Amulet: A secure architecture for mHealth applications for low-power wearable devices

Andres Molina-Markham[†], Ronald Peterson[†], Joseph Skinner[†], Tianlong Yun[†], Bhargav Golla[*]
Kevin Freeman[*], Travis Peters[†], Jacob Sorber[*], Ryan Halter[†], David Kotz[†]
[†] Dartmouth College [*]Clemson University

Abstract

Interest in using mobile technologies for health-related applications (mHealth) has increased. However, none of the available mobile platforms provide the essential properties that are needed by these applications. An mHealth platform must be (i) secure; (ii) provide high availability; and (iii) allow for the deployment of multiple third-party mHealth applications that share access to an individual's devices and data. Smartphones may not be able to provide property (ii) because there are activities and situations in which an individual may not be able to carry them (e.g., while in a contact sport). A low-power wearable device can provide higher availability, remaining attached to the user during most activities. Furthermore, some mHealth applications require integrating multiple on-body or near-body devices, some owned by a single individual, but others shared with multiple individuals. In this paper, we propose a secure system architecture for a low-power bracelet that can run multiple applications and manage access to shared resources in a body-area mHealth network. The wearer can install a personalized mix of third-party applications to support the monitoring of multiple medical conditions or wellness goals, with strong security safeguards. Our preliminary implementation and evaluation supports the hypothesis that our approach allows for the implementation of a resource monitor on far less power than would be consumed by a mobile device running Linux or Android. Our preliminary experiments demonstrate that our secure architecture would enable applications to run for several weeks on a small wearable device without recharging.

Categories and Subject Descriptors

I.5.5 [**Computing Methodologies**]: [Implementation, special architectures]; K.4.m [**Computers and Society**]: [Miscellaneous]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

1st Workshop on Mobile Medical Applications14, November 6, 2014, Memphis, TN, USA.
Copyright 2014 ACM 978-1-4503-3190-6 ...$15.00
http://dx.doi.org/10.1145/2676431.2676432

General Terms

security, design

Keywords

secure architecture, mHealth, low-power, wearable devices

1 Introduction

Mobile health (mHealth) applications deal with highly sensitive information and implement critical functionality. Often, multiple third-party applications share hardware and other resources. Therefore, a platform for running third-party mHealth applications must include security in its core design. On the other hand, key mHealth applications require high availability to be able to deal with emergency situations (e.g., an anaphylactic shock) and/or to adequately record physiological information throughout the day. It remains a challenge to design a platform for mHealth applications that is (i) secure, (ii) provides high availability, and (iii) allows for the deployment of multiple third-party applications that share resources in a body-area network, such as sensor data, actuators, computation, networking, and storage. Indeed, prior work [16] proposed the use of a wrist-worn device for mHealth applications and identified the general goals (i) and (ii) but it does not propose specific mechanisms to achieve (i). In this paper, we propose the design of a software architecture for mHealth applications that meets goals (i)-(iii) using a wearable platform to provide high availability.

Current wearable devices that run third-party applications, such as smart watches, either do not satisfy (i), e.g., they are not designed with security in mind (for example, Pebble Watch), or do not satisfy (ii) because their software architecture is not designed to run on a single charge for several days (e.g., smart watches running Android) while continuously collecting data from internal and external sensors. Some specialized medical devices are designed to be secure and highly available, but these are typically single-purpose embedded devices that do not enable third-party applications (i.e., do not satisfy (iii)).

Our proposed software architecture, **Amulet**, can run multiple third-party mHealth applications simultaneously and provides strong security properties. At a minimum, the security features of an mHealth platform should (1) control access to managed resources (such as sensors, actuators, storage, computation, and network); (2) contain applications so they cannot bypass access-control mechanisms nor read, write or execute from arbitrary memory locations; and (3) log requests

for access to managed resources to provide accountability. Amulet guarantees security through a robust authorization mechanism that manages resources, through application isolation, and through audit logging. In addition to providing security, Amulet is designed for use on small, light and ultra-low power wearable devices that must run on a single charge for several weeks.

Amulet restricts access to managed resources by implementing an onboard authorization manager that checks all requests from applications to access a resource against local policies before granting access. Amulet guarantees application isolation through a compile-time sandboxing mechanism that statically checks third-party applications implemented using a restricted subset of the C programming language. Amulet also encrypts communications when using wireless links. Finally, Amulet ensures accountability by securely logging all requests that the authorization manager receives, and its response for each request.

Thus, this paper's contribution is the description of a secure software architecture suitable for low-power wearable hardware platforms that enables the sharing of resources by multiple third-party applications. We provide preliminary results that show that our approach is feasible – multiple useful mHealth applications can run on low-power hardware for several weeks on a single battery charge, and the application isolation mechanism that we propose does not interfere with the implementation of such mHealth applications.

2 System and Security model

In this paper, we consider personal mobile-health networks (MHNs) that consist of multiple devices and a smart low-power wearable device that manages the resources in an individual's MHN. In our architecture, the wearable device is called an *amulet*. The amulet provides its *internal resources* (computing, storage) to *internal apps*, which run on the amulet. The amulet provides apps with other resources (sensors, actuators, user interface, and network messaging) through a *service API*. Other MHN devices run *external apps:* some consume data from an amulet (such as a smartphone that provides interactive data analysis or a smart treadmill that provides real-time visualization during a workout); others provide services (such as an ECG chest strap that delivers sensor data to other apps, or an infusion pump that delivers medication on command). All software on an amulet is downloaded from a cloud-based system (operated by a trusted authority) that links Amulet system software with apps vetted for their security properties. In many MHN scenarios, the amulet and other devices need to operate for extended periods without access to a smartphone or the Internet.

In this paper, we show how Amulet addresses three main security goals: isolation among internal applications, access control to resources, and accountability for internal and external applications.

2.1 Adversary model

In this paper, we are concerned with adversaries who attempt to construct or compromise apps (internal or external) so those apps will misbehave (perhaps by harming the amulet system or other apps). We are concerned with adversaries who interpose on the communication between an amulet and other devices in the MHN; we assume adversaries can intercept, insert, or alter messages in the communication medium. We assume that adversaries do **not** have (a) physical access to the amulet device, (b) the ability to break encryption primitives, (c) the ability to discover confidential data through indirect means (such as unintended covert channels interpreted via computational-time or power analysis), or (d) the ability to tamper with the cloud-based app-vetting system.

2.2 Threat model

In this paper, we focus on security-related threats (the broader question of safety-related threats is beyond the scope of this paper). Apps (internal or external) may attempt to circumvent the mechanisms provided by Amulet to access amulet-managed resources, including sensors, actuators, storage, computation, and communications. For example, apps may attempt to obtain data from sensors or send commands to cause actuation, bypassing the amulet's access-control mechanisms. Internal apps may seek to read or write to an arbitrary location in any of the amulet memories. Also, internal or external apps may seek to impersonate another app so that an action looks as if that app had requested it. Finally, an app may seek to inject data or events into other apps.

2.3 Trust model

Any secure system rests on a "root of trust" that is assumed trustworthy. Amulet achieves its three security goals through two assumptions: (1) we trust a cloud-based sandboxing compiler system to properly vet the application code it receives and to produce signed firmware images for the amulet to download, and (2) we trust a small bootloader in the amulet firmware to properly verify the signature on any firmware images before it installs and boots the amulet system.

3 Architecture

The Amulet architecture has four main components: (1) an amulet[1] that manages resources and is capable of running internal apps; (2) internal and external apps that share MHN resources managed by the amulet; (3) the MHN resources managed by the amulet, which include the internal sensors and actuators in an amulet, its internal computational capabilities, and its storage and networking capabilities; and (4) a cloud-based compiler that sandboxes internal apps and links them with authentic system software to create a custom firmware image for installation on a given amulet.

To achieve long life on a small battery, we chose a two-tiered architecture. The "base tier" efficiently manages network communications, internal sensors, and other basic services, while the "application tier" runs application code. This two-tier design allows the amulet to shut off power for the application tier to save power when application responses are not immediately required; in our implementation, each tier is a separate board with a separate microcontroller.

In this section, we describe how multiple internal apps run on an amulet, and sketch the three main mechanisms to achieve our security goals: (a) access control to managed resources; (b) internal app isolation; and (c) secure logging.

[1]We use lowercase to refer to the device and uppercase to refer to the architecture.

3.1 Internal apps

In the Amulet software architecture, apps run on the application tier, which remains off most of the time. This approach drives two requirements: first, apps must be able to survive routine system reboots; second, the application tier must be able to return from off state to active state, load its operating system, and reload an application, extremely fast.[2] Our architecture achieves these requirements by providing an event-driven programming model. This approach works well for mHealth apps, many of which are idle most of the time, waiting for an event to occur.

Internal apps are represented as finite-state machines (FSMs) with memory. Each app consists of a set of *states*, a small set of *variables*, and a set of *event handlers* that define the app's response to events that occur. When an event occurs, the system calls the appropriate handler function and transitions the related app to the appropriate next state. Handlers are non-blocking functions that may consume data arriving with the event, update app variable(s), or send events to system services (or to themselves), in any combination. The app subscribes to certain events when it is initialized, and can add or adjust subscriptions as part of the action in an event handler.

This approach makes app state explicit; because handlers run to completion there are no threads with stack-based state information to preserve between events, let alone across processor reboots. As we describe below, app code and variables are kept in persistent storage, as is a record of the current state of each app; thus, when the event queue becomes empty, the app processor is simply powered off. This simplicity is 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.

The application tier remains off until the base tier boots it. On request, the base tier produces messages to carry the output from internal sensors (e.g., temperature, accelerometer), input from the wearer (e.g., a button press), or the reception of a network message from an external mHealth device. The base tier inserts each new message into the inter-processor message queue; the memory controller uses a set of queue-management policies to determine when to wake the processing component (examples include the insertion of a high-priority message or the queue being near full). While the application tier is awake, it draws messages from the inter-processor queue and copies it into a new event message inserted into its internal event queue. Throughout the rest of the paper, we refer to event messages simply as events.

The Amulet software architecture consists of the following components: event-driven apps, an event processor and event-driven framework, Amulet managers providing service to applications, run-time components not directly available to apps, and a set of board-support drivers.

The application tier runs a supervisor that receives, queues, and dispatches events. Each component can create and send events through an API call that adds events to relevant queue(s) – events may be directed to a specific component or *published* to any app that *subscribes* to that event type. Each component has its own event queue and is assigned a distinct priority level: system components have higher priority than apps. (Amulet sets each app's priority at the time the system image is built; the details are beyond the scope of this paper.) The event supervisor dispatches one event at a time, from the highest-priority non-empty queue, waiting for the handler to complete before dispatching the next event. The Amulet supervisor enforces a timeout to prevent handlers from blocking or running too long. To prevent runaway handlers, the supervisor first sets a hardware timer; if the timeout occurs, Amulet interrupts the app (terminating the handler) and places the app in a special state, and the event supervisor resumes control. Most handlers are quick: they make a quick computation, log data, and send the event to another service.

[2] In the apps we consider, tasks execute every few seconds and tasks typically require only a few milliseconds to complete.

3.2 Access control

It is often important to ensure not only that specific apps and system components are allowed to perform an action, but also that the action is only performed under specific conditions [7]. The problem of specifying authorization policies for access control becomes difficult as the number of principals and resources diversifies and the conditions for providing authorization become more specific [5]. In order to address this authorization specification problem, numerous authorization languages have been proposed [3, 4, 5, 8, 11, 12]. Recent approaches have proposed logic-based authorization languages like SecPAL [3], DKAL [11] and others [12].

In our model, the *Authorization Manager* is a piece of software running in the amulet that can check whether a request by a principal is consistent with a set of policies. One of the challenges that we face is that the Authorization Manager running on the amulet must be lightweight if it is to run on a low-power wearable device that has limited computational capabilities.

Logic-based authorization languages have multiple attractive characteristics [1], such as decidability and tractability—i.e., queries always terminate in polynomial time—as well as delegation. However, a suitable authorization service for resolving authorization queries specified in these languages would be too heavy to be implemented in a low-power microcontroller, such as that used in an amulet. To address this, Amulet implements a lite authorization manager that interacts with a full authorization manager for configuration.

The *full authorization manager* is a deductive service capable of resolving authorization requests against policies, both specified in a logic-based authorization language (SecPAL) [3]. This authorization manager could run on a smartphone, personal computer, or possibly a cloud service. The *lite authorization manager* is not capable of resolving authorization requests following logic deduction. Instead, the lite authorization manager decides whether or not an authorization request should be granted by looking up an access control list, derived from the policies and data sources on which the full authorization manager bases its decisions. The full authorization manager allows an individual to create or edit rich policies on her smartphone (or PC or cloud service) and receive updates to policies for her MHN from a trusted entity. When policies are updated on the smartphone, the corresponding access control list is also updated and synchronized with

the lite authorization manager running on the amulet. As a result, Amulet allows for the implementation of fine-grained, expressible policies that can be updated, while enabling authorization requests to be decided when a smartphone is not present. A detailed discussion of the synchronization of these two authorization managers is beyond the scope of this paper.

3.3 Application Isolation

In order to provide a general-purpose application platform for wearable mHealth that is both robust and secure, Amulet isolates applications from each other, using techniques that require a small memory footprint, and performance overhead that is amenable to use on resource-constrained wearable hardware. In this paper, we specifically focus on 1) ensuring that apps can only access Amulet hardware by sending a well-formed request to the Amulet core, which can be checked by the Authorization Manager, and 2) preventing a malicious or buggy app from reading or modifying the memory of either the Amulet runtime system or another app. Our approach leverages compiler-based translation and static analysis: developers upload their application source code to a cloud-based Amulet service that translates, verifies, and compiles the source code, rejecting it if it violates any of the security properties. Later, each amulet owner visits the web page for this service to select the desired apps; the service links these apps with the authentic Amulet core system code and prepares a custom firmware image for download and installation into that amulet. (In future work we expect to explore cross-application optimization, resource-consumption analyses, app prioritization, and personal key distribution as part of this trusted cloud service.)

Isolation techniques used by traditional operating systems (e.g., virtualizing memory and other resources) require significant computational and memory resources. Other more efficient approaches [9, 14, 17], and those designed specifically for embedded systems [20] with smaller memory footprints, can still incur, in the worst case, as much as 240% processing overhead. ARMor [20] also requires application designers to provide a formally specified policy against which memory accesses are checked at runtime—a task which could place a significant burden on system designers, hindering their ability to rapidly deploy new mHealth applications. In contrast, Amulet is designed to reduce the runtime overhead of isolating apps by preventing dangerous operations at the programming language-level and by checking the majority of memory accesses and other unsafe operations at compile-time. Our approach uses static analysis and dynamic checks, similar in principle to those used in XFI [9], to detect and prevent unsafe operations; however, Amulet operates on higher-level language constructs, instead of machine-code, and a more restrictive programming model that requires less runtime memory and processing overhead.

App designers implement Amulet event handlers using a simple variant of the C programming language, which allows programmers to use familiar programming constructs, and facilitates efficient code generation, while excluding many of C's riskier features. Access to arbitrary memory locations (pointers), arbitrary control flows (`goto` statements), and inline assembly, for example, are excluded from the language specification. Since array accesses in C are implemented using equivalent pointer operations, we modified the array syntax, so that arrays can be passed to functions by reference explicitly (not as pointers). In Amulet, arrays also have an associated length that allows for runtime bounds checking, when access behaviors cannot be adequately checked statically.

At compile time, the Amulet event handlers are checked for additional unsafe operations before being translated into C code that can be combined with the Amulet core and the other apps into a single firmware. Specifically, the translator statically analyzes the app's function call graph and detects recursive calls, attempts to access registers directly, and calls to functions not in the Amulet API and not defined by the app designer. Excluding recursive calls protects against crashes and data corruption due to unbounded stack growth, and allows the compiler to accurately predict the app's memory requirements (i.e., depth of the call stack) at any point in its operation.

Finally, in order to prevent naming collisions (intentional or otherwise) between two apps' functions, the Amulet compilation tools will prepend an app's global variables, function declarations and invocations with a unique app identifier. This effectively creates an independent namespace for each app and prevents an app from inadvertently or intentionally calling another's functions.

3.4 Secure Logging

The Authorization Manager in the amulet logs every request that it receives and its result (whether it was granted or not) in an append-only fashion. When the Authorization Manager receives a request, it verifies its authenticity and integrity. If the verification is successful, then it evaluates the request against the local policies. The Authorization Manager finally logs the request and its result, including the identity of the app that made the request. If the verification fails, then the request will be logged without being evaluated – against local policies – and the request would not be granted. In the latter case, the Authorization Manager could be set – by the amulet's owner or her caregiver – to optionally alert the amulet's owner when the Authorization Manager has received numerous fake requests. This would allow an owner to become aware of a misbehaving or misconfigured app. The option of making the owner aware of such repeated requests could also help in preventing a misbehaving app from filling the amulet's logging memory or exhausting its battery processing requests.

4 Preliminary Implementation

We implemented the core elements of the Amulet architecture using development boards to explore the feasibility of our approach and the tradeoffs between computational capabilities and power efficiency. We have also built a wearable amulet prototype that we designed in order to deploy and test Amulet applications. At the time of writing this paper, none of the smart watches on the market are sufficiently open or contain all the hardware components to do low-level software development and prototyping.

We explored two hardware configurations, both consisting of a base tier and an application tier. The base tier is simply a radio, which we used in both cases: an ANT Integrated

Circuit (IC) by Nordic (nRF51422). For the application tier, we explored three possible microcontrollers: an ARM Cortex-M4f by Texas Instruments, an ARM Cortex-M3 by Energy Micro, and an MSP430 by Texas Instruments. The MSP430 MCU requires less power than the ARM MCUs in general and has a non-volatile memory technology called FRAM. FRAM offers persistent storage without power, and is one hundred times faster than flash [18]. During use, FRAM uses 250 times less power than flash ($\approx \mu A$ at 12kB/s) [18].

We implemented the Amulet services and managers in C. The Sensor Manager, Actuator Manager, Storage Manager, Network Manager, Inter-processor Communication Manager, Crypto Service, and Authorization Manager each consist of two parts, an event-driven application — which we implemented in a similar way to Amulet apps — and a set of routines that have access to lower level drivers. The event-driven part allows these managers to receive and process event messages. The set of routines with low level access are not available to Amulet apps.

We implemented the base tier as an event-driven finite state machine. We implemented two devices to simulate a heart-rate monitor and a galvanic skin response sensor that communicate via ANT with our amulet prototype. These devices are development boards with a similar ANT SoC as in our prototype (nRF51422). Our approach to secure network communications is beyond the scope of this paper.

As described in Section 3.3, we propose a variant of C language, which would be used by developers to write apps. To translate the source code from this variant to C, which can then be compiled, we used the parser generator tool ANTLR [2]. We modified a C grammar, distributed by ANTLR, to remove pointers, jumps, and inline asm, and generated a parser for this modified grammar. The code that is passed to the translator is parsed into a parse tree, and the translator walks the parse tree to verify whether recursions exist; change the way arrays are passed as parameters to functions; change element assignment in arrays to verify if the index is within bounds; and also modify all array accesses as we describe in Section 3.3. The translator verifies whether the code that is passed to it is devoid of all such restricted usage and returns the translated code or an error as per the case.

We did not implement the cloud service. For our current approach, developers would have to submit the signed sources of their apps so that the cloud service could verify the source code's authenticity and integrity. Upon successful verification, the cloud service would check it for compliance. When the code is compliant, the service will combine it with other applications and the rest of the Amulet's source code to create a binary with metadata that is signed by the cloud service.

5 Preliminary Measurements

This paper describes a secure architecture for mHealth applications for low-power wearable devices. We listed as general goals for this architecture: (i) security, (ii) high availability, and (iii) customizability via the deployment of multiple third-party mHealth applications. A thorough security analysis to justify that Amulet meets (i) is beyond the scope of this paper because such analysis would require significantly more space. Goal (iii) is met by design. Therefore, this section focuses on providing evidence to support the hypothesis that Amulet meets (ii). This high-availability goal requires that an amulet uses low power. Therefore, it is necessary that (a) the application tier uses a negligible amount of power most of the time; (b) the idle consumption of the overall system is low; and (c) the base tier – including the radio – uses a small amount of power. There are a number of academic and industry efforts to try to achieve (b) and (c). Therefore, a major focus of our preliminary work is to investigate the extent to which our software architecture provides a secure runtime system that satisfies (a). Our measurements on the nRF51422 board show that a full system can achieve an average idle current draw of 0.01 μA, and the MCU driving the ANT radio can have a receiving channel, get 10 Hz data from a low power accelerometer, and send all the data to the application tier while drawing an average of 0.077 mA.

The application tier's current draw depends on which MCU is used, as well as other peripherals. Each time the application tier wakes up, it performs a small set of operations, drives peripherals, and turns itself off again. The power consumption by the MCU will be the power required to resume and run the workload. Low-power microcontrollers, such as the Texas Instruments MSP430 or the EnergyMicro Giant Gecko (ARM Cortex-M3) resume fast (in 11 ms and 17 ms respectively). The Giant Gecko resumes quickly because it supports a deep sleep mode with memory retention.

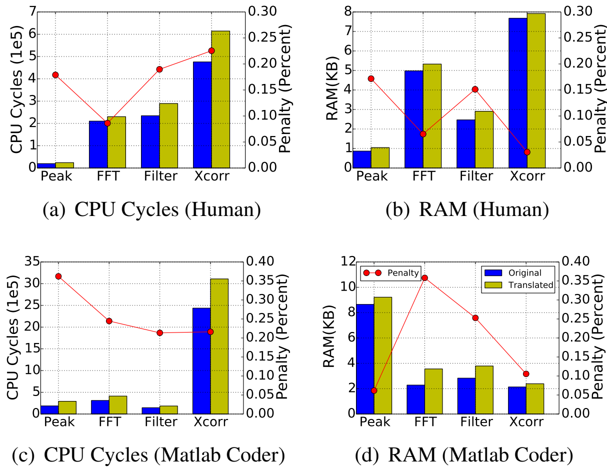
To put this into perspective, consider an application whose workload requires the application tier to work for 70 ms every 2 seconds and remains off otherwise. If the amulet has 150 mAh battery capacity, it could last over 35 days between charges. Optimizations such as wake-on-motion could increase that lifetime.

Android or Linux would take significantly longer to resume from an off state. For example, the Linux version (Linux4SAM) provided by the manufacturers of the application processor Atmel SAMA5D3 suspends/resumes from a suspend to non-volatile storage state (also called hibernation) in more than 15 seconds requiring 0.70 mAh. While suspending to RAM takes approximately 300 ms, the processor remains in a low-power mode that draws over 9 mA and the memory would still have to be powered.

An additional benefit from our light approach is that the size of the trusted computing base is much smaller. The combined implementations of the core Amulet services, including drivers and event framework, compiles to approximately 15 KB. In comparison, in the Motorola MOTOACTV [13] 5.6.0 firmware, the embedded JVM is 656 KB, the Android kernel is 2.7 MB (compressed), and the entire firmware distribution is 71 MB.

Finally, we explored the extent to which our approach for achieving application isolation would require programmers to manually rewrite existing C programs to make them Amulet-compatible. We identified four algorithms that are commonly used in mobile health applications, including FFT, Butterworth Filter, Peak Detection, and Cross Correlation (See Table 1). Developers can build complex algorithms, such as feature extraction for machine learning, using these algorithms. We considered two implementations for each algorithm: one automatically generated by translating Matlab

Figure 1. Translator Evaluation.



(a) CPU Cycles (Human)

(b) RAM (Human)

(c) CPU Cycles (Matlab Coder)

(d) RAM (Matlab Coder)

Table 1. Modifications for Translation.

Algorithm	Modifications/Original Human(LOC)	Modifications/Original Matlab(LOC)
FFT	0/98	0/332
Filter	0/94	10/317
Peak	9/110	40/1840
Correlation	0/80	0/164

implementations to C (via Matlab Coder), and a second implementation coded by a human programmer [15, 10, 19, 6]. We fed these C implementations to our translator, and 5 of 8 required no manual modifications. The remaining required minimal modification due to the use of dynamic memory allocation or the use of pointers. The implementation that required the most changes was a 1840 LOC (lines of code) peak-detection algorithm that needed 40 LOC modified to make it Amulet-compliant. While the performance overhead (CPU and RAM) of the original and translated code varies by program, we observed an average increase of 21% in CPU and a 14% increase in RAM usage when using the translated code (see Figure 1). These increases result from the additional code inserted to do array-bounds checking, but we expect to reduce this overhead through well-known compiler optimization methods.

6 Conclusions

This paper presents Amulet, a software architecture for mHealth networks (MHNs) that allows multiple third-party applications to run on a low-power wearable device, while sharing managed resources securely. Our security analysis demonstrates the extent to which Amulet addresses the desired security goals for the trustworthy management of resources in an MHN. Our implementation of Amulet on our own amulet prototype demonstrates the practicality of our approach. Our security mechanisms have minimal computational and storage overhead. A low-power amulet can run sophisticated mobile applications for longer periods of time than a wearable device running Android or Linux with the same power budget. In particular, an amulet can run for up to 35 days with a 150mAh battery. A wearable running on Linux would drain the same battery in less than 10 hours without necessarily providing security.

7 Acknowledgments

This work results from a research program at the Institute for Security, Technology, and Society, supported by the NSF under award numbers CNS-1314281, and CNS-1314342, 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.

8 References

[1] M. Abadi. Logic in access control. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 228–233, 2003.

[2] ANTLR (ANother Tool for Language Recognition). `http://www.antlr.org/`, Mar. 2014.

[3] M. Y. Becker, C. Fournet, and A. D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4), 2010.

[4] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 139–154. IEEE, 2004.

[5] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, volume 1603, pages 185–210. Springer, 1999.

[6] P. Bourke. Cross correlation, 1996.

[7] M. J. Covington, W. Long, S. Srinivasan, A. K. Dev, M. Ahamad, and G. D. Abowd. Securing context-aware applications using environment roles. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 10–20. ACM, 2001.

[8] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. Technical report, IETF RFC 2693, Sept. 1999.

[9] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88. USENIX, Nov. 2006.

[10] T. Fisher. Interactive digital filter design, 1999.

[11] Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, pages 149–162, 2008.

[12] N. Li and J. C. Mitchell. RT: A role-based trust-management framework. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, volume 1, pages 201–212. IEEE, 2003.

[13] Motoactv. https://motoactv.com, Dec. 2013.

[14] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 199–210, New York, NY, USA, 2013. ACM.

[15] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C*, volume 2. Citeseer, 1996.

[16] J. Sorber, M. Shin, R. Peterson, C. Cornelius, S. Mare, A. Prasad, Z. Marois, E. Smithayer, and D. Kotz. An Amulet for trustworthy wearable mHealth. In *Proceedings of the Workshop on Mobile Computing Systems and Applications (HotMobile)*, Feb. 2012.

[17] M. Sun, G. Tan, J. Siefers, B. Zeng, and G. Morrisett. Bringing java's wild native world under control. *ACM Trans. Inf. Syst. Secur.*, 16(3):9:1–9:28, Dec. 2013.

[18] FRAM Technology. `http://www.ti.com/lsds/ti/microcontroller/16-bit_msp430/fram/overview.page`, Nov. 2013.

[19] xmhk. A library to find peaks in a 1d-list, 2013.

[20] L. Zhao, G. Li, B. De Sutter, and J. Regehr. Armor: Fully verified software fault isolation. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 289–298. IEEE, Oct. 2011.