mutual exclusion. Like a mutex, `MapCell` ensures that there is only one mutable reference to the internal value. However, unlike a mutex, it skips the operation instead of blocking, allowing a single-threaded kernel to continue executing even if one component fails to release a lock. Once compiled, `MapCell` is just as fast as unchecked C code. For example, the following snippet of `MapCell` code

```
struct App {
    count: u32,
    tx_callback: Callback,
    rx_callback: Callback,
    app_read: Option<AppSlice<Shared, u8>>,
    app_write: Option<AppSlice<Shared, u8>>,
}
pub struct Driver {
    app: MapCell<App>,
}

driver.app.map(|app| {
    app.count = app.count + 1
});
```

generates the following ARM assembly, which safely checks if `app` is a null pointer, operates on `app.count` only if it is a value, then stores the result:

```
/* Load App address into r1, replace with null */
ldr     r1, [r0, 0]
movs    r2, 0
str     r2, [r0, 0]
/* If MapCell is empty (null) return */
```

```
cmp     r1, 0
it      eq
bx      lr
/* Non-null: increment count */
ldr     r2, [r1, 0]
add     r2, r2, 1
str     r2, [r1, 0]
/* Store App back to MapCell */
str     r1, [r0, 0]
bx      lr
```

Note that, since the closure is scoped to the stack frame, it requires no special allocation.

3.3 Case Studies

This section briefly describes three kernel abstractions in Rust, showing how given the special cases described in Section 3.2, kernel building blocks are naturally expressed in Rust. These cases are taken from our kernel. The kernel's trusted computing base includes the Rust core library as well as under 1000 lines out of over 6000 lines of kernel code.

3.3.1 Direct Memory Access

Direct memory access (DMA) is a common source of memory safety violations in kernels today. Because the hardware will use whatever address it is given, kernel code using DMA can circumvent virtual memory and other protection mechanisms [41].

A Rust-based kernel exposes memory-mapped registers as typed data structures. Exposing them safely in this manner ensures that kernel code cannot write arbitrary values to them. For example, a DMA interface that uses a Rust slice (dynamically sized array)

```rust
struct DMAChannel {
  ...
  enabled: Cell<bool>,
  buffer: MapCell<&'static mut [u8]>,
}
```

enforces that the `buffer` field is a valid pointer to a block of memory. Furthermore, it can use the buffer length to ensure it does not write past the end of the block. For a caller to pass a `&'static [u8]`, it must have been granted access to a statically allocated byte buffer. This interface also highlights an interesting safety concern that Rust enforces. Rust cannot reason about how long the DMA operation will take, but it needs assurances that the buffer will still be live (not freed) when it completes. The only types of memory that can satisfy this are statically (global) allocated buffers and heap buffers, so it requires that the buffer is `'static`. The only unsafe code in the

DMA implementation is the code that memory-maps registers to write the buffer and length into the DMA registers and enables the completion interrupt.

3.3.2 Universal Serial Bus

Universal Serial Bus (USB) uses in-memory *descriptors* specified by the programmer to configure and control USB endpoints. The hardware assumes these descriptors are laid out in a particular way. Representing these hardware memory structures in Rust is straight-forward.

This example below shows a hardware interface that relies on two levels of data structures that require a particular layout and reference integrity. `USBRegisters.in_endpoints` is reference to an array of endpoint descriptors. `InEndpoint` lays out exactly how the processor lays out these descriptors in memory. Finally, the `EpCtl` type defines the set of valid values which are checked at compile-time. It is worth noting that unlike pointers, references cannot be null; an `InEndpoint` can exist if and only if `dma_address` points to a valid `DMADescriptor` and a `USBRegisters` can exist if and only if `in_endpoints` points to a valid array of 16 `InEndpoints`.

```
enum EpCtl {
  ...
  Enable = 1 << 31,
  ClearNak = 1 << 26,
  Stall = 1 << 21
}
struct InEndpoint {
  control: Cell<EpCtl>,
  dma_address: Cell<&'static DMADescriptor>,
  ...
}
struct USBRegisters {
  ...
  // There can be 16 endpoints
  in_endpoints: Cell<&[InEndpoint; 16]>,
  ...
}
```

3.3.3 Complex Data Structures

It is common for kernel components, such as the buffer cache, page tables, and file systems to rely on data structures with circular references like doubly linked-lists or trees. This often requires multiple aliases to the same mutable data, but those aliases can be logical. For example, a buffer cache entry references a disk block, but this is encoded as a device id and sector number, and so does not require managing Rust's ownership semantics. Cases where data structures use bi-directional pointers such as doubly-linked lists can be handled with the same principles used for circular dependencies between kernel components: `Cell` and `MapCell`. For example:

```
struct ListLink<T>(Cell<Option<&T>>);
struct BufferHead {
  state: BufferState,
  next: &ListLink<BufferHead>,
  prev: &ListLink<BufferHead>,
  page: &Page,
  ...
}
```

3.3.4 Multicore

For the sake of simplicity and brevity, this thesis has examined using Rust in a single-threaded setting. Supporting multicore systems requires managing concurrency within the kernel. Rust was originally designed for parallel systems. As a result, it has language mechanisms that allow the programmer to safely manage concurrency. For example, `Sync` specifies that a structure can safely be shared across threads, while `Channels`, a mechanism to pass data across threads, only admit `Sync` types. A multicore Rust kernel would use such mechanisms to maintain safety of shared data across cores.

Chapter 4

Tock Operating System

The process abstraction common to general-purpose computing usually relies on hardware features provided for that purpose. Processor-enforced privilege levels allow the kernel to prevent applications from accessing hardware directly, and the memory management unit (MMU) provides memory protection and address virtualization. Large reservoirs of RAM make it reasonable to allocate many kernel structures on the heap: this improves the system's ability to support dynamic application requirements while using memory efficiently.

Low-power microcontrollers offer only a limited subset of these hardware features. Some recent microcontrollers include simple privilege levels and a memory protection unit (MPU) which programmers can use to configure access control for address regions, but that lacks virtualized addressing. Additionally, restrictive power budgets for embedded applications mean RAM is scarce: many systems have 64 kB or less of expensive SRAM.

Memory isolation and dynamic memory management have clear software-engineering and performance benefits, but software systems for low-power embedded platforms have mostly provided simpler and easier to implement application execution models.

Low-power embedded operating systems often use the same memory regions for applications and the OS. Merging applications with the kernel makes it easy to share pointers between the two and provides efficient procedure call access to low-level functionality. This monolithic approach usually requires compiling and installing or replacing the applications and OS for a device together, as one unit.

Of course, these restricted features make multiprogramming difficult. Without memory isolation, all code must be trusted absolutely and any misbehaving component threatens the entire system. Even if faults are somehow caught, the entanglement of system and application components via shared pointers means there may not be a safe way to shut down only the failed component at runtime.

Embedded devices require long-running and fault-free operation. To achieve this, software for

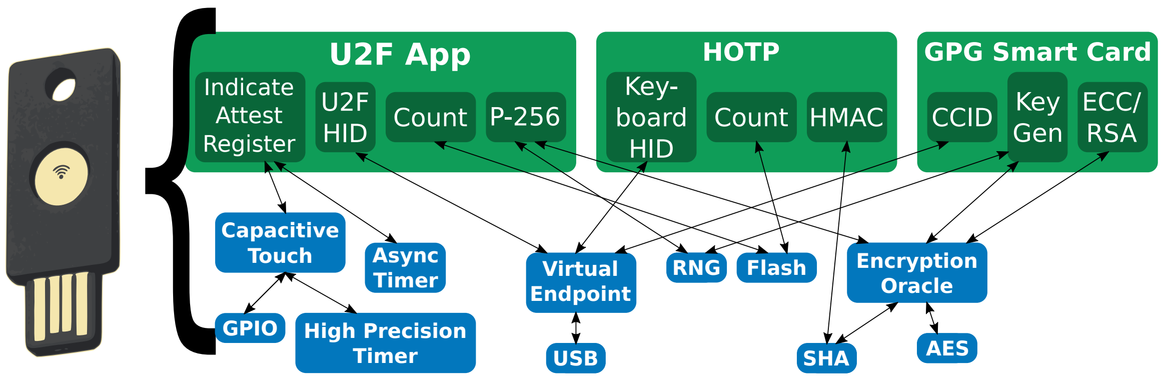

Figure 4.1: A USB authentication device provides a number of related, but independent functions on a single embedded device. Tock is able to enforce this natural division as separate processes that share hardware functionality. An example Tock-based architecture for an authentication key is pictured above. Each application (in green) uses a different combination of common, and often multiplexed, hardware resources exposed by the kernel (in blue).

these platforms usually allocates all memory statically. This avoids hard-to-predict memory exhaustion due to dynamic application behavior. In severely memory-constrained environments, even heap fragmentation poses a significant threat to memory availability.

When memory is statically allocated, system software for managing a shared abstraction like a radio interface must make a static decision about how many concurrent requests it will support, as the kernel must track each request. To support a particular maximum degree of concurrency, the system must pre-allocate memory that may be unused for much of the device's lifetime. This trade-off between concurrency and memory footprint forces developers to guess how to balance resources for optimal performance whenever a system's functional applications are reconfigured.

This thesis presents Tock, a new operating system for low-power embedded platforms that addresses these shortcomings in existing systems to provide a rich multiprogramming environment: it provides fault isolation and allows the kernel to dynamically allocate memory for application requests. The kernel itself is written in Rust, allowing Tock to encapsulate a large fraction of its kernel with granular, type-safe interfaces. Code for these components is trusted only to eventually yield the microcontroller for system liveness. In addition, Tock provides a process abstraction using the hardware isolation mechanisms available on many recent chips. Processes provide complete isolation of memory and CPU resources between applications and the kernel, allowing developers to write applications in C or any other language that targets the hardware.

To avoid trade-offs between memory efficiency and concurrency, Tock allows kernel components to use portions of process memory, called *grants*, to maintain state for the process's requests to kernel services. Grants act as a dynamic kernel heap that is partitioned among processes, so processes cannot starve each other. The kernel can trivially and cheaply reclaim each partition whenever

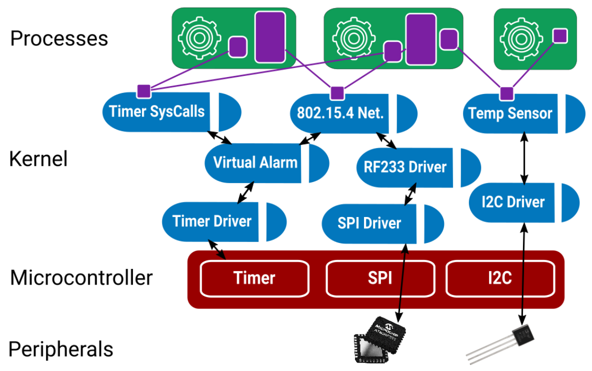

Figure 4.2: Tock system architecture. The kernel, written in Rust, is divided into a trusted core kernel that can use unsafe code, and untrusted capsules. Processes can be written in any language and are isolated from the kernel and each other.

its granting process dies. This approach allows each process to dynamically donate its available memory in order to perform whatever concurrent requests are necessary at a particular moment. It also obviates the need for pre-allocated request structures in the kernel. Although the kernel itself uses only static allocation in order to guarantee continuous operation, this feature simultaneously allows for flexible configuration of applications and efficient use of precious memory.

4.1 Architecture

The Tock architecture has two classes of code: capsules and processes. Each has different goals, is trusted for different properties and is designed for the hardware constraints and application characteristics of embedded systems.

Capsules are units of composition within the kernel. They are constrained by a language sandbox at compile-time and cooperatively scheduled. This scheduling takes advantage of the short operations in the kernel and minimizes context switching overhead.

Processes, in contrast, are similar to processes in other systems: they are scheduled preemptively and memory-isolated by hardware, using system calls to interact with the kernel. Processes may be long-running and can be de-prioritized to conserve energy if needed. The design of both processes and capsules is guided by threat models that favor granular, mutually distrustful components.

4.1.1 Threat Model

Attacker capabilities and system security policies are specific to particular embedded applications. Tock provides the mechanisms required to build a secure system for a variety threat models. Specifically, Tock addresses threats as they relate to four stakeholders: board integrators, kernel component developers, application developers, and end-users. Each is responsible for different parts of a complete system and has different levels of trust in other stakeholders.

Board integrators Integrators combine the Tock kernel with microcontroller-specific glue code, drivers for attached peripherals, and communication-protocol implementations. Board integrators distribute capabilities to kernel components, have complete control over the firmware in the microcontroller, and likely design and build the hardware platform. It is the board integrator's role to determine the end-to-end threat model and structure system components to meet it.

Kernel component developers Kernel developers write most of the kernel functionality, such as peripheral drivers and communication protocols, in capsules. For example, a hardware vendor may supply drivers for a sensor or an open source community may write a networking-protocol stack. Tock's design assumes the source code for kernel components is available for the board integrators to audit before compiling into the kernel. However, it does not assume that auditing will catch all bugs; Tock is able to limit the damage of a misbehaving kernel component. In particular, capsule developers are not trusted to protect the secrecy and integrity of other system components. A capsule may starve the CPU or force a system restart, but it cannot violate other shared-resource restrictions, such as performing unauthorized accesses on peripherals, even if it is authorized to access another peripheral on the same bus. Moreover, capsules are accountable when a fault occurs. Because capsules are isolated, they cannot lead to faults in other capsules by inadvertantly modifying their private state (e.g. due to a buffer overflow).

Application developers Application developers build end-user functionality into processes using the services provided by the kernel. Applications may ship with the hardware platform, or they may be updated after deployment or installed by end users, in which case board integrators cannot generally audit application code. Application developers may be completely unknown before deployment. Therefore, we model applications as malicious: they may attempt to block system progress, to violate the secrecy or integrity of other applications or of the kernel, or to exhaust other shared resources such as memory and communication buses. It is important for a Tock-based system to continue operating in the face of such attacks.

End users Users may install, replace or update applications on a deployed system and may interact with the system's I/O ports in arbitrary ways. They are not assumed to have any particular technical expertise, and may not be able to audit applications before installing them. If a device's construction can prevent the end user from replacing the kernel, then the user need not be trusted to obey security policies attached to sensitive kernel data. For example, a security module on such a device could prevent a master encryption key from leaking to end users.

4.1.2 Capsules

The kernel is built out of components called capsules. Capsules are written in Rust, and restricted from using unsafe features of the language.

A capsule is an instance of a Rust struct, including its fields, associated methods, and accessible static variables (e.g. static variables defined in the same module). The kernel schedules capsules cooperatively. This enables capsules to share a single stack and allows the compiler to eliminate most communication between capsules through inlining. However, it also means that capsules are trusted for system liveness and meeting timing constraints. A capsule could, for instance, interfere with another capsule's ability to receive events by running an expensive computation.

The capsule abstraction provides all but one of the features from Section 2.2.2: it is memory efficient, dependable, supports concurrency, and provides memory fault isolation. However, unlike processes, capsules cannot be loaded at runtime and can exhaust CPU resources.

Capsule Types

There are a number of different kinds of capsules in the kernel that serve different functions and are written by authors with different levels of trust. Understanding these differences, their requirements, and their goals enables policies that ensure the kernel remains safe.

Most capsules are untrusted and cannot subvert the Rust type system. The Rust type and module system ensures that capsules cannot access data in other capsules (i.e. they cannot read/write private fields in other capsules) or process memory. Multiplexing capsules, for example, are written by OS developers or contributed by third parties. These capsules multiplex fixed hardware resources (e.g. timers) to be used by many other capsules. They are purely software constructs and are therefore untrusted. Peripheral drivers for sensors, radios, communication protocols, and other peripherals fall into this category. They are hardware independent since they use hardware-agnostic interfaces for communication buses (e.g. a multiplexed I^2C bus). System-call capsules, which translate between system calls from application processes and internal kernel interfaces of multiplexed abstractions, are also untrusted.

A small number of capsules that must interact directly with hardware are trusted to perform actions outside the Rust type system. This includes low-level abstractions of MCU peripherals that must cast memory mapped registers to type-safe structs. It also includes core kernel capsules, such as the process scheduler, which must manipulate protected CPU registers directly. Because more complex kernel services built from these abstractions can be implemented within the Rust type system, the kernel can maintain the secrecy and integrity of data without having to trust most capsules.

Capsule Isolation

Capsules are isolated from each other using the Rust type and module system. This protects the kernel from buggy or malicious capsules, allows capsules to selectively expose state and methods, and provides a method for abstraction between kernel features.

Since the capsule isolation mechanism is used ubiquitously in the kernel, it is important that it consume minimal memory and have negligible or no computational overhead. Rust enforces type and memory safety at compile-time, so in most cases capsule isolation has no runtime overhead compared to a similar monolithic implementation. For example, a capsule never has to check the validity of a reference, as Rust ensures that all references point to valid memory of the right type. This allows for extremely fine-grained isolation, as there is often no overhead to splitting up components.

Rust's language protection offers strong safety guarantees. An untrusted capsule can only access resources explicitly granted to it, and only in ways permitted by the interfaces those resources expose. For example, direct memory access (DMA) is a common source of kernel memory violations. Because the DMA hardware can manipulate data at any address, kernel code using DMA could circumvent language-level memory protections [41]. To avoid this, chip-specific capsules wrap the DMA memory-mapped registers as a typed data structure that leverages the Rust type system to enforce pointer integrity.

```
struct DMAChannel {
  ...
  enabled: bool,
  buffer: &'static [u8],
}
```

Exposing the DMA base pointer and length as a Rust slice (a bounds-checked array) enforces that the `buffer` field is a pointer to a valid block of memory[1]. Furthermore, it can use the buffer length to ensure it does not write past the end of the block. For a caller to pass a `&'static [u8]`, it must have been granted access to a statically allocated byte buffer. The only code that requires unsafe operations in this DMA implementation is the code that casts the memory-mapped I/O registers to this struct.

Concurrency

The kernel executes capsules cooperatively. The kernel scheduler is event-driven and the entire kernel shares a single stack. Figure 4.3 illustrates the execution model in the kernel. Events are generated from asynchronous hardware interrupts, such as a timer expiring or a physical button being pressed, or from system calls in a running process. Capsules interact with each other directly through function calls or shared state variables.

[1]This particular example works because the hardware DMA interface happens to match the memory layout of Rust's built-in slice. However, Rust's built-in operations are flexible enough to allow the chip-maintainer to write their own type-safe replacement that would match other memory layouts as well.

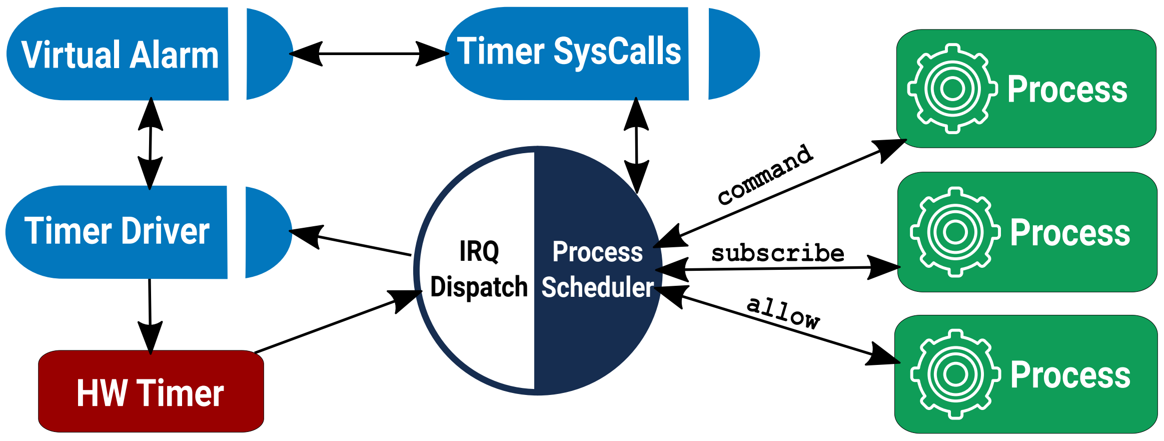

Figure 4.3: The Tock kernel has two sources of events: hardware interrupts and process system calls. The timer driver capsule configures and receives events from a single hardware timer. It dispatches those events to an alarm multiplexing layer which, in turn, delivers appropriate events to a timer system-call driver that enqueues a notification to a process when its timer expires. In the other direction, processes use system calls to configure when to receive timer events.

Capsules cannot generate new events. They interact with the rest of the kernel directly through normal control flow. This has two benefits. First, it reduces overhead since using events would require each interaction between capsules to go through the event scheduler. With simple functions, the interactions compile to a few instructions or are completely inlined away. Second, Tock can statically allocate the event queue since the number of events is known at compile-time. Similar to how TinyOS manages its task queue, this prevents faulty capsules from enqueueing many events, filling the queue, and harming dependability by exhausting the queue resource.

4.1.3 Processes

Tock processes are hardware-isolated concurrent executions of programs, similar to processes in other systems [7]. Each process has a logical region of memory that includes their stack, heap, and static variables, and is independent of the kernel and other processes. Separate stacks allow the kernel to schedule processes preemptively—all kernel events are given higher priority than processes while a round-robin scheduler switches between active processes. Processes interact with the kernel through a system-call interface and with each other using an IPC mechanism. Finally, they can be written in any language, making it convenient to incorporate existing libraries written in other languages such as C.

While similar to processes in systems such as Linux, Tock processes differ in two important ways. First, microcontrollers only support absolute physical addresses so Tock processes never have address spaces larger than their allocated physical memory, nor do they share code through shared

Call	Core/Capsule	Description
`command`	Capsule	Invoke an operation on a capsule
`allow`	Capsule	Give memory for a capsule to use
`subscribe`	Capsule	Register an upcall
`memop`	Core	Increase heap size
`yield`	Core	Block until an upcall completes

Table 4.1: Tock system call interface. `command`, `allow`, and `subscribe` calls are routed to capsules but have an impact on process scheduling. `memop` and `yield` are handled directly by the core kernel.

libraries. Second, the system call API to the kernel, as described in Section 4.1.4, is non-blocking.

Microcontroller memory-protection units provide relatively fine-grained access control. They can set read/write/execute bits on eight memory regions as small as 32 bytes [2]. For example, this allows processes using IPC to directly share memory regions as small as 32 bytes. In principle, the kernel could give processes access to certain memory-mapped I/O registers to enable low-latency direct hardware access. However, for peripherals we have considered so far, such as the Bluetooth Low Energy transceiver of the Nordic nRF51, doing so would would expose side-channel memory access through DMA registers. Finer-grained MPUs or I/O register interfaces designed for safe delegation of resources to untrusted software might eventually make this possible.

Processes provide all five features from Section 2.2.2: they can be loaded and replaced independently; they are concurrent; memory isolation is enforced by hardware; they prevent system resource exhaustion since they have isolated memory regions and are scheduled preemptively; and, as we discuss in Section 4.2, they make efficient use of memory.

4.1.4 System Call Interface

Tock uses a system call interface that is tailored for event-driven systems. Processes interact with the kernel through an extensible interface of five system calls, shown in Table 4.1.

The `command` system call allows processes to make arbitrary requests to capsules by passing word-sized integer arguments. For example, it can be used to configure timers and begin bus transactions. Arguments are passed by value and do not require any special checking by the kernel.

To pass more complicated data, the `allow` system call passes data buffers from processes to capsules. The kernel verifies that the memory, specified by a pointer and length, is within the application's exposed memory bounds and creates a type-safe Rust struct pointing to the array.

[2]Regions 256 bytes or larger can be further subdivided into eight subregions which can be independently enabled/disabled.

Grant<T> Provides access to grant memory of a particular type.	
create()	Reserves an identifier for a new grant used to allocate space for it in process memory.
enter(*proc_id*, *closure*)	Yields the `Owned` value from the specified process to the given closure. Allocates new grant memory if necessary.
each(*closure*)	Iteratively yields the `Owned` value from each process if already allocated. Does not allocate new memory.
Owned<T> A reference to allocated grant memory for a process.	
deref()	Dereferences the value. (sugared in Rust using pointer dereference syntax: `*`)
drop()	Frees allocated space. Automatically called when the `Owned` value goes out of scope.

Table 4.2: API for grants. The interface allows capsules to access dynamically allocated memory on a per-process basis. `Owned` references can only be accessed within a grant, and cannot escape the closure passed to `enter` or `each`. The API ensures that the memory is inaccessible after the process has died or been replaced.

The structure checks that the associated process is alive before each use. This allows processes to explicitly share memory with capsules, for example, to receive network packets.

The `subscribe` system call takes a function pointer and a user-data pointer. The kernel wraps these in an opaque callback structure, which binds the function pointer and user data to a particular process, before passing it to the capsule. The capsule can then invoke this callback which will schedule it in the process's callback queue. For example, a process can request to be notified when a network packet arrives. The network driver capsule will invoke the callback when data is available. The callback structure protects the pointer integrity and checks for process liveness before use.

The `yield` and `memop` system calls invoke the core kernel rather than capsules. `memop` moves the memory break between the heap and grant regions and has similar semantics to `sbrk`.

`yield` blocks process execution until its callback queue is not empty. Callbacks are not serviced until a `yield` is called. Callbacks behave similarly to UNIX signals: the kernel pushes a new frame onto the process's stack and resumes execution at the callback function. When the function completes, execution resumes at the `yield` call.

4.2 Grants

The architecture described in Section 4.1 isolates a dependable kernel with static allocation from processes that can dynamically allocate memory from a heap. However, what happens when the kernel requires dynamic resources to respond to a request from a process? Capsules often need to allocate memory in response to process requests, which cannot be anticipated in advance. For example, a software timer driver must allocate a structure to hold metadata for each new timer any process creates.

Existing techniques for addressing this issue in low resource systems have significant limitations. One technique is to select limits for such resources statically. In the timer example, this would mean setting the maximum number of timers at compile time. If this is set too low, it limits concurrency. If too high, memory is used inefficiently and wasted. Another technique is to use a global kernel heap to dynamically allocate resources. However, this can lead to resource exhaustion, causing unpredictable shortages, and fails to prevent the demands of one process from affecting the capabilities of another. Finally, since process workload is not known until runtime, compile-time counting, as with TinyOS's `uniqueCount`, cannot be used [48].

As previously mentioned, Tock uses a kernel abstraction called *grants* to solve this problem. Grants are separate sections of kernel heap located in each process's memory space along with an API to access them. Unlike normal kernel heap allocation, grant allocations for one process do not affect the kernel's ability to allocate for other processes. While the heap memory is still limited, the rest of the system continues functioning when one process exhausts its grant memory and fails. Moreover, grants guarantee that all resources for a process can be freed immediately and safely if the process dies or is replaced. This is critical since system memory is so limited.

The grant interface leverages the type system to ensure that references created inside a grant cannot escape. As Section 4.3.3 describes, capsules only operate on grant memory through a supplied closure with compile-time enforced Rust lifetimes that guarantee references do not escape the closure. This guarantees that all resources for a process can be freed immediately and safely if the process dies or is replaced.

Table 4.2 summarizes the grant API exposed to capsules. Capsules can create a `Grant` which is the notion of a granted memory section across all processes. This section is conceptual until allocated for a particular process by calling the `enter` method with the process's identifier. In practice, the grant is allocated when a process first makes a call to that capsule. If a process never uses a particular capsule, the grant remains conceptual, requiring no memory space from the process.

Granted memory can be defined as any type. The type can be simple (e.g. an integer) or arbitrarily complex (e.g. a composite data type or a data structure). When accessed, the `Owned` type wraps the memory reference so that it cannot escape the closure and be used without controlled access.

The `enter` method is used to access already allocated memory from a grant. Additionally, it provides an allocator to the closure which can be used to reserve additional memory. This handles both the common case need of dynamic resources on a per process basis, as well as the dynamic needs of a single process (for example requesting multiple timers).

When `enter` is called, it first checks that the supplied process identifier corresponds to a running process. If the process is alive, `enter` executes the supplied closure with a reference to the grant allocated memory. If not, it returns an error. Process identifiers are unique between kernel restarts.

As a consequence of splitting memory across processes, capsules may not use a single data structure to contain all state and must instead iterate across data structures associated with each process. To simplify this, the `each` method provides iterative access to grants, only returning already allocated sections from valid processes.

In order for grants to be safe, the kernel must enforce three properties: allocated memory cannot allow capsules to break the type system; capsules can only be allowed to access grant references while the associated process is alive; and the kernel must be able to reclaim grant memory from a terminated process.

4.2.1 Preserving Type Safety

While grants are physically located within a process's memory space, processes are not allowed to access grant memory. A process that did so could read or write sensitive kernel fields or violate type invariants in capsule data structures. To enforce this, Tock uses the MPU to prevent access by processes. Limiting access to grants from processes allows Tock to preserve Rust's type safety.

4.2.2 Ensuring Liveness

There are two features that allow Tock to ensure capsules can only access grant memory when the associated process is alive. The first is by ensuring that Tock does not run processes in parallel with the kernel. Whatever state (alive or dead) a process was in when the capsule began executing will be the same state it is in when the capsule completes, and the grant will therefore either be valid or invalid for the duration of the capsule's execution.

Second, all accesses to grant memory occur through the limited grant API. Calls to `enter` check the provided process identifier for validity, returning an error if necessary. Calls to `each` only iterate over valid processes. Within the closure, references to grant-allocated values are wrapped in the `Owned` type, which is defined in such a way that these references cannot escape the closure. This ensures that the grant memory cannot be accessed without first being checked. In Section 4.3 we explain how we use Rust's affine type system to enforce these properties.

4.2.3 Grant Region Allocation

The grant region for each process is a dynamically sized heap located within the process's continuous memory. When the kernel loads a process, it allocates a fixed sized block of memory, based on platform configuration or the process's stated requirements. The size of this memory block is the total memory a process may consume between its data segment, stack, heap, and the kernel-controlled grant region.

Process controlled memory—the data segment, stack and heap—is allocated at the bottom of the process memory block and grows upward while the grant region, controlled by the kernel, grows downward from the top of the memory block. A process can expand or contract using `sbrk` and `brk` system calls—for example, Newlib's `malloc` implementation calls `sbrk` to expand the heap. Similarly, grant allocations may expand the grant region downwards.

The process table in the kernel keeps track of the memory break for the currently consumed process memory and grant region. If these memory breaks meet, future allocations, initiated from either the process or grant operations will fail. The caller can either free memory (e.g. by freeing heap memory in the process) or kill the process.

4.2.4 Grant Region Deallocation

Grant memory may be deallocated in two ways. First, when an `Owned` value falls out of scope, the compiler inserts a call to its destructor. `Owned` stores a process identifier alongside the pointer to the value. The destructor uses the process identifier to free the value's memory from the process's grant region.

Second, when a process is terminated, the kernel needs to reclaim its associated memory. Since all accesses from the kernel are made through the grant API, grant space can be freed immediately when the process is terminated without having to wait for a reference count to go to zero or a garbage collector to run. If a capsule tries to `enter` the grant for a non-existent process, it receives an error and knows it can drop any data or requests for that process.

As grant memory is allocated within the same contiguous block of memory as process accessible memory, the kernel deallocates a grant region in the same step as deallocating process memory. In Tock, this means returning the entire process memory block to the processes memory allocator or, if the process will be re-spawned, just resetting metadata fields like the memory break and zeroing the grant region.

This method of reclaiming memory has implications on kernel design. Since the granted memory associated with a process can disappear, it should only store state inherently tied to that process. This precludes the kernel from using a global list of state (e.g. a list of outstanding timers), and instead requires separate per-process lists. When an event occurs (e.g. a hardware timer firing), the capsule has to iterate through the grants for each process in order to service it. In Tock this overhead is acceptable because in practice there is a very limited number of processes (likely fewer

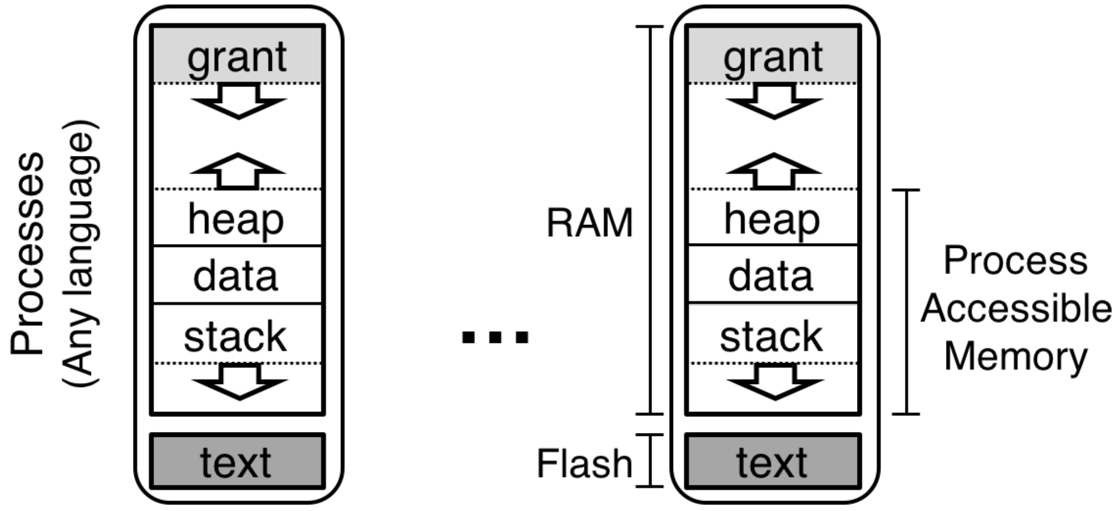

Figure 4.4: Process memory layout. Each process has separate heap, data, stack, and grant regions. Processes are isolated from access the grant region in order to protect kernel state.

than 10 as limited by RAM), and the kernel is I/O bound as capsules cannot perform long-running computation. We discuss optimizations to the implementation of grants in Section 4.3 and evaluate grant overhead in Section 4.4.

4.3 Implementation

We have implemented Tock for ARM Cortex-M based microcontrollers. Our main development platforms are based on the Atmel SAM4L Cortex-M4, which runs at a maximum CPU clock speed of 48 MHz, has 512 kB of flash for code and 64 kB of SRAM. The development platform includes sensors, low-power wireless radios, and a basic user-interface.

The core kernel, which is hardware and platform agnostic, is written in 3554 lines of Rust, as reported by `cloc` [4]. An additional 6824 lines of Rust form the hardware adaptation layer for the SAM4L's hardware peripherals. ARM Cortex-M-specific details such as context switching are 295 lines of (mostly) assembly. The port for our main development platform includes 21 untrusted capsules that implement drivers for the sensors, radios, communication protocols, and hardware multiplexing layers in an additional 12925 lines of Rust code. The Tock kernel on this platform fits in 8.4 kB of SRAM plus an additional 4 kB for the kernel stack. The kernel requires 87 kB of flash. The remaining memory is reserved for processes.

```
impl<T: Default> Grant {
  fn create() -> Grant<T>

  fn enter<F,R>(&self, proc_id: ProcId, func: F) -> Result<R, Error> where
    F: for<'b> FnOnce(&'b mut Owned<T>, &'b mut Allocator) -> R, R: Copy

  fn each<F>(&self, func: F) where
    F: for<'b> Fn(&'b mut Owned<T>)
}
```

Figure 4.5: Rust type signatures for the grant interface. Grants are generic over a type `T` which they allocate space for. `enter` is called with a process ID and a closure that accepts as arguments the grant memory and an allocator. The lifetimes `'b` in the method's signature ensure that the grant memory is only accessible for the duration of the closure. `each` iteratively returns the granted memory from each process, calling the closure repeatedly. Together, these methods allow controlled access to dynamically allocated memory.

4.3.1 MPU Management

Tock uses the ARM Cortex-M's memory protection unit to isolate processes from the kernel and from each other. When context switching into a process, the MPU is configured to allow access to the process's code space in flash, and its data, stack, and heap regions in SRAM, but not the grant region allocated from the process's memory space. When the kernel is executing, the MPU is disabled. The MPU is also used for inter-process communication. The IPC mechanism enables one process to directly share memory blocks with with another process using additional MPU regions.

One difficulty of using the MPU is that MPU regions must be sized to a power of two and must be aligned to an address that is an even multiple of their size. In practice, this means that aligning grant and IPC regions requires additional padding in a process.

4.3.2 Process Memory Layout

Figure 4.4 shows the process memory layout. The process's stack is placed at the bottom of its memory to ensure that any stack overflows trigger an MPU violation. The heap and grant regions grow up and down, respectively, into shared allocation space.

4.3.3 Grants

Figure 4.5 shows the Rust implementation of the grant interface. The type signatures of `enter` and `each` enforce two important properties. First, the lifetimes (`'b`) of the arguments to the closure enforce that they can only be manipulated within the scope of the closure. Their lifetimes are explicitly tied to the life of the closure itself. Second, the closure cannot leak mutable references to grant memory in its return value because return values are copied out of the closure. Specifically, the return type, `R`, must implement the `Copy` trait, and `Owned` does not. Together these properties

```
pub struct GrantData {
  expiration: u32,
  callback: Option<Callback>,
}
impl Syscall for Timer {
  fn subscribe(&self, cb: Callback){
    self.grant.enter(cb.proc_id(), |owned| {
      owned.callback = Some(cb);
    });
  }
  fn command(&self, interval: u32, pid: ProcId){
    self.grant.enter(pid, |owned, _| {
      owned.expiration = self.now() + interval;
      if self.current_alarm > owned.expiration {
        self.set_alarm(owned.expiration);
      }
    });
  }
}
impl HardwareInterface for Timer {
  fn expired(&self, pin_num: u8){
    // timer has expired
    // notify all interested processes
    let now = self.now();
    self.grant.each(|owned| {
      if owned.expiration <= now {
        owned.callback.schedule(...);
      }
    });
    // setup next alarm...
  }
}
```

Figure 4.6: Timer driver demonstrating typical use of grants. Processes can register a callback and request a timer be started. When the hardware timer expires, the capsule notifies the appropriate processes.

ensure that granted memory cannot leak outside of the grant API.

The `Grant` type is implemented with a level of indirection. It contains the unique identifier returned from a call to `create`. This identifier is used to index into a table of allocated grants at the top of the grant region of each process. This table, in turn, is populated with pointers to the allocated memory for that grant. When a grant is accessed for a given process, it indexes to the correct position in the table and passes a pointer to the actual grant memory, wrapped in an `Owned` structure, to the closure.

A memory allocator in the kernel manages blocks in the process's memory pool. Our implementation uses a buddy allocator, however, other allocation strategies are possible and could even be chosen separately for each process.

Case Study: Timer Driver

To demonstrate how grants are used in practice, Figure 4.6 provides a brief example of the Timer driver capsule in Tock. The interface multiplexes a hardware alarm, allowing processes to set virtual timers and receive callbacks when they expire. When a process subscribes, a handle to the callback function is stored in the grant. When the underlying alarm fires, the capsule iterates through the grants for each process, checking if they should be notified. For brevity, some details, including integer wrapping logic and capsule initialization, are elided.

In this example, when an alarm fires, the capsule must iterate through *all* allocated grants to find processes with expired timers.

It is possible to optimize the common case, where previously allocated grants remain accessible, by allocating a combined linked-list. Each link in the list is a timer for a process as well as the process identifier corresponding to the next timer in the list, both allocated in a grant. In the common case, the driver can iterate through the list until it finds a processes with a timer that has not expired. However, when a process fails, entering its grant using the identifier in one of the links in the list will fail, and the driver simply falls back to iterating through all processes.

4.3.4 Case Studies

Tock is open-source and available for anyone to use.[3] A few early adopters in academia and industry are building applications with Tock. To validate the claim that Tock enables embedded hardware platforms with constrained power and memory resources to run third-party processes that are unknown at compile time concurrently and efficiently without memory exhaustion, we consider two such applications: a modular city-scale sensor network platform and a USB security key.

City-Scale Sensing

Signpost [2] is a modular platform for city-scale sensing that provides power, networking, and other resources to sensor modules that attach to it. Rather than predetermining the sensors or applications to deploy, Signpost allows researchers to upgrade or replace sensors and software over time.

Each Signpost comprises a controller that manages module energy allocations and provides time and location resources, a radio module, and a number of independent sensing modules connected to the controller and each other over I^{2}C. All these components run Tock.

The controller and radio, which are built and maintained by the Signpost developers, use multiple processes for logical isolation and development simplicity. The controller performs several independent tasks, while the radio module comprises several communication facilities, running each radio stack in a separate process to ensure failures are isolated.

Sensor modules are developed by research teams wishing to leverage the Signpost platform. Typically, they will extend a basic module schematic (microcontroller, power management, form factor) with peripheral sensors (e.g. audio, RF spectrum analysis, environment sensors) and kernel drivers for those peripherals, written by the Signpost developers or others in the community.

[3]`https://www.tockos.org`

Finally, other researchers may write applications for an existing, already-deployed sensor module. For example, they can use it to validate an audio-event detection algorithm using the already deployed audio module.

At the time of writing, a network of Signposts are deployed on the U.C. Berkeley campus.

USB Security Key

USB authentication keys can provide better security and user experience relative to other second factor authentication options [45]. As a result, many large organizations are deploying security keys internally [28].

A USB authentication key contains a secure element that stores encryption keys and performs cryptographic operations, a simple user interface comprised of an LED and capacitive touch button, and a microcontroller that communicates with a computer over USB. One example, the YubiKey [92], serves several fixed functions: U2F second factor authentication, HMAC-based one-time passwords (HOTP), PGP smart card, and a static password. Other functionality that could be incorporated into such a device includes a Hardware Security Module (HSM), SSH authentication, a password manager, or a bitcoin wallet.

Indeed, a large software organization prototyping an authentication key based on Tock currently uses a JavaCard-based device with custom firmware to implement U2F and SSH authentication. At the core of the prototype is a capsule that has exclusive access to a master symmetric encryption key and acts as an encryption oracle. The remainder of the kernel implements functionality such as USB communication, user-interface drivers, and virtualization layers, as depicted in Figure 4.1.

Using Tock provides several benefits for this application. First, it allows application updates without needing to update the kernel and provides a logical separation between the applications. Second, it enables other developers in the company to build additional applications without risking compromise of the core security applications.

Finally, Tock is able to support the USB authentication key's threat model. While applications will generally be written by other software engineers in the same organization, and are likely not malicious, the core authentication feature is sensitive enough that trusting non-core applications is undesirable. Moreover, limiting access to the master encryption key to an isolated encryption-oracle capsule enables the platform developers to reason carefully about a relatively small amount of code that provides the most important security function of the device.

4.4 Evaluation

Tock explores a new point in the design space of embedded kernels. As a result, quantitative comparisons with existing systems with different design considerations are difficult. Moreover, because

Tock targets application domains unaddressed by previous systems (and does not necessarily subsume previous systems), there are no appropriate benchmark suites to evaluate. Instead, we describe two applications under development that are enabled by Tock. Then, we focus our quantitative evaluation around four main questions:

1. What is the cost of capsule isolation?
2. How do capsules compare to using *only* process isolation?
3. How much memory do grants save compared to alternate solutions?
4. What is the cost of using grants?

All experiments were performed using imix, a Tock development board based on the Atmel SAM4L Cortex-M4. It has three I^2C connected sensors (light, acceleration, and temperature), buttons, LEDs, and Bluetooth Low Energy and IEEE 802.15.4 radios for low power wireless communication. The SAM4L runs at a maximum CPU clock speed of 48 MHz, has 512 kB of flash for code, and 64 kB of SRAM. It has hardware support for a variety of common microcontroller functions, including timers, ADC, I^2C, USART, SPI, and AES encryption. In experiments comparing with TinyOS, we used a port of TinyOS to a predecessor of this platform [5] which has the same microcontroller and similar peripherals.

4.4.1 Capsule Isolation Overhead

Capsule isolation introduces overhead compared to an ideal monolithic implementation. Splitting up a component into multiple capsules requires each capsule to have references to its dependencies. Moreover, in our current implementation, capsules cannot run directly as interrupt service routines (ISRs) so there is some computational overhead associated with re-implementing event-handler routing in software. However, in practice, these overheads are either negligible or incurred anyway in non-isolated systems.

Microbenchmarks

In an optimized monolithic kernel, high-level code such as the user of a peripheral sensor has direct access to the hardware (i.e. memory-mapped registers), and vice-versa (i.e. interrupt handlers). As a result, peripheral-access code can optimize peripheral references to direct, hard-coded memory addresses. In contrast, a small isolated capsule must use references to communicate with other capsules that perform accesses on its behalf.

Figure 4.7 shows the data structure used in a Tock capsule that implements the driver for a light sensor peripheral chip. In addition to driver state, the capsule must also store references to its dependencies—a virtualized I^2C device and a virtual alarm—as well as to its dependents. Each

```
struct Isl29035<'a, AlarmType: time::Alarm + 'a,
                    I2CType: I2CDevice + 'a> {
  i2c: &'a I2CType,
  alarm: &'a AlarmType,
  state: Cell<State>,
  buffer: MapCell<&'static [u8]>,
  client: Cell<Option<&'a AmbientLightClient>>,
}
```

Figure 4.7: The data structure used in the capsule implementing a driver for the ISL29035 light sensor peripheral chip. Capsules use references to "communicate" with other capsules, thus requiring an additional word of memory for each dependency relative to an optimized monolithic implementation: 12 bytes total in this driver for the `i2c`, `alarm`, and `client` references.

reference increases the size of the capsule by a four-byte word, totaling twelve bytes for this particular driver.

While a monolithic kernel may avoid this overhead entirely, in practice, isolation boundaries typically mirror modularity. As a result, as we show in Section 4.4.1, Tock capsules do not consume significantly more memory than comparable systems with no isolation when considering complete systems with applications.

Capsules cannot be invoked directly as interrupt service routines by the hardware. Tock must match the handler routine to the in-memory data structure associated with the handler (the `self` variable). As a result, when a hardware event occurs, such as a timer expiring, Tock takes longer to service.

Table 4.3 compares the latency to service a pin toggle interrupt directly from an ISR, from a capsule in Tock, and from a Tock process. When the CPU is active (i.e. in a busy wait loop), handling an event directly in an ISR is more than twice as fast as servicing it from a capsule. However, the typical state of the CPU is a low-power sleep mode which requires a relatively long wake up period. In this case, the difference between an ISR handler and capsule handler is overshadowed by CPU wake up time.

This implies that certain tasks, such as bit-banging a high-speed communication bus in software, must be implemented in trusted ISR code—without the benefits of capsule isolation. However, most functionality with such high latency sensitivity is typically implemented in hardware on modern microcontrollers.

Macrobenchmarks

To quantify capsule memory overhead, we compare the resource consumption of a capsule-only Tock system with comparable non-isolated embedded systems.

Table 4.4 shows the flash and memory footprint of a kernel-only "blink" application—the "Hello World" of embedded systems—compared to an identical application implemented using TinyOS and

	Handler	**Latency**
No sleep	Interrupt Service routine	0.87 µs
	Tock Capsule	2.03 µs
	Tock Process	36.8 µs
From deep sleep	Interrupt Service routine	2.29 ms
	Tock Capsule	2.29 ms
	Tock Process	2.34 ms

Table 4.3: Latency to handle a hardware event optimally as well as from a Tock capsule and process. Results are shown for an interrupt fired during a busy wait (no sleep) and from the SAM4L's deep sleep state, which requires CPU clocks to boot up and re-stabilize before executing instructions.

System	`text` **(B)**	`data` **(B)**	`bss` **(B)**	**Total RAM (B)**
Tock	3208	812	104	916
TinyOS	5296	0	72	72
FreeRTOS	4848	1080	1904	2984

Table 4.4: Blink application footprint on different embedded operating systems: TinyOS, FreeRTOS, and a Tock implementation using only capsules. Resource consumption is reported as bytes allocated for each of the text, BSS, and data segments in the compiled binary, excluding the stack, which is a tunable parameter in each system. Tock capsules consume comparable resources to existing systems without a similar isolation mechanism.

FreeRTOS. The application toggles an on-board LED once every second then enters a deep-sleep state. Memory overhead for Tock is about 1 kB, nearly 3 kB for FreeRTOS, and under 100 bytes for TinyOS. In all cases, we do not account for the kernel stack, which is a tunable parameter. Tock and FreeRTOS have a larger memory footprint than TinyOS even for a minimal application since they both provide much richer abstractions, such as a preemptive thread scheduler.

The flash footprint is roughly comparable across systems. TinyOS's somewhat larger usage reflects its goal of minimizing memory at the expense of flash usage, in order to match the relatively high flash/RAM ratio on hardware platforms of its time.

This comparison shows that the baseline footprint of a Tock system that uses capsules for isolation is comparable to other embedded systems with no isolation. This is unsurprising since capsules leave no additional runtime artifacts beyond references to other components, which are typically also present in systems without isolation.

We also compare the footprint of a more complete application: an environment sensing application that reports data over an 802.15.4 radio. We used an existing implementation for TinyOS [4] and implemented the application in Tock for the same hardware platform. The application samples periodically from an accelerometer, temperature sensor, and ambient light sensor, then sends a packet containing the readings over a 802.15.4 radio on board. Table 4.5 lists the flash and memory

[4] `https://github.com/SoftwareDefinedBuildings/stormport/tree/rebase0/apps/SensysDemo`

System	`text` (B)	`data` (B)	`bss` (B)	Total RAM (B)
Tock	41744	2824	6880	9704
TinyOS	39604	1228	9232	10460

Table 4.5: Environment sensing application footprint implemented with Tock and TinyOS. The application samples three environment sensors periodically and sends readings over an 802.15.4 radio. Memory reported for both systems include a 4 kB stack.

footprint for each. Tock requires 9 kB of RAM while TinyOS requires 10 kB. Most of the difference in memory consumption is due the TinyOS's more complete network stack which allocates larger static buffers. Importantly, using capsules for isolation does not impose a significant memory overhead in this application.

4.4.2 Capsules vs. Process-only Isolation

Capsules enable isolation at a finer granularity than can be achieved with memory-isolated processes. Capsule isolation requires zero runtime-communication costs and incurs minimal memory overhead (i.e. comparable to architectural separation in monolithic kernels), whereas process isolation suffers both communication and memory overheads.

We evaluate the capsule granularity claim for memory and communication overhead each in turn. As a representative benchmark, we consider the kernel used for Signpost's ambient sensor module, which has 26 capsules, and compare capsule overhead to the same kernel using process isolation.

Memory overhead

From Section 4.4.1, capsule isolation has at most a memory overhead of one word for each other capsule it communicates with. The process abstraction imposes memory overhead per-process, requiring context-switching and state metadata in the kernel as well as dedicated per-process stack memory. In Tock, which is not optimized to support many processes, this overhead is significant. The kernel data structure for process metadata is 164 bytes. Some of this metadata could be discarded (e.g. metadata used for debugging process crashes), but other is essential for context switch performance (e.g. MPU region configurations). Other systems that support threading have smaller process structures. RIOT's [9] minimal process metadata structure is 14 bytes. The minimal process metadata overhead for a system that isolates processes using hardware memory protection lies somewhere between the two.

More significantly, though, pre-emptive processes need separate memory regions to, at least, store their stack whereas cooperatively scheduled capsules share a single stack. Since the stack size must be able to fit the largest depth a process may ever reach, accurately choosing it is difficult and stacks are often over-allocated. Common choices for preemptive embedded system stacks are 256-512 bytes, though many processes must override this default.

To achieve the same degree of isolation granularity as the Signpost ambient sensor using only processes requires at least 13 kB (using 512 byte stacks and RIOT's task structure) and as much as 110 kB (using 4 kB stacks and Tock's current process metadata structure) of RAM, more than the memory available on the SAM4L. In contrast, the capsule-isolated version uses 12 kB, including all static buffers and a shared 4 kB stack.

Communication overhead

Communication overhead for capsules is no more than a pointer indirection or a (often inlined) function call (commonly 0–4 cycles; pessimistically, as many as 25 cycles, or 0.5 µs at 48 MHz). In comparison, communication between processes requires a context switch. A context switch in Tock requires 340 cycles (7 µs at 48 MHz). This limits the kinds of functionality that can be implemented using multiple communicating processes.

4.4.3 Grant Memory Efficiency

Grants allow the kernel to service arbitrary process requests in a memory efficient, highly concurrent manner while maintaining overall system reliability by avoiding a global kernel heap. To our knowledge, no other system provides comparable properties for low-resource hardware. However, it is possible to achieve the most important design goals of Tock—memory isolation and high concurrency—without the grant mechanism, at the expense of memory efficiency. We first examine the memory overhead associated with grants and then compare it to the alternative.

Grants impose memory overhead for both the kernel and for processes. The memory overhead in the kernel is small and fixed at compilation time. For processes, grants impose memory overhead only when requests are being serviced.

The `Grant` type is simply a wrapper around a unique 32-bit grant identifier, thus a reference to a grant in a capsule is only 4 bytes. When a grant is allocated, the kernel adds an entry to a hash table that is stored in the process's grant region that points to the newly allocated memory, imposing an overhead of two words.

The `Owned` type, which wraps references to granted memory (Section 4.2), stores values as a regular pointer in the process's grant memory and an additional word to store the process id. This allows the grant deallocator to know from which process's grant region to deallocate if an `Owned` object falls out of scope. Critically, no memory overhead is imposed on the kernel itself when a process dynamically allocates grant space. A capsule allocates grant memory on demand, using only as much as is needed at that point.

As an example, during a write request, the serial driver allocates two grants, a fixed-size buffer to store metadata about the write (28 bytes), and a dynamically sized buffer to hold the data that will be written. The two `Grant`s impose only a two-byte overhead.

# Threads	Kernel RAM	Syscall RAM	Max Used
1	3506	712	158
2	4216	1422	316
3	4928	2134	474

Table 4.6: TOSThreads has low memory efficiency. Static allocation costs 710-712 bytes per thread, of which at most 158 bytes (22%) can be in use at any time. These numbers do not include the thread stacks, each of which can be less than 100 bytes.

Comparison to Alternatives

To contextualize the memory efficiency of grants, we compare it to the overhead of a modified version of TOSThreads that supports process isolation. TOSThreads, following the TinyOS programming model, uses a static allocation policy. Each multiplexed service is a statically-allocated request queue. Each client (known at compile time) has a single reserved entry in the queue. Each client is therefore assured that it can enqueue one request, and further requests will fail until that operation completes. This functional isolation simplifies thread error handling, as resources are always available for a caller. TinyOS provides no memory isolation: threads and the kernel freely share pointers and overwrite each other's memory. For example, on packet reception the kernel passes a reference to a kernel-allocated packet buffer to a process, which the process can trivially keep, modify, or read beyond.

To show the cost of static allocation when there is isolation, we modified the TOSThreads implementation to copy between processes and the kernel (allocating buffers rather than pointers to buffers in its request queues).

We created a narrow system call interface that samples the six on-board sensors of the TelosB mote [65], sends packets using the CTP collection protocol [34], sends packets over the serial port, and can write to block, log, and configuration flash storage [31]. Table 4.6 shows the code and RAM size of the resulting TinyOS image for 1-3 threads on an MSP43F1612 [60] (the MSP430F1611 has insufficient code space). TinyOS's dead-code elimination means that compiling with zero threads eliminates the entire kernel, while system call interfaces for 4 threads cannot fit in an F1612's RAM.

Each thread requires allocating 710-712 bytes within the kernel for its system calls. The system call to write to configuration storage (small atomic writes to flash) requires the most RAM, 158 bytes, of which 30 bytes is call state and parameters while the data buffer is 128 bytes. Since a TOSThread can only have one outstanding I/O operation, this means at most 22% of a thread's allocated kernel state can be in use at any time and 554 bytes (78%) are wasted.

In contrast, Tock allows concurrency within a process (many operations can be outstanding) and grants allow the system to allocate memory for process requests only as needed. This results in significantly lower memory overhead, with no *wasted* memory.

4.4.4 Algorithmic Overhead

While grants are memory efficient, they require algorithmic changes relative to using a global kernel heap or statically allocating for maximal concurrency. Recall that, as discussed in Section 4.2, from the perspective of a capsule, a grant from a particular process may disappear at any time if the process crashes, restarts, or is replaced. Thus only state inherently tied to that process should be stored in the grant.

Where a traditional driver might use a list of structures each tagged with a process identifier (for example, a list of outstanding timers), with grants this state must be split into separate per-process lists. When an event occurs (such as a hardware timer firing), the capsule must iterate through all processes to find the relevant grant region.

Given enough processes, this algorithmic overhead could prevent the system from meeting timing requirements. However, we argue that this limit is sufficiently permissive and that memory would limit the number of processes first. Moreover, we describe an example mitigation that eliminates the algorithmic overhead in the common case when no processes have failed recently.

Figure 4.8 shows the overhead associated with delivering a timer event as the number of processes with outstanding timers increases. We measure the number of cycles in the timer driver's event handler to enqueue a timer expiration event for the appropriate processes. When only one process has an outstanding timer, the CPU spends 387 cycles (8 µs) in the event handler. Each additional outstanding timer to check adds 44 cycles (< 1 µs). This would allow up to 900 processes to have outstanding timers before exceeding the timer granularity of 1 ms.

Alternatives and Optimizations

Section 4.3 describes a possible optimization to the timer driver: store the process identifier containing the next expected process timer to expire. This allows the timer driver's event handler to enter directly into that process's grant region if the processes is still alive. In addition, each process's grant-allocated timer stores a refernce to the subsequent timer. Because the subsequent timer is also in grant-allocated memory, this is effectively a weak pointer. If any process dies, this chain is broken, and eventually the timer driver must iterate through all grants to fix it. However, in the common case, this optimization allows the timer driver to skip iteration. Applying this optimization in Figure 4.8 reduces time spent in the event handler to a constant 360 cycles.

Finally, we measure the overhead of traversing an optimal, but unsafe, implementation that stores pointers to process-allocated structures (as is the case in existing embedded operating systems). Because storing pointers to data avoids the indirection and liveness checks of grants, this strategy spends only 305 CPU cycles in the event handler—slightly faster than the optimized timer implementation.

The optimized timer demonstrates that grants enable capsule authors to construct efficient (only

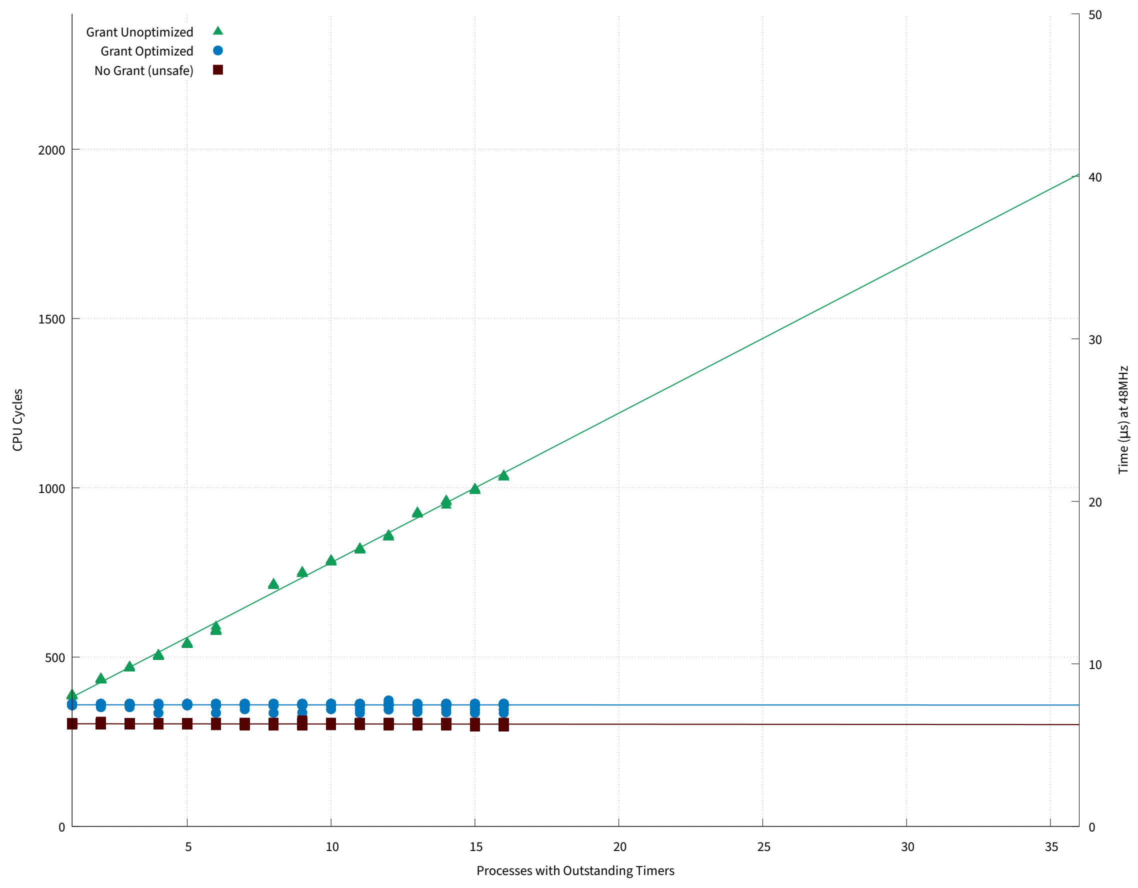

Figure 4.8: The simple implementation requires an event handler to iterate through the grant structure of each process to deliver an event. The graph above shows the overhead in CPU cycles and time of this iteration for a workload of up to 16 processes, projected out to 36 processes. It also shows an optimized version that stores the predicted process separately as well as an unsafe version that uses a combined heap. Full iteration takes an additional 44 cycles for each additional process.

55-cycle overhead) yet safe mechanisms for storing references to numerous, possibly volatile, processes without requiring static allocation of per-process state or an a priori understanding of system load. Without the grant mechanism, capsules could unsafely access process memory (e.g. processes that have been reaped).

Chapter 5

Related Work

Tock draws on a rich ecosystem of embedded operating systems. It is most similar to SOS [37], which also features dynamic loading. Tock uses new hardware facilities and language-based safety to add memory isolation, contributes grants to prevent memory exhaustion, and provides preemptible processes to avoid CPU starvation.

Chapter 2 contrasts the goals and design of Tock with Arduino [10], TinyOS [50], TOSThreads [44], FreeRTOS [12], and RIOT [9]. These were chosen to be representative of a class of systems that includes other research efforts like Contiki [24, 25, 26], TinyThreads [58] and Fibers [86] as well as a variety of industry products such as ARM's mbed [57] and Chromium Embedded Controller [77].

Some non-embedded operating systems use mechanisms that share some characteristics with grants to prevent dynamic allocation of kernel objects from exhausting system memory. Linux cgroups allow the kernel to *charge* dynamic memory allocations to a process namespace [74]. This provides the same flexibility as grants regarding which kinds of objects the kernel may allocate while enabling the system to impose resource limits on process groups. Unlike grants, there are no restrictions on pointers between kernel objects charged to different process groups. This means the Linux kernel does not need to isolate data structures per group, as in Tock, but instead must garbage collect objects when the process group terminates.

The seL4 microkernel, like Tock, avoids dynamic allocation completely in the kernel. Instead, user-level threads can convert their own "untyped" memory into kernel objects for use in system calls [82]. A key difference with grants is that the kernel cannot allocate objects of arbitrary type. In Tock, capsules allocate grants of whatever type they choose directly from process memory. The seL4 microkernel implements only a minimal set of functionality and all kernel objects are specified in the system call API, while most "operating system" functionality is implemented in user-level threads. Conversely, the Tock kernel implements a large and extensible set of functionality that requires a variety of granted types depending on the hardware. Thus, relying on processes to allocate specific kernel objects would be too inflexible.

Previous work has leveraged type-safe languages [36] to build reliable and safe operating systems [52]. Spin [14] allowed applications to extend and optimize kernel performance by downloading modules written in Modula-3 [20]. Spin provides relatively weak isolation between processes, which share a common garbage-collected heap. The Singularity [38] operating system is written in Sing# (a variant of C#) and avoids hardware protection entirely in favor of a software isolated process (SIP) abstraction. Singularity uses threaded SIPs with separate stacks and heaps as the only unit of isolation. It uses linked stacks to mitigate stack over-provisioning, but the minimum stack size is 4 kB, which would allow room for at most 16 processes on our platform. Both systems are inappropriate for memory-constrained embedded devices because Modula-3 and Sing# dynamically allocate most data and use garbage collection for memory management. Moreover, while Modula-3 is defunct (the last release was in 2010) and Sing# is custom-designed for Singularity, Rust is an independent effort with relatively wide adoption.

There has been significant work on using formal methods, instead of type safety, to verify operating systems or their components. For example, FSCQ [21] is a UNIX file-system implementation verified in Coq. seL4 [42] is a verified microkernel. Yang et al. [90] use an automated theorem prover to verify that their C# language runtime correctly enforces type safety. They build an operating system, Verve, using this verified runtime. We view such work as largely complimentary to Tock. For example, similar methods could be used to verify Tock's trusted core kernel while using capsules and processes to isolate unverified drivers and applications.

Finally, both region-based memory management [66, 68, 78, 79] and block-level lock synchronization [63] influenced the design of the grant interface in Tock.

Chapter 6

Conclusion

Language-only isolation techniques can mitigate the performance and granularity issues arising from hardware enforced memory isolation. Moreover, using a type-safe language such as Rust with no garbage collector or other runtime services avoids what would otherwise be one of the largest sections of trusted code base.

For embedded applications like wearables, city-scale sensing, autonomous cars, and personal authentication, resource-constrained computing will continue to be challenging for system designers. Even as computing capability increases, the hardware resources underlying these devices will continue to be constrained in order to lower power, shrink form-factors, and decrease cost. However, the limitations of these systems need not preclude the software abstractions and protections common to general-purpose computers.

Tock is an operating system for resource-constrained systems that provides both dynamic operation and dependability. Tock brings flexible multiprogramming to this tier of computing while isolating processes from the kernel and from each other. To support dynamic demands for kernel resources despite limited system memory, Tock uses a new mechanism, called grants, to split a kernel heap across processes. This allows the system to respond to resource demands from one process without impacting the memory available to other processes or the kernel.

We show how Tock enables multiple system designs whose needs are not adequately met by existing architectures, leading to new capabilities and opportunities for low-power embedded systems.

The lack of isolation or security considerations in many embedded systems has led them to be notoriously vulnerable. As we increasingly connect low-power embedded processors to the physical world, their poor security affects not just our privacy, but also the places we live and work and the things around us. Tock is a first step towards providing a more secure foundation for these increasingly important computers.

Appendix A

System Call Interface

A.1 Overview of System Calls in Tock

System calls are the method used to send information from applications to the kernel. Rather than directly calling a function in the kernel, applications trigger a service call (`svc`) interrupt which causes a context switch to the kernel. The kernel then uses the values in registers and the stack at the time of the interrupt call to determine how to route the system call and which driver function to call with which data values.

Using system calls has three advantages. First, the act of triggering a service call interrupt can be used to change the processor state. Rather than being in unprivileged mode (as applications are run) and limited by the Memory Protection Unit (MPU), after the service call the kernel switches to privileged mode where it has full control of system resources (more detail on ARM processor modes). Second, context switching to the kernel allows it to do other resource handling before returning to the application. This could include running other applications, servicing queued callbacks, or many other activities. Finally, and most importantly, using system calls allows applications to be built independently from the kernel. The entire codebase of the kernel could change, but as long as the system call interface remains identical, applications do not even need to be recompiled to work on the platform. Applications, when separated from the kernel, no longer need to be loaded at the same time as the kernel. They could be uploaded at a later time, modified, and then have a new version uploaded, all without modifying the kernel running on a platform.

A.2 Process State

In Tock, a process can be in one of three states:

- **Running**: Normal operation. A Running process is eligible to be scheduled for execution, although is subject to being paused by Tock to allow interrupt handlers or other processes to run. During normal operation, a process remains in the Running state until it explicitly yields. Callbacks from other kernel operations are not delivered to Running processes (i.e. callbacks do not interrupt processes), rather they are enqueued until the process yields.

- **Yielded**: Suspended operation. A Yielded process will not be scheduled by Tock. Processes often yield while they are waiting for I/O or other operations to complete and have no immediately useful work to do. Whenever the kernel issues a callback to a Yielded process, the process is transitioned to the Running state.
- **Fault**: Erroneous operation. A Fault-ed process will not be scheduled by Tock. Processes enter the Fault state by performing an illegal operation, such as accessing memory outside of their address space.

A.3 Startup

Upon process initialization, a single function call task is added to it's callback queue. The function is determined by the ENTRY point in the process TBF header (typically the `_start` symbol) and is passed the following arguments in registers `r0` - `r3`:

- r0: the base address of the process code
- r1: the base address of the processes allocated memory region
- r2: the total amount of memory in its region
- r3: the current process memory break

A.4 The System Calls

All system calls except Yield (which cannot fail) return an integer return code value to userspace. Negative return codes indicate an error. Values greater than or equal to zero indicate success. Sometimes syscall return values encode useful data, for example in the `gpio` driver, the command for reading the value of a pin returns 0 or 1 based on the status of the pin.

Currently, the following return codes are defined, also available as `#defines` in C from the `tock.h` header (prepended with `TOCK_`):

```
pub enum ReturnCode {
    SuccessWithValue { value: usize }, // Success value must be >= 0
    SUCCESS,
    FAIL, //.......... Generic failure condition
    EBUSY, //......... Underlying system is busy; retry
    EALREADY, //...... The state requested is already set
    EOFF, //.......... The component is powered down
    ERESERVE, //...... Reservation required before use
    EINVAL, //........ An invalid parameter was passed
    ESIZE, //......... Parameter passed was too large
    ECANCEL, //....... Operation cancelled by a call
    ENOMEM, //........ Memory required not available
    ENOSUPPORT, //.... Operation or command is unsupported
    ENODEVICE, //..... Device does not exist
    EUNINSTALLED, //.. Device is not physically installed
    ENOACK, //........ Packet transmission not acknowledged
}
```

A.4.1 0: Yield

Yield transitions the current process from the Running to the Yielded state, and the process will not execute again until another callback re-schedules the process.

If a process has enqueued callbacks waiting to execute when Yield is called, the process immediately re-enters the Running state and the first callback runs.

```
yield()
```

Arguments

None.

Return

None.

A.4.2 1: Subscribe

Subscribe assigns callback functions to be executed in response to various events. A null pointer to a callback disables a previously set callback.

```
subscribe(driver: u32, subscribe_number: u32, callback: u32, userdata: u32)
  -> ReturnCode as u32
```

Arguments

- `driver`: An integer specifying which driver to call.
- `subscribe_number`: An integer index for which function is being subscribed.

- `callback`: A pointer to a callback function to be executed when this event occurs. All callbacks conform to the C-style function signature: `void callback(int arg1, int arg2, int arg3, void* data)`.
- `userdata`: A pointer to a value of any type that will be passed back by the kernel as the last argument to `callback`.

Individual drivers define a mapping for `subscribe_number` to the events that may generate that callback as well as the meaning for each of the `callback` arguments.

Return

- `EINVAL` if the callback pointer is NULL.
- `ENODEVICE` if `driver` does not refer to a valid kernel driver.
- `ENOSUPPORT` if the driver exists but doesn't support the `subscribe_number`.
- Other return codes based on the specific driver.

A.4.3 2: Command

Command instructs the driver to perform a specific action.

```
command(driver: u32, command_number: u32, argument1: u32, argument2: u32) -> ReturnCode as u32
```

Arguments

- `driver`: An integer specifying which driver to call.
- `command_number`: An integer specifying the requested command.
- `argument1`: A command-specific argument.
- `argument2`: A command-specific argument.

The `command_number` tells the driver which command was called from userspace, and the `arguments` are specific to the driver and command number. One example of the argument being used is in the `led` driver, where the command to turn on an LED uses the argument to specify which LED.

One Tock convention with the Command syscall is that command number 0 will always return a value of 0 or greater if the driver is supported by the running kernel. This means that any application can call command number 0 on any driver number to determine if the driver is present and the related functionality is supported. In most cases this command number will return 0, indicating that the driver is present. In other cases, however, the return value can have an additional meaning such as the number of devices present, as is the case in the `led` driver to indicate how many LEDs are present on the board.

Return

- `ENODEVICE` if `driver` does not refer to a valid kernel driver.
- `ENOSUPPORT` if the driver exists but doesn't support the `command_number`.
- Other return codes based on the specific driver.

A.4.4 3: Allow

Allow marks a region of memory as shared between the kernel and application. A null pointer revokes sharing a region.

```
allow(driver: u32, allow_number: u32, pointer: usize, size: u32) -> ReturnCode as u32
```

Arguments

- `driver`: An integer specifying which driver should be granted access.
- `allow_number`: A driver-specific integer specifying the purpose of this buffer.
- `pointer`: A pointer to the start of the buffer in the process memory space.
- `size`: An integer number of bytes specifying the length of the buffer.

Many driver commands require that buffers are Allow-ed before they can execute. A buffer that has been Allow-ed does not need to be Allow-ed to be used again.

As of this writing, most Tock drivers do not provide multiple virtual devices to each application. If one application needs multiple users of a driver (i.e. two libraries on top of I2C), each library will need to re-Allow its buffers before beginning operations.

Return

- `ENODEVICE` if `driver` does not refer to a valid kernel driver.
- `ENOSUPPORT` if the driver exists but doesn't support the `allow_number`.
- `EINVAL` the buffer referred to by `pointer` and `size` lies completely or partially outside of the processes addressable RAM.
- Other return codes based on the specific driver.

A.4.5 4: Memop

Memop expands the memory segment available to the process, allows the process to retrieve pointers to its allocated memory space, provides a mechanism for the process to tell the kernel where its stack and heap start, and other operations involving process memory.

```
memop(op_type: u32, argument: u32) -> [[ VARIES ]] as u32
```

Arguments

- `op_type`: An integer indicating whether this is a `brk` (0), a `sbrk` (1), or another memop call.
- `argument`: The argument to `brk`, `sbrk`, or other call.

Each memop operation is specific and details of each call can be found in the memop syscall documentation.

Return

- Dependent on the particular memop call.

A.5 The Context Switch

Handling a context switch is one of the few pieces of Tock code that is actually architecture dependent and not just chip-specific. The code is located in `lib.rs` within the `arch/` folder under the appropriate architecture. As this code deals with low-level functionality in the processor it is written in assembly wrapped as Rust function calls.

Starting in the kernel before any application has been run but after the process has been created, the kernel calls `switch_to_user`. This code sets up registers for the application, including the PIC base register and the process stack pointer, then triggers a service call interrupt with a call to `svc`. The `svc` handler code automatically determines if the system desired a switch to application or to kernel and sets the processor mode. Finally, the `svc` handler returns, directing the PC to the entry point of the app.

The application runs in unprivileged mode performing whatever its true purpose is until it decides to make a call to the kernel. It calls `svc`. The `svc` handler determines that it should switch to the kernel from an app, sets the processor mode to privileged, and returns. Since the stack has changed to the kernel's stack pointer (rather than the process stack pointer), execution returns to `switch_to_user` immediately after the `svc` that led to the application starting. `switch_to_user` saves registers and returns to the kernel so the system call can be processed.

On the next `switch_to_user` call, the application will resume execution based on the process stack pointer, which points to the instruction after the system call that switched execution to the kernel.

In summary, execution is handled so that the application resumes at the next instruction after a system call is complete and the kernel resumes operation whenever a system call is made.

A.6 How System Calls Connect to Drivers

After a system call is made, Tock routes the call to the appropriate driver.

First, in `sched.rs` the number of the `svc` is matched against the valid syscall types. `yield` and `memop` have special functionality that is handled by the kernel. `command`, `subscribe`, and `allow` are routed to drivers for handling.

To route the `command`, `subscribe`, and `allow` syscalls, each board creates a struct that implements the `Platform` trait. Implementing that trait only requires implementing a `with_driver()` function that takes one argument, the driver number, and returns a reference to the correct driver if it is supported or `None` otherwise. The kernel then calls the appropriate syscall function on that driver with the remaining syscall arguments.

An example board that implements the `Platform` trait looks something like this:

```
struct TestBoard {
    console: &'static Console<'static, usart::USART>,
}

impl Platform for TestBoard {
    fn with_driver<F, R>(&self, driver_num: usize, f: F) -> R
        where F: FnOnce(Option<&kernel::Driver>) -> R
    {

        match driver_num {
            0 => f(Some(self.console)),
            _ => f(None),
        }
    }
}
```

`TestBoard` then supports one driver, the UART console, and maps it to driver number 0. Any `command`, `subscribe`, and `allow` sycalls to driver number 0 will get routed to the console, and all other driver numbers will return `ReturnCode::ENODEVICE`.

Appendix B

Linking & Loading

One key feature of Tock is the ability to load and run multiple applications simultaneously. In a modern computer, the OS uses the Memory Management Unit (MMU) to provide a virtual address space for each process. The code for each process is written assuming that it is the only code in the world and that it can place memory at any address it pleases. The MMU handles translating these virtual addresses into a physical address in real memory, which is shared between all processes. Unfortunately, in the world of embedded systems MMUs are not available. Processors like the ARM Cortex-M series omit them since they are power and area-hungry.

A common approach to handle this issue is to assign addresses to applications at compile-time. For example, Application A can be placed at address 0x21000 and Application B can be placed at address 0x22000. This runs into problems when Application A grows in size in the future. Moreover, with Tock we want to be able to handle dynamically adding, updating, and removing applications at run time. Assigning each an address in advance simply isn't possible.

In Tock, we use position independent code (PIC) [1] to enable loading multiple applications. In PIC, all branches and jumps are PC-relative rather than absolute, allowing code to be placed at any address. All references to the data section are indirected through the Global Offset Table (GOT) [2]. Rather than access data at an absolute address, first the address of the data is loaded from a hard-coded offset into the GOT, and then the data is accessed at that address. This allows the OS to simply relocate all addresses in the GOT at load time based on the actual location of SRAM rather than fixing various instructions throughout the code. The address of the GOT itself is stored in a PIC base register which is set by the OS before switching to application code and is different for each application. The ARM instruction set is optimized for PIC operation, allowing most code to execute with little to no cost in number of instructions.

While PIC handles the majority of addressing issues, it does not fix everything. Data members

[1] Position Independent Code (PIC) in shared libraries
[2] GOT in Tock Applications - August 2016

which are themselves pointers are assigned a value by the compiler rather than indirecting through the GOT. For example, take the statement:

```
const char* str = "Hello";
```

The address of `"Hello"` is stored in the data section (as the value of `str`) and is not relocated like elements of the GOT. In order to solve this issue, we collect the addresses of necessary relocations (such as the address of `"Hello"`) from the ELF and append them to the binary. Tock can then fix each address at load time based on the actual location of the application's text and data segments. Since the application is already compiled as PIC, remaining fixes will only exist in the data segment, making the process of applying relocations simple.

Combining these two solutions together, we reach the Tock application format. Each application binary is compiled as position independent code [3], has a relocation section appended to it [4], and begins with a header structure containing the size and location of the text, data, GOT, and relocation segments as well as an entry point for the app [5]. The app is able to be loaded into any flash and SRAM addresses with no control-flow costs, the recurring cost of additional load instructions when indirecting data accesses through the GOT, and the one-time cost of several simple data address relocations at application load time.

When receiving an application binary, Tock assigns space in flash and SRAM for it, loads the data segment into SRAM, fixes up addresses stored in the GOT [6], walks the relocation section fixing up additional items in the data section [7], sets the process PC to the entry point of the application, and enqueues the newly created processes to be run. Applications can be received through many methods including wireless uploads over protocols such as IEEE 802.15.4 and Bluetooth Low Energy, but the currently implemented system for Tock receives application binaries over a UART serial connection.

[3]GCC Compiler Flags for Tock Applications - August 2016
[4]ELF to Tock Binary - August 2016
[5]Tock Application Header - August 2016
[6]Tock GOT Fixup - August 2016
[7]Tock Relocations Fixup - August 2016

Bibliography

[1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the USENIX 1986 Summer Conference* (July 1986), 93–112. 3

[2] Adkins, Joshua and Campbell, Bradford and Ghena, Branden and Jackson, Neal and Pannuto, Pat and Dutta, Prabal. The Signpost Network: Demo Abstract. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM* (New York, NY, USA, 2016), SenSys '16, ACM, pp. 320–321. 6, 7, 37

[3] Adobe Corporate Communications. Flash & The Future of Interactive Content. `https://theblog.adobe.com/adobe-flash-update/`, 2017. 6

[4] Al Danial. cloc. `http://cloc.sourceforge.net`. Accessed 24-August-2017. 34

[5] Andersen, M. P., Fierro, G., and Culler, D. E. System Design for a Synergistic, Low Power Mote/BLE Embedded Platform. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)* (2016), IEEE, pp. 1–12. 7, 39

[6] Anderson, T., and Dahlin, M. *Operating Systems: Principles and Practice*, 2nd ed. Recursive Books LLC, 2014, ch. 1.1, p. 4. 3

[7] Anderson, T., and Dahlin, M. *Operating Systems: Principles and Practice*, 2nd ed. Recursive Books LLC, 2014, ch. 2.1, pp. 43–44. 28

[8] Atmel. *ARM SAM4L Low Power MCU*, 3 2014. 4203G. 14

[9] Baccelli, E., Hahm, O., Wähilsch, M., Günes, M., and Schmidt, T. C. RIOT: One OS to Rule them All in IoT. Tech. rep., INRIA, Dec 2012. Research Report, No. RR–8176. 8, 42, 48

[10] Banzi, M., Cuartielles, D., Igoe, T., Martino, G., Mellis, D., et al. Arduino. `https://www.arduino.cc/`. Accessed 09-May-2016. 8, 9, 48

[11] Barr, T. W., Smith, R., and Rixner, S. Design and Implementation of an Embedded Python Run-time System. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC'12, USENIX Association, pp. 27–27. 9

[12] Barry, R., et al. FreeRTOS. `http://www.freertos.org/`. Accessed 09-July-2016. 8, 48

[13] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 49–65. 4

[14] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 267–283. 3, 5, 49

[15] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility safety and performance in the SPIN operating system. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29, ACM, pp. 267–283. 10

[16] Boese, Elizabeth, S. *Java Applets: Interactive Programming*, 2nd ed. Lulu.com, 2007. 6

[17] Bomberger, A. C., Frantz, W. S., Hardy, A. C., Hardy, N., Landau, C. R., and Shapiro, J. S. The KeyKOS Nanokernel Architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures* (Berkeley, CA, USA, 1992), USENIX Association, pp. 95–112. 3

[18] Bonwick, J. The Slab Allocator: An Object-caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Berkeley, CA, USA, 1994), USTC'94, USENIX Association, pp. 6–6. 17

[19] Cardelli, L., Donahue, J., Jordan, M., Kalsow, B., and Nelson, G. The Modula–Type System. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1989), POPL '89, ACM, pp. 202–212. 11

[20] Cardelli, L., Donahue, J., Jordan, M., Kalsow, B., and Nelson, G. The Modula–3 Type System. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1989), POPL '89, ACM, pp. 202–212. 10, 49

[21] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th*

Symposium on Operating Systems Principles (New York, NY, USA, 2015), SOSP '15, ACM, pp. 18–37. 49

[22] Chromium Blog. Goodbye PNaCl, Hello WebAssembly! `https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html`, 2017. 5

[23] Coggins, J., McDonald, A., Plank, G., Pannell, M., Ward, R., and Parsons, S. Snow web 2.0: The next generation of antarctic meteorological monitoring systems? In *EGU General Assembly Conference Abstracts* (2012), vol. 14, p. 591. 6, 8

[24] Dunkels, A., et al. Contiki Mulithreading. `https://github.com/contiki-os/contiki/wiki/Multithreading`. Accessed 09-May-2016. 48

[25] Dunkels, A., Gronvall, B., and Voigt, T. Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks* (Washington, DC, USA, 2004), LCN '04, IEEE Computer Society, pp. 455–462. 8, 48

[26] Dunkels, A., Schmidt, O., Voigt, T., and Ali, M. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2006), SenSys '06, ACM, pp. 29–42. Updated documentation: `http://contiki.sourceforge.net/docs/2.6/a01802.html`. 48

[27] Engler, D. R., Kaashoek, M. F., and O'Toole, Jr., J. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 251–266. 3

[28] FIDO Alliance. FIDO Certified Showcase. `https://fidoalliance.org/fido-certified-showcase/`, April 2017. 38

[29] Fitbit. FitBit: Official Site for Activity Trackers and More, 2017. Accessed: 04-20-2017. 6

[30] Garmin. vìvoactive 3. `https://buy.garmin.com/en-US/US/p/571520`, September 2017. 6

[31] Gay, D., and Hui, J. TEP 103: Permanent Data Storage (Flash). `http://www.tinyos.net/tinyos-2.x/doc/txt/tep103.txt`, 2007. 44

[32] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2003). 9

[33] Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K. J., Uhlig, V., Tidswell, J. E., Deller, L., and Reuther, L. The SawMill Multiserver Approach. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System* (New York, NY, USA, 2000), EW 9, ACM, pp. 109–114. 3

[34] Gnawali, O., Fonseca, R., Jamieson, K., Moss, D., and Levis, P. Collection Tree Protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2009), SenSys '09, ACM, pp. 1–14. 44

[35] Grossman, D. Existential Types for Imperative Languages. In *Proceedings of the 11th European Symposium on Programming Languages and Systems* (London, UK, UK, 2002), ESOP '02, Springer-Verlag, pp. 21–35. 13

[36] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. Region-based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 282–293. 49

[37] Han, C.-C., Kumar, R., Shea, R., Kohler, E., and Srivastava, M. A Dynamic Operating System for Sensor Nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2005), MobiSys '05, ACM, pp. 163–176. 8, 9, 48

[38] Hunt, G. C., and Larus, J. R. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review 41*, 2 (April 2007), 37–49. 3, 5, 10, 49

[39] Intel. *Intel® Digital Random Number Generator Software Implementation Guide*, 8 2012. Rev. 1.1. 14

[40] Jackson, D. WebGL 2.0 Specification. Khronos working draft, Khronos, July 2017. https://www.khronos.org/registry/webgl/specs/latest/2.0/. 6

[41] King, S. T., Chen, P. M., Wang, Y.-M., Verbowski, C., Wang, H. J., and Lorch, J. R. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2006), SP '06, IEEE Computer Society, pp. 314–327. 19, 27

[42] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 207–220. 4, 10, 49

[43] Klues, K., Handziski, V., Lu, C., Wolisz, A., Culler, D., Gay, D., and Levis, P. Integrating Concurrency Control and Energy Management in Device Drivers. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 251–264. 8

[44] Klues, K., Liang, C.-J. M., Paek, J., Musăloiu-E, R., Levis, P., Terzis, A., and Govindan, R. TOSThreads: Thread-safe and Non-invasive Preemption in TinyOS. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2009), SenSys '09, ACM, pp. 127–140. 8, 48

[45] Lang, J., Czeskis, A., Balfanz, D., and Schilder, M. Security Keys: Practical Cryptographic Second Factors for the Modern Web. In *Financial Cryptography* (2016). 38

[46] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–. 16

[47] Lédeczi, A., Nádas, A., Völgyesi, P., Balogh, G., Kusy, B., Sallai, J., Pap, G., Dóra, S., Molnár, K., Maróti, M., and Simon, G. Countersniper System for Urban Warfare. *ACM Trans. Sen. Netw. 1*, 2 (Nov. 2005), 153–177. 6

[48] Levis, P. Experiences from a Decade of TinyOS Development. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)* (October 2012). 31

[49] Levis, P., and Culler, D. MatÉ: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ASPLOS X, ACM, pp. 85–95. 9

[50] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., and Culler, D. *Ambient Intelligence.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, ch. TinyOS: An Operating System for Sensor Networks, pp. 115–148. 8, 48

[51] Levy, A., Andersen, M. P., Campbell, B., Culler, D., Dutta, P., Ghena, B., Levis, P., and Pannuto, P. Ownership is Theft: Experiences Building an Embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems* (New York, NY, USA, 2015), PLOS '15, ACM, pp. 21–26. 11

[52] Levy, A., Andersen, M. P., Campbell, B., Culler, D., Dutta, P., Ghena, B., Levis, P., and Pannuto, P. Ownership is Theft: Experiences Building an Embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems* (New York, NY, USA, 2015), PLOS '15, ACM, pp. 21–26. 49

[53] Liskov, B. H. The Design of the Venus Operating System. In *Proceedings of the Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1971), SOSP '71, ACM, pp. 11–. 3

[54] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 49–64. 10

[55] Maas, M., Asanović, K., Harris, T., and Kubiatowicz, J. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 457–471. 6

[56] Matsakis, N. D., and Klock, II, F. S. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (New York, NY, USA, 2014), HILT '14, ACM, pp. 103–104. 10

[57] mbed. mbed OS 5. `https://developer.mbed.org/`, 2017. Accessed: 04-20-2017. 48

[58] McCartney, W. P. *Simplifying Concurrent Programming in Sensornets with Threading.* PhD thesis, Cleveland State University, 2006. 48

[59] Metz, C. The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire. `https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/`. 11

[60] MSP430 ultra-low-power Microcontrollers. `http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/overview.page`. 7, 44

[61] Myers, A. C., and Liskov, B. Protecting Privacy Using the Decentralized Label Model. *ACM Trans. Softw. Eng. Methodol. 9*, 4 (Oct. 2000), 410–442. 5

[62] Nest Labs. Meet the Nest Protect smoke and carbon monoxide alarm. `https://nest.com/smoke-co-alarm/meet-nest-protect/`, 2017. 6

[63] Oracle Java Documentation. Intrinsic Locks and Synchronization. `https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html`, 2017. Accessed: 04-20-2017. 49

[64] Peter, S., Li, J., Zhang, I., Ports, D. R., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS) 33*, 4 (2016), 11. 4

[65] Polastre, J., Szewczyk, R., and Culler, D. Telos: Enabling Ultra-low Power Wireless Research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks* (Piscataway, NJ, USA, 2005), IPSN '05, IEEE Press. 7, 44

[66] PostgreSQL 9.6.2 Documentation. Memory Management. `https://www.postgresql.org/docs/current/static/spi-memory.html`, 2017. Accessed: 04-20-2017. 49

[67] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., and Purcell, S. C. Pilot: An Operating System for a Personal Computer (Summary). In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1979), SOSP '79, ACM, pp. 106–107. 5

[68] Ross, D. T. The AED Free Storage Package. *Commun. ACM 10*, 8 (Aug. 1967), 481–492. 49

[69] Rossberg, A. WebAssembly Core Specification. W3C working draft, W3C, Feb. 2018. https://www.w3.org/TR/2018/WD-wasm-core-1-20180215/. 6

[70] SAM4L ARM Cortex-M4 Microcontrollers . `http://www.atmel.com/products/microcontrollers/arm/sam4l.aspx`. 7

[71] Stefan, D., Russo, A., Mitchell, J. C., and Mazières, D. Flexible Dynamic Information Flow Control in Haskell. *SIGPLAN Not. 46*, 12 (Sept. 2011), 95–106. 5

[72] Suunto. Ambit3 Sport. `http://www.suunto.com/en-US/Products/Sports-Watches/Suunto-Ambit3-Sport/Suunto-Ambit3-Sport-White/`, September 2017. 6

[73] Swift, M. M., Martin, S., Levy, H. M., and Eggers, S. J. Nooks: An Architecture for Reliable Device Drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (New York, NY, USA, 2002), EW 10, ACM, pp. 102–107. 10

[74] Tejun Heo. Control Group v2. `https://www.kernel.org/doc/Documentation/cgroup-v2.txt`, 2015. 48

[75] Terei, D., Marlow, S., Peyton Jones, S., and Mazières, D. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium* (New York, NY, USA, 2012), Haskell '12, ACM, pp. 137–148. 5

[76] Terei, D., Marlow, S., Peyton Jones, S., and Mazières, D. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium* (New York, NY, USA, 2012), Haskell '12, ACM, pp. 137–148. 11

[77] The Chromium Project. Chromium Embedded Controller (EC) Development. `https://www.chromium.org/chromium-os/ec-development`, 2017. Accessed: 04-20-2017. 48

[78] Tofte, M., and Birkedal, L. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst. 20*, 4 (July 1998), 724–767. 49

[79] Tofte, M., Birkedal, L., Elsman, M., and Hallenberg, N. A Retrospective on Region-Based Memory Management. *Higher Order Symbol. Comput. 17*, 3 (Sept. 2004), 245–265. 49

[80] Tolle, G., Polastre, J., Szewczyk, R., Culler, D. a., Turner, N., Tu, K., Burgess, S., Dawson, T., Bu onadonna, P., Gay, D., and Hong, W. A Macroscope in the Redwoods. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2005), SenSys '05, ACM, pp. 51–63. 6

[81] Tov, J. A., and Pucella, R. Practical Affine Types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 447–458. 11, 12

[82] Trustworthy Systems Team, Data61. *seL4 Reference Manual Version 7.0.0*, Sept. 2017. `https://sel4.systems/Info/Docs/seL4-manual-7.0.0.pdf`. 48

[83] VMWare. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. `https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html`, 2008. 5

[84] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 203–216. 4

[85] Watson, M. Web Cryptography API. W3C recommendation, W3C, Jan. 2017. https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/. 6

[86] Welsh, M., and Mainland, G. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (Berkeley, CA, USA, 2004), NSDI'04, USENIX Association, pp. 3–3. 48

[87] Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J. o., and Welsh, M. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 381–396. 6

[88] Wetherall, D. Active Network Vision and Reality: Lessions from a Capsule-based System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1999), SOSP '99, ACM, pp. 64–79. 5

[89] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: The Kernel of a Multiprocessor Operating System. *Commun. ACM 17*, 6 (June 1974), 337–345. 3

[90] Yang, J., and Hawblitzel, C. Safe to the Last Instruction: Automated Verification of a Type-safe Operating System. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 99–110. 49

[91] Yee, B., Sehr, D., Dardyk, G., Chen, B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (Oakland'09)* (IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016, 2009). 5, 6

[92] Yubico. Yubikey Hardware. `https://www.yubico.com/products/yubikey-hardware/`. 38

[93] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making Information Flow Explicit in HiStar. *Commun. ACM 54*, 11 (Nov. 2011), 93–101. 3