# Amulet: An Energy-Efficient, Multi-Application Wearable Platform

Josiah Hester*, Travis Peters†, Tianlong Yun†, Ronald Peterson†, Joseph Skinner†,
Bhargav Golla*, Kevin Storer*, Steven Hearndon*, Kevin Freeman*,
Sarah Lord†, Ryan Halter†, David Kotz†, Jacob Sorber*

*Clemson University and †Dartmouth College

## ABSTRACT

Wearable technology enables a range of exciting new applications in health, commerce, and beyond. For many important applications, wearables must have battery life measured in weeks or months, not hours and days as in most current devices. Our vision of wearable platforms aims for long battery life but with the flexibility and security to support multiple applications. To achieve long battery life with a workload comprising apps from multiple developers, these platforms must have robust mechanisms for app isolation and developer tools for optimizing resource usage.

We introduce the Amulet Platform for constrained wearable devices, which includes an ultra-low-power hardware architecture and a companion software framework, including a highly efficient event-driven programming model, low-power operating system, and developer tools for profiling ultra-low-power applications at compile time. We present the design and evaluation of our prototype Amulet hardware and software, and show how the framework enables developers to write energy-efficient applications. Our prototype has *battery lifetime lasting weeks or even months*, depending on the application, and our interactive resource-profiling tool predicts battery lifetime within 6–10% of the measured lifetime.

## CCS Concepts

•**Computer systems organization** → **Architectures;** •**Human-centered computing** → *Ubiquitous and mobile computing systems and tools;* Interaction paradigms;

## Keywords

Wearables, Mobile health, Energy, Low power

## 1. INTRODUCTION

Wearable wristbands are increasingly popular devices for health and fitness sensing, and the increasing variety of applications is driving the market from single-function devices (like the Fitbit Flex) toward multi-application platforms (like the Apple Watch or Pebble Time).



*SenSys '16, November 14 - 16, 2016, Stanford, CA, USA*



ISBN 978-1-4503-4263-6/16/11…$15.00

DOI: http://dx.doi.org/10.1145/2994551.2994554


**Figure 1: Perspective and interior views of our open-hardware wearable device, part of the open-source Amulet Platform. The platform supports development of energy-efficient, body-area-network sensing applications on multi-application wearable devices.**

These devices enable new sensing paradigms; they are worn continuously, they can interact through a body-area network with computers, smart phones, and other wearables, and they can provide at-a-glance information to the wearer. Some existing devices are flexible and full-featured, with supportive development environments, but have inadequate battery life (about a day). Most others are single-purpose devices with better battery life that users cannot easily reprogram or customize for different applications and conditions.

Although the line between "smartband" and "smartwatch" products is blurring, we think of the former as having great battery life but limited flexibility, and the latter as having programmability but limited battery life. Battery life is a critical feature for mobile and wearable devices – by far the most-important feature as rated by users of today's smartphones and wearable gadgets [9, 42]. We aim to enable devices that have the week-long or month-long battery lifetimes of a smartband with the multi-application flexibility and full-featured development environment of a smartwatch.

To support multiple applications, especially in critical domains like health, the platform must also provide strong security properties, including isolation between apps. To realize these goals, wearables must effectively manage energy, share resources, and isolate applications on low-power microcontrollers that cannot support hardware memory management units (MMUs).

In this paper, we present Amulet, a hardware and software platform for developing energy- and resource-efficient applications on multi-application wearable devices. This platform, which includes the Amulet Firmware Toolchain, the Amulet-OS Runtime, the ARP-View graphical tool, and open reference hardware, efficiently protects applications from each other without MMU support, allows developers to interactively explore how their implementation decisions impact battery life without the need for hardware modeling and additional software development, and represents a new approach to developing long-lived wearable applications.

We also aim to equip the health-behavior science community with a wearable platform researchers can field for long-duration experiments on human subjects in a wide variety of studies, by providing the entire Amulet Platform as an open-source, open-hardware alternative to the available commercial platforms that have so far been used for wearables research. We envision the Amulet Platform as being broadly applicable to those in the sensing communities, as well as domain scientists and practitioners in human-centered fields like health and fitness. With the Amulet Platform, sensor researchers can prototype new wearable devices and test new sensing technology without building from scratch.

**Contributions:** The Amulet Platform enables developers to write energy- and memory-efficient sensing applications that achieve long battery life on a secure, open-source, multi-application wearable device. Specifically, this paper makes four contributions:

1. the Amulet Firmware Toolchain, a firmware-production toolchain that guarantees application isolation (protecting the system and applications from errant apps),
2. the Amulet-OS Runtime, a multi-application runtime system for resource-constrained wearables that is built on a low-power variant of the QP runtime [24],
3. a graphical tool called ARP-View that helps developers predict Amulet battery lifetimes and understand how their decisions affect those lifetimes, and
4. an open-source, open-hardware release of the Amulet platform and its tools[1].

Although the focus of our current implementation is on a smartwatch form factor and on applications related to health and wellness, these contributions can be generalized to any embedded platform that needs to support multiple third-party applications with extremely low power consumption. Indeed, Amulet is not a single system – it is a novel approach to developing long-lived wearable platforms that is suitable for a wide spectrum of multi-application wearable devices.

## 2. BACKGROUND

Although the Amulet Platform has potential to support a broad range of applications, we focus our design on mobile health (mHealth) applications because they are increasingly prevalent and their need for a robust, secure, long-lived platform poses important design challenges. The current generation of mHealth wearables, such as the Fitbit Flex and the Withings Pulse, are single-application devices that focus on specific health goals like physical activity or sleep quality. These devices run one application, created by the device developers; they cannot run multiple applications nor be extended with applications from third-party developers. Meanwhile, "smartwatches" like the Apple Watch and the Microsoft Band are general-purpose wrist wearables that support multiple applications and third-party developers. Neither class of devices address our goals, for several reasons.

First, we are not convinced that all users want a general-purpose large-screen smartwatch with a battery life measured in hours. Our architecture aims to enable smaller wristbands (and other constrained wearables) with battery lifetimes measured in weeks or months and support for critical and sensitive applications like those related to chronic disease and behavioral health.

Second, developer tools for these wearables are in their infancy. Battery lifetime (i.e., amount of time between battery charges) is a critical concern for any wearable; although some developer frameworks provide general guidelines for writing efficient applications, developers are unable to accurately predict how their applications will perform when deployed. The Amulet Platform includes tools that forecast battery lifetimes and an application's resource usage. More importantly, these tools help developers conceptualize how their design decisions impact energy consumption and identify specific opportunities for improvement.

Third, current solutions do not provide open-source hardware and software; the Amulet Platform is fully open-source and open-hardware, enabling new opportunities for innovation by health and technology researchers alike.

## 3. SYSTEM OVERVIEW

We designed Amulet to support a multi-developer, multi-application vision, aiming at four goals not faced by single-purpose wearables, single-developer wearables, or power-hungry platforms that need to be recharged daily.

**Goal 1: Multiple applications.** *Amulet platforms enable sensing applications written by third-party developers,* even on resource-constrained wearable devices. The Amulet Platform masks the complexity of embedded-system development, and supports a variety of internal and external sensors, actuators, and user-interface elements. Since users are unlikely to wear multiple single-function devices, the Amulet Platform aims to support multiple concurrent applications.

**Goal 2: Application isolation.** *Amulet platforms isolate applications from each other and from the system.* With multiple concurrent applications, sensitive user information must be protected and applications must be prevented from interfering with the system or other applications. Amulet uses creative compile-time and run-time isolation mechanisms to achieve these properties on ultra-low-power microcontrollers that do not provide memory virtualization or memory protection. In this paper we focus on memory isolation and resource management; later papers will address encryption, key management, and related security aspects.

**Goal 3: Long battery life.** *Amulet platforms enable wearable devices with battery life measured in weeks.* Today's

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

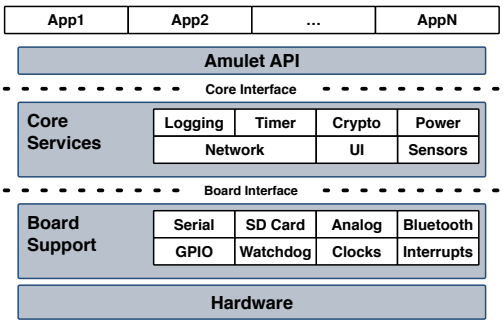


**Figure 2: The Amulet-OS software stack; applications access core services through the Amulet API layer.**

multi-application wearable devices have poor battery life, including research devices like ZOE [23] and commercially significant devices like the Apple Watch [7]. Even the longest-lived commercial devices, like the Pebble [36], have lifetimes measured in days. When wearables can run for weeks or months, new applications are enabled and users are likely to benefit from apps that support long-term 24/7 health monitoring and interventional behavior change.

**Goal 4: Resource-usage prediction.** *Amulet platforms include tools that provide interactive analysis of resource usage*, including energy impact and memory usage for applications and the underlying system. Existing tools for third-party application developers on wearable platforms are very limited, focused on documenting best practices and measuring resource usage of running applications; they do not provide compile-time, app-developer tools for predicting the battery impact of an app or combination of applications.

We next describe our initial implementation, focusing on the above goals in two parts: the Amulet-OS and the Amulet Firmware Toolchain (AFT).

## 3.1 Amulet-OS

The Amulet-OS software architecture (Figure 2) achieves all the above goals by providing a low-power, event-driven programming model, a rich API, and efficient app isolation and optimization through compile-time techniques.

**Event-driven programming:** Many sensing-based apps, including the health-oriented apps that motivate our work, tend to remain idle waiting for user interaction or new sensor data. Thus, Amulet uses an event-driven programming model to simplify developer tasks and enable low-power operation. Each application is represented as a state machine with memory, that is, each app consists of a set of *states* and *transitions* between states, and a small set of persistent *variables*. Each transition is triggered by the arrival of an *event*, which themselves result from expired timers, user interactions like a button press, or data arriving from internal and external sensors. Apps can specify optional *event handlers* for each state and each event type. Handlers are non-blocking functions that may consume data arriving with the event, update app variables, call Amulet APIs, or send events, in any combination.

This state-machine approach makes app state explicit, easing analysis and optimization. App code, state, and variables are kept in persistent storage. Handlers run to completion, so there are no threads with stack-based state information to preserve between events, let alone across processor reboots. The Amulet-OS leverages this simplicity for deep power savings; when there are no events to handle, the processor can go into deep sleep or even shut off. The Amulet Firmware Toolchain leverages this structure to enable the analysis and profiling tools described in the next section. This approach is also a major advantage over the alternative of running a larger operating system, such as embedded Linux or a real-time OS, in which applications are represented as processes or threads with complex state and limited opportunity for deep sleep. (For more info, see Section 8.)

**Amulet API:** Amulet-OS provides an application programmer interface (API) that allows for sensing, storage, signal processing, communication, timing, and user interaction. As with any OS, this API provides abstraction and resource management. Amulet-OS simplifies data gathering by providing applications the ability to *subscribe* to internal and external sensors; multiple applications can share a single data stream, each receiving an event when new data arrives. Amulet-OS also includes a logging API so apps can log sensor information to files on an internal microSD card, and a timer API so apps can arrange for an event in the future. Finally, the API provides apps access to interface elements (display, LEDs, buzzer, buttons, and capacitive touch, in our implementation) and multiplexes access across apps. These APIs call into the Amulet-OS core services shown in Figure 2. All such calls are non-blocking because event handlers must run to completion; where needed, a response is delivered to the app later as an event.

The Amulet-OS is more of a lightweight run-time system than an operating system, but nonetheless supports multiple applications on a low-power microcontroller without memory protection or management. These challenges are the focus of the Amulet Firmware Toolchain.

## 3.2 Amulet Firmware Toolchain

The Amulet Firmware Toolchain (AFT), shown in Figure 3, manages the analysis, translation, and compilation of firmware. Here, we focus on two critical AFT roles: application isolation and resource profiling. First, the AFT ensures that apps can only access Amulet hardware by sending a well-formed request to the Amulet-OS core via the Amulet API, and prevents malicious or buggy apps from reading or modifying the memory of either the OS or another app. Second, with the AFT's profiling tools an app developer can predict her app's resource usage.

**Analysis and Translation:** Our approach leverages compiler-based translation and static analysis: application developers pass their code to the Amulet Firmware Toolchain, which translates and analyzes the source code, rejecting any code that is either not well formed or violates the isolation principle. App designers implement Amulet state machines using a simple variant of the C programming language, "Amulet C," which offers programmers familiar programming constructs and facilitates efficient code generation, while excluding many of C's riskier features (Section 5). These modifications, and the addition of loop invariants and automatic annotations by the AFT, allow rigorous analysis of an application's memory safety. The AFT uses static-analysis tools to examine the code to identify memory-safety violations and present the

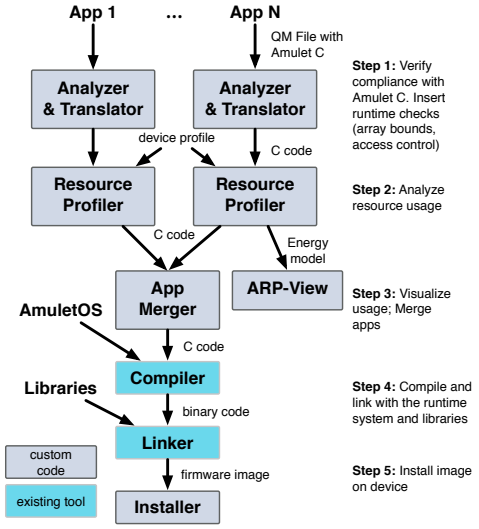


**Figure 3: Architecture of the Amulet Firmware Toolchain, showing the steps in producing the firmware image for a given Amulet device.**

developer with compile-time errors. The AFT inserts run-time validation code where static analysis is inconclusive; for example, the AFT inserts code for array-bounds checking when the array index is not computable at compile time.

The Translator also implements authorization policies that define which applications can access which resources. For example, these policies may say that a fall-detecting app may subscribe to data from the accelerometer, that an emergency-response app may write a file to the SD card, or that the EMA app may use the buttons and the display. By addressing these policies at the time of app translation and analysis, the AFT can flag illegal access at compile time, and insert run-time checks only when needed. The resulting code is smaller, faster, and safer. We omit description of the authorization policy module in this paper, for lack of space.

**Resource Profiler:** Although our experiments in Section 6 characterize the performance of the Amulet hardware and software under a representative app workload, and those results show it is possible to develop efficient applications for the Amulet hardware, new app developers may not have access to the inner Amulet hardware or to our measurement infrastructure. With effective compile-time tools that predict an app's resource usage, developers can effectively manage critical resources and protect the user's experience even when combining apps from many developers.

The Amulet Resource Profiler (ARP) tool leverages the other modules in the Amulet Firmware Toolchain to predict an application's resource usage; here, we focus on memory and energy. Profiling these resources gives insight into the app's impact on battery lifetime, and the app's impact on constrained, shared, device computing resources. Lifetime is the most important consideration for a wearable device; thus, giving developers insight on how their design decisions affect lifetime is of great value to the developer and the user. ARP captures information about each app's code space and memory requirements, using a combination of

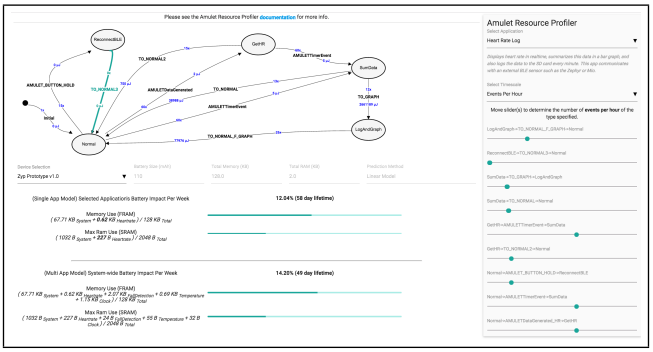


**Figure 4: Screenshot of the ARP-View tool. A selected application is rendered on the screen where the transitions have been annotated with both the estimated cost of executing that transition and the rate at which the transition executes. The sliders along the right-hand side of the window allow the developer to adjust the rate of specific transitions. The bottom area of the screen displays information about an application's impact on the overall battery lifetime per week and memory usage. To view a full-size interactive demo of ARP-View, please visit `https://arpview.herokuapp.com/`.**

compiler tools and static analysis. To profile energy, ARP builds a parameterized model of the application's energy consumption, as described in the next section. The results are then exported for use by ARP-View.

**ARP-View.** This interactive tool presents developers a graphical view of the resource profile and sliders that allow them to immediately see the battery-life impact when they adjust application parameters. The tool guides developers towards a better understanding of their application and enables them to explore trade-offs of different design decisions. See Figure 4 for a snapshot of the ARP-View in action.[2]

In ARP-View, developers are presented with an annotated visualization of their application that resembles their original finite-state machine. Annotations on edges (representing state transitions) provide immediate feedback about the rate in which that event handler executes and the energy usage of that particular event handler. In addition, the ARP-View consists of a set of 'knobs' that can be individually scaled to simulate a particular handler running more/less frequently ($R_t$); this allows developers to evaluate the impact of specific application activities on the overall battery lifetime.

For perspective, ARP-View imports the device profile for information about the device (such as battery and memory capacity) obtained earlier by the Profiler; see the next section for more details.

**Merge, Compile, Link:** The next step in producing a final firmware image, referring back to Figure 3, is for the AFT to merge the desired set of applications, then compile and link them with the Amulet-OS code. For Amulet we leverage an event-driven programming framework (QP) that automatically generates the C code for state transitions from each app's state diagram. The AFT Merger combines this C code with the translated application code (now in C), and the

[2]We acknowledge that this snapshot may be too small to read without zooming. Readers are encouraged to visit our interactive demo.

**Table 1: Model notation**

| | |
|---|---|
| **Device profile** | |
| $M_d$ | storage capacity of data memory |
| $M_c$ | storage capacity of code memory |
| $E_B$ | energy capacity of full battery |
| $E_\ell$ | average energy consumed by a line $\ell$ |
| $E_f$ | average energy consumed by one call to API function $f$ |
| $P_0$ | baseline power draw |
| $P_s$ | average power draw for subscription to sensor $s$ |
| $S$ | the set of all sensors on this device |
| $F$ | the set of all API functions available in Amulet-OS |
| **Application parameters** | |
| $A$ | the set of all applications on this device |
| $a$ | an application in the set of all applications; $a \in A$ |
| $f$ | a function in the set of all Amulet API functions; $f \in F$ |
| $F_a$ | the set of all API functions used by application $a$; $F_a \subseteq F$ |
| $s$ | a sensor in the set of all sensors; $s \in S$ |
| $S_a$ | the set of sensors used by application $a$; $S_a \subseteq S$ |
| $t$ | transition $t$ in the set of all application transitions, $T$ |
| $T_a$ | the set of all transitions in application $a$; $T_a \subseteq T$ |
| $L$ | the set of all lines of code outside Amulet-OS |
| $\ell$ | line of code in the set of all lines of code; $\ell \in L$ |
| $N_{\ell,t}$ | number of times line $\ell$ is executed by transition $t$ |
| $N_{f,t}$ | number of times function $f$ is called in transition $t$ |
| $R_t$ | rate transition $t$ executes (transitions per second) |
| **Energy estimates** | |
| $E_t$ | estimated energy of each occurrence of transition $t$ |
| $E_a$ | estimated energy consumed by application $a$ |
| $E_A$ | estimated energy consumed by set of applications $A$ |

underlying event-driven application framework, into a single C file. This file is compiled and linked with the Amulet-OS code, incorporating only the system components needed for this particular set of applications. (We anticipate users will select apps for their personal Amulet device from a store hosting apps from many developers and which compiles a custom firmware image comprising their selected apps.)

# 4. RESOURCE MODEL

As noted above, the Amulet Resource Profiler (ARP) constructs a predictive energy and memory model for each application. In doing so, the ARP proceeds in four phases, with reference to the notation in Table 1.

**Phase I: Import device profile.** The ARP imports a *device profile*, specific to the target Amulet model but independent of any particular application. The device profile lists the amount of energy consumed for each API call and for other fundamental operations, based on empirical measurements collected earlier on a given hardware and system software configuration; a device profile would be prepared and provided by the Amulet manufacturer with each new hardware and Amulet-OS release. (The AFT can assist by automatically producing the code to create this profile by generating a specially instrumented *Amulet Device Profiler app* that exhaustively tests each of the Amulet API functions that draw significant amounts of energy, for example, sampling the Gyro, writing to the SD card, or turning on the radio. Using simple monitoring hardware, Amulet manufacturers can gather these statistics once and distribute the profile to application developers for their own testing. Similar to current Android manufacturer practice [4].)

The device profile includes information about the device capacity (memory, battery) as well as empirically derived measures of average energy consumed $E_f$ for each Amulet API function $f$; the average energy consumed $E_\ell$ for executing a specific line of C code $\ell$; and the average power draw $P_s$ for a subscription to sensor $s$ (see Table 1). Energy is measured in joules (J); power draw is measured in watts (J/s).

**Phase II: Analyze code.** The ARP examines the application's state diagram – a graph in which nodes represent states and directed edges represent transitions from one state to another. The result is a set of transitions $T_a$ for application $a$. For each transition $t \in T_a$ the ARP identifies all non-system code executed when transition $t$ occurs (using static analysis to count the number of executions $N_{\ell,t}$ of each line of code $\ell$, summing across loop iterations and recursively examining code in helper and library functions). That is, $N_{\ell,t}$ counts the number of times line $\ell$ will be *executed* during the handling of transition $t$, accounting for loops and function calls.[3] Similarly, for each transition $t$ the ARP determines the number of times the code for transition $t$ will invoke each Amulet API function $f$, which we denote $N_{f,t}$. Finally, the ARP examines the sensor-related API calls to identify the set of sensors $S_a$ to which application $a$ subscribes. The constraints of Amulet C (no recursion, no pointers, no dynamic memory allocation) make this static analysis feasible.

**Phase III: Construct model.** The ARP constructs a parameterized model of the total energy cost for the app. For each transition $t$, it estimates the average energy consumed $E_t$ for an occurrence of that state transition, incorporating the cost of executing the code and API calls in that transition:

$$E_t = \sum_{\ell \in L} N_{\ell,t} E_\ell \; + \; \sum_{f \in F} N_{f,t} E_f \qquad (1)$$

If the app subscribes to any sensors to feed it sensor data, we must also account for their average power draw: $\sum_{s \in S_a} P_s$.

Finally, the Amulet hardware incurs a baseline power draw when it is inactive; we use $P_0$ to represent the average power draw of the baseline system (the microcontrollers, the display, and the input devices). Because Amulet-OS has no background activity, this baseline power draw represents all of the Amulet-OS power draw not captured in the above equations.

To estimate the total energy consumption for application $a$, the ARP needs to know how often each transition $t$ will occur. While some of these rates may be discerned from static analysis on the code, for others the ARP needs advice from the developer – which the developer provides through annotations on the app's state machine. These rates $R_t$ are the 'knobs' for the energy model – knobs the developer can tweak to explore the power draw for various design options (for example, the period of a timer that duty-cycles a key part of the application behavior). The total energy consumed for application $a$, over a time period $\tau$, is thus predicted from the baseline power and the above equations, factoring in the rate of every transition $t$:

$$E_a(\tau) = \tau P_0 + \sum_{s \in S_a} \tau P_s + \sum_{t \in T_a} \tau R_t E_t \qquad (2)$$

Over a week, then, application $a$ consumes a fraction of the total battery capacity, $E_a(\omega)/E_B$, where $\omega = 1$ week; we leverage this calculation in our evaluation below.

**Phase IV: Count memory usage.** Every application

[3]We count lines of code as a proxy for code complexity; to improve accuracy we could use the code generator to count instructions of assembly. Since instruction execution has a relatively small impact on power consumption, our implementation assumes $E_\ell$ to be the same for all lines of code; we focus on modeling the API calls and sensor usage.

requires memory for storage of its code and its data; like any embedded system, low-power wearable platforms have severely limited memory space. After parsing the application's code and generating its firmware image, ARP reports the amount of memory to store the application's code and determines an upper bound on the amount of data memory consumed by the application. These numbers are presented as fractions of $M_c$ and $M_d$. An application's data memory comprises global variables and local variables (on the stack). Amulet C does not allow dynamic memory allocation, so ARP easily counts the size of all global variables; Amulet C does not allow recursion, so ARP can compute the maximum stack depth (including local variables).

**Multiple applications.** The ARP is also capable of estimating the energy consumption for some mix of applications $A$. (The view shows one app's state diagram along with total energy consumption for all apps.) To estimate the total energy consumed by $A$, we cannot simply sum the energy consumed by all of the individual applications. That is, the total is not simply $\sum_a E_a$, because we need to avoid double-counting the baseline power draw as well as the sensor subscriptions (which are shared across all apps). Instead, we need to account for the union of all sensors used by the mix of apps: $S_A = \bigcup_{a \in A} S_a$. The total energy consumed for the application mix $A$ is therefore estimated by a variant of Equation 2:

$$E_A(\tau) = \tau P_0 + \sum_{s \in S_A} \tau P_s + \sum_{a \in A} \sum_{t \in T_a} \tau R_t E_t \qquad (3)$$

# 5. IMPLEMENTATION

We developed a Amulet reference device and implemented the Amulet-OS and Amulet Firmware Toolchain software described above. In this section we describe the details of each, as well as nine applications we wrote to demonstrate and evaluate our Framework.

**Amulet device prototype.** The Amulet wearable prototype shown in Figure 1 is mostly a single-board system. The main board, battery, haptic buzzer, and secondary storage board are all housed in a custom-designed 3D-printed case that fits a standard 22mm off-the-shelf watchband. The hardware architecture of the prototype is shown in Figure 5.

A Texas Instruments (TI) MSP430FR5989 microcontroller with 2 KB of SRAM and 128 KB of integrated FRAM serves as the main computational device. The wearable is equipped with internal sensors for use by developers: an Analog Devices (AD) ADMP510 microphone, an Avago Tech APDS-9008 light sensor, a TI TMP20 temperature sensor, an STMicroelectronics L3GD20H gyroscope and an AD ADXL362 accelerometer. The board includes a Nordic nRF51822 used as a modem for communicating with peripheral BLE devices (such as a heart-rate monitor); the MSP430 communicates with this radio chip over a SPI bus. The board's GPIO and ADC ports connect the microcontroller to two buttons, three capacitive touch sensors, a haptic buzzer, and two LEDs embedded in the case. The small secondary storage board holds a microSD card reader. The board also includes a USB battery charger (MCP73831) that can recharge the 110 mA h battery.[4] A Sharp LS013B7DH03 display with 128x128 resolution is mounted on the backside of the PCB opposite the

[4]A survey of popular smartwatches [39] shows an average battery capacity in today's smartwatches of about 350 mA h with a low of 250 mA h and a high of 570 mA h. Our 110 mA h battery is much smaller.

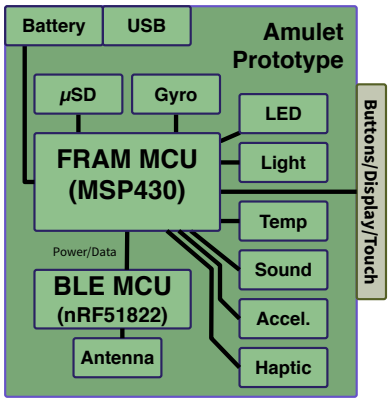


**Figure 5: The hardware architecture of our two-processor Amulet prototype: the MSP430 runs applications, and the nRF51822 manages communication.**

components. In batches of 1000, including all components (except the enclosure and wristband), PCB fabrication, and assembly, **we estimate the cost of a Amulet wearable to be only $54.85**.

**Ultra Low Power Operation:** We use a variety of techniques to achieve low power operation. The MSP430 runs Amulet-OS and is responsible for running applications. The MSP430 spends most of its duty cycle in a low-power deep sleep (3 µA); it wakes from sleep when action is triggered by the user interface, a timer, or inbound message from the BLE radio chip. The MSP430 gates power to high-powered components (like the microSD card, and BLE radio) using on-board MOSFETs. Thus, when these components are not in use by applications, they can be completely "turned off" to reduce power consumption. Always-on components like the display draw only 6 µW.

**Amulet-OS.** We implemented the Amulet-OS run-time system on top of the QP event-driven framework [24]. As shown in Figure 2 the Amulet architecture (and our implementation) has three major layers: 1) a board-support layer, running directly on the hardware and abstracting some of the hardware-dependent nuances; 2) a set of core services that provide core functionality like networking, time, logging, and power management; and 3) a set of application services accessible through a thin set of functions in the Amulet API. The AFT static-analysis tools recognize Amulet API functions and verify an app's authorization to use specific application services (much as Android uses the Manifest file to determine app permissions).

Recalling Section 3.1, the event-driven Amulet-OS has no processes or threads, and no context-switching overhead. Quick interrupt and event handlers allow the system to stay in low-power sleep mode most of the time. For the purposes of our experiments below, and to support the Amulet Device Profiler app, Amulet-OS can to toggle GPIO pins when executing app event handlers, Amulet API functions, or interrupt handlers; our external measurement chassis (Section 6) monitors these pins to derive detailed energy and temporal measurements about application and system modules.

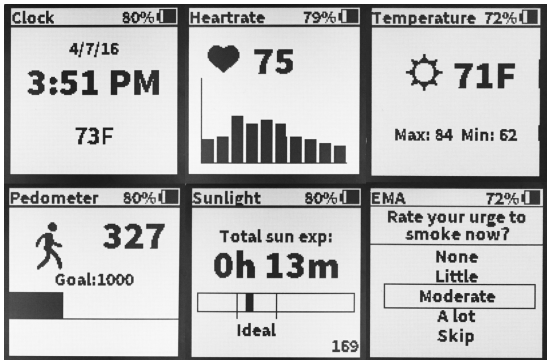


**Figure 6: Screenshots of six of the nine applications presented in this paper.**

**Amulet apps.** We implemented nine applications to demonstrate and evaluate the Amulet Platform. Amulet application developers construct their applications using the QP event-based programming framework. Following QP, each app is defined as a finite-state machine; for each possible state, the app can respond to a set of events by providing a handler function for each type of event. We used vanilla QP with the non-preemptive kernel (version 5.3), and required applications to use the Amulet API to request services from Amulet-OS. Screenshots of six apps are shown in Figure 6.

The QP framework exports the application in the XML-based QM format, which embeds all of the programmer-supplied C code along with information about the application's finite state machine.

**Amulet C.** As mentioned earlier, app isolation is an important goal of the Amulet Platform. To achieve app isolation without the support of memory-management hardware, and without incurring excessive run-time overhead, the AFT conducts most app isolation at compile time. Applications (that is, their handler functions) are written in a custom variant of C that removes many of C's riskier features: access to arbitrary memory locations (pointers), arbitrary control flows (goto statements), recursive function calls, and in-line assembly. Since array access in C is implemented using equivalent pointer operations, we modified the array syntax so that arrays can be passed to functions explicitly 'by reference' (not as pointers). In Amulet C, arrays also have an associated length that allows for run-time bounds checking whenever access behaviors cannot be adequately checked statically. Although this approach imposes some effort on the developer (to adapt their code for Amulet C) it allows us to estimate the runtime of code executed in the state machine, giving tighter bounds on energy predictions made by the Amulet Resource Profiler.

**Amulet Firmware Toolchain.** We implemented the AFT as a series of programs that translate, analyze, validate, and profile apps. Each app includes 1) a state machine, 2) event handlers (written in Amulet C), and 3) attributes specifying the app's global variables. The QP framework combines application information into XML-formatted QM files. AFT tools are written in Java and use its built-in XML libraries to parse the submitted apps. Our tools translate the Amulet C code to safe C code using a modified C grammar and the ANTLR parser generator [6]. Violations against Amulet C coding rules and non-authorized requests to the core API trigger AFT compile-time errors.

After all these steps, applications are merged together in a single QM file, which is then converted to C using QP. Immediately before the merge, all app resources (e.g., variables, handlers, helper functions) are isolated by mapping them to a unique namespace based on the application's name (no two apps installed on the system can have the same name). This code is then compiled and linked using Texas Instrument's open-source GCC for MSP430. This firmware image can then be installed onto the application chip (MSP430) of our Amulet device prototype.

**Resource Profiler.** We implemented the Amulet Resource Profiler (ARP) in Java and integrated it with the other tools of the Amulet Firmware Toolchain (AFT). After validation and translation, the ARP uses an ANTLR-generated parser to extract model parameters from the application's code and QM file. Specifically, from the application code it extracts estimates of lines executed per transition ($N_{\ell,t}$), sensor subscriptions ($S_a$), and Amulet API calls ($F_a$ and $N_{f,t}$). From the QM file it extracts the state machine and its transitions ($T_a$), and the developer's annotations about transition rates ($R_t$). From the firmware's symbol table it extracts the code size for both the application and the core, and the amount of FRAM memory used by the application and the core.

Separately, the ARP uses the Amulet Device Profiler app, and the measurement chassis described in the next section, to extract the parameters for the device profile (Table 1). The Device Profiler app also informs ARP about the scaling factors for certain operations, allowing for fine-grained estimates of function cost (for example, assigning lower energy costs to drawing a 2x2 rectangle as opposed to drawing a 128x128 rectangle). The Resource Profiler combines these measurements to build a parameterized model of the energy cost for each application, following Equation 2. Our prototype hardware has two kinds of internal memory: SRAM used for the execution stack and local data, and FRAM used for code and global data; thus our ARP implementation reports on SRAM and FRAM usage (rather than code and data usage as in Section 4).

**Resource Profiler Developer View.** Our developer-facing tool, ARP-View, leverages the ARP's fine-grained data about the structure and behavior of applications to 1) give developers insight into how certain user actions, sampling rates, and blocks of code consume energy, enabling developers to make concrete the links between certain parts of code and energy draw; and 2) provide meaningful battery-lifetime estimates for an application or suite of applications. The ARP-View currently presents a wealth of information including system and OS level details (e.g., FRAM available and its usage), sliders to adjust event frequency and view results in real-time, and the battery impact (percentage of battery consumed per week) and lifetime in days for a selected application as well as an entire suite of applications. Our implementation provides **real-time** feedback to the developer regarding application impact on the battery by running a daemon process that monitors application files and re-profiles applications upon detecting changes to those files; note that application translation, analysis, validation, and profiling all happens prior to actually compiling the source code and, thus, runs fast even on common laptop and desktop machines. Providing memory stats to the developer via the

**Table 2: Amulet applications used for evaluation**

| Name | Description |
|---|---|
| Clock | Display time of day and temperature |
| Fall Detection | Detect falls using the accelerometer |
| Pedometer | Record number of steps walked |
| Sun Exposure | Monitor exposure to light over time |
| Temperature | Monitor ambient temperature over time |
| Heart-rate | Monitor and display HR from BLE sensor |
| Battery Logger | Monitor, display, and log battery statistics |
| Heart-rate Logger | Monitor, display, and log HR statistics |
| EMA | Deliver surveys to users with touch input |

ARP-View requires compiling all of the application/system code and parsing the resulting binary, which incurs a noticeable – but small – amount of time (approximately 1-2 seconds) on common laptop and desktop machines.

# 6. EVALUATION

In this section we evaluate the performance of Amulet-OS and the Amulet Firmware Toolchain against the goals from Section 3. Goal 1 (Multiple applications) and Goal 2 (Application isolation) are met by design of Amulet-OS and the Amulet Firmware Toolchain. We thus focus on an experimental evaluation of Goal 3 (Long battery life) and Goal 4 (Resource-usage prediction). First, though, we describe the experimental infrastructure that allowed us to collect precise controlled measurements of the Amulet system (and its apps) without incurring measurement artifacts or overhead on the Amulet itself.

**Measurement chassis:** For detailed measurements under controlled conditions, we built a chassis comprising multiple ARM breakout boards equipped with 12-bit DACs and 16-bit ADCs (to control and monitor signals on the prototype Amulet), MOSFETs to gate power to the prototypes (allowing repeatable testing by power cycling), and INA225 auto-ranging current-sense circuits to gather power readings. This programmable chassis allowed us to stimulate the user interface and trigger state changes in the applications under test. Thus, we could automatically and repeatably test the Amulet prototype under controlled and consistent conditions.

**Applications:** We developed nine apps for use in the evaluation of Amulet (Table 2), selected to represent a range of compelling applications and exercise many of the features available in the current prototype. Three of the apps were used in the pilot study described in Section 7.

**Event automation:** We used the measurement chassis along with minor modifications to the core code to partially automate the state transitions of the applications running on our prototype Amulet devices. User sessions can be simulated by providing an interaction script that defines the interaction (specifically the named state transitions), and the delay between these transitions. This method was minimally invasive to the Amulet software and hardware (one dedicated I/O pin and a small portion of RAM). By executing interaction scripts multiple times, the full functionality of an application can be tested repeatably and without human involvement. We refer to this program as the "Event Automator".

## 6.1 Battery lifetime

In this section we quantify the average power draw of our Amulet prototype for multiple loads. We represent this power

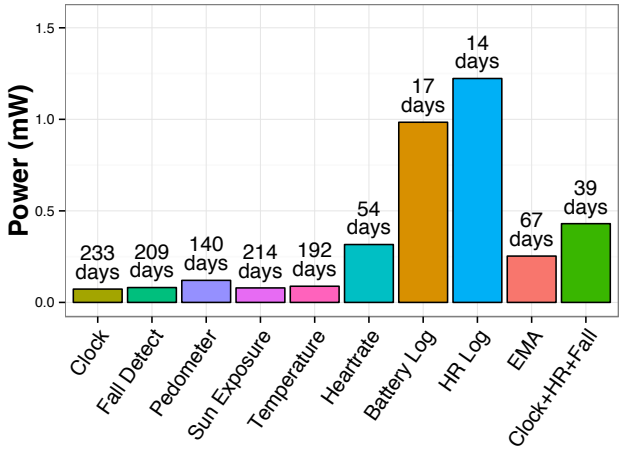


**Figure 7: Average power draw for each app on the current prototype. The expected lifetime with the 110mAh battery is shown above each bar.**

draw in terms of battery lifetime using the 110 mA h battery currently encased in the prototype. (Larger batteries up to 570 mA h are used in current smartwatch products [39].) We acknowledge that the accuracy of battery lifetime estimates depends heavily on battery wear, quality, leakage, and other factors. These estimates serve to place the actual measured power draw in an understandable form.

For our first experiment, we installed a firmware image containing a single application and measured the average power draw using the measurement chassis. We used the Event Automator to emulate user sessions, where a 'session' lasted as long as it took to exercise states in the critical path of an application. We repeated this experiment for each of the nine applications, for a single session, determining average power by summing the energy of the session and dividing by the time. The results of this experiment are shown in Figure 7. This figure shows the power draw of an app was highly dependent on the hardware components used, and the frequency of their use, further motivating the use of ARP-View. Lifetimes for all of the apps exceeded two weeks.

On our current prototype, an application load comprising Clock, Fall Detection, Pedometer, Sun Exposure and Temperature would allow an Amulet battery to last **over four months**. With constant BLE communication to the heart-rate sensor (we used commercial Zephyr and Mio heart-rate sensors) our Amulet's power draw allows for nearly a **two-month** battery life. Two energy-hungry applications (Battery Log and HR Log) used significant amounts of energy logging data to the microSD card, drastically reducing battery life.

To determine the effect multiple applications have on the device lifetime, we assembled a firmware image that included three applications (Clock, Heartrate, and Fall Detection), and measured the steady-state power draw as above. With this configuration, using the accelerometer, Bluetooth communication, and updating the display, the expected lifetime was **39 days**. In summary, an Amulet with applications using on-board sensors will last for many months, an Amulet using an external BLE sensor feeding regular heart-rate data will last for 1-2 months, and applications making heavy use of the logging operations or capacitive touch features will

Table 3: ARP battery-%impact predictor.

| App Name | Obs. ( µW) | Pred. ( µW) | Error ( µW) | Error (%) |
|---|---|---|---|---|
| Clock | 72.73 | 77.45 | 4.72 | 6.1 |
| Fall Detect | 81.20 | 89.21 | 8.01 | 9.0 |
| Pedometer | 120.47 | 129.98 | 9.51 | 6.3 |
| Sun Exposure | 79.19 | 85.13 | 5.94 | 7.0 |
| Temperature | 88.37 | 92.89 | 4.52 | 4.9 |
| Heart rate | 316.16 | 320.34 | 4.18 | 1.3 |
| Battery Log | 984.10 | 1089.63 | 105.53 | 9.7 |
| HR Log | 1223.11 | 1318.68 | 95.57 | 7.2 |
| EMA | 253.07 | 265.05 | 11.98 | 4.5 |
| Multi | 430.01 | 438.86 | 8.85 | 2.0 |

last for a few weeks. This lifetime enables long-term usage for many application domains.

## 6.2 Resource Profiler

Recall that the ARP tool predicts the effect of an app on the battery lifetime of an Amulet device. Specifically, ARP predicts the app's total energy cost per week, which can be used to determine the impact an application has on the battery lifetime. To evaluate the accuracy of these predictions, we compared the ARP battery-impact prediction, for each app, with the actual battery impact computed from the measured average power draw presented in Figure 7.

Each app's event frequencies were set to match the event frequencies used by the Event Automator in collecting the measurements that resulted in in Figure 7. For some apps, specifically Clock, Temperature, and Sun Exposure, these event frequencies were gathered from the developer's code. These apps sense intermittently, on a timer whose value is set by the developer inside the code for a transition. These apps only respond to the timer, not to user or environmental input, so their energy impact depends completely on the timer value. Event frequencies were set carefully for Apps like Pedometer and Fall Detection, but their energy impact was dominated by the Amulet's baseline energy, paired with the number of times they compute over a series of acceleration values. Table 3 presents the predicted (and observed) percent impact on battery life, for each application, along with the percent error in the ARP prediction. (We quantify the error as the difference between expected average power draw and observed average power draw.) The results indicate that our Resource Profiler was reasonably accurate at estimating the battery life for these applications.

Applications can have different battery lifetimes depending on the amount of user interaction, the data rates or sensing schedules, the environment, and of course, the choices the developer makes in implementation. We capture four types of events in ARP: 1) User interaction, 2) Data delivery, 3) Timers, and 4) Programmer defined. While data delivery events are static (sensor subscription schedules are determined beforehand in our current system), each of the other three can be modified by the developer to explore the effect on battery lifetime. The ARP predictions are heavily dependent on the accuracy of the underlying device profile. Small measurement errors in generating the device profile can compound as event frequency increases. ARP has trouble quantifying certain types of operations. For instance, the energy required for microSD card writes is heavily dependent on the size of the write. ARP does not currently account for parameter length in SD writes, contributing to the higher error for applications that use SD functions (Battery Log, HR Log). Nonetheless, in our experiments, the highest error rate was only 9.7%, and we expect it will improve.

However, most developers will not care if their application is predicted to last 90 days and instead lasts 80 or 100 days; they care about how the code they write, and the frequency of events, proportionally affect the lifetime. Of course, further tests 'in the wild', with a wider variety of apps, will be necessary to generalize this conclusion.

## 6.3 ARP-View User Study

We claim that ARP-View helps developers reason about how different parts of their code, and their decisions, map to energy costs of the application. To test this claim, we conducted a preliminary study of 10 computing students (Sophomore to Graduate) who all had some experience with low-power embedded systems.Each subject was asked to consider developing an embedded application that required sensing temperature at an undefined frequency. Subjects were asked to decide how frequently to take temperature readings, and to calculate the fictional application's energy usage given their chosen frequency. To assist in this process, subjects were first provided the QM IDE and a data sheet containing relevant information about the temperature sensor being used. Subsequently, each subject was provided ARP-View, and asked to repeat the same tasks as above. Each subject went through a structured interview at the end of the study.

We found that ARP-View helps developers better understand the relationship between event frequency and energy use. Prior to using ARP-View, only 2 subjects reported considering energy consumption as a determining factor in deciding how many events to use, while 9 in 10 subjects reported accounting for energy in their decision-making process when using ARP-View. Additionally, ARP-View made event-frequency decisions easier for developers: 7 of 10 subjects reported their decision-making process was less difficult when using ARP-View. Further, after using both tools, 8 in 10 subjects indicated they would not consider using the QM IDE and a data sheet to aid in their future application development, if able to use ARP-View instead. We believe these findings indicate there is room for further exploration into developer understanding of resource use on constrained devices, and tools like ARP-View can assist in their reasoning.

## 6.4 Overhead

**Runtime:** Energy consumption is significantly impacted by the fraction of time the application microcontroller (MSP430) is active; the rest of the time it can be in a low-power deep sleep mode. The MSP430 is active whenever it is running application code (handler) or system code (Amulet-OS and QP system code). The latter time is *overhead*, from the application's point of view. To measure this overhead, we instrumented the Amulet-OS to trigger I/O pins whenever it 1) puts the system to sleep, 2) was active and executing Amulet-OS code, and 3) was active and executing application code. We then used our measurement chassis to obtain precise measurements of the time spent in each mode, for each of our applications, as shown in Table 4. For each application listed, we ran three short sessions while the application was conducting its normal duty cycle, but not being triggered by user-interface events. For all apps this meant polling sensors, and waking up for timer events. The

Table 4: Temporal overhead.

| Application | %Sleep | %OS | %App |
|---|---|---|---|
| *Clock* | 98.1 | 0.9 | 1.0 |
| *EMA* | 98.2 | 1.0 | 0.8 |
| *Heart rate* | 91.1 | 0.9 | 8.0 |
| *Pedometer* | 93.8 | 2.2 | 4.0 |
| *Pedometer+HR* | 87.5 | 1.9 | 10.6 |
| *Pedometer+HR+Clock* | 85.4 | 2.8 | 11.8 |

low temporal overhead of 0.9-2.8% confirms the efficiency of our approach.

**Memory:** Amulet-OS uses a portion of the limited memory space available to applications, limiting the quantity and size of apps that can be installed in a single firmware image. In our prototype, applications and the OS must share the limited FRAM memory space (128 KB). For a firmware image comprising five applications that used most of the functionality available, Amulet-OS consumed 55.91 KB of the 128 KB available FRAM code space, while applications consumed 14.48 KB; the OS consumes nearly half of the current FRAM. Meanwhile, Amulet-OS claims 1.078 KB of SRAM, leaving 0.922 KB of SRAM for applications; recall that apps use SRAM only for their execution stack, and only when actively executing an event handler; only one app is active at a time. Moreover, FRAM and SRAM are continuous on the MSP430; we anticipate making larger blocks of FRAM available (to be treated as RAM) to the application.

The memory and runtime overhead of our Amulet implementation – while sufficient to develop multiple interesting apps on constrained hardware – could be improved. Memory limitations could be sidestepped by adding 256KB of external FRAM on a secondary storage board to "swap" applications. We expect runtime overhead to further improve as we tune the display driver, which dominates the system overhead.

# 7. PILOT STUDY: MHEALTH

We conducted a preliminary pilot study with the Ecological Momentary Assessment (EMA) application in collaboration with our School of Medicine.EMA tools are often used in behavioral medicine research. These tools allow participants to report on symptoms, affect, behavior, and experiences at the time of action, and in the participants' natural environment. This preliminary study was intended to evaluate user acceptance of the Amulet wearable, and technical feasibility of multi-source data collection with an mHealth EMA application. This type of application can be used in both clinical and research settings for tracking stress, cravings, sleeping and eating habits, and other events in a non-intrusive way.

An EMA tool needs to be non-invasive to the users' daily routine. A wrist-worn EMA device is better positioned than a mobile phone to facilitate a quick, unobtrusive response to an EMA prompt. Moreover, a wearable with long battery life allows for continuous, un-interrupted monitoring, does not burden the user with the need to recharge the device, and is less likely to be left behind or misplaced.

For this preliminary study, we recruited six participants (medical students) to simulate the behavior and user interactions of the target population (habitual cigarette smokers over the age of 18). Each participant wore the Amulet prototype and an external heart-rate sensor (a Mio LINK) for 5–10 hours of a normal workday. We loaded their Amulet with the Clock, Heart-rate Logger, and EMA applications. The EMA application asked participants about their tobacco use at random intervals during the day, prompting them to answer a few multiple-choice questions using the buttons and capacitive-touch slider. A screen shot of the EMA app starting one of these surveys is shown in Figure 6. Participants returned both devices at the end of the day, and then filled out a short usability survey.

Table 5: Data gathered during pilot deployment.

| | Packets received | Battery drain per hour | Duration worn | Mio BLE disconnect |
|---|---|---|---|---|
| P1 | 13,381 | 3.2% | 9.3 hr | 62.0 min |
| P2 | 15,526 | 2.1% | 9.6 hr | 0.2 min |
| P3 | 10,267 | 2.0% | 6.6 hr | 0.3 min |
| P4 | 10,893 | 1.9% | 7.3 hr | 0.2 min |
| P5 | 12,709 | 2.4% | 8.7 hr | 42.5 min |
| P6 | 8,645 | 1.7% | 5.4 hr | 0.0 min |

**Data Gathered:** we logged EMA responses, heart-rate data from the Mio LINK HRM sensor, and battery-status analytics to the microSD card throughout the study. Amulet prototypes were worn for a total of **47.8 hours**, and gathered over **71,421 heart-rate readings**. Table 5 shows statistics for each participant. The amount of time the Mio was disconnected correlates strongly with the amount of energy used; whenever a connection was lost, the Amulet prototype goes into a high-power discovery mode attempting to re-establish connection. The battery drain was higher than expected; we identified after the study that the capacitive touch sensors were drawing a continuous 1400 µW during the deployment, reducing the battery much faster than expected. These sensors were not exposed to the developer API or ARP at the time of development. After the pilot, we added API functions to allow developers to enable and disable the touch sensors. As these user-input elements are only needed when the user is presented with EMA questions, this duty cycling will significantly increase lifetime. (We used this newer EMA application for Figure 7.)

**Usability Results:** Each participant filled out the System Usability Survey after their session [12]. The participants all thought the Amulet prototype was a bit bulky and uncomfortable, but a majority did not consider it a nuisance and thought it was easy to use, while a plurality noted that they learned something about themselves by using the sensor. The biggest problem identified was the small size and difficult placement of the buttons. All of the participants enjoyed the real-time feedback about their heart rate. We have since addressed these usability issues with a smaller, more comfortable case (and larger buttons).

This preliminary deployment demonstrated the feasibility of the Amulet prototype for EMA, and for continuous monitoring applications on human subjects. Indeed, we now plan further studies with more subjects and for longer durations.

# 8. RELATED WORK

The collection of software and hardware techniques in Amulet draw extensively from the wireless sensor network literature. In this section we address related work in software architectures for sensor-focused devices and other constrained devices (including wearables); approaches for isolating the

execution of application code; and techniques for modeling an application's energy and resource usage.

**Open wearable platforms.** Current commercial platforms such as Pebble, Android Wear, and Apple watchOS only document best practices and measure resource usage of running applications [3, 8, 35], and are closed source, closed hardware, or both. However, many open platforms have been developed concurrently with Amulet as the wearable hardware ecosystem has evolved; many are not yet available to users or developers, or do not address one or more of the Amulet goals. BLOCKS [43], ZWear [48], Angel Sensor [5] and Sony's Open SmartWatch Project [40] are all in development. Little information about their developer tools are available; none of them appear to provide compile-time, app-developer tools for predicting battery life for a given app or combination of apps, and none are engineered to give battery lifetime measured in months. Hexiwear [19] is completely open source and open hardware built for the mbed platform. However, it lacks comprehensive developer tools that allow application isolation and evaluation of energy costs. Hexiwear was not built specifically for low power operation like Amulet. The choice of high powered components like a color OLED screen and ARM Cortex-M4 make lifetimes significantly less than the Amulet.

Wearable platform such as ZOE [23] Mercury [27] and Opo [20] have been designed by the research community to address specific sensing problems, or to explore specific research areas (like BodyScan [16]).

In contrast, Amulet provides open-source hardware and software in a general platform allowing wearable system designers to innovate at both the operating system and application level – importantly, Amulet provides a tool that gives application developers insight into the tradeoffs between energy and utility. The Amulet Firmware Toolchain and Resource Profiler provide novel capabilities essential to app development for long-running multi-application multi-developer wearables.

**Software architectures.** Software architectures and operating systems exist for sensor networks and the Internet of Things, including TinyOS [25], Contiki [15], RIOT OS [10], and mbed OS [29]. SafeTinyOS [14] allows for the static analysis of code to improve reliability and programmer confidence in a solution for single TinyOS applications. Amulet uses similar static analysis techniques to provide application isolation and energy prediction for *multiple* applications. All but RIOT OS are event-driven (like Amulet), and only one (mbed OS) provides application isolation and access control. The mbed OS provides app isolation using the target processor's Memory Protection Unit (MPU) – a capability not available on the ultra-low-power microcontrollers targeted by Amulet.

Amulet's architecture combines high- and low-power components to achieve our lifetime, performance, and availability goals. This design is similar to other hierarchical power-management systems [1, 11, 41] that seek to provide resource-rich computing platforms with wide dynamic power ranges, by combining a hierarchy of functional "tiers" into a single integrated platform. None of the other IoT frameworks in development appear to offer the multi-application and app-isolation features of Amulet.

**Application isolation.** Traditionally, hardware memory management units (MMUs) prevent applications from interfering with each other and with core system functions. Software fault-isolation techniques extend the MMU by isolating malicious code and been implemented for x86 and ARM architectures [37, 46, 47] or require hardware (MMU) support. These approaches require hardware (MMU) support not available on low power processors, and incur significant runtime overhead. They are not tenable on the constrained hardware platforms that enabled long-lived wearable deployment.

Language-based techniques provide a more efficient alternative, by changing the programming model [18, 21] to make dangerous actions impossible or easy to detect, and using a combination of compile-time static analysis and inserted run-time checks to detect the dangerous actions that remain [13, 14]. The Amulet architecture builds on these techniques, with a new focus on safely combining multiple applications without hardware memory-protection support and providing insight into each apps' share of system resources.

**Energy and resource modeling.** Previous research on energy modeling for constrained devices has focused on techniques for smartphones and wireless sensors. Tools like Sandra [31], eDoctor [28], Eprof [34], Carat [33] and PowerForecaster [30] help users make decisions about energy efficiency at install time, observe the effect of use from day to day, or diagnose abnormal battery drain. Other research has tried to identify how users and batteries interact—investigating the battery mental model of users [17]. These works focus on meeting user needs while ARP-View is focused on enabling the developer. Further, these systems are all in-situ; gathering information about usage and then presenting information to the user. However, integrating these tools with ARP-view could provide interesting avenues for future work.

Developer-focused tools have also emerged for smartphones. Tools have focused on identifying energy bugs at compile time [44], in-situ energy metering using kernel additions for energy model building [45], and then using those models to predict smartphone lifetime [22]. These tools, if carefully applied to constrained wearable platforms, could complement the developer insights gleaned from the Amulet Resource Profiler and ARP-view.

Other tools profile changes in the frequency and amount of system calls as a proxy for hardware profiling, to estimate changes in application energy efficiency [2]. Some tools estimate energy cost per line of source code [26]. These tools provide insight for the developer into energy usage and efficiency, but are not tuned to the specific needs of wearable development. Wearable applications are more energy constrained than cell phones, and rely on periodic sensing activities for their function. The developer must be able to easily identify energy expensive code segments that are periodically executed. This is why ARP-view exposes timers to the developer, allowing them to manipulate code, and frequency of sensing actions in a GUI, without additional profiling or emulation steps.

Simulation and emulation techniques have attempted to provide forecasts of energy usage, starting with tools from the wireless sensor network literature like Power TOSSIM [38]. WattsOn [32] extends the idea of emulation to provide insights and what-if analysis for Android applications, after running the application in an emulator. WattsOn specifically focuses on empowering developers to understand energy efficiency, and is most closely aligned with the goals of ARP-view. However, WattsOn is phone-focused, and does not account for periodic sensing tasks that are crucial to wearable operation. Nor does it provide real time feedback on code and

duty cycling changes. Wearables require special attention to be paid to periodic sensing tasks, timing and duty cycling, and costs of each API function. The Amulet Resource Profiler seeks to empower wearable developers without requiring specialized hardware knowledge, or costly profiling and emulation, all while accounting for sensing and user-interaction costs.

# 9. DISCUSSION AND FUTURE WORK

The Amulet project (see `http://Amulet-project.org`) is a large, ongoing research effort, and this paper is focused on just one aspect of the Amulet vision and its novel approach to enabling secure co-existence of multiple applications on an ultra-low-power wearable device. There remain some limitations and areas for future work.

**Profiler limitations:** Predicting software energy costs is difficult in any context, and the Amulet Resource Profiler has some limitations. First, it currently assumes that every line of code has the same energy cost $E_\ell$, regardless of complexity (though it does account separately for any function the line calls; each of the system API calls are explicitly measured such that we know the amount of energy consumed by making such a call and thus, do not have to estimate their energy cost). We could improve our analysis tools (and benchmark measurements) to reflect a more nuanced view.

Second, it cannot handle data-dependent loop conditions (all loops are expected to execute a statically-determined maximum number of times) or conditional statements (ARP conservatively assumes both branches are always taken).

Third, it assumes that each sensor has an average power draw $P_s$, although some sensors may have different rates or modes of operation that affect power consumption. Indeed, it assumes that the set of sensor subscriptions does not change over time, whereas some apps may start (or end) sensor subscriptions (triggered by new data or by user input).

Fourth, as applications become more complex, the ARP-view interface becomes much larger, and the amount of parameters increases, which could lead to usability problems, We plan to address these limitations by refining our energy model and analysis tools. Further improvements to ARP-View are planned that take advantage of the wealth of information provided by ARP. By integrating ARP-View with a code editor, developers will be able to get real time updates as they code, and view energy costs per line of code.

**App combinations:** We anticipate that Amulet users will be able to select a set of apps for their personal Amulet device, perhaps from a sort of app store. The app store would compile a custom firmware image comprising their selected apps. When a developer submits her app to the app store, the results of the Resource Profiler are included, which later allows the app store to evaluate a given combination of apps for their impact on Amulet resources and advise the user in battery lifetime impact.

**Applications:** To fully explore the capability of Amulet and its tools, we need to explore a wider variety of applications. Researchers at Worcester Polytechnic Institute (WPI) have developed an app that correlates data from multiple physiological signals in an effort to mathematically determine whether one sensor is faulty (or perhaps tampered). We encourage the community to download our code, fabricate an Amulet device, and write apps.

**Security and privacy:** Wearable devices, especially those focused on mHealth applications, raise important security and privacy questions. Together, Amulet-OS and AFT provide the foundation for securing the system and applications from misbehaving applications, but more effort is needed on issues related to access control, encrypted storage, secure communication, and key management.

# 10. SUMMARY AND CONCLUSIONS

Wearable platforms must give developers insight into how their application (and specifically its usage of system resources) will affect the device's battery lifetime. Currently, developers can not easily determine how an application's sensing regimen will impact battery lifetime, or easily tweak sensing and interaction rates to explore their effect on battery life. Furthermore, today's multi-application wearable platforms have battery life measured in hours and days, whereas many compelling applications (particularly in mHealth) benefit from battery life on the order of weeks. In this paper we present the Amulet Platform, which comprises a toolchain (the Amulet Firmware Toolchain) and runtime (the Amulet-OS) for development of energy- and resource-efficient applications on low-power multi-application wearable devices. Using this framework, and the interactive ARP-View tool, developers are given guarantees on app isolation, insight into the tradeoff between resource usage and app performance, and bounded predictions of the application's effect on battery lifetime.

This paper's contributions include 1) a firmware-production toolchain that guarantees application isolation (protecting the system and applications from errant applications), 2) a multi-application runtime system for resource-constrained wearables, and 3) the ARP-View tool to help developers predict battery lifetimes and understand how their decisions affect those lifetimes, 4) an open-source, open-hardware release of the Amulet platform and its tools;[5] From our evaluation, we found that our Amulet prototype could sustain multiple applications for weeks on a single, small battery, that our resource profiler was accurate to within 6-10% of the actual power draw on our prototype hardware, and that Amulet has great potential as an mHealth research device.

# 11. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Omprakash Gnawali, for their comments. We are grateful to George Boateng for developing the EMA and Heart-rate Logger apps, to Taylor Hardin for the final preparation of the open-source release, and to the whole Amulet team for their advice and input. We are especially grateful to Andrés Molina-Markham for his early guidance of the Amulet design and development.

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, and TC-0910842, and by the Department of Health and Human Services (SHARP program) under award number 90TR0003-01. 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.

[5]The Amulet Platform can be found at `https://github.com/AmuletGroup/amulet-project`

## 12. REFERENCES


[1] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting network interfaces to reduce PC energy usage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 365–380. USENIX Association, 2009.

[2] K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia. The power of system call traces: Predicting the software energy consumption impact of changes. In *Proceedings of Annual International Conference on Computer Science and Software Engineering*, (CASCON), pages 219–233. IBM Corp., 2014. Online at http://portal.acm.org/citation.cfm?id=2735546.

[3] Android. Optimizing performance and battery life. Online at http://developer.android.com/training/wearables/watch-faces/performance.html, visited Dec. 2015.

[4] Android. Power profiles for Android. Online at https://source.android.com/devices/tech/power/index.html, visited Nov. 2015.

[5] Angel. Angel Sensor. Online at http://www.angelsensor.com, visited Dec. 2015.

[6] ANTLR (ANother Tool for Language Recognition). Online at http://www.antlr.org/, visited Mar. 2014.

[7] Apple. Apple Watch. Online at http://www.apple.com/watch/, visited Dec. 2015.

[8] Apple. Energy efficiency and the user experience. Online at https://developer.apple.com/library/prerelease/watchos/documentation/Performance/Conceptual/EnergyGuide-iOS/, visited Dec. 2015.

[9] C. Arthur. Your smartphone's best app? battery life, say 89% of Britons. *The Guardian*, May 2014. Online at http://www.theguardian.com/technology/2014/may/21/your-smartphones-best-app-battery-life-say-89-of-britons.

[10] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, T. Schmidt, et al. RIOT OS: Towards an OS for the Internet of Things (poster). In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013. DOI 10.1109/INFCOMW.2013.6970748.

[11] N. Banerjee, J. Sorber, M. D. Corner, S. Rollins, and D. Ganesan. Triage: balancing energy and quality of service in a microserver. In *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 152–164. ACM, 2007. DOI 10.1145/1247660.1247680.

[12] J. Brooke et al. SUS – a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[13] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor network software faults. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, (SOSP), pages 235–246. ACM, 2009. DOI 10.1145/1629575.1629598.

[14] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for tinyos. In *Proceedings of the International Conference on Embedded Networked Sensor Systems*, (SenSys), pages 205–218. ACM, 2007. DOI 10.1145/1322263.1322283.

[15] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 29–42. ACM, 2006. DOI 10.1145/1182807.1182811.

[16] B. Fang, N. D. Lane, M. Zhang, A. Boran, and F. Kawsar. Bodyscan: Enabling radio-based sensing on wearable devices for contactless activity and vital sign monitoring. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services*, (MobiSys ), pages 97–110. ACM, 2016. DOI 10.1145/2906388.2906411.

[17] D. Ferreira, E. Ferreira, J. Goncalves, V. Kostakos, and A. K. Dey. Revisiting human-battery interaction with an interactive battery interface. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*, (UbiComp), pages 563–572. ACM, 2013. DOI 10.1145/2493432.2493465.

[18] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, 2003. DOI 10.1145/781131.781133.

[19] Hexiwear. Hexiwear open iot development solution. Online at http://www.hexiwear.com/, visited Aug. 2016.

[20] W. Huang, Y. S. Kuo, P. Pannuto, and P. Dutta. Opo: A wearable sensor for capturing high-fidelity face-to-face interactions. In *Proceedings of the ACM Conference on Embedded Network Sensor Systems*, (SenSys), pages 61–75. ACM, 2014. DOI 10.1145/2668332.2668338.

[21] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, pages 275–288. USENIX Association, 2002.

[22] D. Kim, Y. Chon, W. Jung, Y. Kim, and H. Cha. Accurate prediction of available battery time for mobile applications. *ACM Transactions on Embedded Computing Systems*, 15(3), May 2016. DOI 10.1145/2875423.

[23] N. D. Lane, P. Georgiev, C. Mascolo, and Y. Gao. ZOE: A cloud-less dialog-enabled continuous sensing wearable exploiting heterogeneous computation. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 273–286. ACM, May 2015. DOI 10.1145/2742647.2742672.

[24] Q. Leaps. QP/C Framework. Online at http://www.state-machine.com/qpc/index.html, visited Dec. 2015.

[25] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer, 2005.

[26] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for Android applications. In *Proceedings of the International Symposium on Software Testing and*

*Analysis*, (ISSTA), pages 78–89. ACM, 2013. DOI 10.1145/2483760.2483780.

[27] K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, and M. Welsh. Mercury: A wearable sensor network platform for high-fidelity motion analysis. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 183–196, Nov. 2009. DOI 10.1145/1644038.1644057.

[28] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, (NSDI), pages 57–70. USENIX Association, 2013. Online at https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ma.

[29] ARM. Technology / mbed OS. ARM mbed IoT Device Platform, 2014. Online at http://mbed.org/technology/os/.

[30] C. Min, Y. Lee, C. Yoo, S. Kang, S. Choi, P. Park, I. Hwang, Y. Ju, S. Choi, and J. Song. PowerForecaster: Predicting smartphone power impact of continuous sensing applications at pre-installation time. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 31–44. ACM, 2015. DOI 10.1145/2809695.2809728.

[31] C. Min, C. Yoo, I. Hwang, S. Kang, Y. Lee, S. Lee, P. Park, C. Lee, S. Choi, and J. Song. Sandra helps you learn: The more you walk, the more battery your phone drains. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*, (UbiComp), pages 421–432. ACM, 2015. DOI 10.1145/2750858.2807553.

[32] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Proceedings of the Annual International Conference on Mobile Computing and Networking*, (Mobicom), pages 317–328. ACM, 2012. DOI 10.1145/2348543.2348583.

[33] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma. Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, (SenSys). ACM, 2013. DOI 10.1145/2517351.2517354.

[34] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with Eprof. In *Proceedings of the ACM European Conference on Computer Systems*, (EuroSys), pages 29–42. ACM, 2012. DOI 10.1145/2168836.2168841.

[35] Pebble. Battery performance guide. Online at https://developer.getpebble.com/guides/best-practices/battery-perform-guide/, visited Nov. 2015.

[36] Pebble. Pebble. Online at https://www.pebble.com/, visited Nov. 2015.

[37] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security Symposium*, pages 1–12, 2010.

[38] V. Shnayder, M. Hempstead, B. R. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the International Conference on Embedded Networked Sensor Systems*, (SenSys), pages 188–200. ACM, 2004. DOI 10.1145/1031495.1031518.

[39] SocialCompare. Comparison of popular smartwatches. Online at http://socialcompare.com/en/comparison/comparison-of-popular-smartwatches, visited Dec. 2015.

[40] Sony. Open SmartWatch Project. Online at http://developer.sonymobile.com/services/open-smartwatch-project/, visited Dec. 2015.

[41] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 261–274. ACM, 2005. DOI 10.1145/1067170.1067198.

[42] TE Connectivity. New survey from TE Connectivity uncovers America's desire for wearables. PR Newswire, May 2015, Online at https://perma.cc/Y55G-5RRF.

[43] The Creators Project. Choose BLOCKS. Online at http://www.chooseblocks.com, visited Dec. 2015.

[44] H. Wu, S. Yang, and A. Rountev. Static detection of energy defect patterns in android applications. In *Proceedings of the International Conference on Compiler Construction*, (CC), pages 185–195. ACM, 2016. DOI 10.1145/2892208.2892218.

[45] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha. AppScope: Application energy metering framework for Android smartphones using kernel activity monitoring. In *Proceedings of the USENIX Conference on Annual Technical Conference*, (USENIX ATC), page 36. USENIX Association, 2012. Online at https://www.usenix.org/conference/atc12/technical-sessions/presentation/yoon.

[46] L. Zhao, G. Li, B. De Sutter, and J. Regehr. ARMor: Fully verified software fault isolation. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 289–298. ACM, 2011. DOI 10.1145/2038642.2038687.

[47] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. ARMlock: Hardware-based fault isolation for ARM. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 558–569. ACM, 2014. DOI 10.1145/2660267.2660344.

[48] ZWear. ZWear – a wearable platform for makers. Online at http://zwear.org, visited Dec. 2015.