Multiprogramming a 64 kB Computer Safely and Efficiently

Amit Levy
levya@cs.stanford.edu
Stanford University

Bradford Campbell
bradjc@virginia.edu
University of Virginia

Branden Ghena
brghena@berkeley.edu
University of California, Berkeley

Daniel B. Giffin
dbg@scs.stanford.edu
Stanford University

Pat Pannuto
ppannuto@berkeley.edu
University of California, Berkeley

Prabal Dutta
prabal@berkeley.edu
University of California, Berkeley

Philip Levis
pal@cs.stanford.edu
Stanford University

ABSTRACT

Low-power microcontrollers 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, an emerging class of embedded applications are software platforms, rather than single purpose devices, and need these multiprogramming features. Tock, a new operating system for low-power platforms, takes advantage of limited hardware-protection mechanisms as well as the type-safety 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.
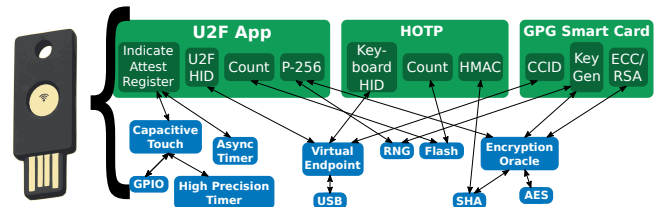
Figure 1: A USB authentication device provides a number of related, but independent functions on a single embedded device. Tock is able to enforce this natural division as separate processes that share hardware functionality. An example Tock-based architecture for an authentication key is pictured above. Each application (in green) uses a different combination of common, and often multiplexed, hardware resources exposed by the kernel (in blue).

1 INTRODUCTION

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 these platforms usually allocate 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 paper 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 [36], a type-safe language whose memory efficiency and performance is close to C. Rust allows 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 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.

2 BACKGROUND & MOTIVATION

Historically, embedded applications have been designed to solve a specific problem: collecting environmental data [12, 51, 54], localizing a sniper [30], recording fitness data [17], or detecting household fires [40]. 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 [18, 46]; USB authentication devices need to isolate multiple services from each other for security reasons (Figure 1); and city sensing infrastructure can run multiple applications written by different stakeholders [1].

Unfortunately, current operating systems cannot meet the requirements of these applications given the resource limitations of embedded microcontrollers.

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 1 shows the clock speed, RAM, and flash memory of two embedded research platforms, the TelosB mote (2004) [42] used in a decade of sensor network research, and Signpost (2017) [1], a recent platform

	TelosB [42] (2004)	Signpost [1] (2017)
MCU	MSP430F1611 [39]	ATSAM4LC8CA [45]
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 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.

System	Concurrency	Memory Efficiency	Dependability	Fault Isolation	Loadable Applications
Arduino [6]		✓			
RIOT OS [5]		✓			
Contiki [14]	✓	✓			✓
FreeRTOS [8]	✓		✓		
TinyOS [33]	✓	✓	✓		
TOSThreads [28]	✓		✓		✓
SOS [23]	✓	✓			✓
Tock	✓	✓	✓	✓	✓

Table 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.

for city-scale sensing which is representative of other recent platforms [3]. 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.

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 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 [12] and cannot 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 [33]) or restricting dynamic allocation to boot time (FreeRTOS [8]) 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 [27]. As a result, most embedded operating systems allow many operations to occur in parallel. Systems such as TinyOS and SOS [23] 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 [28], FreeRTOS [8], and RIOT OS [5] use preemptive threads to run some or all code.

Efficiency. As discussed in Section 2.1, 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 [20], while Arduino [6] 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.
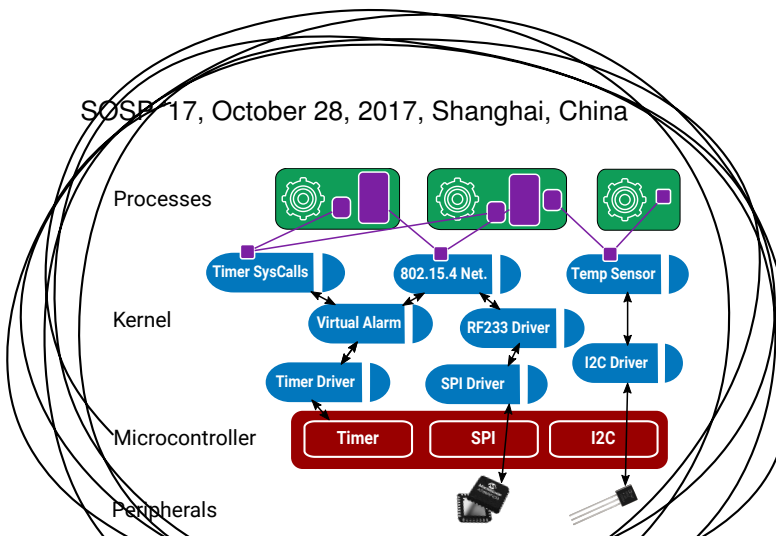
Figure 2: Tock system architecture. The kernel, written in Rust, is divided into a trusted core kernel that can use unsafe code, and untrusted capsules. Processes can be written in any language and are isolated from the kernel and each other.

However, this is inefficient: in typical use cases, 80% of this RAM is wasted (Section 6.4).

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 [23]. Some existing systems run applications in a bytecode interpreter [7, 32] 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.

3 TOCK 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 are guided by threat models that favors granular, mutually distrustful components.

3.1 Threat Model

Tock does not aim to address any specific threat model, as attacker capabilities and system security policies are specific to particular embedded applications. Instead, it provides the mechanisms required to build a secure system. 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, and 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.

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. Thus board integrators cannot generally audit application code. Even the 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.

3.2   Capsules

The kernel is built out of components called capsules. Capsules are written in Rust [36], which is an attractive language for low-level systems because it preserves memory-safety (e.g. no double frees or buffer overflows) and type-safety while providing performance close to C's [35].

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 capsule boundaries 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: 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.

3.2.1   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 I²C 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.

3.2.2   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 [25]. 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

---

[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.
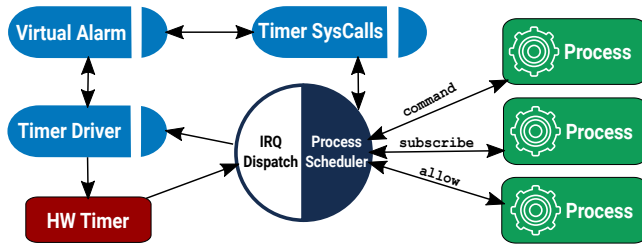
Figure 3: The Tock kernel has two sources of events: hardware interrupts and process system calls. The timer driver capsule configures and receives events from a single hardware timer. It dispatches those events to an alarm multiplexing layer which, in turn delivers appropriate events to a timer system 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.

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.

3.2.3 Concurrency. The kernel executes capsules cooperatively. The kernel scheduler is event-driven and the entire kernel shares a single stack. Figure 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.

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.

3.3 Processes

Tock processes are hardware-isolated concurrent executions of programs, similar to processes in other systems [4]. They have 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.

While similar to processes in systems such as Linux, Tock processes differ in two important ways. First, because microcontrollers only support absolute physical addresses, Tock does not provide the illusion of infinite memory through virtual memory nor do processes share code through shared libraries. Second, the system call API to the kernel, as described in Section 3.4, is non-blocking.

Processes have two main advantages over capsules. First, because they are hardware-isolated rather than sandboxed by a type-system, they can be written in any language. As a result, they make it convenient to work with and incorporate existing libraries written in other languages, like C. Second, they are preemptively scheduled, so they can safely execute long-running computations such as encryption or signal processing.

Microcontroller memory-protection units provide a relatively high granularity of 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, processes could be given access to certain memory-mapped I/O registers by the kernel 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, it is not possible to do so without exposing side-channel memory access through DMA registers. Finer-grained MPUs or I/O register interfaces designed for this functionality might eventually make this possible.

Processes provide all five features from Section 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, they make efficient use of memory.

3.4 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 3.

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

---

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

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 3: 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.

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: 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.

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. 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 GRANTS

The architecture described in Section 3 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 [31].

Tock solves this problem with a kernel abstraction called *grants*. 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 5.3 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 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. 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 also 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).

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.

4.1 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 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 5 we explain how we use Rust's affine type system to enforce these properties.

4.3 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.

4.4 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.
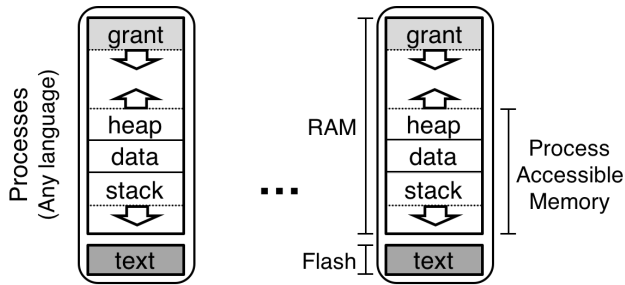
Figure 4: Process memory layout. Each process has separate heap, data, stack, and grant regions. Processes are isolated from access the grant region in order to protect kernel state.

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 then 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 5 and evaluate grant overhead in Section 6.

5 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` [2]. 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.

5.1 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.

5.2 Process Memory Layout

Figure 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.

5.3 Grants

Figure 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 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

```rust
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 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.

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.

5.3.1   Case Study: Timer Driver. To demonstrate how grants are used in practice, Figure 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. Using this optimization the driver can iterate through the list until it finds a processes with a timer that has not expired. At any point, entering a grant in this manner might fail, in which case the driver simply falls back to iterating through all processes.

6   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:

```rust
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 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.

(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^{2}C 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^{2}C, USART, SPI, and AES encryption. In experiments comparing with TinyOS, we used a port of TinyOS to a predecessor of this platform [3] which has the same microcontroller and similar peripherals.

6.1 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.

6.1.1 City-Scale Sensing. Signpost [1] 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.

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.

6.1.2 USB Security Key. USB authentication keys can provide better security and user experience relative to other second factor authentication options [29]. As a result, many large organizations are deploying security keys internally [16].

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 [56], 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 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.

6.2 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

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

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

Figure 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.

	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 5: 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.

(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.

6.2.1 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 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 I²C device and a virtual alarm—as well as to its dependents. Each 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 6.2.2, 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 5 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.

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

Table 6 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 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

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 6: 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.

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

Table 7: 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.

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 7 lists the flash and memory 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.

6.3 Capsules vs. Process-only Isolation

Capsules enable isolation at a finer granularity than can be achieved with memory-isolated processes. Capsule isolation

---

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

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.

6.3.1 Memory overhead. From Section 6.2.1, 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 [5] 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.

6.3.2 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.

6.4 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), 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 `Grants` impose only a two-byte overhead.

6.4.1 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.

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

Table 8: 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.

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 [42], sends packets using the CTP collection protocol [21], sends packets over the serial port, and can write to block, log, and configuration flash storage [19]. Table 8 shows the code and RAM size of the resulting TinyOS image for 1-3 threads on an MSP43F1612 [39] (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.

6.5 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, 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
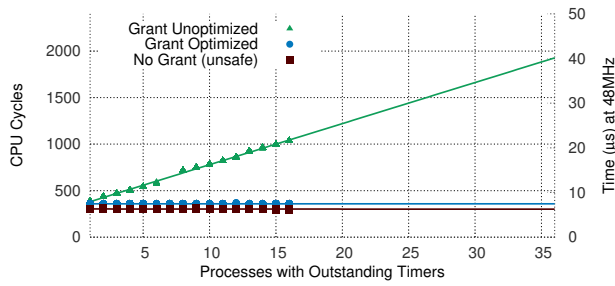
Figure 8: The simple implementation requires an event handler to iterate through the grant structure of each process to deliver an event. The graph above shows the overhead in CPU cycles and time of this iteration for a workload of up to 16 processes, projected out to 36 processes. It also shows an optimized 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.

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 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.

6.5.1 Alternatives and Optimizations. Section 5 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 stores a weak pointer to the subsequent timer, and so on. 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 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 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).

7 RELATED WORK

Tock draws on a rich ecosystem of embedded operating systems. It is most similar to SOS [23], 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.

Section 2 contrasts the goals and design of Tock with Arduino [6], TinyOS [33], TOSThreads [28], FreeRTOS [8], and RIOT [5]. These were chosen to be representative of a class of systems that includes other research efforts like Contiki [13–15], TinyThreads [38] and Fibers [53] as well as a variety of industry products such as ARM's mbed [37] and Chromium Embedded Controller [48].

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 [47]. 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 [52]. 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 [22] to build reliable and safe operating systems [34]. Spin [9] allowed applications to extend and optimize kernel performance by downloading modules written in Modula-3 [10]. Spin provides relatively weak isolation between processes, which share a common garbage-collected heap. The Singularity [24] 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 [11] is a UNIX file-system implementation verified in Coq. seL4 [26] is a verified microkernel. Yang et al. [55] 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 [43, 44, 49, 50] and block-level lock synchronization [41] influenced the design of the grant interface in Tock.

8 CONCLUSION

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.

9 ACKNOWLEDGMENTS

REFERENCES

[1] 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.

[2] AL DANIAL. cloc. http://cloc.sourceforge.net. Accessed 24-August-2017.

[3] 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.

[4] ANDERSON, T., AND DAHLIN, M. *Operating Systems: Principles and Practice*, 2nd ed. Recursive Books LLC, 2014, ch. 2.1, pp. 43–44.

[5] 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.

[6] BANZI, M., CUARTIELLES, D., IGOE, T., MARTINO, G., MELLIS, D., ET AL. Arduino. https://www.arduino.cc/. Accessed 09-May-2016.

[7] 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.

[8] BARRY, R., ET AL. FreeRTOS. http://www.freertos.org/. Accessed 09-July-2016.

[9] 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.

[10] 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.

[11] 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.

[12] 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? 591–.

[13] DUNKELS, A., ET AL. Contiki mulithreading. https://github.com/contiki-os/contiki/wiki/Multithreading. Accessed 09-May-2016.

[14] 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.

[15] 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.

[16] FIDO ALLIANCE. FIDO Certified Showcase. https://fidoalliance.org/fido-certified-showcase/, April 2017.

[17] FITBIT. FitBit: Official site for activity trackers and more, 2017. Accessed: 04-20-2017.

[18] GARMIN. vìvoactive 3. https://buy.garmin.com/en-US/US/p/571520, September 2017.

[19] GAY, D., AND HUI, J. TEP 103: Permanent Data Storage (Flash). http://www.tinyos.net/tinyos-2.x/doc/txt/tep103.txt, 2007.

[20] 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).

[21] 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.

[22] 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.

[23] 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.

[24] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review 41*, 2 (April 2007), 37–49.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] LANG, J., CZESKIS, A., BALFANZ, D., AND SCHILDER, M. Security keys: Practical cryptographic second factors for the modern web. In *Financial Cryptography* (2016).

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] LEVY, A., CAMPBELL, B., GHENA, B., PANNUTO, P., DUTTA, P., AND LEVIS, P. The Case for Writing a Kernel in Rust. In *Proceedings of the Eighth ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2017)* (September 2017).

[36] 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.

[37] MBED. mbed OS 5. https://developer.mbed.org/, 2017. Accessed: 04-20-2017.

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

[39] MSP430 ULTRA-LOW-POWER MICROCONTROLLERS. http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/overview.page.

[40] NEST LABS. Meet the Nest Protect smoke and carbon monoxide alarm. https://nest.com/smoke-co-alarm/meet-nest-protect/, 2017.

[41] ORACLE JAVA DOCUMENTATION. Intrinsic Locks and Synchronization. https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html, 2017. Accessed: 04-20-2017.

[42] 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.

[43] POSTGRESQL 9.6.2 DOCUMENTATION. Memory Management. https://www.postgresql.org/docs/current/static/spi-memory.html, 2017. Accessed: 04-20-2017.

[44] ROSS, D. T. The AED free storage package. *Commun. ACM 10*, 8 (Aug. 1967), 481–492.

[45] SAM4L ARM CORTEX-M4 MICROCONTROLLERS . http://www.atmel.com/products/microcontrollers/arm/sam4l.aspx.

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

[47] TEJUN HEO. Control Group v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt, 2015.

[48] THE CHROMIUM PROJECT. Chromium Embedded Controller (EC) Development. https://www.chromium.org/chromium-os/ec-development, 2017. Accessed: 04-20-2017.

[49] TOFTE, M., AND BIRKEDAL, L. A region inference algorithm. *ACM Trans. Program. Lang. Syst. 20*, 4 (July 1998), 724–767.

[50] 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.

[51] 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.

[52] TRUSTWORTHY SYSTEMS TEAM, DATA61. *seL4 Reference Manual Version 7.0.0*, Sept. 2017. https://sel4.systems/Info/Docs/seL4-manual-7.0.0.pdf.

[53] 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.

[54] 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.

[55] 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.

[56] YUBICO. Yubikey hardware. https://www.yubico.com/products/yubikey-hardware/.