Beetle: Flexible Communication for Bluetooth Low Energy

Amit Levy, James Hong, Laurynas Riliskis, Philip Levis, Keith Winstein
Computer Science Department, Stanford University
Stanford, CA, U.S.A.
levya,jhong16,lauril,pal,keithw@cs.stanford.edu

ABSTRACT

The next generation of computing peripherals will be low-power ubiquitous computing devices such as door locks, smart watches, and heart rate monitors. Bluetooth Low Energy is a primary protocol for connecting such peripherals to mobile and gateway devices. Current operating system support for Bluetooth Low Energy forces peripherals into vertical application silos. As a result, simple, intuitive applications such as opening a door with a smart watch or simultaneously logging and viewing heart rate data are impossible. We present Beetle, a new hardware interface that virtualizes peripherals at the application layer, allowing safe access by multiple programs without requiring the operating system to understand hardware functionality, fine-grained access control to peripheral device resources, and transparent access to peripherals connected over the network. We describe a series of novel applications that are impossible with existing abstractions but simple to implement with Beetle.

1. INTRODUCTION

As a mobile device moves through the world, it encounters hundreds or thousands of different peripherals it might interact with. In the space of wireless peripherals, personal area networks such as Bluetooth Low Energy have become increasingly common due to low hardware costs and support for ultra-low power operation: an integrated 32-bit processor and Bluetooth Low Energy radio can last a year on a coin cell battery and costs ≈$1. [19] Bluetooth Low Energy is particularly attractive due to a highly structured application-level protocol which enables interoperability between peripherals and applications. Today, there are Bluetooth Low Energy health trackers [2], medical devices [13], basketballs [24], door locks [11], fishing lures [17], light bulbs [16], and smart watches [15].

Current operating system abstractions are not designed to handle this peripheral proliferation. Traditionally, OS support for peripherals has come through standard, well-defined interfaces. For instance, frame buffers for displays, USB human interface devices (HIDs) for keyboards and mice, and LPD for printers. When a peripheral performs a function not anticipated by the OS (e.g., a logic analyzer), the operating system exposes the peripheral to an application as a raw hardware channel. As it has no knowledge of the underlying operations or semantics of the interface, the OS cannot safely multiplex access to the peripheral or enforce a security policy.

The diversity of available Bluetooth Low Energy peripherals makes it impractical for operating systems to anticipate the function of each new device. Instead, today's operating systems impose frustrating constraints on how applications may access Bluetooth Low Energy peripherals:

- Only a single application (process) can access a peripheral. This forces vertical software stacks (application, cloud, etc.) despite many Bluetooth Low Energy peripherals ability to interoperate with many applications. For example, many applications understand how to read data from a cycling cadence meter, but only one can do so at a time.

- The operating system does not impose any restrictions on how an application may use a peripheral. Once the an application connects to a peripheral, the application has complete control it. Allowing an application to check the battery charge of a glucose monitor *also* allows it to read a patient's blood sugar level.

- Only directly connected Bluetooth Low Energy peers may communicate with each other. As a result, applications on different computers cannot communicate with the same peripheral, and peripherals have no way of communicating with each other. For example, in order for a wireless switch to control a wireless light, a gateway they are both directly connected to must have an application that mediates that particular interaction.

This paper presents Beetle, an operating system service that mediates access between applications and mobile peripherals using GATT (Generic Attribute Profile), the application-level protocol of Bluetooth Low Energy. While GATT was designed for low-power personal area networks, its connection-oriented interface, naming hierarchy, and transactional semantics give sufficient structure for an OS to manage and understand application behavior and properly manage access.

A Beetle enabled operating system can support a peripheral's function without understanding it or knowing what it

MobiSys'16, June 25 - 30, 2016, Singapore, Singapore

ISBN 978-1-4503-4269-8/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2906388.2906414

is. Beetle leverages the transactional nature of Bluetooth Low Energy to provide isolation to multiple applications concurrently accessing the device. Beetle takes advantage of Bluetooth Low Energy's structured protocol to enforce fine-grained security policies. Beetle is agnostic to the link used to connect to a peripheral: the peripheral may be connected directly over Bluetooth, remotely over TCP/IP or even emulated by a local process. Beetle includes a simple policy language that allows the operating system, the community or device manufacturers to provide sane default access and security policies.

We have implemented Beetle for Linux and Android (Section 4) and built three end-to-end applications that demonstrate Beetle's capabilities (Section 5). Each of these three applications is impossible with existing operating systems today:

- A battery monitoring application that collects battery levels from all nearby peripherals, The monitoring application is restricted to reading only battery levels but other applications can simultaneously have more permissive access.
- Two applications concurrently read the current heart rate from a heart-rate monitor. Both applications can make configuration changes (such as turning notifications on or off) without impacting the other.
- A door-lock that can be accessed by different types of users—*residents* can lock and unlock the door while *observers* can only see whether the door is currently locked. The door-lock connects through a generic home gateway, rather than one built specifically for the lock.

Our evaluation (Section 5) shows that Beetle's performance is comparable to or better than that of existing operating system Bluetooth Low Energy interfaces, even when peripherals are accessed from multiple apps concurrently.

Finally, we discuss some of the limitations of Beetle and how changes to the operating system or extensions to the Bluetooth Low Energy protocol might help resolve some of these limitations.

2. BACKGROUND & MOTIVATION

This section provides relevant details on the Bluetooth Low Energy protocol stack and how operating systems such as Linux and Android present the stack to applications. The limitations and restrictions in these APIs motivate requirements for a new abstraction.

2.1 Bluetooth Low Energy

Bluetooth Low Energy is an ultra-low power wireless protocol for single-hop networks at ranges of 1-10 meters. A Bluetooth Low Energy peripheral's average power draw can be low enough ($< 10\mu$A) to run for weeks, months or even years on a small, coin-cell battery.

Every Bluetooth Low Energy device has one of two roles. A *peripheral* (such as a heart-rate monitor, door lock or smart watch) can only connect to a single *central*[1] (such as a mobile phone or laptop) but many peripherals can connect to the same central. Bluetooth Low Energy networks therefore form star topologies, with peripherals at the spokes and a central as the hub. Bluetooth Low Energy's energy efficiency comes from tight time synchronization between a central and its peripherals, with the central deciding the communication schedule.

[1]Version 4.2 of the Bluetooth specification allows Peripherals to maintain an active connection with more than one Central at a time. However, it is very rare for actual devices to do so.

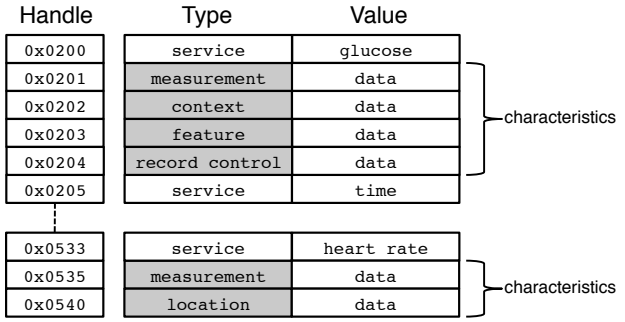

Figure 1: Bluetooth Low Energy attributes, services, characteristics, and handles.

Bluetooth Low Energy provides an application-level protocol called GATT (Generic Attribute Profile) on top of a link-layer connection (called L2CAP). Each side of a GATT connection exposes data objects through key-type-value tuples called *attributes*. An attribute's key, called a *handle*, is 16 bits long and is local to a connection (like OS-assigned ports for TCP and UDP connections). The type field is a 128-bit UUID. Using GATT, a central or peripheral can access attributes on the other using atomic commands like `READ`, `WRITE`, and `NOTIFY`.

GATT attributes provides a structured, fixed-depth hierarchical namespace, as shown in Figure 1. A device has zero or more *service* attributes. Each service attribute specifies a collection of related data values called *characteristics*. For example a heart rate monitor service provides characteristics on heart rate, location of the monitor, and battery level, while a lock service has a characteristic for whether the door is locked or unlocked. Characteristics can be read-only, write-only, or read-write.

The Bluetooth SIG defines common services and characteristics. For example, the battery service [4] defines a service and characteristics that battery powered devices can expose their battery level through, allowing a battery monitoring application to interoperate with any such device. In addition, developers can define their own services and characteristics. For example, the Angel Sensor (an activity monitor) [20] defines services for activity monitoring (e.g. step count, fall detection, etc.) allowing third-party client applications to interact with features not yet standardized by the Bluetooth SIG.

2.2 Existing OS support for BLE

As Figure 2 shows, operating systems expose the connection-oriented nature of GATT directly to applications. Linux and Windows, for example, provide a socket over which an application sends and receives binary GATT messages. Android, following its general object-oriented architecture, has an API of remote procedure calls and Java classes.

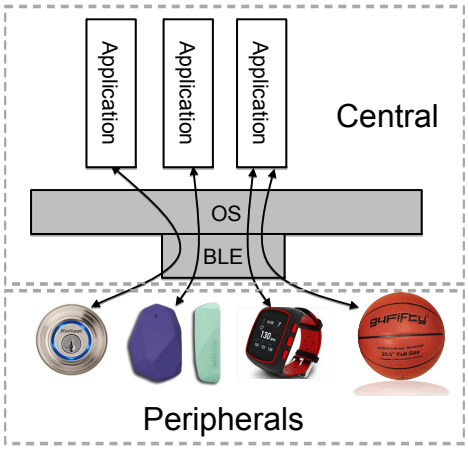

Figure 2: Bluetooth Low Energy communication pattern. Peripherals can only communicate with a central, and each peripheral has at most one open connection.

Existing operating system interfaces expose that a peripheral can have at most one connection. In Windows, OS X and Linux, only one application may open a socket to a peripheral, because in order to open a new socket the operating system always tries to establish a new connection to the peripheral. Android and OS X applications first scan for a peripheral device and connect when a callback from the system provides them an opaque handle to an object representing the device. Because scanning is the only way to get such a handle, applications cannot obtain one once the peripheral is connected to another application.

It is sometimes, none-the-less, possible to share a peripheral between applications. On OS X and Linux, for example, a process can explicitly pass a socket to child processes. On Android, if two applications happen to be scanning for peripherals simultaneously, they *may* be able to both obtain a handle to the same connection. However, in these cases, existing operating systems do nothing to help isolate applications' use of the connection from one another. For example, on Linux, each packet sent from the peripheral will go to one or the other application, but not both.

Moreover, existing operating systems provide all or nothing access to a peripheral. It is not possible, for example, to allow an application to access some services on a device but not others or to allow an application to read from but not write to characteristics. On both Android and iOS, applications can specify particular services to look for when scanning for peripherals – e.g. an application may only care about peripherals that have the Heart Rate service. However, this is merely used to filter which peripherals an application sees while scanning. Once the application establishes a connection to the peripheral it has full access to all of the peripherals services and characteristics.

2.3 Application Needs

Bluetooth Low Energy's communication restrictions and operating system APIs to peripherals prevent many applications and use cases that would seem natural and simple. We present three examples: a heart rate monitor used by multiple applications, monitoring the battery level of all of your peripherals, and managing a smart lock as well as opening it with a smart watch.

2.3.1 *Battery Monitoring*

The battery monitoring application reads the battery state of all of the peripherals paired with your phone and displays this information on a single screen. It provides warnings when peripheral batteries are low. This application can only access battery state information: it can tell that a scale is low on power, for example, but not read the user's weight.

This application needs to be able to read the battery status of many peripherals without preventing other applications from using those peripherals. It also needs to be restricted to only be able to read battery state.

2.3.2 *Shared Heart Tracker*

Wireless heart rate trackers are a common tool for professional as well as recreational athletes [23]. There is a wide range of software applications, with different features and interfaces. Strava [21], for example, has excellent recording capabilities, storing data in the cloud for later analysis. Other applications, such as Pear [14], provide excellent active interfaces that show the current workout.

Today, it is impossible to run both apps at the same time. A person must choose between logging data for later analysis or having a good real-time interface. Applications should be able to share data and peripherals, such that a user can gain the benefits of using multiple applications.

2.3.3 *Generic Gateways*

It is common for Bluetooth Low Energy enabled home peripherals (door-locks, lights, etc) to use a wall-powered gateway device to provide cloud connectivity. For example, the August Connect is a gateway allowing users to monitor their door lock over the Internet. The Lively smart-watch ships with a gateway that supports the emergency response features of the watch with a 3G modem that is too power consumptive to integrate into the battery powered watch.

Today, external connectivity depends on an application specific gateway for the particular device. For example, the August lock cannot connect to the Internet through the Lively gateway, and vice-versa. An increasing number of home-automation devices in the home means a similar increase in the number of gateways. Instead, a single gateway with generic ways of exporting Bluetooth Low Energy services of the network and specifying policies could serve all such devices.

2.4 Requirements

These three example applications motivate three requirements for how an operating system should provide access to peripherals:

1. **Sharing**: Multiple applications should be able to read, write and receive notifications from a peripheral. These applications should be isolated from one another, such that each one can behave as though it were the only one connected to the peripheral.

2. **Access control**: A person should be able to specify fine-grained policies about which applications can access which services and with what permissions.

3. **Many-to-many communication**: Multiple peripherals should be able to interact with one another. A

peripheral should be able to read, write and receive notifications from other peripherals.

3. DESIGN

Beetle reflects the insight that an operating system can effectively multiplex and control access to an *arbitrary* Bluetooth Low Energy peripheral—and to each of its individual attributes—by interposing itself at the layer of GATT transactions between an application and the device. These transactions include `GET`, `WRITE`, and `NOTIFY` commands.

By contrast, today's operating systems expose Bluetooth peripherals simply by routing a serialized byte-stream of GATT transactions between the device and a single application at a time. The OS doesn't try to interpret the byte-stream, even to know where one command stops and another starts. This prevents multiplexing and per-attribute access control (or even "read-only" access control).

Traditionally, an OS would provide special-case support for multiplexing certain kinds of devices at a more semantically-appropriate layer (e.g., printers via a print-specific API, or keyboards through an input bus). The assumption underlying this work is that it will not be practical to build a special-case multiplexing facility for every new type of Bluetooth Low Energy device—hence the benefit from Beetle's more general-purpose mechanism.

Beetle represents every Bluetooth Low Energy resource as a *Virtual Device*. A virtual device may be backed by a physical peripheral device, a user-level process running locally, or a network service. Virtual devices allow Beetle to enforce fine-grained access control policies, provide many-to-many communication between devices, and enhance peripheral functionality in user-space drivers.

3.1 Transaction Layer Interposition

As we described in Section 2, Bluetooth Low Energy devices use the GATT protocol to communicate with applications. GATT has a well-defined set of commands for reading, modifying, and discovering attributes. Beetle interposes at this layer, which provides enough semantic information to share a device between multiple applications while being generic across *all* Bluetooth Low Energy devices. Specifically, there are three command types that Beetle must handle: service discovery, notifications and read/write.

Read/Write.

The most common GATT commands are to read or write the value of an attribute. Typically these attributes contain a characteristic value (e.g. the current battery level) or represent an action (e.g. lock or unlock a door). For these types of attributes, Beetle simply forwards the read or write command from the application if authorized, changing only routing information in the request, and forwards back the reply (Section 3.2).

Notifications.

For some attributes, multiplexing access among applications calls for a different approach. When a GATT client wants to receive notifications of changes to a characteristic from a Bluetooth Low Energy peripheral, it subscribes to notifications by writing a "1" to the *Client Characteristic Descriptor*—a type of metadata attribute of a characteristic—and a "0" to unsubscribe. If one application subscribes while another unsubscribes for the same notifications, Beetle must handle the situation appropriately.

```
var device1 = find_heart_rate_band();
var device2 = find_cadence_meter();

var hrm = device1.find_services(HRM_SRVC)[0];
var cad = device2.find_services(CADENCE_SRVC)[0];

var my_server = beetle.create_server();

my_server.register_service(hrm);
my_server.register_service(cad);
```

Figure 3: Beetle lets users "mix and match" BLE peripherals. Above, example code to combine a heart-rate monitor and a bicycle cadence meter into one merged virtual device. The result has all the services and characteristics of the original devices, but appears like a single device across the Bluetooth bus—which allows unmodified Bluetooth peripherals to behave as if they were speaking with multiple peer devices at a time, by reading and writing with the merged virtual device. Beetle automatically renumbers the attributes in the constituent devices if necessary to avoid collisions and obey the consecutiveness requirements of the Bluetooth spec. In this example, the user is including all characteristics from the constituent devices in the combined device, but it is also possible to combine devices more selectively.

Beetle identifies writes to Client Characteristic Descriptor attributes, and maintains an associated subscription list. Beetle only writes to the Client Characteristic Descriptor when the first client subscribes or when the list becomes empty. Similarly, when the value of the underlying attribute changes and the device sends a `NOTIFY` command, Beetle consults the subscription list and forwards the command only to currently subscribed applications.

Service discovery.

GATT is a self-describing protocol. Connected devices perform discovery on each other to learn which services and characteristics each provides and with which handles they are associated. Beetle performs discovery once for each device, caches the result, and responds to discovery requests on behalf of the device. This is important for two reasons. First, when multiple applications might perform discovery for the same peripheral, it conserves peripheral device energy as only the first discovery process is performed. Second, properly dealing with notifications as described above requires Beetle to know the type of each attribute.

3.2 Virtual Devices

Interposing on the transaction layer (Section 3.1) allows multiple applications to communicate with the same device. However, it is not sufficient for controlling access to particular services and characteristics or for enabling many-to-many communication between peripherals.

Each connection between Bluetooth Low Energy devices has a GATT server and GATT client on each side of the connection. When a peripheral device is multiplexed across multiple applications, there are, logically, multiple GATT servers—one for each application. Beetle represents logical GATT servers as *Virtual Devices*. Each GATT client (be it an application, network service, or device) has a corresponding virtual device.

A virtual device may be backed by a single peripheral device, an application, a network service, or a combination of services from all of the above. It encapsulates state like notification subscriptions, applicable access control rules and attribute mappings.

Virtual devices enable many-to-many peripheral communication by re-exporting multiple device services over the same connection. For example, a virtual device might export GATT services from a connected glucose monitor as well as from a simultaneously connected heart rate monitor. An insulin pump, connected to that virtual device will see services from both devices.

Virtual devices also allow Beetle to enforce fine-grained access control by hiding services or characteristics or blocking commands from certain clients. For example, instead of an application connecting directly to a peripheral that has both the heart rate service and battery service, Beetle allows the user to connect the application to a virtual device that hides the heart rate service, so that only the battery service is exposed.

A key challenge for enabling these features is to honor GATT semantics regarding handle ordering and continuity. For example, all attributes in a GATT service (e.g. characteristics and included sub-services) must be consecutively numbered. Simply removing a characteristic from the middle of a service would create a "hole" in the service, violating the protocol.

Fortunately, the GATT protocol ensures that handles are constant for a particular connection, but does not require that handles are preserved between subsequent connections to the GATT client, or between concurrent connections to other clients. This allows Beetle to fill "holes" in the handle space by changing the handles visible to GATT clients.

Beetle keeps a mapping, for each virtual device, between virtual handles—the handles the client of that virtual device uses—and real handles on the peripheral or application. In the common case, handles do not change. However, there are two important cases where virtual handles are used.

Most Bluetooth Low Energy peripherals only support a single connection. Therefore, if a peripheral wants to consume services from multiple applications or other peripherals, those services must appear to come from the same GATT server. To support this, Beetle can combine attributes from multiple GATT servers into a single virtual device. As a result, services will likely appear at handles in the virtual device that are different from the handles in the backing peripheral or application.

To implement some access control policies (Section 3.3), Beetle may hide certain characteristics from a service. To compensate, Beetle shifts the virtual handles of the remaining attributes in the services to eliminate holes.

In Bluetooth Low Energy, if a device adds, removes or changes a service, it notifies connected GATT clients through the "Service Changed Characteristic" with a range of affected handle identifiers. A device must handle such notifications and adjust to the new settings. Beetle leverages this feature to model a change in connection status of a GATT server as a service change to the GATT client. This is important if a virtual device combines services from multiple GATT servers and one of them becomes disconnected, becomes available again, or if the access policy changes to hide or reveal an attribute in a virtual device.

```
{
  "ServerControler": "this-controller",
  "DeviceUUID": "5C:31:3E:42:BF:57",
  "ServiceUUID": "*",
  "CharacteristicUUID": "*",
  "LogicalResourceName": "my-heart-rate-monitor",
  "GatewayController": "this-controller",
  "AppGroup": "heart-rate-apps",
  "AccessMode": "rw",
  "TimeRestrictions" : "none",
  "Expiration": "never"
}
```

Figure 4: On Linux, users specify access policies in a JSON configuration file. Above, an example policy to allows read and write access to all services and characteristics on a device with a given UUID connected to this controller from any app running with effective group `heart-rate-apps` and also connected to this controller. The same rule can also be viewed or specified in the Beetle controller's web interface.

3.3 Access Control

Beetle exposes an interface to control a client's access to devices at a fine grain—allowing no access, read-only access, or read-write access to each individual attribute.

In some cases, access to a set of attributes, a whole service, or a whole device may need to be exclusive—only a single GATT client should be allowed access at a time. Although it is not common, writing to one attribute may affect the behavior of other attributes, for example, by changing the units or sampling rate of a sensor measurement.[2]

Future devices with these semantics would represent an exception to Beetle's assumption that access to a Bluetooth Low Energy peripheral may safely be multiplexed by merging read/write commands to each attribute separately. In these cases, an application should only be able to write to the attribute in question as long as no other GATT client can do so. That application may, in turn, re-expose those services in a device-specific manner.

To support these access control semantics, Beetle determines the set of rules to apply to a virtual device upon connection. Exclusivity can be enforced for an actual GATT server (an actual peripheral or application). The restriction will be carried through even to virtual devices that include services from the peripheral or application in question. More generally, Beetle can enforce exclusive access permissions across any arbitrary grouping of devices, services, and attributes under its control through the use of logical resource names. This can be useful for management purposes, for instance to enforce that one application uniformly controls all of the Bluetooth Low Energy light bulbs in a room.

The Beetle design is agnostic to exactly how access control policies are specified. The Linux implementation of Beetle uses configuration files (Figure 4), while the Android implementation prompts the user to specify a policy interactively when a connection is established (Figure 7). We also built a web interface to configure access policies on Linux for use in gateway devices. In both cases, the user specifies policy rules in terms of four columns: a service or characteristic

[2]We can't give an example of such a "problematic" device yet, because in currently standardized GATT profiles, measurement units are fixed: temperature is in Celsius, power in watts, etc. But we don't doubt that such devices may emerge in the future.

```
class HeartRateCallback extends BluetoothGattCallback
      {
  void onServicesDiscovered(BluetoothGatt gatt) {
    BluetoothGattCharacteristics hrmChar = gatt.
        getService(HRM_SRVC).getCharacteristic(
        HRM_CHAR);
    gatt.setCharacteristicNotification(hrmChar, true)
        ;
    BluetoothGattDescriptor desc = hrmChar.
        getDescriptor(CLIENT_CHAR_CONFIG_UUID);
    desc.setValue(BluetoothGattDescriptor.
        ENABLE_NOTIFICATION_VALUE);
    gatt.writeDescriptor(desc);
  }

  void onCharacteristicChanged(
      BluetoothGattCharacteristic c) {
    bpm = c.getIntValue(FORMAT_UINT8, 1);
    textBox.setText(Integer.toString(bpm, 10));
  }
}

bleScanner.startScan(new ScanCallback() {
  void onScanResult(ScanResult result) {
    BluetoothDevice hrm = result.getDevice();
    hrmGatt = hrm.connectGatt(this, false, new
        HeartRateCallback(heartRate));
  };
});
```

Figure 5: Example Android code, demonstrating the Beetle API for finding a device with a heart-rate-monitor service and subscribing to its stream of measurements. Some unused parameters and details omitted for brevity.

type, a device address, read and write flags and an exclusive flag. Both the service/characteristic and the device columns can be a wild-card, allowing the user to specify rules that apply for a particular service on all devices (e.g. the battery service on all devices that expose it) as well as all services on a particular device. Beetle has a base policy exposing the GAP service (which includes the device name and other metadata) as read-only. Additional rules are applied on top of the base policy and may override it.

4. IMPLEMENTATION

We implemented Beetle for Linux as a user-level process and for Android as a privileged system app. Linux and Android expose significantly different interfaces to applications for interacting with Bluetooth Low Energy peripherals. As a result, our Beetle implementation on each system differs significantly as well. This section describes and compares the two implementations, providing intuition on how difficult it is to incorporate Beetle into an existing operating system.

4.1 Android

Android applications interact with peripherals through an Android-specific IPC mechanism called Binder. For each peripheral type—GPS, camera, and Bluetooth Low Energy—there is a specially privileged system service that issues commands to the hardware through a C API provided by the device manufacturer.

In the case of Bluetooth Low Energy, the existing service is called `GattService`. The GattService exposes relatively high level API calls such as `startScan`, `connectGatt` and `readCharacteristic`. Figure 5 shows an example of an Android app using this API to connect to a heart rate monitor and subscribe to notifications from the heart rate characteristic.

There are a large number of existing Android apps that connect to Bluetooth Low Energy peripherals. Most of them are closed source and we could not easily modify or recompile them. In order to support unmodified existing applications our implementation of Beetle for Android is an API compatible replacement for the `GattService`.

4.1.1 Scanning for Peripherals

To scan for connectable peripherals, an Android application calls `startScan` passing in scan parameters (e.g. whether the scan should be low-power or low-latency), filters (e.g. a device name or services exposed) and a callback that accepts one scanned device per invocation.

When invoked, Beetle asks the user to approve the scan. Beetle interprets the passed in filter as a signal that the app may only require those services. Therefore, the confirmation prompt to the user allows her the option to pre-approve access to those services by the application for all connected devices.

If there are already connected devices matching the filter Beetle immediately returns them to through the app-provided callback. In addition, Beetle begins a scan for new devices. Beetle continues to return matching peripherals through the callback, both if they are found through subsequent scans or if other apps connect to those peripherals. This process continues until the app stops the scan explicitly. Most apps call `stopScan` immediately prior to connecting to their device of interest. However, this is not always the case. The battery monitor application in Section 5.1, for example, never stops scanning, as it is interested in any peripherals with the battery service that the phone may connect to in the future.

4.1.2 Establishing Connections

Once an application has discovered a device it would like to connect to, it calls `connectGatt`. Beetle establishes a connection, but does not give the application access to the connection until the user approves that the application may use that device. This approval can either be implicit or explicit. Implicit approval occurs when the user had pre-approved access to services in a scan. In this case, Beetle creates a virtual device containing only those pre-approved services and gives the application access to it. Explicit approval occurs when Beetle prompts the user asking to approve the connection and allows the user to restrict access to specific services.

4.1.3 Accessing Characteristics

Once a connection is established, Beetle generates a unique connection identifier that the application uses for all commands that access a characteristic. Beetle uses the connection identifier to store access control rules derived from user prompts during the connection phase.

The Android Bluetooth Low Energy interface does not expose GATT handles directly to applications, but instead uses characteristic and service "instance identifiers" generated by the mobile device manufacturers proprietary drivers [3]. As a result, Beetle on Android does not need to translate the handle specified in an application request to the real handle on a GATT server.

[3]In fact, on the Android devices we tested, these seem to correspond to the underlying handle. However, ordering of these identifiers is not guaranteed to be meaningful, so applications must treat them as opaque anyway.

```
beetle.scan([HRM_SRVC], function(scan_record) {
  if (scan_record.device_name == "MyHRM") {
    beetle.connect(scan_record.address, on_connect);
    return false;
  }
  return true;
});

function on_connect(device) {
  var srvc = device.find_services(HRM_SRVC)[0];
  var hrm = srvc.find_characteristics(HRM_CHAR)[0];
  hrm.subscribe(function(val) {
    console.log("Heart rate: ", val);
  });
}
```

Figure 6: Example code, demonstrating the same use-case as Figure 5 – finding and streaming heart-rate service data – in a Node.js app on Linux.

However, instance identifiers are forgeable (they are simply 32 bit integers and are generally sequential). Therefore, when an application requests to, e.g., write a characteristic, Beetle must consult the rules-set to decide whether the command is permissible.

4.2 Linux

The Linux Bluetooth subsystem [3] manages connections to Bluetooth Low Energy peripherals inside the kernel and exports a socket interface to user-level applications. To connect to a particular peripheral, an application creates an `L2CAP` socket and connects to the peripherals L2CAP address. The kernel handles connection establishment and when the `connect` system call returns, the application can communicate with the peripheral over the socket using GATT directly.

In our implementation of Beetle for Linux, on the other hand, all `L2CAP` sockets are owned by a Beetle daemon. Applications do not directly initiate connections with peripherals, but instead, the user uses a shell interface to the Beetle daemon to establish connections which applications can then access over UNIX domain sockets. However, once a connection is established, applications interact with peripherals the same—by sending and receiving GATT commands over a socket. This design differs from the current Linux Bluetooth Low Energy interface but more closely resembles the interface for basic rate Bluetooth.

An application finds peripherals to communicate with by navigating the filesystem depicted in Figure 8. Each directory represents a connected peripheral or virtual device. The contents of each directory includes the peripherals original advertising data and a a UNIX domain socket for each exposed service, named by the GATT UUID of the service. Applications access individual services by connecting to the respective UNIX domain sockets.

Once connected to a particular service, Linux applications issue GATT commands to the socket just as though they were connected over an L2CAP socket. Importantly, because GATT does not have a length field, applications must open sockets with the `SEQ_PACKET` option so only complete packets are ever returned from the `read` system call. The Beetle daemon intercepts GATT commands written to the socket and forwards them to the appropriate peripheral or virtual device.

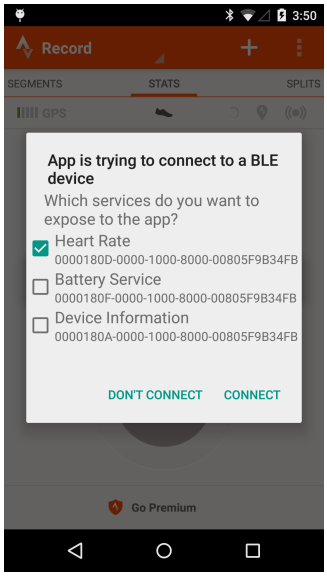

Figure 7: On Android, Beetle asks the user at runtime which services an app should be able to access if the app is not covered under any existing policy rules.

```
MyHRM/      ──►00:12:55:3D:2C:FF/

00:12:55:3D:2C:FF/
  ├──00001800-0000-1000-8000-00805F9B34FB=  # GAP Service
  ├──0000180D-0000-1000-8000-00805F9B34FB=  # Heart Rate
  ├──0000180F-0000-1000-8000-00805F9B34FB=  # Battery Service
  └──advertising_data
MyBike/     ──►E6:1D:1E:D0:16:0B/

E6:1D:1E:D0:16:0B/
  ├──00001800-0000-1000-8000-00805F9B34FB=  # GAP Service
  ├──00001816-0000-1000-8000-00805F9B34FB=  # Cadence Meter
  ├──0000180F-0000-1000-8000-00805F9B34FB=  # Battery Service
  └──advertising_data
VirtualFit/ ──►DF:19:D5:77:44:9B/

DF:19:D5:77:44:9B/
  ├──0000180D-0000-1000-8000-00805F9B34FB=    ──► ../MyHRM/00001800...
  ├──00001816-0000-1000-8000-00805F9B34FB=    ──► ../MyBike/00001800...
  └──advertising_data
```

Figure 8: On Linux, Beetle represents Bluetooth Low Energy connected devices as a directory tree. Each device is a subdirectory, with each service as a Unix-domain socket inside. A user can create a merged virtual device by making a new directory with symbolic links to existing services.

4.3 GATT over TCP

Both the Linux and Android implementations of Beetle allow users to connect to GATT services over the LAN or even the Internet. Beetle on Linux exposes peripherals or virtual devices by giving each one a particular TCP port and listening for incoming connections over that port. To expose a device, the user uses the `export_tcp` command in the Beetle shell passing in the local device address and a TCP port to listen on.

Similarly, to connect to a remote service, the user specifies the IP and port of the Beetle server. On Linux, the user uses the `connect_tcp` command in the shell. Our Android implementation has a settings activity where users can enter the remote address.

Once a connection is established, endpoints communicate using a slight extension to the GATT protocol. First, each side exchanges its device address with the other by sending it as the first six bytes. Second, all GATT commands are prefixed with their length.

Just like with L2CAP sockets, in both implementations Beetle immediately performs a service discovery for a new TCP connection. Beetle uses this step to get the same information obtained during the advertising phase, allowing it to populate the directory tree and respond to filtered scan requests, on Linux and Android, respectively.

5. EVALUATION

We evaluated the Beetle design by implementing three applications: two heart rate apps on an Android phone simultaneously accessing the same heart rate monitor, a battery monitoring app on Android that tracks the battery usage of all connected Bluetooth Low Energy peripherals, and a generic gateway device for the home that provides access control and Internet tunneling to a keyless door-lock. We also evaluated the performance of Beetle on latency and throughput metrics.

5.1 Applications

5.1.1 *Multi-app Heart Rate Monitor*

Fitness applications, such as the Strava Android app [21], often connect to a user's Bluetooth Low Energy heart-rate monitor and record the heart-rate over the course of a workout for later analysis. Unfortunately, during the workout itself, the heart-rate is not displayed prominently, and might be difficult for the user to see while, e.g., running. There are several apps that only display the heart rate or some that display it on an smart-watch like the Pebble. However, it is not possible to use Strava to record the heart rate while simultaneously using another app to view it.

In order to support this functionality on existing operating systems, one of the two applications would get exclusive access to the heart-rate monitor and expose the heart-rate data through a separate interface (on Android this would most likely be a Binder IPC service). For example, Strava could would serve heart-rate data through a specially crafted IPC service. The Pebble app would have to be designed specifically to use that interface and, importantly, would have a dependency on the Strava app or else implement a path for accessing the heart-rate monitor over both interfaces. Moreover, there is no dependency management in the Android Play store; the user must install the Strava app manually for the Pebble app to work.

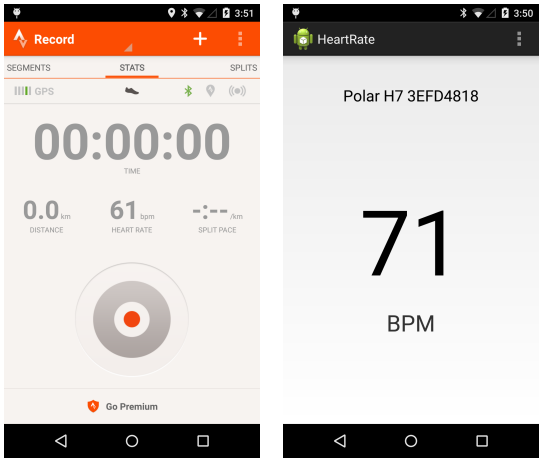

Figure 9: The Strava app (left) records the user's heart rate over time for later analysis, while our heart rate app (right) displays the current heart-rate more prominently. Both apps shown are running concurrently, connected to the same heart rate strap on the same phone.

Using Beetle, on the other hand, the desired functionality—accessing the heart-rate service from both apps—just works. We were able to run Strava, unmodified, alongside a simple app we built that just displays the current heart rate at the same time. Because Beetle keeps a subscription list for the heart-rate service data instead of issuing all subscription and unsubscription commands on behalf of applications, each app will continue to receive notifications regardless of whether the other has unsubscribed.

5.1.2 *Battery Monitor*

Many battery powered Bluetooth Low Energy devices implement the Bluetooth SIG standardized *Battery Service*. The battery service notifies a subscribing GATT client of changes in the battery level. One useful application is a dashboard that monitors the battery levels of all connected Bluetooth Low Energy devices. This dashboard could, for example, notify the user when one of her devices drops below a certain threshold.

Unfortunately, using the interfaces exposed by Linux and Android today, it is impossible to build such a dashboard without preventing other applications from using those peripherals. Since the dashboard would have to *own* the connection to all connected peripherals, the OS cannot allow other applications to safely access them at the same time. Moreover, even if the OS allowed multiple apps to access the peripheral (after all, it is unlikely for a battery monitoring dashboard and fitness application to interfere with each other in practice), the battery dashboard would be given access to *all services on all peripherals*.

Using our implementation of Beetle for Android, we were able to easily build such an app. Our dashboard, shown in Figure 10 monitors the battery level of each connected peripheral device connected to a phone. If the battery level drops below 10%, the app alerts the user.

The dashboard is a regular Android app. When launched, it makes a request to Beetle to scan for available Bluetooth Low Energy peripherals with the battery service. Beetle in-

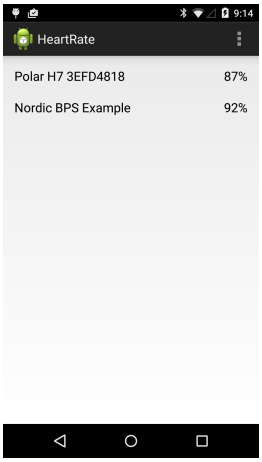

Figure 10: The battery monitor app can read the battery state of all connected Bluetooth Low Energy peripherals. It does not interfere with other applications accessing the same peripherals and it does not have access to other services.

fers from the scan request that the app might want access to the battery service across all devices. It prompts the user to approve the scan and provides the option to pre-approve restricted access to the battery service for any peripheral rather than have to approve each peripheral separately.

Once the user approves the scan request, Beetle first returns currently connected peripherals to the app. It also initiates an actual Bluetooth Low Energy scan, and continues to return unconnected peripherals to the app. Our dashboard app wants to be notified of all peripherals, even those that might connect in the future, so it does not explicitly stop the scan. Beetle automatically times out the scan on the radio after ten seconds, but if new matching peripherals are connected (e.g. by other apps) Beetle will return them to the dashboard app as well.

As soon as the dashboard app begins receiving peripherals in response to its scan request, it requests access to them from Beetle. If the user pre-approved access to the battery-service on all peripherals, Beetle does not need to prompt the user for each one and automatically exposes the battery service for each requested peripheral to the app. Otherwise, the user is prompted to approve access for each peripheral. Once the app gets access to a peripheral it subscribes to the battery level notification and updates an entry in the UI for the peripheral appropriately.

5.1.3 *Generic Home Gateway*

Many Bluetooth Low Energy enabled devices in the home require a proprietary gateway to make them accessible, e.g., over the Internet. Usually this is done to enable connectivity through a more power intensive link, such as WiFi or LTE. For example, the August smart lock uses the August Connect (a wall plug with WiFi and Bluetooth) to allow users to monitor the lock remotely. Similarly, the Lively watch (a medical alert smart-watch) comes with a dedicated gateway that bridges the Bluetooth Low Energy watch to Lively servers over the mobile phone network. One of the primary goals of Beetle is to remove reliance on different gateways for each device by directly forwarding GATT over

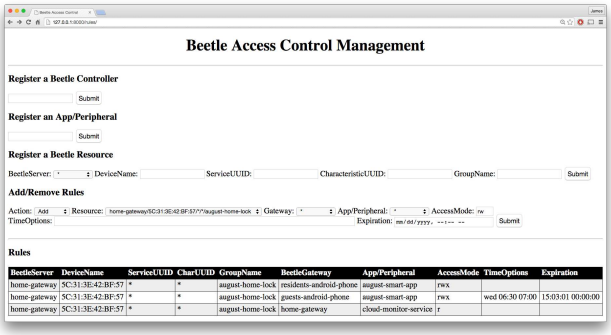

Figure 11: The access control policy web interface, with separate rules specified regarding residents, guests, and cloud-monitor for the August door lock.

non-Bluetooth links, like TCP, and providing a generic access control interface.

To evaluate our design we used Beetle to bridge an August lock with Android phones over a wireless network. We deploy a Beetle enabled Linux-based gateway machine near the lock that connects to the lock over Bluetooth Low Energy as well as to the local WiFi network. Users can connect their Beetle enabled Android phones to the door lock over TCP using the Android Beetle interface. The gateway exports the lock's GATT services over each TCP connection and Beetle running on the user's phone, in turn, re-exports the device to apps running locally—in this case, the August lock app.

Administrators grant access to different users through a general interface on the gateway. In our gateway, we use a browser based administrative interface, similar to most home WiFi routers. However, other common administrative interfaces, such as configuration files, are possible.

In the case of a door lock, there are three kinds of relevant access control rules, all of which are supported by Beetle.

1. *Residents* are people who should have full access to the lock at any time.

2. *Guests* should get full access to the lock, but only for a limited period of time.

3. *Monitors* are remote services that can monitor the state of the lock, but should not be able to lock or unlock it. For example, a cloud service might alert users via SMS when a door is unlocked at certain times during the day.

5.2 Performance

Performance is not a main goal of Beetle. However, it is nonetheless important that Beetle does not hurt performance in legacy applications and that performance for new applications is acceptable. In some cases, Beetle presents opportunities for better performance over alternative designs. We evaluated Beetle experimentally and present the results along with a discussion of ways in which to improve performance. In particular, we were concerned with round-trip latency in a multi-hop Bluetooth Low Energy network in addition to throughput and tail latency when multiple applications access the same peripheral concurrently.

We used three kinds of hardware for all of our experiments:

1. The `nrf8001` [18] Bluetooth Low Energy IC connected to an Arduino Uno [8] application controller. The `nrf8001` is a peripheral-only Bluetooth Low Energy radio. It communicates with the Arduino over an SPI bus using a high-level, proprietary protocol.

2. The `nrf51822` [19] is a Bluetooth Low Energy enabled, ARM Cortex-M0 [1] SoC. The `nrf51822` can act as both a peripheral and controller and exposes a relatively low-level Bluetooth Low Energy interface to the application controller. The ARM Cortex-M0 allowed us to perform high-resolution timing measurements on peripheral hardware.

3. A desktop with an Intel i7 quad-core CPU, 16GB of RAM and on-motherboard Bluetooth Low Energy hardware which served as our controller.

Bluetooth Low Energy performance is dominated by the connection interval—the frequency at which two connected Bluetooth Low Energy devices exchange data. Since the minimum connection interval is relatively high ($7.5ms$) the overhead of Beetle is negligible. We measured the latency between a Linux gateway application and a peripheral using connection intervals between $7.5ms$ and $960ms$, with random delays between requests. We sampled 30 requests for each interval with and without Beetle.

The average latencies were within one standard deviation of each other. For the lowest connection interval, $7.5ms$, request latency averaged $14.25ms$ ($\pm 3.52ms$) with Beetle and $13.35ms$ ($\pm 2.63ms$) without. For a $120ms$ interval, Beetle averaged $199.2ms$ ($\pm 43ms$) versus $206.4ms$ ($\pm 44.4ms$). Overall, latency averaged 1.7 times the connection interval, both with and without Beetle, with a slightly higher variance for Beetle (0.45 versus 0.37). We believe most of the variance is due to the timing of requests relative to the next connection event.

5.3 Round-trip Latency

An exception is when Beetle is used to bridge two peripheral devices. Each peripheral device's connection events with the central Beetle node occur independently. As a result, Beetle must queue a request from one peripheral to the other for the time between the two connection events.

We measured the round-trip latencies for different connection intervals in two scenarios, in both cases using the Arduino with `nrf8001` to service requests. First, we used the desktop as a client connected directly to the `nrf8001`, simulating two end-devices connected directly to each other. We also measured the round-trip latency of sending requests from a second peripheral, the `nrf51822`, with both peripherals connected to Beetle on the desktop. In both cases we performed read requests since the `nrf8001` serves those directly rather than pass them up to the Arduino. Thus we avoided measuring the cost of the SPI bus communication – which would have minimized the impact of Beetle.

We found in a single hop connection, it takes between one and two connection intervals to serve a request. This is probably due to a combination of the time spent queuing the request on the client until the next connection event and processing time on the server to serve the request.

Conversely, the overhead of adding a hop through a Beetle node is twice the round-trip latency of a single hop when the connection interval is short, and up to three times when the connection interval is long. For example, with a connection interval of 15 ms, the average round-trip time on a single hop was 26.1 ms, while with Beetle it was 58 ms. With a connection interval of 960 ms the round trip time with Beetle was 3.7 seconds versus 1.4 seconds without Beetle. Of course, peripheral-to-peripheral communication is not currently possible in Bluetooth Low Energy without an intermediate Beetle node.

We believe that most of the increase in latency is due to poor coordination of each peripheral's connection events. Currently, Beetle does nothing to manage connection timings. However, if the server's connection events were timed to occur just before and just after the client's connection events, Beetle Beetle could pass the request and response along immediately, reducing the time spent queuing. Unfortunately, this would require changes to the Bluetooth hardware on the central device to expose timing decisions to Beetle. We believe this is an interesting avenue for further research and are working on custom Bluetooth Low Energy firmware for the `nrf51822` that would enable such functionality.

5.4 Multi-application Throughput

Bluetooth Low Energy supports a limited number of data exchanges per connection event (typically 1 to 6). As a result, multiple apps accessing the same peripheral device will be unable to, naively, achieve linear increases in throughput. We evaluated the impact on throughput in the worst case (where each request from an app *must* be forwarded to the peripheral) as well as in cases like cacheable read requests or notifications, where Beetle can achieve linear throughput despite hardware restrictions.

In our experiments, we used one `nrf8001` as the GATT server with a connection interval of 25 ms connected to a desktop running Beetle and between one and 180 client applications issuing requests in a closed loop. For uncacheable commands (`WRITE` commands), overall throughput is constant at 39 queries per second (just under one command per connection event). Conversely, cacheable reads scaled linearly with the number of clients, reaching 7092 queries per second for 180 clients (just under one request per client per connection event).

6. RELATED WORK

The Bluetooth Low Energy protocol specification is large and complex, almost 3000 pages long. We have explained only a small number of pertinent details on how the protocol works. For example, we did not discuss its security model and the implications for Beetle. Curious readers can read more in several good overviews [10, 7] or the specification itself [5].

There has been extensive prior work on flexible network architectures for IP networks and some for Bluetooth Low Energy. In addition, existing operating systems have added some provisions to allow more flexibile communication with Bluetooth Low Energy peripherals in particular classes of applications. We summarize some of the work in both cases.

REST [9] describes the architectural principals underpinning the World Wide Web. Bluetooth Low Energy adheres to some of REST's principals, such the client-server model. Beetle shares a philosophical motivation with REST but differs in technical details. Beetle draws on principals of REST, specifically statelesness and cachable resources, to extend

Bluetooth Low Energy support in the operating system. Most notably, Beetle adds a layered system on top of Bluetooth Low Energy, where clients do not necessarily know if they are directly connected to the server or through a virtual device. Unlike REST, Beetle does not allow servers to extend the functionality of clients through code extensions.

Zachariah et al. [25] point out the need to use gateway devices to connect Bluetooth Low Energy peripherals to the broader internet. Beetle focuses on compatibility with existing peripherals as well as on communication between peripherals. Moreover, their security and access control model focuses on protecting the gateway while Beetle's protects peripheral devices.

Discovery systems like multicast DNS (mDNS) [6] and Universal Plug and Play (UPnP) [22] allow peers on an IP network to advertise and discover services (e.g. a printer) in a decentralized fashion. Clients reach discovered services directly over IP and use out-of-band means for authentication or access control (e.g. Kerberos, SSL). Bluetooth Low Energy peripherals cannot communicate directly and have no means of multi-client authentication or access control of. Beetle addresses all three problems (service discovery, many-to-many communication and access control) by allowing the controller to act as a mediator.

Android's Google Fit and iOS's HealthKit allow applications to access health and fitness related data, including from Bluetooth Low Energy connected heart rate monitors and pedometers. While these systems enable sharing of these devices in a safe manner, they are restricted to a narrow use case and they do not allow access to other services on of the peripherals like the battery level or unsupported sensors (e.g. body temperature).

Early research in sensor networks and ultra-low power wireless networks concluded that the connection-oriented nature of Bluetooth was a poor fit [12]. These networks, however, focused on multi-hop wireless meshes, which forced nodes to simultaneously be centrals and peripherals. Personal area networks, in contrast, fit well with Bluetooth's single-hop communication model. The emergence of proximity networks and Bluetooth Low Energy's dominance of them suggests that it is worth revisiting this decade-old conclusion.

7. LIMITATIONS & FUTURE WORK

While our results our encouraging there are several important limitations and avenues for future work. First, as noted in Section 3, Beetle cannot currently handle peripherals that do not conform to GATT's transactional semantics. In particular a GATT server where the ordering of writes to one attribute would affect the value read from another could not be exposed directly to multiple applications. For such devices, users can give one application exclusive access to that attribute and that application could re-expose the peripheral as a virtual device in an device-specific, safe manner.

Beetle's access control design assumes that users can easily name both peripherals and applications. All peripherals have addresses which are (statistically) unique but not human-readable and human readable names that are likely to collide. For example, all August locks are named "August 57". Furthermore, both addresses and names are forgeable, so it is up to the user to verify that the device they are connecting to is in fact the one named in their policy. On the other side of the connection, while Android has a notion of an application (each application gets a unique user ID), Linux does not. As a result, our implementation of Beetle on Linux can enforce access control policies on users running particular processes, but not on applications.

8. CONCLUSION

Beetle allows the operating system to provide a generic interface to Bluetooth Low Energy peripherals while satisfying our three goals:

1. **sharing** access to peripherals between multiple applications
2. fined grained **access control**, and
3. **many-to-many communication** between Bluetooth Low Energy peripherals.

Beetle is able to achieve these goals by leveraging three important properties of the GATT protocol. First, the GATT protocol is used by all Bluetooth Low Energy peripherals, regardless of their particular function, allowing Beetle to interpose on the communication between apps and peripherals without understanding the content of the communication. Second, the transactional nature of GATT means that retaining atomicity of individual commands is sufficient to provide isolation for the majority of peripherals. Finally, the GATT naming hierarchy of devices, services and characteristics allow peripherals to describe their functionality directly to applications.

We demonstrated the feasibility of Beetle by building prototype implementations for Android and Linux. Using these prototypes, we built three applications that are simple, but nonetheless impossible today. We also evaluated the performance of Beetle and demonstrated that virtualizing Bluetooth Low Energy connections at the application layer does not significantly limit performance.

By transforming Bluetooth Low Energy from a star topology to one in which peripherals can communicate, Beetle elevates it from a link-layer to a network-layer abstraction, with controllers routing data at the application level. This forwarding model resembles virtual circuits, giving a router explicit knowledge of each flow that might pass through it. This is a good thing: the security implications of Internet-style default-on connectivity could have disastrous implications to personal devices that are continually coming into range of potential new attackers. Beetle's properties suggest some ways in which the constraints and technologies of "Internet of Things" may lead it to differ from the Internet of today.

Acknowledgments

We would like to thank Jason Flinn, David Mazières, David Terei and the anonymous reviewers for their thoughtful comments. This work was supported by Intel/NSF CPS Security grant #1505728, the Secure Internet of Things Project, and gifts from VMware and Analog Devices International.

9. REFERENCES

[1] ARM. Arm cortex-m0 product specification, 2014.

[2] BASIS Science Inc. Basis health tracker. http://www.mybasis.com/, Sept. 2014.

[3] Beutel, J., and Krasnyanskiy, M. Linux bluez howto, 2014.

[4] Bluetooth SIG. Battery service specification, Dec. 2011.

[5] Bluetooth SIG. Bluetooth core specification 4.1, December 2013.

[6] Cheshire, S., and Krochmal, M. Multicast dns. RFC 6762, IEFT, Feb. 2013.

[7] Decuir, J. Introducing bluetooth smart: Part ii: Applications and updates. *Consumer Electronics Magazine, IEEE 3*, 2 (April 2014), 25–29.

[8] DâĂŹAusilio, A. Arduino: A low-cost multipurpose lab equipment. *Behavior research methods 44*, 2 (2012), 305–313.

[9] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[10] Gomez, C., Oller, J., and Paradells, J. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors 12*, 9 (2012), 11734–11753.

[11] Kwikset. Door locks. http://www.kwikset.com/kevo/default.aspx, Sept. 2014.

[12] Leopold, M., Dydensborg, M. B., and Bonnet, P. Bluetooth and sensor networks: A reality check. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems* (New York, NY, USA, 2003), SenSys '03, ACM, pp. 103–113.

[13] OneTouch. Veriosync blood glucose meter. http://www.onetouch.com/veriosync, Mar. 2015.

[14] Pear. Pear mobile bluetooth heart rate monitor. http://pearsports.com/shop/pear-bluetooth-heart-rate-strap.html, Mar. 2015.

[15] Pebble, I. Pebble Smartwatch. https://getpebble.com, Mar. 2015.

[16] Phillips. HUE: Personal Wireless Lighting. http://www2.meethue.com/en-us/, Mar. 2015.

[17] ReelSonar. Fish finders. http://reelsonar.com/, Sept. 2014.

[18] Semi, N. nrf8001 single-chip bluetooth bluetooth low energy product specification 1.2, 2013.

[19] Semi, N. nrf51822 product specification, 2014.

[20] Seraphim Sense Ltd. Angel Sensor - Open Mobile Health Wearable. http://angelsensor.com, Apr. 2016.

[21] Strava. Running and cycling gps tracker, performance analytics, maps, clubs and competition. https://www.strava.com, Mar. 2015.

[22] UPnP Forum Technical Committee. Upnp device architecture v1.0, Sept. 2014.

[23] USA, P. H7 heart rate sensor. http://www.polar.com/us-en/products/accessories/H7_heart_rate_sensor, Mar. 2015.

[24] Wilson Basketball. Smart basketballs. http://www.wilson.com/smart/, Sept. 2014.

[25] Zachariah, T., Klugman, N., Campbell, B., Adkins, J., Jackson, N., and Dutta, P. The internet of things has a gateway problem. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications* (New York, NY, USA, 2015), HotMobile '15, ACM, pp. 27–32.