Microcontrollers Deserve Protection Too

Michael P Andersen[†], *Tom Bauer*[‡], *Sergio Benitez*[‡], *Bradford Campbell*[§], *David Culler*[†], *Prabal Dutta*[§], *Philip Levis*[‡], *Amit Levy*[‡], *Pat Pannuto*[§] *and Laurynas Riliskis*[‡]

[†]*University of California, Berkeley* [‡]*Stanford University* [§]*University of Michigan*

{*m.andersen,culler*}*@berkeley.edu* {*tbauer01,sbenitez,pal,levya*}*@stanford.edu* {*bradjc,prabal,ppannuto*}*@umich.edu*

Abstract

Microcontroller operating systems and frameworks typically assume that a single, monolithic application will run on an embedded system. Traditionally, the need to optimize for power and to squeeze applications into minimal code and memory allocations have constrained microcontroller applications to be single-function. The absence of hardware primitives capable of providing protection has eliminated security and isolation as considerations. Newer hardware, however, has changed this paradigm. The microcontroller is growing up and can now support a secure, trusted kernel and multiple, isolated, concurrent, and dynamically-loaded applications, all while operating on the power budgets that originally made this device class feasible. While the hardware support is now present, the software ecosystem to leverage these advances is lagging behind. To remedy this, we propose Tock, a new embedded operating system that builds on established operating system principles adapted to the embedded system environment. Tock exploits memory protection units, advancements in modern systems programming languages, and the event driven nature of embedded applications to allow a core kernel, device specific drivers, and untrusted applications to coexist on a single microcontroller. This new operating system will allow embedded devices to mature beyond program-once, deploy-once systems into re-usable, ubiquitous, and reliable computing platforms.

1 Introduction

As networking became an integral part of computing through the Internet and world wide web, system security grew in importance from a niche interest of the military into a core requirement for operating systems. While far from perfect, today's general purpose operating systems support firewalls, NATs, encrypted networking, no-execute bits, randomized memory allocation, signed packages, and a host of other security features. With the growth in networked embedded devices, it is time for embedded operating systems to follow suit.

Today, we see the emergence of "intelligent" *things* that integrate computing with physical devices. Smart pill bottles such as AdhereTech's [1] monitor our medications and remind us of missed doses and wearable personal fitness trackers like Fitbit [3] and UP [23] measure our steps and sleep patterns. But these devices are siloed: Fitbit cannot load a "pill app" that vibrates when it is time to take today's medication. As embedded systems grow from narrow applications into a more general purpose computing platforms, there will be a need to dynamically load applications and share hardware across multiple applications. In fact, this trend is already underway: the Pebble watch [7], for example, supports loading executable code that can use the built-in sensors and display as well as communicate with apps on a connected mobile phone.

Traditionally, embedded systems have been single-purpose, single-application devices. A developer compiled a monolithic application image which incorporated the OS and gave the application full access to underlying hardware. The software was co-designed with hardware—i.e. by the same people—and this paradigm was what the underlying microcontrollers could support. Without virtual memory or segmentation features embedded "operating systems" have no security mechanisms and applications must be trusted completely. Even designs which separate kernel and application code [4, 18, 21] require trusted applications.

However, a new generation of microcontrollers has emerged recently. These processors, based on the ARM Cortex-M architecture, sit in a middle ground between full-fledged CPUs and older microcontrollers. Like some mid-1980s processors, they have a 32-bit address space, memory protection, and run at tens of MHz. Like microcontrollers, they have tens to a few hundred KB of RAM, provide many peripherals and bus interconnects (SPI, I2C, one-wire, UART/USART, etc.) and have sub-µA sleep currents, allowing them to operate on a battery for months or years. The Atmel SAM4L [12], for example, is a microcontroller with a Cortex-M4 processor (40 MHz, 64 KB RAM, 512 KB flash), a plethora of peripherals (4 USARTs, 2 SPIs, 6 timers, native AES and a random number generator) and has a sleep current as low as 0.9 µA.

Most importantly to an operating system, this new generation of microcontrollers includes memory protection. This can protect kernel memory, memory-mapped I/O, and application memory from misbehaving applications. However, isolation and safety is only as good as the kernel itself. As numerous recent results show, OS kernels contain scores of memory access bugs that applications can exploit to crash a system or gain unauthorized ac-

Authors listed in alphabetical order.

cess [16, 35]. By writing the OS kernel in a type-safe, memory-safe language with strong semantics, we can eliminate whole classes of vulnerabilities. The relative simplicity of Cortex-M processors and embedded operating system APIs means that writing such a kernel is a manageable task. This makes an embedded OS especially well suited for research in secure operating systems design.

It is time for embedded operating systems to evolve! In this paper, we argue that an embedded operating system should be more than a hardware API. It should enable and encourage multiple concurrent, isolated applications, provide hardware abstractions as well as low-level hardware access, and offer critical services to Internet of Things applications. We discuss three protection mechanisms that operating systems should utilize to achieve these goals—language level isolation, software mediated hardware protection, and multiple isolated processors.

2 Operating System Considerations

Tanenbaum's *Modern Operating Systems* asserts, "The main property which distinguishes embedded systems from handhelds is the certainty that no untrusted software will ever run on it" [33]. This was once true, but is no longer the case. The Pebble [7] watch, for example, allows users to load multiple third-party applications dynamically. Embedded systems have traditionally been categorized by event-driven computation, minuscule sleep currents, and constrained battery life. Existing embedded operating systems have provided support for these properties and constraints. Providing support for concurrent, isolated applications, as traditional operating systems do, provides a different set of goals and challenges. We argue that these these requirements and constraints create a new design space for operating systems research.

2.1 General-Purpose Requirements

In traditional embedded systems, application code is completely trusted with full access to all hardware resources. When a single developer both writes the application and designs the hardware, this design is reasonable. However, new embedded devices run third-party applications and allow the end-user to dynamically load applications from app stores. A modern operating system for embedded devices must support loading applications dynamically by the end-user. It must also protect the hardware and kernel from these applications and applications from one another. Modern general-purpose operating systems use hardware features such as virtual memory and hardware virtualization to isolate applications. While these features are not available in embedded microcontrollers, embedded operating systems must use the tools at their disposal to achieve similar goals.

2.2 Embedded Requirements

While attractive from a compatibility and bootstrapping perspective, porting general-purpose operating systems, like Linux, to microcontrollers is not the best solution. First, embedded systems today are still bound by power and battery constraints. Second, embedded applications are significantly different than applications on general-purpose systems. Embedded applications are event-driven, responding to timers, packet arrival, sensor interrupts, etc., with short bursts of computation. General-purpose applications typically use threaded execution, which may include long-running computations or latency sensitive user interactions. General-purpose system support for event-driven models has largely focused on latency, throughput, and scalability [9, 19, 28], rather than power draw, which is typically more important in embedded devices. Finally, embedded applications often require direct access to underlying hardware resources. Some general-purpose operating systems have attempted to make dynamically loaded drivers more safe by using lightweight hardware protection domains [32] but do not offer the flexibility that many embedded applications may need. We argue that in embedded systems, *applications* should have direct access to hardware resources when those resources are not shared or the hardware can do isolation by itself [14].

3 Protection *is* a New Primitive

This section examines recent advances and trends in embedded hardware and programming languages that enable novel protection primitives in constrained systems.

While traditional operating systems handle independently developed applications, these OSes cannot simply be applied to the embedded domain. They assume virtual memory for protection and isolation and are not designed to take advantage of memory protection subsystems present in embedded hardware. They trust that drivers are written correctly, granting them full access to kernel structures and memory, and fail when driver bugs are present. Lastly, they are designed for cooperative cores with shared memory, rather than physically isolated microcontrollers with private memories.

3.1 Hardware Advances and Trends

New embedded system platforms will be built on the Cortex-M series of microcontrollers. While ARM's A-series is substantially more capable and is used in wall-powered devices such as the HummingBoard [5] and BeagleBone Black [2], its significant power draw (over 1 W in active mode) make it infeasible for low-power, battery operated devices. In contrast, Cortex-M series microcontrollers have power draws conducive to low-power operation while adding hardware features that enable protection.

Memory Protection Units. ARM's Cortex-M is a microcontroller design—a System-on-Chip with tightly integrated memory—and does not (nor will it likely ever) integrate enough memory to merit a Memory Management Unit (MMU). Instead, the Cortex-M series includes a Memory Protection Unit (MPU), a lightweight, efficient subsystem that provides the memory protection features similar to those found in an MMU (e.g. it will trap illegal memory accesses such as writing to read-only memory) but without address translation and at a much finer granularity. With OS support an MPU can enable isolated applications, shared libraries, and even possible relocation of running code. MPUs are much more fine-grained than typical virtual memory. MPUs are able to address regions as small a 32 bytes, whereas MMUs typically use at least 4 KB pages. Moreover, MPUs support regions of any size that is a power of two larger than 32 bytes (27 sizes in practice) [11], whereas MMUs support a limited number of page sizes.[1] MPUs do not, however, perform any translation, so mechanisms such as swap and dynamic page allocation may not be feasible.

Multi-Processor Platforms. Modern embedded platforms increasingly include multiple microcontrollers. For example, the Nest Protect [6] includes a main Cortex-M4 application controller, a secondary Cortex-M0+ peripheral processor, and an EM357 802.15.4 radio SoC with an onboard Cortex-M3 [30]. Adding additional microcontrollers can allow offloading timing-critical communication functionality or running certain computations on a microcontroller with a different energy profile. Multiple processors provide an opportunity for the strictest isolation, stochastic and deterministic real-time schedules, and computation offloading. Existing operating systems have not needed to provide abstractions for multiple microcontrollers what these abstractions should be is an open question.

3.2 Language Level Protection

The C programming language has long been the language of choice for system-level programming, including operating systems development. C is inherently an unsafe programming language [34, 35]. As such, much effort has been made to develop operating systems in languages with stricter semantics [20, 22, 26]. These efforts have shown that language features like type safety, memory safety, and strict aliasing can provide support for operating system protection.

Memory Safety. Memory safety bugs—dangling pointers, double-frees, access to unallocated memory, and pointer arithmetic errors—plague operating systems written in languages with weak memory semantics. Languages that guarantee memory safety obviate such bugs by catching them at compile-time[2]. For instance, a kernel written in a memory safe language can enforce memory isolation of drivers at compile time without relying on hardware support [15].

Type Safety. Strong, strict types enable low-level hardware to be exposed through typed abstractions that cannot be subverted by untrusted code. Because client code utilizing these interfaces is constrained to the particular hardware operations exposed by the interface, careful interface design can result in fully safe hardware access for untrusted code. As a result, careful interface design can allow for minimum unsafe kernel code. We argue that principled methods for designing such interfaces should be a goal of future operating systems research.

Strict Aliasing. Operating systems typically enforce, but do not guarantee, thread safety at runtime by using atomic primitives. In contrast, languages with strict aliasing rules, such as unique and read/write references, guarantee thread safety at compile-time. *Unique references* ensure that only a single active execution context can access the reference, while *read/write references* ensure that multiple references to the same state are immutable. If multiple applications require write access to the same resource, a broker with a unique reference to the resource can mediate access. Consequently, by modeling hardware resources as unique or read/write references, the kernel can guarantee the preclusion of hardware race conditions.

4 Tock: a Secure Embedded OS

Tock is an operating system designed for embedded platforms with multiple processors running multiple concurrent applications. Tock's design centers around protection, both from potentially malicious applications and from device drivers. Tock uses three mechanisms to protect different components of the operating system. First, the kernel and device drivers are written in Rust [27], a systems programming language that provides compile-time memory safety, type safety and strict aliasing. Tock uses Rust to protect the kernel (e.g. the scheduler and hardware abstraction layer) from platform specific device drivers as well as isolate device drivers from each other. Second, Tock uses memory protection units to isolate applications from each other and the kernel. Finally, when available, Tock uses multiple microcontrollers to protect applications with timing concerns against starvation or leaking information through side-channels. The remainder of this section gives an overview of the Tock architecture, focusing on how the kernel uses the MPU for protection while simultaneously giving applications direct access to hardware.

[1]The Itanium architecture supports eight page sizes, while x86 and ARMv7 both only support three.

[2]Some languages, like Java, are considered memory safe, but do not offer such strong guarantees. However many, such as Haskell and ML, do.

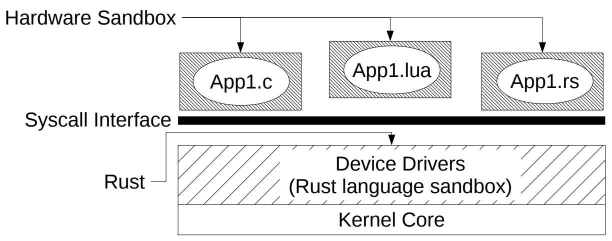

Figure 1: Tock is written in Rust, a type-safe, memory-safe systems language. The Tock kernel's core has full access to the underlying hardware. Device drivers run with no hardware protection, but within a language sandbox that constrains their access to hardware. Applications are sandboxed using the MPU, a hardware memory protection mechanism, and can be written in any language (e.g. C, Rust, or even a scripting language like Lua).

4.1 Architecture

The Tock kernel has complete system access, including arbitrary memory access and interrupt control. Device drivers are compiled into the kernel and run with the same *hardware* privileges but inside a *language-level* sandbox. This sandbox statically enforces that device drivers only use kernel hardware interfaces, limits dynamic memory allocation to load time, and prevents uncoordinated access of underlying hardware resources.

Applications are separated from the kernel by a syscall ABI and can be written in any language that can interface with the ABI. Applications have an execution stack, an application memory heap, and a kernel memory heap. Kernel (and driver) dynamic allocations take place in the application memory space, thus charging applications for buffers the kernel creates on their behalf and ensuring that an out-of-memory condition terminates the application and does not cause a kernel error. The kernel uses the MPU to protect the application-level kernel memory heap from application access.

The ABI can provide direct access to system hardware resources. If an application requires direct access, it can request it from the kernel. If granted, the kernel uses the MPU to make the memory mapped I/O region of specific peripherals accessible to the application. Because processors such as the Cortex M support access permissions on as fine grained boundaries as 32 bytes, the kernel can expose one port of GPIO pins but not others. Enabling this kind of access safely requires that kernel-controlled peripherals be connected only to GPIO ports that cannot be exposed.

4.2 Execution Model

When a device powers on or resets, Tock loads each application by allocating a fixed sized memory region for it's stack, heap and data (Section 4.3), configuring the MPU, switching to an unprivileged execution mode and jumping to the application's `init` function. While executing, the application can request resources (e.g. access to the *TRNG* memory mapped registers), register for callbacks to events (e.g. a timer or sensor interrupt) and issue commands (e.g. send a packet over the radio). Applications run in a single thread and will not be interrupted by event callbacks until either invoking the `wait` system call, or returning from `init` (which is an implicit `wait`). Once an application has issued the `wait` system call, the kernel suspends it until a relevant event has fired, and a callback can be invoked. Finally, the kernel invokes callbacks by pushing an additional stack frame onto the application's stack and jumping to the callback function. Thus, when a callback function returns, execution is resumed where the `wait` system call was invoked.

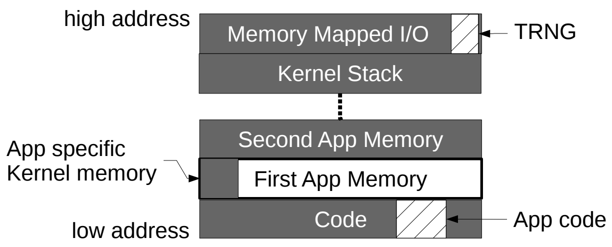

Figure 2: Memory layout and access permissions while executing an application. Tock lays out memory into four types of regions: code, memory mapped I/O, application memory and the kernel stack. Greyed regions are marked unreadable and unwritable, while hatched regions are read-only. The application memory contains a subregion used by the kernel for application-specific kernel data structures and is thus inaccessible to the application. Conversely, some hardware registers are exposed directly to applications by marking them read-only.

4.3 Tock Memory Design in Detail

One major challenge in Tock is defining a memory management design that allows untrusted application code to interact with a type-safe Rust kernel, for the kernel to be able to issue event upcalls to the application, for device driver isolation, and for applications to directly access hardware. To achieve this Tock defines four types of memory regions: code regions, memory-mapped I/O regions, kernel stack and static data regions and application regions. Figure 2 shows these regions and their access rules from an application.

Code. A code region contains kernel and application code. While an application is executing code, it only has read and instruction fetch access to its own code segment. An application cannot write to its own code section because typically the code is backed by internal flash, which has limited write cycles.

Memory-Mapped I/O. Hardware registers are typically located in a single, continuous region of the memory space.[3] Tock may provide read-only or read-write access over small ranges of memory registers to applications, bypassing the kernel for direct hardware access. For example, our development platform uses the Atmel SAM4L Cortex-M4 [12] which provides a true random number generator (TRNG) peripheral. Tock exposes random numbers directly to applications by allowing read-only access to the TRNG registers.

Kernel Stack and Static Data. These regions include all kernel and device driver memory, including local variables. The MPU restricts any application-level access to this memory section.

Application Memory. Tock allocates a continuous, fixed size region for each application. Application memory and the kernel stack grow towards each other to balance application count with kernel stack size. If the kernel stack grows too large, Tock terminates the application whose memory region is closest to the kernel stack. Thus, applications with higher priority should be allocated memory regions at lower memory addresses. We intend to explore an alternate strategy of allocating application memory from a shared heap using the high granularity of the MPU. This would trade-off a simpler process for reclaiming stack space for better support of heterogeneously sized applications.

Most of an application's memory is used for its stack, static variables, and heap. However, the kernel may also "borrow" memory from the top of an application's memory region for application-specific kernel data structures. This region, which may change in size, is marked as read and write protected when the application is running. This allows the kernel and device drivers to allocate memory dynamically in response to an application without potentially causing an out-of-memory kernel error and while maintaining the integrity and confidentiality of shared kernel data structures. As an example, when an application registers a callback for a timer, the timer device driver allocates a new link node for the callback in the application's memory. Since the link includes forward and backward list pointers, it is imperative that, despite being in the application's memory, this link's integrity be maintained. Moreover, the values of the links might leak information about other applications.

5 Prior and Future Work

Embedded systems have historically had highly fragmented application domains, each with a vertically integrated set of software and hardware that a single group of developers is responsible for. This complete oversight has typically meant that embedded operating systems have minimal or no system security to protect themselves from malicious code or applications.

One extreme example of this approach is TinyOS [25], which makes no distinction between application and operating system code. Application code specifies which abstractions and services it needs from the operating system, and the compilation toolchain uses this information to build an application-specific version of the OS that is compiled and directly linked with application code. Even with the threading library, TOSThreads [24], and dynamically linked applications, the application developer and the system maintainer are assumed to be the same person and applications have unlimited access to the entire system. The Contiki [18] operating system is another microcontroller-based operating system with extensive networking support. Security in Contiki is limited to link-layer encryption and integrity and the OS can run only a single application at once. SOS [21] provides dynamically loadable applications and drivers on top of a core kernel but provides no isolation or protection, therefore requiring both be trusted completely.

FreeRTOS [4] is an open-source commercial operating system intended to support a wide range of applications on many different processors. It is used in applications from the Nest Protect to smart meters to thermal monitoring systems. FreeRTOS supports network security through TLS [17] and DTLS [31], but operates under the assumption that all code is trusted. For example, any task in the system can spawn a task that runs in privileged mode and arbitrarily modify the system [8, 10].

Recent paravirtualization-based systems such as Dune [13], Arrakis [29], and IX [14], have explored how hardware can be directly exposed to applications for high-performance or greater flexibility. These approaches all depend on the hardware virtualization (e.g., NICs, page tables). In contrast, embedded systems have no virtualization. Tock applications do not access a virtualized SPI bus, they can gain direct access to hardware registers with no subsequent OS or hardware arbitration.

Emerging embedded applications and platforms require new abstractions and services from their operating systems. Simultaneously, new embedded processors provide new mechanisms that are similar to, but distinct from modern and earlier CPUs. This combination of application pull and technology push creates many interesting problems in operating system design. As pervasive computing, ubiquitous computing, sensor networks and the Internet of Things all gain momentum, solutions to these problems will be increasingly important. We have presented one initial design for such an operating system, called Tock, which combines hardware memory protection with a modern safe systems language to enable novel capabilities that meet the needs of these emerging domains.

[3]The location of hardware memory registers and flash is determined by the chip manufacturer, however, chips typically place flash at the bottom of memory, followed by RAM, and peripheral memory registers towards the top of memory.

References

[1] AdhereTech Smart Wireless Pill Bottles. http://adheretech.com/.

[2] BeagleBone Black. http://beagleboard.org/black.

[3] Fitbit One Wireless Activity + Sleep Tracker. https://www.fitbit.com/one.

[4] FreeRTOS. https://www.freertos.orf/.

[5] HummingBoard. http://www.solid-run.com/products/hummingboard/.

[6] Nest Protect. https://nest.com/smoke-co-alarm/life-with-nest-protect/.

[7] Pebble smartwatch. http://www.getpebble.org.

[8] FreeRTOS-MPU security (privileges). http://sourceforge.net/p/freertos/discussion/382005/thread/7b135e95, September 2013.

[9] *epoll(7) Linux Programmer's Manual*, July 2014.

[10] FreeROTS: xTaskCreate. http://www.freertos.org/a00125.html, December 2014.

[11] ARM. *Cortex-M4 Devices User Guide*, Dec 2010. Issue A.

[12] ATMEL. *ARM SAM4L Low Power MCU*, 2013.

[13] BELAY, A., BITTAU, A., MASHTIZADEH, A. J., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012* (2012), pp. 335–348.

[14] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 49–65.

[15] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. J. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating Systems Principles* (1995), ACM Press, pp. 267–284.

[16] CHEN, H., MAO, Y., WANG, X., ZHOU, D., ZELDOVICH, N., AND KAASHOEK, M. F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (New York, NY, USA, 2011), APSys '11, ACM, pp. 5:1–5:5.

[17] DIERKS, T., AND RESCORLA, E. The transport layer security protocol version 1.2 (TLS). RFC 5246, August 2008.

[18] DUNKELS, A., GRÖNVALL, B., AND VOIGT, T. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE Conference on Local Computer Networks (LCN 2004), 16-18 November 2004, Tampa, FL, USA, Proceedings* (2004), pp. 455–462.

[19] GRIBBLE, S. D., WELSH, M., VON BEHREN, J. R., BREWER, E. A., CULLER, D. E., BORISOV, N., CZERWINSKI, S. E., GUMMADI, R., HILL, J. R., JOSEPH, A. D., KATZ, R. H., MAO, Z. M., ROSS, S. J., AND ZHAO, B. Y. The ninja architecture for robust internet-scale systems and services. *Computer Networks 35*, 4 (2001), 473–497.

[20] HALLGREN, T., JONES, M. P., LESLIE, R., AND TOLMACH, A. P. A principled approach to operating system construction in Haskell. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005* (2005), pp. 116–128.

[21] 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.

[22] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (Apr. 2007), 37–49.

[23] JAWBONE. Up by Jawbone. https://jawbone.com/up.

[24] KLUES, K., LIANG, C.-J. M., PAEK, J., MUSALOIU-ELEFTERI, R., LEVIS, P., TERZIS, A., AND GOVINDAN, R. TOSThreads: Thread-safe and non-invasive preemption in TinyOS. In *SenSys* (2009), vol. 9, pp. 127–140.

[25] LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., ET AL. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. Springer, 2005, pp. 115–148.

[26] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D. J., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *ASPLOS* (2013), V. Sarkar and R. Bodk, Eds., ACM, pp. 461–472.

[27] 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.

[28] MAZIÈRES, D. A toolkit for user-level file systems. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA* (2001), pp. 261–274.

[29] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 1–16.

[30] POOLE, N. Nest Protect teardown. https://learn.sparkfun.com/tutorials/nest-protect-teardown.

[31] RESCORLA, E., AND MODADUGU, N. Datagram transport layer security. RFC 4347, April 2006.

[32] SWIFT, M. M., MARTIN, S., LEVY, H. M., AND EGGERS, S. J. Nooks: an architecture for reliable device drivers. In *Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France, July 1, 2002* (2002), pp. 102–107.

[33] TANENBAUM, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[34] WANG, X., CHEN, H., CHEUNG, A., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Undefined behavior: What happened to my code? In *Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012* (2012), p. 9.

[35] WANG, X., CHEN, H., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Improving integer security for systems with KINT. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012* (2012), pp. 163–177.