BLESS: Bluetooth Low Energy Service Switching using SDN

Mostafa Uddin Sarit Mukherjee Hyunseok Chang T.V. Lakshman

Nokia Bell Labs, Holmdel, USA
Email: {firstname.lastname}@nokia-bell-labs.com

***Abstract*—Bluetooth Low Energy (BLE) is a low-energy personal area wireless network technology that is of increasing importance for large-scale Internet of Things (IoT) deployment. By design, BLE is localized to small regions and to simple network topologies. Moreover, it is not designed for dynamic enforcement of policy and access controls. These limitations are impediments to BLE based IoT service deployment, where IoT devices for a service may need to be grouped into a network slice spanning a sizable geographic region and with strong per-slice policy enforcement. This paper presents an architecture for creating wide-area IoT service slices which utilize BLE devices at the edge. For this, we introduce an SDN-controlled "switch node", called BLE service switch (BLESS) that is transparently inserted between two communicating BLE devices. It can be programmed at the service layer by a central controller and enables flexible, policy-based switching between the devices. We describe the design of BLESS, and illustrate its utility through a few practical use cases.**

I. Introduction

Bluetooth Low Energy (BLE) [1] is undergoing wide adoption for interconnecting a large number of wireless devices, particularly with the growth in the Internet of Things (IoT). BLE devices are finding applications in medical and health care services, home automation, factory automation, sensing and control, consumer electronics, smart phones and tablets, etc [2]. It is expected that by 2020, the global value of BLE devices in IoT technology could be as much as $5.57 billion, with most of the value coming from devices in health care and manufacturing [3]. This wide adoption of BLE technology can be attributed to two of its major features [2]: a simple low power physical and data link layer that is suited for low cost devices with long lasting operating life on battery; and client-server based service layer (comprising of Attribute Protocol (ATT) and Generic Attribute Profile (GATT) with ready-made and extensible service profiles) that catalyzes fast development of different services on devices.

As new and feature-rich devices with useful service profiles and capabilities are introduced, the scope for creating a variety of practical, large scale, networked services is expanding [4]. Consider for example a large scale health care service (beyond current personal health care monitors like FitBit) that is based on an automated patient care system in a hospital using different types of BLE devices [5]. Devices used for such a service broadly fall into two categories [6]. Peripheral devices like heart rate monitors, blood pressure monitors, weighing machines are used at the patients' end to measure and report patient vitals. Central devices like smart phones, tablets, laptops and other GUI devices are used by the medical and administrative professionals to subscribe to services offered by the peripheral devices, systematically read them, organize patient records, and take appropriate actions.

In such an environment, it is desirable to create different network service slices that are overlaid on the large number of IoT devices underneath. For example, a physician's slice consists of a private registered central device and all the health related services provided by the peripheral devices that connect to this physician's patients, regardless of the distance (within the premise or outside when a patient with wearable devices is monitored remotely). Similarly, an administrative professional's slice may span across his central devices and the battery status service of all the peripheral devices. These two slices must remain isolated at the service layer, but do overlap at the device connectivity layer. Slices may overlap at the service layer, for example, when a patient is treated by more than one physician. Needless to mention, network service slices must be dynamically created, and be amenable to real-time membership changes in both central and peripheral devices and/or the services. The overall service depends on the underlying communication amongst peripheral and central devices that constitutes a BLE network. Communication and networking requirements in a sizable geographic area (beyond BLE's range) with a large number of devices can be categorized as follows:

Connectivity and reachability: Different types of devices that are part of a service are connected to the BLE network and are reachable when required. As new devices join the service, they also get connected into the same network and become reachable from other devices. For example, when a medical professional wants to check the vitals of a patient, the corresponding peripherals are readable from the central device regardless of the distance.

Policy and access control: Access to a peripheral device by a central one is governed by some policy so that security and social responsibility of the service can be ensured. The policy is implemented and enforced on each device to device communication in real-time. In addition, the policy can be updated without any service interruption. For example, a heart rate monitor is accessible by both the medical and administrative professionals; while the former can read the patient vitals, the latter can check only the battery status.

BLE standard does not address either of these two requirements adequately for a large scale networked service environment. BLE is a connection-oriented peer-to-peer communication technology that is extendable to a star topology, where a central device can connect to multiple peripheral devices within a short range. Although the most recent version (where a peripheral device can connect to multiple central devices) allows the creation of a mesh, the BLE protocol does not support packet routing natively. So reachability from a device to other devices is severely limited unless the devices are directly connected. Access control to a device in BLE is mostly limited to the link layer during connection setup. As a consequence, once a central device gets connected to a peripheral one, the former can access any service offered by the latter. So dynamic device selection based on service is not possible at a higher layer where the service is created [7].

The focus of this paper is on addressing both shortcomings mentioned above and enable the creation of service slices, while maintaining the native BLE service layer intact and not entailing changes to BLE devices or applications. We do this by introducing software defined networking (SDN) into BLE networking, drawing upon the ability of SDN to perform scalable and flexible policy-driven flow switching and routing. To control BLE communication using SDN flow rules, we introduce a new "switch node", which we call the BLE Service Switch (BLESS), in the communication path between peripheral and central devices. BLESS maintains link layer connections to the devices to support peer-to-peer connectivity, but controls ATT packet flows at the service layer using flow rules that are installed by a central controller. With practical use cases, we show how incorporating SDN technology enables service slicing in a BLE network. To the best of our knowledge, this is the first contribution of using SDN technology in BLE.

II. Challenges and Contributions

The BLE standard defines devices with two key roles; peripheral and central [1]. In a BLE network, a pair of peripheral and central devices communicate in a peer-to-peer fashion. Prior to the communication, peripheral devices announce their presence and connectability via BLE advertisement packets. Any central device listening to the advertisement can initiate a connection with the peripheral device. After establishing a link layer connection, communication can go on in both directions. Typically, the central devices that run the client applications, send requests to the peripheral devices for the services they support, and the peripheral devices respond. A central device can connect to multiple peripheral devices, and similarly a peripheral device can connect to multiple central devices simultaneously.

The first and foremost challenge in incorporating SDN in BLE networking is to decide at which protocol layer packet flow rules should be introduced. In the BLE standard, if a central device can establish a link layer connection to a peripheral device, it can access all the services that the peripheral device supports. To offer flexibility in service creation, the access to the service profile (or different sub-profiles within a profile) must be dynamically controllable depending on the role of the central device (refer to the example scenario described in Section I). We choose to apply policy control at the service layer of BLE (composed of ATT and GATT) where service profiles are maintained and accessed. This implies that even if a peripheral device is connected to multiple central devices, accessibility to service profiles is controlled based on the identity of the central devices. Moreover, the access policy is dynamically modifiable from SDN's central controller so that new services can be added, and existing services can be modified without interrupting the ongoing service.

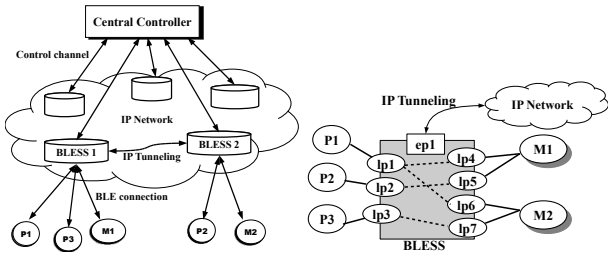
The next challenge is to determine a node in the BLE network (like a switch or router in IP network) that is suitable for applying the policy control as ATT packets flow through the node. BLE is a peer-to-peer protocol and does not require packet forwarding using an intermediary switch between the devices. Without a switch in place it is not possible to inspect the packet flow between the devices, which in turn prohibits any policy enforcement. To address this challenge, we introduce BLESS, a switch in the BLE network that is transparently inserted between the devices without violating any BLE protocol. BLESS resides in the data plane, has access to the full protocol stack, and is used to enforce policy at the service layer.

Challenges in building such a switch is manifold. Existing SDN switches are IP protocol based. Their connectionless architecture is not suitable for BLESS as BLE is natively connection oriented between the peers. We, therefore, design and develop BLESS bottom-up and make it behave as a central device to a peripheral device and a peripheral device to a central device. In its role as a peripheral device, BLESS advertises on behalf of an actual peripheral device that is connected to it. In its role as a central device, BLESS maintains a link layer connection with each peripheral device within its range. Since a central device connects to each peripheral device separately, a single link layer connection between BLESS and a central device is not enough to multiplex across different peripheral devices. Therefore, in its role as a peripheral device, BLESS maintains one connection per peripheral device that the central device needs to communicate with.

As evident from the above description, BLESS (transparently) breaks the native peer-to-peer connection model and introduces the ability to "forward" packets. This necessitates device addressing and identification in BLESS, which is not available in the service layer. BLESS adopts metadata structures and exploits ATT protocol's serialization to resolve correct packet addressing and forwarding (see Section III).

Once BLESS is inserted in a network of BLE devices, a central controller pushes ATT and GATT layer service rules into it in real-time. BLESS keeps any connection related state information in the packet forwarding path. This enables it to employ a stateless match-action packet forwarding model to examine each packet, and then to apply the action of the matched rule that enforces access and policy control.

In order to extend the reachability, we augment BLESS



(a) (b)

Fig. 1: (a) SDN controlled BLE network. (b) BLESS and BLE device link layer connection.

with backhaul IP connectivity so that multiple such switches are interconnected over an IP network (see Fig. 1a). Once a peripheral device gets connected with a BLESS node, the central controller determines the set of BLESS nodes that should announce on behalf of the device via BLE advertisements. This makes the peripheral device "reachable" to a central device that is connected to a remote BLESS, beyond the physical range of the peripheral device. If the central device wants to connect to the peripheral device, it simply responds to the advertisement packet originated from the local BLESS. For example, in Fig. 1a, BLESS 2 can advertise for peripheral device P_3 and let central device M_2 connect to it even though they could be far apart. If peripheral and central devices are connected to two different switches, SDN controller installs forwarding rules to carry the packets from one BLESS to another within an IP tunnel.

III. BLESS Architecture

In order to implement SDN control in BLE, we interconnect two communicating devices via BLESS and instantiate packet forwarding through it. When the devices get connected to BLESS, the SDN controller identifies the devices and discovers the services offered by the peripheral devices. This allows the controller to install appropriate flow rules in BLESS. Once rules are installed, BLESS examines each packet and enforces the rules using match-action paradigm. Below we describe the steps in more detail.

A. Packet forwarding using BLESS

In the link layer, BLESS needs to provide a transparent one-to-one connection between peripheral and central devices. To achieve this, BLESS responds to the advertisement of each peripheral device and sets up a link layer connection with it. Connection by peripheral devices only to BLESS can be ensured by white-listing only the BLESS nodes in them and/or by adopting an advertisement jamming scheme [8]. BLESS creates ports (for example, `lp1`, `lp2`, `lp3` of Fig. 1b) for each connected peripheral device. From then on, it advertises on behalf of the connected peripheral devices. A central device connects to BLESS at the link layer in response to such an advertisement. A central device can establish multiple link layer connections with BLESS, one for each peripheral device that it wants to communicate with. This is in conformance with BLE standard 4.1 and later that allows BLESS as a peripheral device to connect with multiple central devices. Moreover, by allowing multiple link layer connections from a central device to BLESS (instead of a single connection) the client application(s) running on the central device can remain unmodified. Note that, each link layer connection from a central device creates a port in BLESS that has one-to-one mapping to a port of a peripheral device (i.e., as shown in Fig. 1b).

After BLESS is (transparently) placed in between two communicating devices, it examines the packet flow at the service layer for enforcing access and policy control. In this layer, BLESS introduces a packet forwarding model, instead of a peer-to-peer communication model. One of the necessities of packet forwarding model is to have source and destination addresses for each packet. Unfortunately, the service layer of BLE uses L2CAP packets, that is devoid of device addresses. In our model, the central controller is responsible for assigning a unique address to each device when it connects to BLESS. A public or a static device address can be used as is since it remains fixed for the duration of the connection. Only a random address may be mapped to a fixed ephemeral address by the controller (for example, by using hashing on some of the device parameters). The address is used globally for addressing a device (including advertisement as described above) and orchestrating flow rule instantiation and packet forwarding in the network, while the device is connected to a BLESS. The address is carried as metadata with a packet in the service layer to make packet forwarding decisions.

BLESS creates additional ports for any remotely connected peripheral devices for which it advertises. It maps a central device's port connection to the port corresponding to the peripheral device the central device wants to communicate with. In BLESS, by default, packets are forwarded based on the destination device address. It is the responsibility of BLESS to gather the device address (and update the metadata field) based on where the packet is coming from. Since a central device has a unique port that connects to each peripheral device separately, an incoming packet from a central device will always have a unique destination peripheral device address. However, since a peripheral device maintains only one connection with BLESS, an incoming packet from it will be destined to BLESS, not to any central device. BLESS resolves the destination device address by leveraging ATT's Sequential Protocol feature [1], which restricts a central device to have at most one outstanding request per peripheral device at any time.

BLESS implements a serialization mechanism that emulates a sequential request-response at each peripheral device's port by controlling the flow of requests from the central devices to a peripheral device. Only the port that is locally connected with the peripheral device implements this serialization mechanism. The goal is to uniquely match the response from a peripheral device to the corresponding request to determine the destination of the response. BLESS can take a naive approach of sending one ATT request packet at a time till it receives a response from the peripheral device. In a more pipelined

approach, BLESS can send multiple ATT request packets at a time so long as they are not conflicting in terms of protocol parameters and source device's address. With this serialization mechanism, BLESS leverages the request-response pattern of ATT protocol to map the response packet with the request packet. Thus, BLESS identifies the destination address of the packet received from the peripheral device. There are a few ATT operations (i.e., Notify), which does not follow request-response model. In those cases, Controller keeps additional state information for finding out the destination device address.

There could be scenarios, where a device could have the role of both peripheral and central. For example, an Air Conditioner (AC) can act as a central device with respect to a thermostat, where AC can access the temperature service of the thermostat. At the same time, AC can also act as a peripheral for a smart phone/tablet, which runs the AC control application. In such a case, AC will have one link layer connection with BLESS as a peripheral device, and one or more link layer connection(s) as a central device.

B. Control Plane Operations

In a network of BLESS, each BLESS node connects to a central controller that is responsible for overall operation of the network. An Openflow-like protocol is used for BLESS to controller communication. Once a peripheral device connects to a BLESS, the controller learns all the services it can offer, and then makes it discoverable from one or more BLESS nodes. It also installs service access policies for the device at the relevant BLESS nodes and sets up tunneling information for remote access between two devices. Below we describe two BLE specific control applications, namely Device Discovery and Connection (DDC), and Service Discovery and Access Policy (SDAP).

Device Discovery and Connection (DDC): This application installs rules in a BLESS that sends to DDC any captured BLE advertisement packet from a peripheral device. The advertisement packet contains the device's public or random device address and the advertising data. Upon receiving the advertisement packet, DDC determines if BLESS should connect to the device, and if the policy allows, DDC commands BLESS to initiate the connection request command. Once connection is established, BLESS confirms it to DDC which then assigns a unique address for the connected peripheral device (refer to Section III-A). DDC is also responsible for assigning a unique address to a newly connected central device. DDC uses the assigned address of the peripheral device and instructs one or more (appropriate) BLESS nodes to advertise on behalf of the peripheral device using the captured advertising data.

Service Discovery and Access Policy (SDAP): This application is responsible for discovering services offered by each peripheral device connected to a BLESS. In BLE, service is expressed in terms of attribute type or UUID and the associated device specific handle for it. SDAP builds a Handle-UUID map table for each peripheral device. After the establishment of a connection with a peripheral device, SDAP initiates back and forth exchanges of multiple ATT request-response packets (e.g., *Find information request/response*) between the peripheral device and BLESS. Such exchange results in a Handle-UUID map table for the peripheral device. For any declaration type UUID, SDAP retrieves its corresponding value. This enables the controller to learn the services offered by the device. In addition, based on this table and the user provided access policy, the controller can insert appropriate flow rules in the match-action table of BLESS. When a peripheral device is disconnected from BLESS, SDAP destroys the Handle-UUID map table.

C. Data Plane Operations

BLESS's data plane works at the service layer between a pair of central and peripheral devices, where they act as client and server, respectively, for the service. The ATT protocol is used to communicate the services described using GATT protocol. The GATT server maintains a list of *attribute*'s where each entry is described as a key-type-value tuple. A fixed length attribute's key, called a *handle*, is locally unique per server-client connection. The 16 or 128 bit type field, called *attribute type* (UUID), is defined mostly by Bluetooth SIG [9]. The value field, called *attribute value*, can be variable in length based on the type. Service policy and access control in BLESS is defined (statically) in terms of attribute type. Rules are instantiated in terms of the (ephemeral) handles translated from the attribute type using the Handle-UUID map table.

BLESS uses a number of BLE packet fields and metadata (see Fig. 2) for the match-action rules. The metadata is generated at a local BLESS, and is carried along with the packet to a remote BLESS. The fields in `BLE_packet` appears in all packets. We augment it with metadata which contains relevant lower layer information like source device address (`SID`), destination device address (`DID`), packet's input port (`in_port`). The `in_port` can be a normal port or external port based on whether the packet is local to the BLESS or delivered from a remote one, respectively. The `enable` and `counter` fields are used for (re)matching the same ATT command packet multiple times through the match-action rules when the command requires checking more than one attribute types or values (e.g., *Read by type request/response*).

ATT protocol supports three categories of commands: read attributes, write attributes and attribute notification. As a data plane operation, BLESS applies match-action rules on these commands to enforce different access policies. We keep match-action rules stateless for all the ATT commands, and maintain the connection's state at the forwarding path (ports). Note that different ATT commands have different packet formats and can have different match fields. While all the ATT commands can be implemented as match-action rules, we briefly describe one popular command from each class in the following.

Read Attributes: The *Read request/Read response* ATT command pair allows a client to read attribute's value from a server using attribute's handle. The *Read request* packet contains the attribute handle which, along with a few other fields (see Fig. 3), is used by BLESS for finding a matching

```
fields BLE_packet{
     l2cap_CID     :16;
     Length        :16;
     ATT_opcode    :8;
     }
```

```
field BLE_packet_metadata{
  SID           :48;
  DID           :48;
  in_port       :16;
  enable        :8 /*initial value 1*/
  counter       :8 /*initial value 0*/
  }
```

Fig. 2: Packet fields and metadata used for match-action rules.

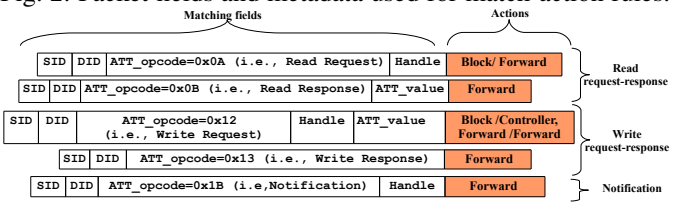


Fig. 3: Match-action rules for different ATT commands.

rule. Recall that the match-action rule contains the handle corresponding to the attribute type or UUID from the Handle-UUID mapping table. As an action, BLESS can either **forward** or **block** the Read request packet. In case of blocking, BLESS sends an ATT *Error response* packet to the client with the ATT error code "Read Not Permitted". The *Read response* packet is **forward**ed to the client.

Write Attributes: The *Write request/Write response* command pair allows a client to write a single attribute value at the server based on the handle information. When a request packet passes through BLESS, based on the handle information, BLESS either **block**s or **forward**s the packet. The response packet is simply **forward**ed to the destination. There is a special case of write request when a client subscribes to a service attribute for notification from the server. In this case, the client first sends a write request with the attribute handle and value information to the server. If the attribute handle corresponds to a service attribute of type *client characteristic configuration* (i.e., 0x2902), then, in addition to **forward** the request packet to the server, BLESS applies action (i.e., `controller`) to send the packet to the controller as "packet in" message. This allow controller to create the set of client devices, referred to as Notify-Set, that subscribes to the notification for the corresponding service attribute.

Attribute Notification: This is a server initiated ATT command that sends attribute notification to the subscribed clients. In our model, a server sends only one notification packet for an attribute value as it connects to only one BLESS acting as a central device's client. It is the responsibility of the controller to push rules in the BLESS nodes to multicast the notification packet to each client in Notify-Set.

Besides the above mentioned actions, BLESS also uses a few more to handle more involved ATT commands. For example, **Rematch** action is used to match the packet with the match-action rules repetitively. This is useful for ATT commands that carry a list of attributes. Before rematching, this action updates metadata fields such as `counter`[1] and `enable`.[2]

[1] `counter` is used as an index that increments by one each time **Rematch** action is performed on a given packet.

[2] `enable` (1 or 0) indicates whether or not **Rematch** action can be applied for the matched rule.

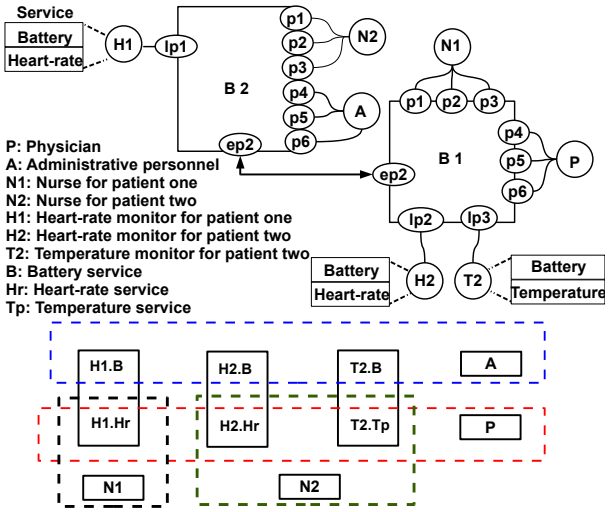


Fig. 4: The hospital scenario with BLESS. Dotted rectangles depict network slices among different groups of BLE devices and services (represented in solid rectangles).

The **RemoveATTValue** action removes the matching attribute values from a packet, and updates the rest of the packet content accordingly.

IV. Example Use cases

To illustrate the inner workings of BLESS, we use the network service slicing example described in Section I, where service slices are created, on which access control is enforced. Fig. 4 shows the hospital scenario with central and peripheral devices and their roles. Fig. 4 depicts the GATT services offered by the peripheral devices. Also shown are several network service slices created using different devices and services. As evident, slices can be independent (e.g., Blue and Red) or overlapping at various services (e.g., Red and Green). BLESS ensures that each slice is created and maintained in isolation without interrupting any ongoing services. To illustrate further, consider the blue dotted rectangle representing the administrative professional's network service slice composed of his central device **A** and the battery services of all the peripheral devices **H1**, **H2**, and **T2**. BLESS must ensure that in this slice, device **A** can only access the battery status information from the peripheral devices. Fig. 5 shows the match-action rules needed to apply such service access policy. Match-action rules are listed in decreasing order of priority (priorites are needed when a packet matches several rules).

We describe the operation of the slice assuming that the link layer connections are already set up. There are a few different ATT commands that **A** can use to access the service from the peripheral devices. We describe the steps when *Read request/Read response* pair is used between **A** and **H2**.

First, **B2**, physically connected to **A**, receive *Read request* packet to port `p5`, which directly maps with the port `lp2` at **B1**. If the packet carries handle = 0x0202 (assume which corresponds to the attribute of "battery status" in the heart rate monitor **H2**), then based on rule ① (which has higher priority than rule ②), **B2** forwards the packet to port `ep2`, which

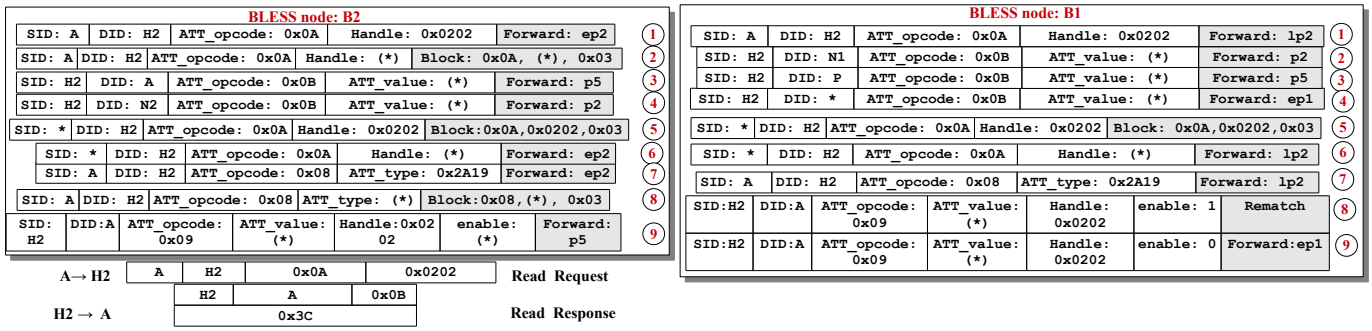


Fig. 5: Match-action rules in the BLESS nodes for the service access policies.

ultimately tunnels the packet to **B1**. At **B1** rule ① triggers and the packet is forwarded to **H2** through port `lp2`. Note that *Read request* packet from **A** with any other handle value triggers rule ② in **B2**, which blocks the request from entering the slice. When **H2** replies with *Read response* packet, the packet matches rule ② at **B1**, which forwards the packet to **B2** through port `ep1`. At **B2** the packet matches rule ③, which forwards the packet to **A** through port `p5`. Rules ⑤ in **B2** and rules ⑤ in **B1**, restrict the access to the battery status service from any central device other than **A**, when *Read request/Read response* command pair is used.

Note that device **A** can also access the service from **H2** using other read attribute commands. For example, in case of *Read by type request/Read by type response* (i.e., ATT_opcode = 0x08/0x09), Fig. 5 shows the match-action rules for restricting **A** to access only the battery status service from **H2** (using rules ⑦ to ⑨ in **B2** and rules ⑦ to ⑨ in **B1**). These match-action rules involve more complex actions such as *Rematch* and *RemoveATTValue* as the command may carry a list of attributes. The details are not included due to lack of space.

V. Prior Work

IPv6 over BLE [6] has been proposed for wider reachability of BLE devices across the Internet. However, TCP/IP-based solutions are not suitable in this environment as BLE's very small MTU size significantly degrades TCP/IP protocol performance. In a mesh topology, routing becomes far more expensive, and IP-multicasting for notification becomes inefficient due to time-to-time sleep mode of BLE nodes [10]. Running BLE's native service layer (ATT, GATT) over IPv6 introduces extra burden on the resource contained, low power devices. In addition, such design choices require changes in BLE's native implementation (e.g., BlueZ [11]). In order to preserve BLE's native service layer, and at the same time address reachability, BLE service-IP gateways have been proposed [7], [5], [12], [13]. In this architecture, a service gateway maintains BLE connection to a device, and connects to a remote service agent over the Internet. The gateway translates BLE actions into a form understood by the service agent, and vice versa. However, this requires development of special application level or operating system support (e.g., programming model [5] or virtualization of BLE device [7]). Similar to BLESS, Beetle [7] has introduced the requirement of flexible and policy controlled access to services from BLE devices. However, their solution of virtualizing BLE devices intrinsically requires application level proxies and content caches at the gateway device. This is inefficient and does not scalable in dynamic network environment with intermittent connectivity. Cloud-based access control approaches [14], [15] offload all policy decisions to the cloud via IoT gateways. This not only requires all device-to-device communication to detour via the cloud, but also fails to support unmodified client applications. In fact, BLESS does not preclude cloud integration. If needed BLESS can forward traffic to the cloud with backhaul IP connectivity.

VI. Discussion

In this paper, we make a case for network service slicing in a large scale BLE-based IoT service deployment, and its broader implication of dynamic policy and access controls. Towards that effort, we introduce a "switch node", called BLESS, that resides transparently between the link layer connection of peripheral and central devices, and controls packet flow at the service layer. In BLE security encryption happens at the Link Layer, therefore BLESS introduce no additional concern of "man-in-the-middle" attack. We aim at exploring the challenges and the design issues of realizing BLESS . In the future, we intend to report on the feasibility of building BLESS using P4 [16] paradigm with special focus on construction of optimal match-action rule set, scalable connection state management etc., so that overall system performance is maximized in a large scale deployment with keeping the native BLE security aspect intact.

References

[1] "Bluetooth Core Specification Version 4.2," 2014.
[2] C. Gomez *et al.*, "Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology," *Sensors*, 2012.
[3] "Bluetooth SIG 2014 Annual Report," 2014.
[4] T. Yu *et al.*, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things," in *ACM HotNets*, 2015.
[5] W. McGrath *et al.*, "Fabryq: Using Phones as Gateways to Prototype Internet of Things Applications Using Web Scripting," in *ACM SIGCHI*, 2015.
[6] J. Nieminen *et al.*, "IPv6 over BLUETOOTH(R) Low Energy," Internet Requests for Comments, RFC 7668, October 2015.
[7] A. A. Levy *et al.*, "Beetle: Flexible Communication for Bluetooth Low Energy," in *ACM MobiSys*, 2016.
[8] K. Fawaz *et al.*, "Protecting Privacy of BLE Device Users," in *USENIX Security*, 2016.
[9] "Bluetooth SIG Specification."
[10] W. Shang *et al.*, "Challenges in IoT Networking via TCP/IP Architecture," NDN Project, Tech. Rep., 2016.

[11] "BlueZ," http://www.bluez.org.
[12] M. Andersson, "Use case possibilities with Bluetooth low energy in IoT applications," *White Paper*, 2014.
[13] T. Zachariah *et al.*, "The Internet of Things Has a Gateway Problem," in *ACM HotMobile*, 2015.
[14] "AWS IoT," https://aws.amazon.com/iot/.
[15] "Cisco Jasper," https://www.jasper.com.
[16] P. Bosshart *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, 2014.