Application Memory Isolation on Ultra-Low-Power MCUs

Taylor Hardin, *Dartmouth College;* Ryan Scott, *Clemson University;*
Patrick Proctor, *Dartmouth College;* Josiah Hester, *Northwestern University;*
Jacob Sorber, *Clemson University;* David Kotz, *Dartmouth College*

https://www.usenix.org/conference/atc18/presentation/hardin

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Application Memory Isolation on Ultra-Low-Power MCUs

Taylor Hardin
Dartmouth College

Ryan Scott
Clemson University

Patrick Proctor
Dartmouth College

Josiah Hester
Northwestern University

Jacob Sorber
Clemson University

David Kotz
Dartmouth College

Abstract

The proliferation of applications that handle sensitive user data on wearable platforms generates a critical need for embedded systems that offer strong security without sacrificing flexibility and long battery life. To secure sensitive information, such as health data, ultra-low-power wearables must isolate applications from each other and protect the underlying system from errant or malicious application code. These platforms typically use microcontrollers that lack sophisticated Memory Management Units (MMU). Some include a Memory Protection Unit (MPU), but current MPUs are inadequate to the task, leading platform developers to software-based memory-protection solutions. In this paper, we present our memory isolation technique, which leverages compiler inserted code and MPU-hardware support to achieve better runtime performance than software-only counterparts.

1 Introduction

Smart watches and smart bands offer novel opportunities for individuals to monitor and control their health, manage a chronic disease, pursue athletic excellence, recover from surgery, or steer their lifestyle toward healthier behaviors. Smart watches can run a variety of apps, including third-party apps installed by the user. However, the battery life for a typical smart watch is about one day, far shorter than the weeks-long battery life typical of single-purpose fitness bands. To balance these tradeoffs, some devices (such as the Amulet [10]) seek to achieve the battery life of a closed-source fitness band (like a Fitbit) and the capability to run multiple third-party apps, while retaining strong security properties. These low-energy multi-app wearable platforms employ ultra-low-power microcontrollers (MCUs), with tiny RAM, limited secondary storage, and which lack the hardware-based memory-protection mechanisms – such as Memory Management Units (MMU) – needed to ensure that applications cannot interfere with each other. This makes it difficult to provide long battery life *and* strong security properties that allow multiple third-party apps to coexist.

This work focuses on a fundamental security property: *memory isolation*, which ensures that no application can read, write, or execute memory locations outside its own allocated region, or call functions outside a designated system API. In this paper, we present a novel memory isolation technique, which leverages compiler inserted code and a low-sophistication Memory Protection Unit (MPU) found in many microcontrollers, to achieve better performance than software-only counterparts.

We use the open-source Amulet platform [10] to implement the following isolation methods for comparison: (1) compiler-enforced language limitations (no pointers, no recursion), (2) compiler-inserted run-time memory isolation (address-space bounds verification), and (3) MPU-supported memory isolation (hardware enforced failure). The first option is the approach taken by the Amulet team, which limits the programmer to a subset of C. Pointers are disallowed, and the compiler inserts code for run-time bounds-checking on arrays. In the second approach, we modify the Amulet implementation to allow for pointers and recursion, but our custom compiler inserts code to validate each pointer dereference to ensure the application stays within its bounds. In the third approach we implement a novel combination, in which the OS and compiler coordinate the dynamic assignment of the MPU's limited functionality – and limited compiler-inserted pointer checking – to enable the desired isolation. Finally, we automate this process through an extension to the Amulet Build System. We make the following **contributions**:

1. an analysis of design considerations, including security issues, that enable multiple applications on ultra-low-power wearables, with minimal burden on the programmer or the user;
2. a novel technique, using the limited-function hardware memory protection unit (MPU) found in commodity ultra-low-power microcontrollers, combined

with compile-time analysis of application code, to sandbox application code and memory;

3. a prototype implementation as a refinement of the open-source Amulet platform;
4. an evaluation that compares the performance of the Amulet platform's limited language-based memory-isolation mechanism, a full-featured software-only approach, and a full-featured MPU-assisted mechanism.

2 Background and Related Work

Early operating systems for wireless sensors like TinyOS [14] and others [1, 6, 9] reduced complexity [13], enabled dynamic reprogramming [15], and provided interfaces for concurrent execution [7]. These platforms did not provide memory isolation, nor did they allow installation of multiple third-party applications. As the application space grows, security mechanisms that enable multiprogramming of multi-tenant microcontroller units (MCUs) must be developed. Recent work has explored approaches for memory isolation on microcontrollers.

Some approaches change the language: AmuletOS [10] uses a dialect of ANSI C, termed AmuletC, which disallows pointers and recursion. TockOS [16] writes kernel code in Rust, a type-safe and memory-safe language, and isolates their apps using an MPU. While language modifications can make compile-time analysis easier [17], they tend to limit expressiveness and are rarely enough to ensure complete application isolation.

Language features are often coupled with compiler checks, binary-code rewriting, or system-implemented dynamic checks. For example, AmuletOS has a compiler that inserts run-time bounds-checking code around all array accesses [10]. Deputy [4, 5] enforces type safety at compile time; Harbor [12], built on top of SOS [9], rewrites binary code to check any pointer reference and function call. T-Kernel [8] modifies code at load time to secure application memory. Each of these compile and run-time techniques come with limitations: compile-time techniques depend on language features (or modifications) and clear OS rules, while dynamic checking requires expensive run-time overhead to check memory accesses.

Other systems virtualize the single memory space to isolate applications, like Maté [13], or rely on novel hardware mechanisms such as a Secure Loader hardware unit between the CPU, peripherals, and RAM [11].

Many ultra-low-power MCUs like the MSP430 FRAM series [18] are equipped with a basic Memory Protection Unit, but they have some or all of the following shortcomings: (1) they support too few distinct regions, not enough to sandbox each application; (2) they leave certain segments of memory, like hardware registers or RAM, unprotected; and (3) they have arcane protection boundary rules, because they depend on opaque hardware implementations.

Given all these prior techniques, we see the potential for a *new* approach that leverages the meager capabilities of the new class of MPU, and the lessons learned from years of isolation techniques using software approaches. In this paper, we evaluate the performance of our memory-isolation technique, which leverages compiler-inserted code and MPU-hardware support, against: a language-limited software-based approach (the native Amulet approach [10]), and a full-featured compiler-inserted-check approach.

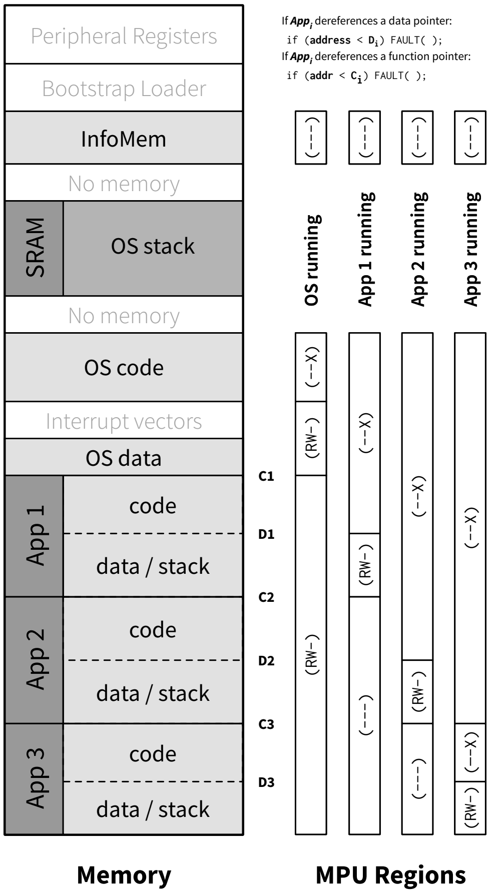
3 System Design

We apply our memory-isolation technique to the latest open-source build of Amulet[1]. Amulet implements memory isolation through compiler-enforced language limitations (no pointers, no recursion, no goto statements and no inline assembly). We remove the most burdensome restrictions by allowing app programmers to use recursion and C pointers (including function pointers) in their code, which reduces the effort to port code to the Amulet and allows developers to write new apps in a customary fashion. In our approach we implement two methods to allow these language features and still ensure memory isolation – use of the memory protection unit (MPU) and compiler-inserted run-time memory isolation.

The Amulet system allows an Amulet user to select a customized mix of applications to run on her Amulet wristband, from a suite of applications developed independently by separate app developers. The Amulet system consists of three core parts – AmuletOS, Amulet Runtime, and the Amulet Firmware Toolchain (AFT). AmuletOS provides the core system services and an event-based scheduler that drives the apps' state machines, delivering events by calling the appropriate event-handler function with parameters representing the details of the event. Amulet Runtime provides a state-machine environment in which all applications run. The Amulet Firmware Toolchain (AFT) [10], analyzes, transforms, merges, and compiles the user's desired applications with the AmuletOS to construct a firmware image for installation on the user's Amulet device.

Amulet devices use a TI MSP430FR5969 MCU, which have a memory protection unit (MPU), with limited capabilities as described in Section 2. The MPU is not a memory-management unit (MMU), nor does it provide full memory protection: it cannot protect all regions of memory (the MPU will not prevent instructions from reading or writing the peripheral registers, InfoMem, SRAM,

[1] The latest open-source release of the Amulet platform can be found at `https://github.com/AmuletGroup/amulet-project`



If ***App$_i$*** dereferences a data pointer:

```
if (address < D_i) FAULT( );
```

If ***App$_i$*** dereferences a function pointer:

```
if (addr < C_i) FAULT( );
```

Figure 1: Memory diagram of our approach, and MPU regions per application.

or interrupt vectors), and its limited selection of three MPU-controlled segments does not allow us to subdivide memory into the four regions we desire (app code, app data/stack, off-limits memory below the app, and off-limits memory above the app). The MPU only has the ability to protect accesses to memory above the higher app bound but not below the lower app bound. To protect lower memory the compiler inserts a lower bound check. Thus, the MPU memory isolation method consists of configuring the MPU for an app and inserting lower-bound checks, while the compiler inserted (software-only) method mentioned earlier consists of not using the MPU and inserting both an upper and lower memory bound check. Although the MSP430's MPU itself is not sufficient to protect the system and other applications from pointer misuse by a buggy (or malicious) app, it is useful: in our approach, we strategically leverage both the MPU and the compiler to accomplish the necessary protections. This section details the memory map used for MPU, as well as how we handle memory accesses and context switches.

Memory Map: Use of the MPU requires a different memory mapping than in the original Amulet implementation. Figure 1 diagrams our approach. We leverage the SRAM for the AmuletOS stack, the low FRAM for AmuletOS code and data, and the high FRAM for app code and data, grouped by app.[2] Each app's code and data are separated, with its code in lower addresses than its data. The MPU has four segments, of which we can make good use of three.[3]InfoMem, the first segment, is fixed to a certain address range and its configuration can be changed any time by any code. Furthermore, only two boundaries are adjustable: the boundary between segment 1 and 2 and the boundary between segment 2 and 3.

To allow application developers to use C pointers, we leverage previously described MPU hardware. While an app is running, we configure the MPU segments as follows: 0: InfoMem (unused; no access); 1: OS, lower-memory apps, and current-running app's code (execute-only); 2: current-running app's data and stack (read-write only); 3: higher-memory apps (no access).

Consider, as an example, Application 2 in Figure 1. All of the app's code is gathered in one region, all of its data and stack in another region. The MPU configuration triggers a fault if a stray pointer references anything in higher regions (shown as Application 3 in the figure), but the MPU cannot fully protect regions in addresses below the application's code segment.

While the OS is running, we configure the MPU segments as follows: 0: InfoMem (unused; no access); 1: OS code (execute-only); 2: interrupt vectors and OS data (read-write only); 3: apps (read-write only). This configuration allows the AmuletOS to run its own code and, as needed, to manipulate data in both the app and OS regions.

It's important to note an important design change from Amulet as it was originally introduced. The Amulet system uses a single stack – shared by both the OS and the current application. This approach is possible because at most one app runs at any time, so there is no need to retain a stack for non-running apps. It is also possible because app code cannot use pointers, and thus cannot read any memory outside its statically allocated global variables, or outside its current stack frame. If we were to stick with the same single-stack model, we would need to `bzero` the stack region every time we switched apps, lest the new app glean information from the stack tailings left behind by the prior app. We chose instead to allocate a distinct region of memory for each app's stack, removing this cost (and other costs to ensure stack references remain

[2] If the AmuletOS is too large to fit in the low FRAM, it could span the interrupt vectors, but for simplicity we do not show it as such in the diagram.

[3] MPU segment 0 is pinned to the InfoMem, which is only 512 bytes and which we currently do not use. We anticipate using the InfoMem in future revisions, for a return-address stack that protects the return address from stack overflow bugs and attacks.

in-bounds) at the cost of increased memory usage.

That brings us to another important design decision related to security and the application stack. Languages such as C traditionally place a function's return address on the stack, and jump indirectly through that address as part of the function-return instruction. Stack overflows in buggy or malicious code can overwrite that entry on the stack, however, causing the function to return to a different address. We leverage the compiler to insert code to bounds-check the return address before every function return. Furthermore, we place the top of the app stack below the app's data in the app's data/stack segment, and allow the stack to grow downward. The compiler and linker can compute the size of the app's data region, and estimate the maximum stack depth, to ensure the data/stack segment is large enough for the app's needs. If the app overflows its stack, for example by too-deep recursive calls, it will cross an MPU boundary into an execute-only code region and trigger a fault.

Memory accesses: An important role for the runtime system is to handle application faults; when the app attempts an invalid memory access, it jumps to a `FAULT` function to log app-specific information about the fault. At compile time, the AFT uses its transformation tools to verify that the app only calls approved API functions and reads approved system global variables, and to insert code that verifies (at run-time) every pointer dereference before it occurs. Notice that every one of these checks is a simple comparison against a constant, followed by a conditional branch (jump) to the fault-handling code. Because all app code is processed by the AFT, and the app cannot inline any of its own assembly code, the resulting code is guaranteed to check every pointer used by the app.

Context Switches: The AmuletOS provides an API for applications to access utilities and system services. We need to swap MPU configurations and change stacks on each transition, and we need to carefully handle application-provided pointers passed through API calls to the OS. Furthermore, because each app, and the OS, has a separate stack segment, we need to change the stack pointer on every transition between the OS and an app.

AFT Implementation: We extend the AFT to implement the MPU and software-only method checks previously mentioned. These tasks are accomplished by the AFT in a four-phase code analysis. In the first phase, the AFT checks for any still unsupported language features – such as inline assembly and GOTO statements. In addition, the AFT enumerates each memory access and OS API call on an app by app basis. Examination of the application call graph and the stack frame for each function determines the maximum stack size for each app. In the event of recursion, the maximum stack size cannot be determined and the AFT cannot guarantee a large enough stack to prevent overflow. During the second phase, the MPU configuration code and the previously mentioned memory access checks (with placeholder values for app boundaries) are injected into the code. The third phase marks apps with memory section attributes for the linker, as well as injecting the assembly code needed to manipulate the stack pointer. The last phase involves determining the code size of each app, updating the linker script to place each app in high memory (as detailed in Figure 1), and updating the memory access checks from phase two with the correct app boundaries. The AFT completes by recompiling the modified code into the final firmware image.

Operation	No Isolation	Feature Limited	MPU	Software Only
Memory Access	23	41	29	32
Context Switch	90	90	142	98

Table 1: Average cycle count for basic memory isolation operations.

4 Evaluation

In this section we evaluate the costs of application isolation. Our proposed system allows developers to write pure C, instead of a constrained Amulet C, enabling them to more easily write (or port) application code to the Amulet platform. We look at the isolation overhead of a large set of Amulet applications for three methods in Section 4.1, and see that while the overhead of our isolation method is higher than a feature-limited Amulet C, the impact of the overhead on battery lifetime is negligible. In Section 4.2 we describe three benchmark applications, and the tradeoffs they display between computation-intensive and OS-intensive applications.

4.1 Isolation Overhead

We use the Amulet Resource Profiler (ARP) and the ARP-view tool to count the number of memory accesses and context switches per state and transition, per application. Using ARP-view, we can account for the rate of environmental, user, and timer events set by the developer, combine this information with the counted number of memory accesses and context switches, and extrapolate the number of cycles of overhead for isolating applications. We can then convert the estimated cycles into energy cost (in Joules) to estimate the negative impact of isolation on battery lifetime. The results of this experiment are shown in Figure 2 for nine applications that are part of the Amulet platform. These applications comprise thousands of lines of code, and many have been deployed in user studies [2, 3]. **For all applications, isolation using either the MPU or Software Only methods has less than a 0.5% impact on battery lifetime.**

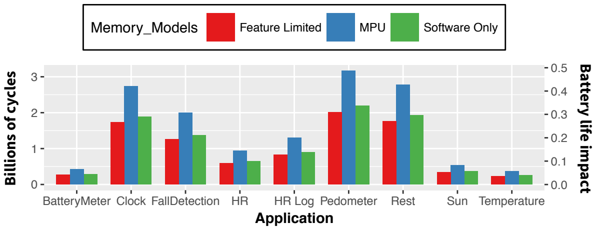


Figure 2: Isolation overhead in billions of cycles per week, and battery lifetime impact percentage for a variety of applications. Gathered using the Amulet Resource Profiler infrastructure.

4.2 Benchmark Applications

We further explore the system overhead of application isolation through several benchmark applications with varying levels of memory accesses. We designed a **Synthetic App** a simple application whose purpose is to test the two fundamental actions that incur memory-protection overheads: *memory accesses* and *context switches*. We then investigate two major functions in our **Activity Detection App**, which correspond to *Activity Case 1* and *Activity Case 2* in Figure 3. These functions have a high number of memory accesses compared to context switches. Finally, we design a **Quicksort App:** an application that runs the quicksort algorithm with a high number of memory accesses and no context switches. Each application was run 200 times and a hardware timer on the MSP430FR5969 MCU was used to measure the time of each iteration (with a precision of 16 cycles).

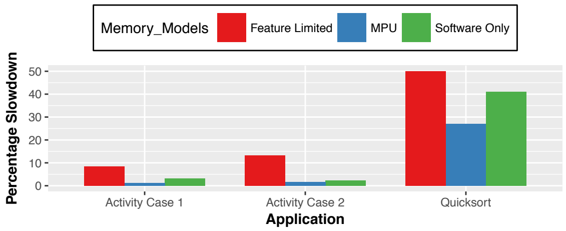


Figure 3: Percentage slowdown for each memory isolation method calculated by comparing them to running apps with no isolation method.

The results from the synthetic app test in Table 1 show that our MPU method had the fastest memory accesses, but the slowest context switches. This result was expected, and validates the simulation results, as our method only requires half the number of bounds checks as the Software Only approach, but incurs extra overhead for reconfiguring the MPU during context switches. Figure 3 further confirms the results from Table 1, which is that our method is the most effective when used for computationally heavy applications.

5 Discussion and Conclusion

In this paper we explore the challenge of memory isolation on ultra-low-power microcontrollers, which offer primitive hardware support for memory protection. Traditional approaches use a range of language limitations, compiler analysis, or dynamic checks (inserted by compiler or other tools); few have leveraged the capabilities of emerging MPUs.

Our solution employs MPU hardware to protect most regions of memory from inappropriate access by application code. Our proof-of-concept implementation (on an Amulet) is limited by the capabilities of the MSP430 MPU, which cannot protect the region below the current app's allocation; thus, the compiler still needs to insert some code for bounds checks – albeit half as many as in the software-only solution. We envision extending our approach to work with more advanced MPUs to further reduce our runtime overheads; MPUs that can protect *all* of memory and support 4 or more regions would negate the need for our compiler-inserted bounds checks. We may also explore more robust error handling techniques, such as restart policies for applications that trigger a memory access fault, or the use of a shadow return-address stack to prevent applications from jumping outside their code bounds.

In conclusion, our exploration shows that (1) it is possible to efficiently support memory isolation without resorting to language limitations, as in the original Amulet approach, and (2) a hybrid approach that leverages compiler-inserted code and MPU-hardware support can provide performance benefits over a software-only approach. While our approach leveraging the MPU was not effective for apps that make frequent API calls, our MPU isolation approach had, for all applications, less than 0.5% impact on battery lifetime.

Acknowledgements

This research results from a research program at the Institute for Security, Technology, and Society, supported by the National Science Foundation under award numbers CNS-1314281, CNS-1314342, CNS-1619970, and CNS-1619950. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsors.

References

[1] Bhatti, S., Carlson, J., Dai, H., Deng, J., Rose, J., Sheth, A., Shucker, B., Gruenwald, C., Torgerson, A., and Han, R. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications 10*, 4 (Aug. 2005), 563–579.

[2] Boateng, G., and Kotz, D. StressAware: An app for real-time stress monitoring on the Amulet wearable platform. In *Pro-*

ceedings of the IEEE MIT Undergraduate Research Technology Conference (URTC) (Jan. 2017), IEEE.

[3] BOATENG, G. G. ActivityAware: Wearable system for real-time physical activity monitoring among the elderly. Master's thesis, Dartmouth Computer Science, May 2017. Available as Dartmouth Computer Science Technical Report TR2017-824.

[4] CONDIT, J., HARREN, M., ANDERSON, Z., GAY, D., AND NECULA, G. C. Dependent types for low-level programming. In *European Symposium on Programming* (2007), vol. 4421, Springer, pp. 520–535.

[5] COOPRIDER, N., ARCHER, W., EIDE, E., GAY, D., AND REGEHR, J. Efficient memory safety for TinyOS. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)* (2007), ACM, pp. 205–218.

[6] DUNKELS, A., GRONVALL, B., AND VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the Annual IEEE International Conference on Local Computer Networks (LCN)* (2004), pp. 455–462.

[7] 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.

[8] GU, L., AND STANKOVIC, J. A. T-Kernel: providing reliable os support to wireless sensor networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys)* (2006), pp. 1–14.

[9] HAN, C.-C., KUMAR, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. A dynamic operating system for sensor nodes. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2005), ACM, pp. 163–176.

[10] HESTER, J., PETERS, T., YUN, T., PETERSON, R., SKINNER, J., GOLLA, B., STORER, K., HEARNDON, S., FREEMAN, K., LORD, S., HALTER, R., KOTZ, D., AND SORBER, J. Amulet: An energy-efficient, multi-application wearable platform. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (Nov. 2016), ACM Press, pp. 216–229.

[11] KOEBERL, P., SCHULZ, S., SADEGHI, A.-R., AND VARADHARAJAN, V. TrustLite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)* (2014), ACM.

[12] KUMAR, R., KOHLER, E., AND SRIVASTAVA, M. Harbor: software-based memory protection for sensor nodes. In *Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)* (2007), ACM, pp. 340–349.

[13] LEVIS, P., AND CULLER, D. MatÉ: A tiny virtual machine for sensor networks. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2002), ACM, pp. 85–95.

[14] LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND OTHERS. Tinyos: An operating system for sensor networks. *Ambient intelligence 35* (2005), 115–148.

[15] LEVIS, P., PATEL, N., CULLER, D., AND SHENKER, S. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. 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. 2–2.

[16] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, D., AND LEVIS, P. Multiprogramming a 64 kb computer safely and efficiently. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2017), ACM.

[17] SANT'ANNA, F., RODRIGUEZ, N., IERUSALIMSCHY, R., LANDSIEDEL, O., AND TSIGAS, P. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)* (2013), ACM.

[18] TEXAS INSTRUMENTS. Msp430fr5969 16 mhz ultra-low-power microcontroller. `http://ti.com/product/MSP430FR5969`, Oct. 2017.