

Dartmouth College

## Dartmouth Digital Commons

---

Dartmouth College Undergraduate Theses

Theses and Dissertations

---

5-1-2017

# Cryptographic transfer of sensor data from the Amulet to a smartphone

David B. Harmon  
*Dartmouth College*

Follow this and additional works at: [https://digitalcommons.dartmouth.edu/senior\\_theses](https://digitalcommons.dartmouth.edu/senior_theses)



Part of the [Computer Sciences Commons](#)

---

### Recommended Citation

Harmon, David B., "Cryptographic transfer of sensor data from the Amulet to a smartphone" (2017).  
*Dartmouth College Undergraduate Theses*. 123.  
[https://digitalcommons.dartmouth.edu/senior\\_theses/123](https://digitalcommons.dartmouth.edu/senior_theses/123)

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact [dartmouthdigitalcommons@groups.dartmouth.edu](mailto:dartmouthdigitalcommons@groups.dartmouth.edu).

# Cryptographic transfer of sensor data from the Amulet to a smartphone

Thesis by David Harmon

Adviser: David Kotz

Dartmouth College

Dartmouth Computer Science Technical Report TR2017-826

## ABSTRACT

The authenticity, confidentiality, and integrity of data streams from wearable healthcare devices are critical to patients, researchers, physicians, and others who depend on this data to measure the effectiveness of treatment plans and clinical trials. Many forms of mHealth data are highly sensitive; in the hands of unintended parties such data may reveal indicators of a patient’s disorder, disability, or identity. Furthermore, if a malicious party tampers with the data, it can affect the diagnosis or treatment of patients, or the results of a research study. Although existing network protocols leverage encryption for confidentiality and integrity, network-level encryption does not provide end-to-end security from the device, through the smartphone and database, to downstream data consumers. In this thesis we provide a new open protocol that provides end-to-end authentication, confidentiality, and integrity for healthcare data in such a pipeline. We present and evaluate a prototype implementation to demonstrate this protocol’s feasibility on low-power wearable devices, and present a case for the system’s ability to meet critical security properties under a specific adversary model and trust assumptions.

## 1 INTRODUCTION

Smart health devices – such as smartwatches or fitness bands, insulin pumps, or heart-rate monitors – are becoming more common in personal wellness, health research and treatment settings. The authenticity, confidentiality, and integrity of data streams from “mHealth” devices are critical to patients, researchers, physicians, and others who depend on this data. To protect the confidentiality of their patients or research subjects, these parties need to minimize risk of the disclosure or leakage of patient information to untrusted parties or eavesdroppers. In addition, authorized data consumers require the ability to decrypt and validate mHealth data streams. Securing the personal health data streams emitted from wearable devices is therefore critical to enabling these technologies and health applications. Many of these mHealth devices are limited in memory capacity, energy, and processor cycles and may not be able to support traditional encryption systems.

These data-producing sensors allow the wearer to share data with entities that can make decisions in response to the data obtained. These communication specifications meet the low-power requirements of the devices they live on, but lack the range of large-scale wireless networks and so are often limited in range to around 50 feet. Most of these so called body-area network standards, such as Bluetooth Low Energy (BLE), provide some security at the link layer via methods such as channel hopping. In addition, symmetric-key encryption at the network layer is practical for many devices,

but this protection ends when the data reaches the smartphone that acts as a companion to the smart health device. Data can be re-encrypted for Internet protocols such as TLS that protect the data on the way to the database and downstream data-consumers, but this encryption does not provide end-to-end authenticity and integrity from the source device to the companion smartphone application, to the database, and to end-user data consumers. Our goal is to provide the data-consuming application some assurance about the authenticity and integrity of the data from the mHealth device.

Many mHealth devices use body-area wireless network protocols to link to a smartphone that has persistent access to the Internet via cellular or Wi-Fi networks. Our platform uses the smartphone as a staging point for data on its way to the database from the device. The companion smartphone application bridges the healthcare device to a database service, and is responsible for detecting corrupted data and reformatting data into a data format that the database can interpret. Data sent over body-area networks need to be compressed to meet tight bandwidth constraints and power restrictions. Once a data point reaches the companion application it can be attached to identifying factors such as a device ID, application ID, and a data type ID. These identifying factors allow the data-producer to control which data-consumers have access to what data, or subset of data. This method could allow us to support existing database solutions such as open mHealth by translating the data format within the smartphone [1].

To transmute binary data into structured data types, we envision application-specific “plugin” components built into the companion app. These should be written by the Amulet application writers to unpack binary data and reformat the data into a standard representation of a structured data type that downstream data-consumers can store appropriately. The binary data can be whatever format the Amulet application writers find most effective; the structured format may be a standard format compatible with common data-consuming applications. The formatted data can then be uploaded to the database by applications we call *database storage handlers*.

We note above that body-area networks often include encryption at the link layer to accomplish authenticity, confidentiality, and integrity of data in transit. Protocols such as TLS provide this same protection to Internet messages. In our work, we trust the smartphone with data confidentiality because we need it to be able to translate raw data into structured data types. However, because we do not trust the smartphone with the authenticity and integrity of data, downstream data consumers should be able to verify the authenticity of the data source and the integrity of the data (unchanged from the data source). In our solution, authenticity and

integrity can be verified by the ultimate data consumer via a message authentication code (MAC); our solution uses MACs that are derived from the message and from a shared secret like a symmetric key. The companion application needs to forward this value along with the original binary data to the database, and later from the database to the data consumer, to provide a method for verification by downstream data consumers.

*Our contributions.* Our primary contribution is an efficient protocol with end-to-end authenticity, confidentiality, and integrity between wearable devices and downstream data-consumers. We accomplish confidentiality through the use of standard practices for encryption in both body-area and Internet protocols to accomplish confidentiality from attackers listening in on communications. We use established authentication and data integrity practices such as MACs to accomplish end-to-end authentication and integrity. In our approach, the data consumer can verify the authenticity and integrity of the original data from a healthcare device, despite translation by a smartphone, transit through the Internet, and storage in a database, none of which are trusted with data integrity or authenticity.

## 1.1 Organization

We begin by giving necessary background information and pertinent definitions in Section 2. We continue with a discussion of related work in Section 3. We characterize the anticipated security model on which our system is based in Section 4. We then present our solution, especially focused on the wearable device and companion application, in Section 5. We detail our implementation in Section 6. We follow with an evaluation of the system in Section 7. We discuss limitations to our system and interesting extensions in Section 8, and finally conclude with Section 9.

## 2 BACKGROUND

Our solution is integrated into an open-source hardware and software platform called the Amulet, a multi-application smartwatch [9]. The Amulet’s primary microcontroller is the MSP430FR5989, which has a 16-Bit RISC Architecture and operates at a clock speed up to 16 MHz [14]. The normal supply voltage ranges from 1.8V to 3.6V, using 100  $\mu$ A/MHz in typical active mode and about 0.4  $\mu$ A/MHz in typical standby mode. It comes with 128Kb of nonvolatile FRAM that is capable of ultra-low-power writes.

The Amulet chipset includes an AES accelerator for performing AES-128 encryption on 128-bit data in 168 cycles that we use to encrypt our messages and their headers [13]. It also comes standard with a hardware module for pseudorandom number generator algorithms that can be used to generate private keys securely [14]. The AES accelerator consumes 21  $\mu$ A/MHz in typical usage. The AES code used to interact with the accelerator was based off of the TI library of sample C code for the MSP430FR5989 microcontroller [15].

The MSP430 also has a hardware module to generate 16-bit Cyclic Redundancy Check (CRC) codes that we use as a Frame Check Sequence (FCS) on each packet. The FCS allows the smartphone to verify that the message was not corrupted by noise in transmission. Our current implementation discards corrupted messages; we do

not attempt retransmission. The code we use to interact with this accelerator is based on code from the MSP430FR5989 microcontroller sample C code library [15].

We use a Hashed Message Authentication Code (HMAC) to protect the integrity and authenticity of the data from the Amulet. The HMAC is built using a shared symmetric key, a hash algorithm, and a payload [10]. The resilience of an HMAC to brute force is limited by the strength of the shared symmetric key [16]. The hash algorithm chosen as a primitive for the HMAC algorithm is substantially more resistant to recent practical attacks against algorithms such as MD5 and SHA-1. It is important to use cryptographic hash algorithms such as the SHA-2 family to avoid possible security pitfalls of algorithms such as MD5 and SHA-1. Thus we choose SHA-256 as our hash algorithm.

SHA-256 takes 512-bit blocks of data as input, and outputs 256 bits. Our implementation of SHA-256 comes from the TI library of SHA-2 code for the MSP430 series microprocessors [8]. The code is measured to complete in 67 kilo cycles for 2 blocks of data. Input to this function must already be in binary format in 32-bit integers.

## 3 RELATED WORK

Other papers have explored security holes in Bluetooth, and even proposed practical tools to eavesdrop on live messages. Ryan and Albazraq et al. independently proposed practical methods to eavesdrop on live messages without listening in on the pairing process in BLE [2, 12]. These papers contribute practical methods of sniffing previously established connections by performing traffic analysis to derive connection-specific values (such as the hop interval) that allow an adversary to intercept plaintext messages. The methods proposed by these papers are dependent on the implementation using all bands of the BLE radio spectrum, which is not required by the BLE specification but is generally true for many common commercial devices.

Other prior work has proposed methods to snoop on the application-layer encryption key exchange and expand on link-layer attacks to intercept cipher text messages and decrypt them. Ryan established that one can intercept shared secrets during the BLE key-exchange procedure [12]. Ryan’s process is specific to older BLE specs; other key-exchange protocols have been added to newer versions of the BLE specification. All values used to build a short-term link key are exchanged in plaintext; if you can observe them all, computing the long-term key is a matter of brute-forcing a 6-digit PIN code, which Ryan shows can be performed in under a second on commodity hardware. Armed with the link key, the attacker is capable of independently generating the long-term key used at the start of each session to generate a new session key for AES-CCM. If the long-term key is known, and you observe the session-initiation exchange, it is trivial to recompute the keys used in each session. Even if you do not observe the session-key exchange, but have the long-term key, then you can jam the signal and force a new session key exchange once the session has timed out.

Das et al. propose a traffic-analysis attack as a tool to identify fitness information (such as activity type) and to track users [4]. They show that most healthcare devices use unchanged device addresses and, as a result, allow an observer to track the wearer. They note that many fitness devices only periodically connect to

their paired smartphones to dump data, and at all other times are advertising themselves, revealing the user’s location to anyone in the advertising range, such as in a gym setting. In addition, they found that user data traffic is tightly correlated to the intensity of the activity type. This traffic can be used to identify an individual from a group, since each user walks with distinct gait that is revealed by traffic analysis.

Other papers have proposed other practical measures for increased Bluetooth security. Fawaz et al. designed and implemented BLE-Guardian, a privacy focused precaution designed to limit the devices that can discover and connect with the protected device [5]. BLE-Guardian protects host hardware by acting in tandem. It is a hardware module with an interface designed to allow the user to analyze the advertising patterns of nearby devices and apply active jamming to hide the user’s device. It then acts as an intermediary for the protected device, alerting the user to new devices in the area that attempt to scan the protected device. The module then blocks non-accepted devices and advertises services on behalf of the protected device to user-accepted devices within range.

In BLE version 4.2, security was greatly enhanced with the inclusion of the Diffie-Hellman key exchange protocol. This protocol provides a significantly stronger method of device pairing and long-term key generation [3]. Developers have shown a way to update the firmware of specific devices to this new version [11]. They acknowledge, however, that there is no way to authenticate firmware and there are few security precautions in the installation process.

## 4 SECURITY MODEL

Our security and privacy goals are to maintain the confidentiality, integrity, and authenticity of data produced by the data owner in possession of an Amulet smartwatch for the purpose of a study or treatment plan, and to allow a data consumer to verify the authenticity and integrity of the data retrieved from a database. The completeness of our solution must therefore be judged in the context of an anticipated security model. Here we next present our adversary model, threat model, and trust model.

### 4.1 Adversary model

Our adversary is anyone with the intent of obtaining confidential information produced by the Amulet apps, of tampering with that information, or of injecting false information as if it came from the Amulet of a specific person. We assume several capabilities and limitations of the adversary. We assume the adversary does not have physical access to any system component and cannot compromise the hardware or software of any component. This assumption is reasonable due to the wearable nature of the Amulet and the personal nature of the smartphone. We assume the attacker cannot break the cryptographic primitives that we use: SHA-256, and AES. These primitives are computationally hard and resilient to brute-force attacks. We assume that all cryptographic keys are generated and shared securely so that the adversary cannot steal them during some initialization phase. The adversary has access to the network channels between the Amulet and the phone, and the phone and the database. The adversary is assumed to be capable of intercepting, changing, injecting, replaying or blocking an arbitrary subset of

messages between the Amulet and the database, or between the database and the consumer.

### 4.2 Threat model

Given the capabilities of the adversary, we focus on the following threats.

*Threat to confidentiality:* The adversary attempts to obtain plaintext of a message in a data stream in order to learn sensitive information about the Amulet owner, such as medical conditions (e.g., disease or treatment type), mHealth usage (e.g., types or number of apps/devices), or other personal information deemed private (e.g., location or activity). The adversary may eavesdrop on the system, including all communications between the Amulet, the connected smartphone, the database, and data consumers, to discover sensitive information from the messages. The adversary may try to compromise the database to determine this sensitive information, or the database itself may be adversarial.

*Threat to data integrity and authenticity:* The adversary attempts to cause the companion application, database, or the data consumer, to accept incorrect, invalid, or duplicate data by either forging an entry that looks legitimate to the database, tampering with a legitimate entry from the companion application, or replaying a previously submitted entry. The adversary may inject, tamper, or replay communications among parties.

### 4.3 Trust model

We make certain trust assumptions about each system component: the Amulet, the companion smartphone application, the database, and the data-consumer portal. All components are trusted with the confidentiality of the data, and securely preconfigured with the shared keys required to participate in our protocol. All components are uncompromised, and store the data and key material securely. The patient or research participant trusts his healthcare professional or researcher with the confidentiality of the data and with the responsibility to dispose of the keys and data securely at the end of the study or treatment plan.

## 5 OUR APPROACH

We extend the Amulet operating system to enable its applications to send *data messages* to the Amulet’s companion smartphone. Each Amulet application produces one or more *data streams*, each of which produces a series of *messages*, each of which has one or more *data values*. Applications must include a timestamp in each message on the data to provide the highest possible accuracy on the time associated with a message as well as to provide some uniqueness to each message. The system prepends several fields to form a *packet*, labeled with an app ID, stream ID, format version number, and protocol version number. The Amulet system keeps track of unique identifiers for each app installed, allowing for simple app ID inclusion. Each app has the ability to define up to eight data streams specified by a stream identifier on each call to our protocol. Applications may update the format of one or more message types; the format specifier allows the smartphone companion application to recognize backwards compatible messages. The last identifier is

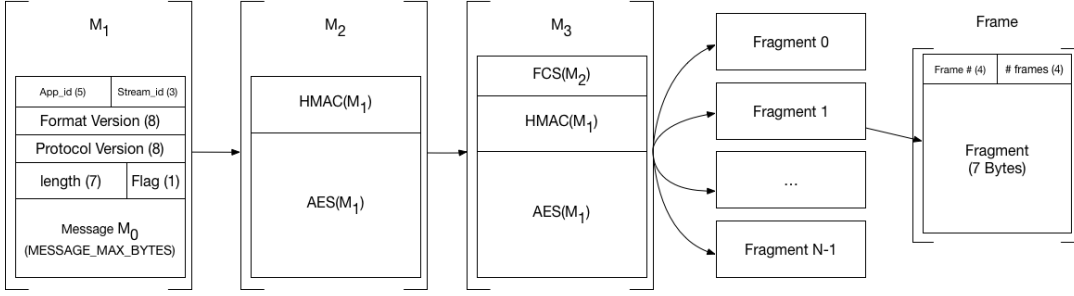


Figure 1: Structure of packets at each stage in their construction

used to identify the protocol version, allowing for backward compatibility by the companion application should changes be made to the protocol. Packets are encrypted to preserve data confidentiality of the message and the identifying fields. All packets flow from the Amulet to the smartphone companion application through our protocol.

Once a packet arrives at the smartphone companion application it is decrypted, and unpacked from binary data into textual format by an app-specific *plugin*. The smartphone transfers the translated data to the database via *database storage handlers* that have the required permission set. *Data consumers* can then retrieve data from the database assuming they have the relevant access permission; the details of this database and access-control policy are out of scope of this thesis.

Confidentiality is maintained via encryption between the Amulet and smartphone using one key, and encryption between the database storage handler and the database using TLS or another key, and between the database and data consumers using TLS or another key. Our focus is on the first hop.

Authentication and integrity are maintained end-to-end from the Amulet to the downstream data consumer via a message authentication code, specifically an HMAC. The Amulet generates the original HMAC over the plaintext message and sends it to the companion application as part of a packet. The relevant key has been previously shared with downstream data consumers who are able to pull the reformatted data, the original binary data, and the HMAC from the database and independently calculate and verify the HMAC as well as the binary-text translation.

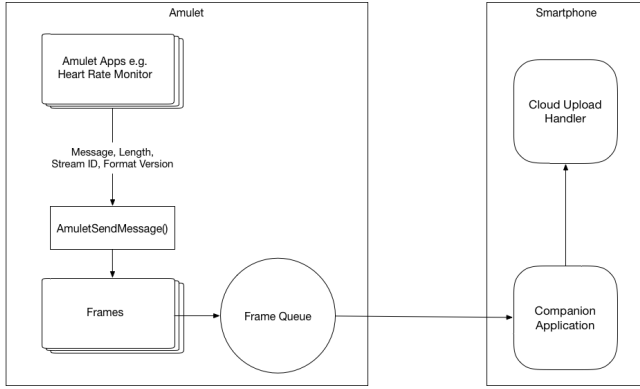
Armed with these tools, we next define our protocol for Amulet applications to push data from wearable devices through a BLE connection to a smartphone companion application. Amulet applications can define up to eight data streams at their discretion. If an app wants to change the definition of a data stream, or add, remove, or replace a data stream, then they increment the format version number, allowing the plugins in the companion app to preserve backwards compatibility. The OS provides the application ID and a protocol version. The length of the message, and a reserved flag that is not currently used, make up the last of the plaintext fields. When an application provides a message to be sent, identifying the stream id and format version number, the system assembles a packet by adding the application ID and protocol version number as shown in Figure 1.

We denote the data provided by an Amulet app  $m_0$  with length  $L$ ; with the addition of a format version  $fv$ , stream id  $s_{id}$ , app id  $a_{id}$ , protocol version  $pv$ , and reserved flag  $f$ , the system builds a string denoted  $m_1 = a_{id}|s_{id}|fv|pv|L|f|m_0$  as shown in Figure 1. Then let  $m_2 = HMAC(K_{HMAC}, m_1)|AES(K_{AES}, m_1)$ , where  $K_{HMAC}$  is the shared key used to generate an HMAC and  $K_{AES}$  is the shared key used to encrypt data with AES. Then let  $m_3 = FCS(m_2)|m_2$ . At each stage, different operations are performed on the message; in  $m_1$  the plaintext fields are prepended, in  $m_2$  the HMAC is calculated from  $m_1$  and  $m_1$  is encrypted, and finally in  $m_3$  we calculate the FCS over  $m_2$ , producing the final packet  $P = m_3$ . Once a packet  $P$  is constructed, we create 7-byte *fragments* of the packet such that  $P = f_0, f_1, \dots, f_{n-1}$  where  $n$  is the number of 7-byte fragments in the packet, and  $f_0$  to  $f_{n-2}$  are filled with data and  $f_{n-1}$  may only be partially filled and is padded with trailing 0s. A frame that fits the BLE data size of 8-bytes is constructed from each fragment and the fragment identifiers  $F_i = i|n-1|f_i$ . A depiction of each stage is shown in Figure 1.

Two shared keys are required, one shared with the smartphone for decryption and one shared only with the data consumer to verify the HMAC. Before the packet is encrypted, we use  $K_{HMAC}$  to generate a 256-bit HMAC and prepend the higher order 128 bits to message  $m_1$ . The AES key, denoted  $K_{AES}$ , is used to encrypt  $m_1$  using AES-ECB, resulting in  $m_2$ . We acknowledge the flaws in using this block cipher mode, and with the ordering of our cryptographic primitives, we discuss it further in section 8. Finally, a 16-bit Frame Check Sequence (FCS) is calculated over  $m_2$  and prepended to  $m_2$ , forming a packet  $P$ .

As seen in Figure 1, frames are marked with the identifier of a specific frame in a series comprising a packet. The system schedules frames to be sent when BLE is next available. The overall System structure is shown in Figure 2.

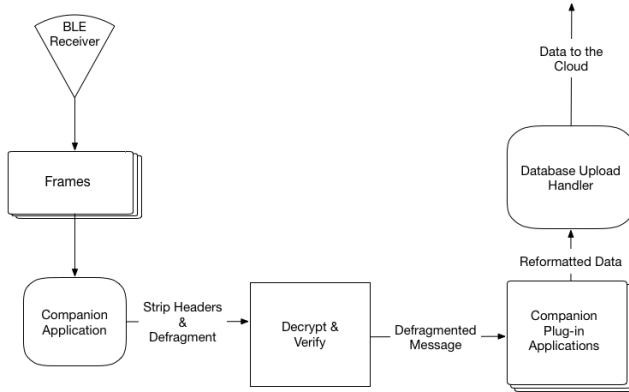
The Amulet radio controller is responsible for transferring data to the smartphone companion application. The Bluetooth client on the smartphone requests data from the Amulet at regular intervals and delivers each frame to the companion application one at a time. The companion application strips the frame id  $i$  and frame count  $n-1$  from each frame, assembles them in numeric order, and combines each one into a single buffer as they arrive. If a frame is missing, it discards the entire packet (support for retransmission is discussed in Section 8). Once the smartphone companion application has received all the frames of a packet, it recalculates the FCS and use it to verify the FCS provided in the packet as an initial check against



**Figure 2: The system architecture of the protocol between the Amulet and smartphone companion application**

data corruption. Then we copy the HMAC from the packet, and decrypt the remaining ciphertext using the previously shared key  $K_{AES}$ .

Once we have the plaintext bytes making up  $m_1$ , we extract the identifying fields provided by the application and system. From this we have the application ID, stream ID, format and protocol versions, the length of the message, the reserved flag bit, and the bytes comprising the message  $m_0$ . Using the application ID and stream ID, we call the plugin, passing in the message  $m_0$ . The result is a structured textual representation of the data, e.g. in JSON format. To this structure we add the binary HMAC data and the original binary message data  $m_0$  translated into a textual representation, e.g. Base64. The companion application then passes this bundle to the database storage handler to upload to its database. The architecture of the smartphone companion application is shown in Figure 3.



**Figure 3: The smartphone companion application architecture**

Downstream data consumers can then pull data from the database via means specific to each database provider. Because consumers do not trust the database provider with authenticity and integrity, the HMAC is available along with the original binary packet received by the companion application. As a result, data

consumers can verify the HMAC and the binary-to-text translation, using the previously received key  $K_{HMAC}$  required to recalculate the HMAC. This verification step represents the final step in the end-to-end authentication and integrity provided by our protocol.

## 6 IMPLEMENTATION

The scope of our project consists of two components, the Amulet code and the companion application code. The database upload provider, database, and downstream data consumer are all within the scope of our architecture but are implemented outside the scope of this project. The main contribution of this implementation is to build a protocol between the Amulet and smartphone companion application that is agnostic to the database but provides a means to authenticate data that data consumers receive from a potentially untrusted database.

### 6.1 Amulet

The Amulet API is stateless, and consists of a single function for app designers to call written in standard GNU C. It takes four parameters: a stream identifier, a format number, version number, and a message payload, including the length of the message, and the application ID number. Messages provided by Amulet applications must include a timestamp, to provide the highest possible accuracy on the time associated with a message as well as to provide some uniqueness to each message. The function prototype is shown in Listing 1, and an example call is shown in Listing 2. Note that the example call does not include the requester (app ID) field, because it is automatically filled in by the system-call mechanism in Amulet OS, and the Amulet C compiler.

```

1 /**
2  * Send a binary message to the companion device when
3  * available. The message will be copied into a queue
4  * of pending messages. If the queue becomes full, the
5  * oldest messages are overwritten by newer messages.
6  * @Param stream_id: the data type identifier, apps can
7  * emit up to 8 types
8  * @Param format_v: the format version of the data type
9  * being emitted
10 * @Param message: the data being emitted
11 * @Param length: the length of the data being emitted
12 * @Return can indicate simple errors such as message
13 * too long, invalid stream_id, connection not available.
14 */
15 int
16 AmuletSendMessage(uint8_t stream_id, uint8_t format_v,
17                  __uint8_t_array message, uint8_t requestor);

```

**Listing 1: Amulet API function header**

```

1
2 uint8_t sendmsg[15] = "This is a Test";
3 AmuletSendMessage(0, 1, sendmsg);

```

**Listing 2: Amulet API sample call**

During an `AmuletSendMessage()` call, we copy the message  $m_0$  and identifying fields into a buffer comprising  $m_1$ . We use  $m_1$  to generate an HMAC with  $K_{HMAC}$ , which needs to be padded to the length of the input block size of the hash algorithm resulting in  $K^+$ .

$K^+$  is then XOR'd with the inner (0x36) and outer pads (0x5c) to increase Hamming distance of the key, and resulting in  $K_{inner}^+$  and  $K_{outer}^+$ . To save computation time from doing this step on every call to the protocol, we precompute these values and save them. The high-order 128 bits of the HMAC are copied into  $m_2$  byte by byte to preserve byte ordering. We use the high-order bits as a means of reducing our transmission size, while preserving the strength of the HMAC [6].

The HMAC makes use of the SHA-256 hashing algorithm, which takes in 512-bit blocks and outputs 256-bit blocks. First we build the inner buffer by concatenating the keyed inner pad with the message and then padding it out to a multiple of 512-bits, or 64 bytes. The result of the hash is then appended to the keyed outer pad and hashed again to produce the final HMAC. Pseudocode for this function is shown in Algorithm 1. The SHA-256 library we use is from the TI library of auxiliary code [15].

---

**Algorithm 1** Compute the HMAC

---

- 1: Concat the message  $m_1$  with  $K_{inner}^+$
  - 2: Pad the resulting buffer to a multiple of the input block size
  - 3: Hash the buffer with SHA-256
  - 4: Clear the buffer
  - 5: Concat the hash with  $K_{outer}^+$
  - 6: Hash the buffer with SHA-256 again to calculate the HMAC
  - 7: Copy HMAC into the message  $m_2$ , truncate the lower order bits
- 

The plaintext message  $m_1$  is then encrypted in 16-byte blocks, and the resulting ciphertext blocks are inserted into  $m_2$  as they are generated. The AES accelerator on the MSP430 takes keys of multiple sizes; we use a 128-bit key as our  $K_{AES}$ .  $K_{AES}$  should be generated and installed prior to the distribution of the Amulet (methods for key exchange between the Amulet and smartphone or between the Amulet and data consumer are out of the scope of this thesis). The AES library we use implements AES-ECB. When we encrypt a message we divide it into 16-byte blocks and copy them into a manipulable buffer. This buffer is pushed block by block through the AES accelerator and then the cipher text is copied into the packet over top of the plain text.

Finally, we compute a shorter frame check sequence (FCS), used by the smartphone to verify the message. The MSP430 has a CRC-16 hardware accelerator that uses the CRC-CCITT algorithm and takes input in either big-endian or little-endian ordering; we use little-endian because it is the new standard for MSP microcontrollers [13]. This algorithm specifies an initial value of 0xFFFF, so we begin by initializing the result register with this value every time we generate a CRC. The accelerator works by taking input blocks and then performing operations to combine the value-so-far with the new data. The accelerator takes either 8 or 16-bit blocks; we push 8-bit blocks into the high-order byte of the register for simplicity. The result is copied to  $m_3$  with  $m_2$ , thus completing our packet.

The Amulet supports the transmission of 8-byte frames over the BLE “heart-rate” profile; we adopt this profile for convenience. Once the entire packet is built, it is broken into 7-byte fragments that are copied into a frame with a 1-byte header identifying the current frame number in the sequence and the total number of

frames, and pushed onto a ring buffer called the *frame buffer*. This ring buffer is implemented as a “virtual stream” such that the *put* and *get* pointers increase until they overrun their size limitations. On put and get operations, we modulus the access into the buffer with the size of the buffer to avoid buffer overflow errors [7]. This approach has the benefit of removing the case where the buffer could be empty or full if the *put* and *get* indices are equal to each other; if they are equal then the buffer is full, and if they are unequal then there is data to send. If the buffer fills before data is sent, then older packets are overwritten. If some of the packet space is not used and there are frames with no data, then those frames are not created. However, in order to prevent artifacts from remaining if a packet is overwritten partially we always increment the *put* index by the maximum number of frames in a packet.

## 6.2 Companion Application

The companion application receives individual frames from the Android Bluetooth device manager. As it receives each frame, it verifies that none are missing or out of order and then strips the frame header and copies them into a buffer representing the original packet. Each frame indicates the total number of fragments and the fragment number. If a frame arrives out of order then we know that one was lost, and so we drop the fragments received so far and wait for the next packet to begin.

Once we have a fully reconstructed packet buffer, we verify the CRC16 by performing the same operation performed by the Amulet on the binary buffer that we received. If the CRC16 code is not the same then one or more frames were corrupted in transit; we drop the buffer and wait for the next packet. After the verification step, the CRC is stripped from the packet buffer resulting in  $m_2$ .

Then the HMAC is stripped from  $m_2$  and set aside to be sent to the database later. We are left with the encrypted  $m_1$ ; we decrypt  $m_1$  using the key  $K_{AES}$  and the corresponding AES decryption function. Android uses the standard `javax.crypto` library that implements AES-ECB in 128-bit mode.

Given the decrypted headers and message, we decode the binary header into the original header fields and copy out the byte array representing the message. The message and format specifier is passed to the relevant app-specific parser plugin (selected by the application and stream ID identifiers) and parsed into a structured data format in ASCII – specifically, in a JSON serializable format – and shared with database storage handlers. After receiving the JSON object, the data is sent on an intent to applications who have been given permission to listen to intents from our application, including one or more database upload handlers.

## 7 EVALUATION

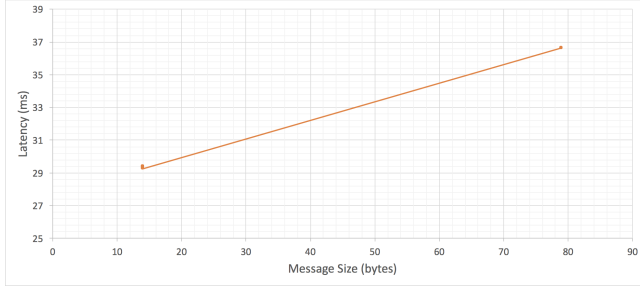
In this section we analyze the practicality of our implementation in the context of the Amulet platform, then describe the security implications in the context of our security model.

### 7.1 Memory

Figure 4 shows the amount of FRAM in bytes required for the base image without our protocol, and for the same system after the addition of our extensions. We use several constants in our code, including the installed keys  $K_{AES}$  and  $K_{HMAC}$  as well as

	Base Image	Base Image & Protocol	Difference
.rodata	8060	8432	+372
.data	240	240	0
.bss	588	1342	+754
.noinit	7306	7434	+128
.text	25932	29508	+3576

**Figure 4: Bytes of memory used with and without our protocol in the Amulet image**



**Figure 5: Latency as a function of message  $m_0$  size**

several other important variables. The .bss section, the initialized data variable section, is over twice as large primarily because the frame buffer, which uses 562 bytes, is stored here. The .text section is larger particularly because of the function code to build each packet.

## 7.2 Latency

We define the *latency* to be the time it takes for a packet to be processed, defined as the time from the start of the Amulet API function to the end of fragmenting a packet and pushing it to the frame buffer. We add this time to the time it takes the companion application to process a packet once all the frames have been received in the companion application and to send the data intent to database storage handlers. This definition excludes the actual network transmission time, which is out of our control so we focus on our computational overhead. We further define the *head latency* as the minimum time to process a packet, and the *tail latency* as the maximum time. The latency distribution pictured in Figure 5 shows the scaling of latency with packet size in bytes. This outcome is due to the nature of the AES, FCS, and SHA-256 algorithms that split data into blocks with sizes specific to each algorithm and process them one at a time. The maximum latency that appeared in our measurements was 36.6ms at 79 bytes and the minimum was 29.3ms at 14 bytes; these values signify that there are significant constant costs in addition to scaling costs.

## 7.3 Throughput

Given our use of the BLE profile for a heart rate monitor, we are allowed 8 bytes per fragment. The shortest connection interval (the rate at which a central will ask for new data from the peripheral) for BLE is 7.5ms. Therefore the maximum throughput is  $1/0.0075 = 133.33$  fragments per second. The maximum number of frames for the biggest possible message is 16, therefore there is a theoretical

cap of 8 full messages per second or 1,064 bytes per second. As shown in Section 7.2, we are able to push frames from a 14-byte message in 29.3ms, and a 79-byte message in 36.6ms. For a 14-byte message we would push eight frames, and for a 79-byte message we would push seventeen frames. For both of those sizes, and for BLE operating at the optimal rate, we would be able to produce a new message during the time it took to send the first message. Therefore our protocol is not a bottleneck to throughput.

## 7.4 Security analysis

Given the security model outlined in Section 4, and our approach described in Section 5, we now argue that our approach resists threats to confidentiality, integrity, and authenticity.

*Threat to confidentiality.* The attacker attempts to violate confidentiality by decrypting ciphertext messages or by observing patterns in the ciphertext that may violate the privacy of the Amulet owner. In our trust model, we trust the smartphone and database to protect the confidentiality of unencrypted data, but need to protect confidentiality of messages in transit that can be intercepted by an attacker. Our protocol is focused on the network hop between the Amulet and the smartphone companion application. Our protocol implements AES 128-bit encryption. Therefore the legitimacy of the threat to confidentiality is the ability of an attacker to break AES encryption, or to obtain the shared key  $K_{AES}$ . We trust that  $K_{AES}$  has been shared between the Amulet and the smartphone companion application securely; key exchange is out of the scope of this thesis. We also trust that the attacker cannot obtain  $K_{AES}$  through the UI or by compromising the hardware or software of the Amulet or smartphone. Our last relevant assumption is that the attacker cannot discover  $K_{AES}$ , or decrypt messages via brute force. This approach implies that ciphertext messages are safe from decryption.

Additionally, an adversary may attempt to recognize patterns in a series of messages. Given the vulnerabilities of AES-ECB, it is obvious when the same message is resent since both the AES and HMAC value would be the same. This situation is unlikely because Amulet applications are required to include a precise timestamp with each message, but a more secure implementation of AES-CCM is planned for future work and is described further in Section 8.

*Threat to authenticity and integrity.* The attacker may violate the authenticity or integrity of data by causing the companion application, or the data consumer, to accept incorrect, invalid, or duplicate data by either forging an entry that looks legitimate to the database, tampering with a legitimate entry from the companion application, or replaying a previously submitted entry. Additionally, the smartphone may incorrectly translate the binary data to text, or the database may change the text form of the data. We do not trust the smartphone or the database with the authentication and integrity of the data, but provide end-to-end authentication and integrity through the use of an HMAC. The strength of the HMAC is measured by the strength of the shared secret used to create it; we use a 128-bit key  $K_{HMAC}$  distinct from the key used for AES  $K_{AES}$ . We assume that the attacker is not capable of brute-forcing 128-bit keys, and that  $K_{HMAC}$  was generated and distributed securely. Key exchange is out of the scope of this thesis. Attackers can

change ciphertext in transit, attack the database, or the database may act adversarial; but any changes will be detected when the data consumer recalculates the HMAC and notices a different value than the one provided to them. Without the shared key  $K_{HMAC}$ , it is not feasible for the attacker to generate valid HMACs for invalid messages. This protects data streams from injection, and modification. Replay attacks can be stopped by timestamps on each data point.

## 8 DISCUSSION AND FUTURE WORK

Our protocol does not support frame retransmission; it just drops packets for which any frame is missing, reordered, or corrupted. Supporting frame acknowledgements or some sort of packet-sized sliding window would be possible, but the ability to buffer sent frames is limited by the Amulet’s memory restrictions. Additionally, supporting retransmissions could leave our protocol open to more sophisticated denial-of-service attacks and tie up resources on either or both sides.

The Amulet implements the heart-rate measurement BLE profile with its 8-byte frame size, limiting our throughput and increasing the number of frames comprising a packet. This increases the risk for frame loss or corruption; changing the profile implementation could increase throughput and decrease risk of loss. Implementing the BLE service profile to one with a larger size such as 20 bytes would significantly increase our throughput and reduce such risks.

Additional shortcuts involved the transfer of data between the companion and database storage handlers in intents rather than in data files on the phone. Storing data on the phone as well as in the database could data consumer portals to exist on the smartphone, but would also require strict access requirements that are out of the scope of this project.

The Amulet AES library implements AES-ECB; there are known issues with using this block chain mode from the lack of pseudo-randomness. Packets that have been captured entirely by an attacker are vulnerable to reveal patterns in the data, or to make transmissions with the same message  $m_1$  obvious. Patterns in the ciphertext could potentially reveal unintended personal information such as usage patterns, applications used on the Amulet, or other factors. A more secure block cipher implementation that includes a pseudo-random initial value such as AES-CCM is highly recommended for future work. When the block cipher implementation changes, the order in which we compute the ciphertext and the HMAC should be revisited.

## 9 CONCLUSION

In this project, we produced a secure protocol for downstream data consumers to verify data from a BLE-enabled wearable healthcare device like the Amulet. This ensures the consumer, the researcher, physician, or other healthcare professional that the data they are reviewing was in fact generated by the device that claims to be generating it and that the data has not been changed since it was produced. We described a simple, stateless API for Amulet applications to send data over Bluetooth to the smartphone and a framework for application-specific plugins attached to restructure the binary data as a structured, text-based data type such that it can be used by the database and downstream consumers. We implemented our

protocol as an addition to the Amulet operating system and a simple companion smartphone application built on the Android platform, and within a larger vision including a database and a downstream data consumer. In our evaluation we showed that the addition of our protocol does not overwhelmingly restrict the Amulet’s limited resources, and that our contributions protect private data from predicted attacks. We conclude that our protocol is a practical addition to the Amulet system to provide end-to-end authentication, confidentiality, and integrity to healthcare data produced by the Amulet.

## 10 ACKNOWLEDGEMENTS

We are grateful to many for their assistance with this thesis.

Ron Peterson is an important member of the Amulet project, and has been an asset in implementing the radio code for BLE. His implementation of BLE using the Nrf5 SDK for the Amulet has made this thesis possible by providing an environment in which we can contribute our protocol. He has also been critical to the ability to incorporate the protocol into the existing BLE radio API.

Patrick Proctor has been involved in our greater vision for this protocol by designing and implementing a database for handling data offloaded from the Amulet by this thesis. He has been an important reference for helping to debug problems and provide insight to our work.

Taylor Hardin is an immense source of knowledge on the Amulet operating system and has helped identify locations in the firmware source code where our protocol now lives. He also provided significant knowledge of the Amulet debugging environment and to debug significant blocks when they arose. He also contributed knowledge to measuring the performance of this protocol on the Amulet side.

Emily Greene implemented a significant portion of the greater vision for this protocol by implementing a database upload handler, a secure database, and a sample data consumer portal. She also previously worked on BLE protocol interactions between the Amulet and companion smartphone application; the companion smartphone application is built from that initial work.

David Kotz is the director of the Amulet project at Dartmouth and is responsible for perhaps the biggest contribution to this thesis. He has given feedback and ideas for further exploration and effort throughout the process of this thesis. His guidance allowed for success and this thesis would not have been possible without him.

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

## REFERENCES

- [1] Open mhealth, 2015. Online at <https://open-mhealth.org>.
- [2] Wahhab Albazraqoe, Jun Huang, and Guoliang Xing. Practical Bluetooth Traffic Sniffing: Systems and Privacy Implications. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services, MobiSys*, pages 333–345. ACM, 2016. DOI 10.1145/2906388.2906403.
- [3] Bluetooth SIG. Security, Bluetooth Low Energy, 2017. Online at <https://www.bluetooth.com/what-is-bluetooth-technology/how-it-works/low-energy>.

- [4] Aveek K. Das, Parth H. Pathak, Chen N. Chuah, and Prasant Mohapatra. Uncovering Privacy Leakage in BLE Network Traffic of Wearable Fitness Trackers. In *Proceedings of the International Workshop on Mobile Computing Systems and Applications*, HotMobile, pages 99–104. ACM, 2016. DOI 10.1145/2873587.2873594.
- [5] Kassem Fawaz, Kyu-Han Kim, and Kang G. Shin. Protecting Privacy of BLE Device Users. In *Proceedings of the USENIX Security Symposium*, pages 1205–1221. USENIX Association, 2016. Online at <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/fawaz>.
- [6] Sheila Frankel and Scott G. Kelly. Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec. Technical Report 4868, RFC Editor, Fremont, CA, USA, May 2007. Online at <http://www.rfc-editor.org/rfc/rfc4868.txt>.
- [7] Fabian Giesen. Ring buffers and queues, December 2010. Online at <https://fgiesen.wordpress.com/2010/12/14/ring-buffers-and-queues/>.
- [8] Jace H. Hall. *C Implementation of Cryptographic Algorithms*. Texas Instruments, Dallas, Texas, July 2013. Rev. A, Online at <http://www.ti.com/lit/an/slaa547a/slaa547a.pdf>.
- [9] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearnson, Kevin Freeman, Sarah Lord, Ryan Halter, David Kotz, and Jacob Sorber. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 216–229. ACM, November 2016. DOI 10.1145/2994551.2994554.
- [10] Jongsung Kim, Alex Biryukov, Bart Preneel, and Seokhie Hong. On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1. *IACR Cryptology*, 2006, 2006. Online at <https://www.semanticscholar.org/paper/On-the-Security-of-HMAC-and-NMAC-Based-on-HAVAL-MD-Kim-Biryukov/23843d19ce3ecd348ea7ccae08fc47b3a26d50a9>.
- [11] mbed IoT Device Platform. *Firmware Over the Air FOTA Updates*, 2017. Online at <https://developer.mbed.org/teams/Bluetooth-Low-Energy/wiki/Firmware-Over-the-Air-FOTA-Updates>.
- [12] Mike Ryan. Bluetooth: With Low Energy Comes Low Security. In *Proceedings of the USENIX Workshop on Offensive Technologies*, Washington, D.C., 2013. USENIX. Online at <https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan>.
- [13] Texas Instruments, Dallas, Texas. *MSP430FR58xx, MSP430FR59xx, MSP430FR68xx, and MSP430FR69xx Family User's Guide*, January 2017. Online at <http://www.ti.com/lit/ug/slau367m/slau367m.pdf>.
- [14] Texas Instruments. *MSP430FR698x(1), MSP430FR598x(1) Mixed-Signal Microcontrollers* Rev. C, March 2017. Online at <http://www.ti.com/product/MSP430FR5989/datasheet>.
- [15] Texas Instruments. *MSP430FR5x8x, MSP430FR692x, MSP430FR6x7x, MSP430FR6x8x Code Examples*, August 2016. Rev. F, Online at <http://www.ti.com/product/MSP430FR5989/toolssoftware>.
- [16] Sean Turner and Lily Chen. RFC 6151: Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms. Internet Requests for Comment, March 2011. Online at <http://www.rfc-editor.org/rfc/rfc6151.txt>.