Secure Seamless Bluetooth Low Energy Connection Migration for Unmodified IoT Devices

Syed Rafiul Hussain, *Member, IEEE*, Shagufta Mehnaz, *Member, IEEE*, Shahriar Nirjon, *Member, IEEE*, and Elisa Bertino, *Fellow, IEEE*

Abstract—At present, Bluetooth Low Energy (BLE) is dominantly used in commercially available Internet of Things (IoT) devices–such as smart watches, fitness trackers, and smart appliances. Compared to classic Bluetooth, BLE has been simplified in many ways that include its connection establishment, data exchange, and encryption processes. Unfortunately, this simplification comes at a cost. For example, only a star topology is supported in BLE environments and a peripheral (an IoT device) can communicate with only one gateway (e.g., a smartphone, or a BLE hub) at any given set time. When a peripheral goes out of range and thus loses connectivity to a gateway, it cannot connect and seamlessly communicate with another gateway without user interventions. In other words, BLE connections are not automatically migrated or handed-off to another gateway. In this paper, we propose *SeamBlue*[1], which brings secure seamless connectivity to BLE-capable mobile IoT devices in an environment that consists of a network of gateways. Our framework ensures that unmodified, commercial off-the-shelf BLE devices seamlessly and securely connect to a nearby gateway without any user intervention.

Index Terms—Bluetooth low energy (BLE), IoT, seamless connectivity, secure connection migration

1 Introduction

IoT devices and services have entered the commercial market much faster than expected. IoT industry predicts that the total number of 'smart things' will be more than 30 billion [2] by the year 2020–which will outnumber the total number of smartphones. IoT technology is already being adopted in many places such as factories, airports, offices, homes, hospitals, and schools, and is being used in applications such as asset tracking, health monitoring, predictive maintenance, environmental monitoring, energy metering, and elder care. In a typical scenario, an IoT device connects to a gateway (e.g., a smartphone or a smart hub) over a low-power wireless network, and the gateway enables its access to the Internet. Because the connection process between an IoT device and a gateway requires the active engagement of a user, *seamless connectivity of mobile IoT devices in a network of gateways* is still not happening. Ideally, an IoT device should be able to seamlessly communicate with a nearby gateway, without requiring an end-user to enter pins and passwords every time it moves near a different gateway in the same trusted network environment.

There are a number of wireless protocols, such as Bluetooth LE (BLE) [3], ZigBee [4], and NFC [5], that have been used in different IoT communication scenarios. Among these, BLE is the most popular because of its simplicity, openness, and its several orders of magnitude of energy savings. The BLE protocol allows multiple devices ('peripherals') to connect to a single gateway (the 'central'), but it restricts the mobility of the peripherals outside and within the range of a gateway. Carrying the gateway along with a mobile IoT device seems like an option, but it is not always feasible as it causes disconnections of other IoT devices that are either static or moving in a different direction from the gateway. For instance, if a personal smartphone is used as a gateway for the IoT devices deployed for a home automation system, BLE-enabled IoT devices may get disconnected when the smartphone is taken outside of the home. Similarly, in a hospital scenario, patients wearing BLE enabled IoT devices may move inside and outside of the hospitals for which simple smartphones may not be used as a BLE gateway. Furthermore, IoT devices and gateways deployed in battlefields and agricultural farms can be mobile, and in these use cases continuous connectivity through smartphones is not be a viable solution.

In order to ensure continuous BLE connectivity [3], Zachariah et al. [6] proposed an architecture where an IoT device may connect to multiple gateways located at different places. However, establishing a distinct connection with every gateway requires a device to reset and broadcast advertising signals separately for all the gateways. This behavior is observed in many BLE devices including Moto 360 [7] and Samsung Gear watches [8]. Some of the Android Wear watches are so dependent with their proprietary smartphone applications that these devices do not even

- *S. R. Hussain, S. Mehnaz, and E. Bertino are with the Department of Computer Science, Purdue University, West Lafayette, IN 47907. E-mail: {hussain1, smehnaz, bertino}@purdue.edu.*
- *S. Nirjon is with the Department of Computer Science, University of North Carolina Chapel Hill, Chapel Hill, NC 27599. E-mail: nirjon@cs.unc.edu.*

Manuscript received 16 Mar. 2017; revised 10 June 2017; accepted 26 July 2017. Date of publication 16 Aug. 2017; date of current version 2 Mar. 2018. (Corresponding author: Syed Rafiul Hussain.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMC.2017.2739742

1. A preliminary version [1] of this paper has been published in a conference.

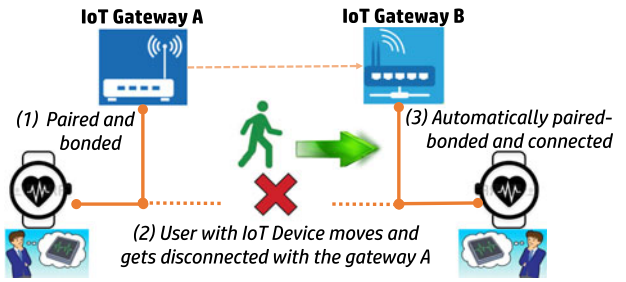

Fig. 1. Seamless BLE connectivity architecture.

allow themselves to pair with non-proprietary smartphones. In fact, the scenario is so restricted in its functionality that even the smartphones model matters. For example, the latest Gear devices only pair with Samsung S4 smartphones and above. In such scenarios, the possibility of a Utopian environment where smart devices seamlessly communicate with each other seems a far reality.

Even if connections with multiple gateways are made possible by changes to the BLE protocol [9], it comes at the cost of disconnecting the device from its previous gateway and then connecting to a new one. This incurs significant CPU, memory, energy, and bandwidth overhead in resource constrained IoT devices as even a single connection establishment requires advertisements, discovery, pairing and bonding [10], and several mutual agreements in different layers of BLE protocol stack. Consequently, connection establishment with multiple devices is neither a time efficient nor a cost effective process. In addition, the process requires repeated manual interventions that disrupt the ongoing communication between a device and a remote service. Because of these practical issues, we argue that an IoT device should be able to seamlessly communicate with different gateways [6] without having to create a separate connection with each of them.

Our vision of a seamless BLE migration is illustrated in Fig. 1, where a user at first connects (pairs) his fitness tracker to gateway A like he does for any BLE device. When he moves to gateway B, connection states are automatically migrated from gateway A to gateway B over a different communication channel, without interrupting ongoing communications between the device and any remote service it is talking to. Finally, when the user enters into the range of B, the fitness tracker is completely handed-off to gateway B, without requiring the user to manually pairing the tracker with it. While this seems similar to hand-offs [11], [12], [13] in cellular or WiFi networks, a major distinction between BLE migration and a cellular/WiFi connection migration is that in the case of BLE, we are constrained by the billions of already deployed IoT devices and many other legacy devices that are running Bluetooth 4.0, for which we cannot change their BLE implementation. This practical constraint makes it difficult even to detect the presence of a device at the time it is in the connected state with a central. In addition to this, migrating a connection requires transferring a set of state variables between gateways that define the state of a BLE connection. Finding the set of state variables by browsing a large code base is itself a time-consuming and error-prone process. Finally, selecting the next gateway among the available ones and then securely transferring the connection states pose further challenges to seamless BLE migration.

In this paper, we propose *SeamBlue*, which addresses these challenges and enables seamless BLE connection migration for mobile IoT devices in a network of static or mobile BLE gateways. Several salient features combined together make *SeamBlue* unique. First, we develop a systematic approach based on static program analysis technique that automatically finds a set of variables and objects in the BLE code base which define the connection state, i.e., the internal protocol state. Second, we propose two modes of connection state extraction: *partial stack cloning* and *full stack cloning* in order to support connection migration for a wide range of IoT devices. Third, we leverage existing approaches to user movement prediction [14], and propose a mechanism to select the best candidate gateway for a connection migration. Fourth, while transferring connection from one gateway to another we consider both trusted and untrusted gateways and ensure secure connection migration. Fifth, we have developed a testbed that consists of unmodified, BLE-capable IoT devices (e.g., Android smartphones, and a tablet, and a Moto 360 watch) and BLE gateways (e.g., customized smartphones acting as central). We perform an in-depth evaluation of *SeamBlue* in this testbed to quantify its effectiveness as well as its overhead. In summary, the contributions of this paper are the following:

- We propose a framework that ensures secure seamless communication between an unmodified, BLE-enabled mobile IoT device and a remote service in a network of static or mobile BLE gateway environments, without requiring pairing-bonding and connections to individual gateways.
- We develop a systematic approach based on static program analysis to identify the state variables in the BLE code base that are required for transferring pairing-bonding and connection information from one gateway to another gateway.
- We propose two approaches–*partial stack cloning* and *full stack cloning*—for capturing a snapshot of connection states at the current gateway and then updating them at the next gateway during BLE connection migration.
- We propose a gateway selection mechanism for transferring the connection state to the most suitable gateway when an IoT device requires to migrate its connection and there are multiple gateways in its range.
- We design a secure sharing of connection information between two gateways so that adversaries cannot obtain the pairing-bonding keys and connection parameters while the connection is migrating.
- We design a storage/memory management component for our system so that adversaries cannot run the BLE gateways into memory exhaustion problems.
- We extensively analyze the security and privacy issues of our proposed system.

Organization. The rest of the paper is organized as follows: Section 2 presents some use cases of the *SeamBlue* system. Section 3 provides a brief primer on BLE. Section 4 discusses the challenges of building a system supporting seamless connection migration for BLE. Section 5 provides an overview of the workflow of *SeamBlue* system and discusses the adversary model. Section 6 describes the details of *SeamBlue*. Section 7 presents experiment results. Section 8 analyzes the security and privacy of our proposed system. Section 9 discusses the state-of-the-art and finally Section 10 concludes the paper by outlining future work.

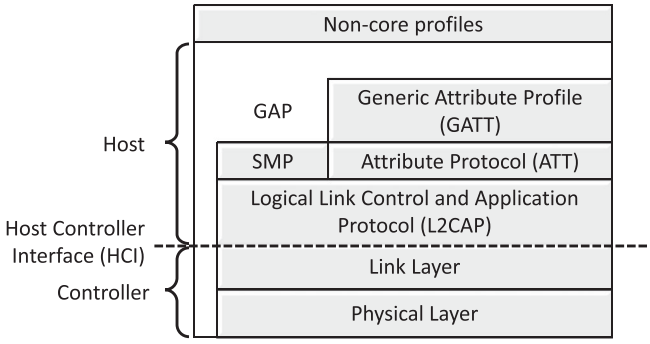

Fig. 2. The Bluetooth LE protocol stack.

2 Usage Scenarios

We briefly describe some of the use cases of seamless BLE connection migration.

1) *In Hospitals:* Patients wearing BLE devices in hospitals can be localized and tracked, and their heart rate and other physiological signals can be monitored using a network of gateways deployed at different locations in the hospital. Even if the patient moves, these devices will provide continuous monitoring and uninterrupted services by connecting to nearby gateways.
2) *In Airports:* Many airports [15] use BLE enabled tracking devices to monitor the location and movement of passengers and airport equipment. Upon arrival at the airport, passengers (and baggage) which are equipped with BLE beacons can voluntarily report their location and status to the deployed gateways from anywhere within the airport and, in return, receive personalized services and notifications.
3) *In Theme Parks:* With the help of a BLE-enabled wristband worn by children and static gateways deployed at different locations inside a theme park, parents can monitor and locate their children via their mobile phone. The IoT wristband concept has already been implemented and successfully tested at Disney World in Orlando [16], Florida. Disneys MagicBand is a customizable wristband that functions as a passport for just about everything in the park. These bands can serve as digital entrance tickets for guests or even store credit card information to facilitate transactions.

3 Background

3.1 Roles of BLE Device

A BLE device assumes either a *peripheral* or a *central* role. A peripheral, typically an IoT device, such as a heart rate monitor, a blood pressure monitor, a smart lock, or a smart watch, has limited capabilities and contains advertisement information. A central device, such as an access point, a personal computer, or a smartphone, scans for BLE advertisements, receives an advertisement, and initiates a connection.

3.2 BLE Protocol Stack

Similar to classic Bluetooth [17], the BLE protocol stack [18] is composed of two main parts: a *controller* and a *host* as shown in Fig. 2.

Physical Layer: BLE operates in the 2.4 GHz Industrial Scientific Medical (ISM) band and defines 40 Radio Frequency (RF) channels with 2 MHz channel spacing. There are two types of BLE RF channels: (1) three advertising channels used for device discovery, connection establishment and broadcast transmission, and (2) thirty-seven data channels used for bidirectional communication between connected devices. In order to avoid interference, an adaptive frequency hopping pattern consisting of 37 frequencies is used for data channels.

Link Layer: BLE defines two device roles at the Link Layer for a connection: the *master* and the *slave*. Once a connection between a master and a slave is created, the physical channel is divided into non-overlapping time units called connection events. In order to detect bit error, all data units include a 24-bit Cyclic Redundancy Check (CRCInit) code. For a new connection event, master and slave use a new data channel frequency, which is computed using the frequency hopping algorithm. `Access Address (AA)`, embedded in a Link Layer packet, is used to identify communications on a physical link, and to exclude or ignore packets on different physical links that are using the same physical channels in physical proximity.

L2CAP: It works as a logical link layer and multiplexes the data of higher layers on top of a Link Layer connection.

ATT: The ATT defines the communication between two devices playing the roles of server and client, respectively. The server maintains a set of attributes. An attribute is a data structure that stores the information managed by the GATT.

GATT: A framework defined by GATT uses the ATT for the discovery of services that includes characteristics. A characteristic is a set of data which includes a value and a set of properties. The data related to services and characteristics are stored in attributes. For example, a server that runs a heart rate monitoring service may account with a heart rate characteristic that uses an attribute for describing the sensor, another attribute for storing heart rate measurement values and a further attribute for specifying the measurement units.

GAP and Application Profiles: GAP specifies device roles, modes, and procedures for the discovery of devices and services, the management of connection establishment and security. A device may support various roles, but only one role can be adopted at a given time. Application profiles specify general behaviors that Bluetooth-enabled devices use to communicate with other Bluetooth devices. For example, a heart-rate monitor is able to send its sensor values to a gateway only if the corresponding gateway implements a heart-rate profile.

3.3 Modes of Communication

Two modes of communication are available: *broadcast* and *connected* modes. The broadcast mode enables a peripheral to send data to any other device listening for transmissions. If two devices need to exchange data they can use the *connected* mode. In this mode, a peripheral device broadcasts its presence by sending advertisement packets. The central can initiate a connection following a received broadcast. Once a connection is established the devices can exchange data.

3.4 BLE Security and Privacy

Pairing: In *connected* mode, if two devices (one acting as peripheral and another as central) want to exchange data securely, they perform a *pairing* process where as a first step, the parties involved in the communication exchange their identity information to set up the trust and then establish the encryption keys for future data exchange. The Security Manager Protocol (SMP) used for the pairing procedure

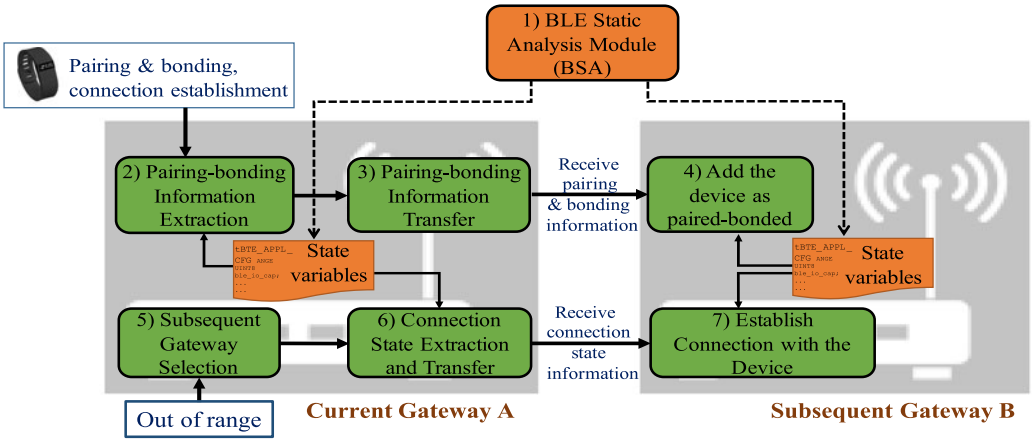

Fig. 3. Basic workflow of *SeamBlue*.

results in the following keys that are shared between the peripheral and the central.

- *Identity Resolving key (IRK)*: 128-bit key used to generate and resolve a random address.
- *Connection Signature Resolving Key (CSRK)*: 128-bit key used to sign data and verify signatures on the receiving device.
- *Long Term Key (LTK)*: 128-bit key used to generate the session key for an encrypted connection.
- *Encrypted Diversifier (EDIV)*: 16-bit stored value used to identify the LTK. A new EDIV is generated each time a new LTK is distributed.
- *Random Number (RAND)*: 64-bit stored value used to identify the LTK. A new RAND is generated each time a new LTK is distributed. Note that both the EDIV and RAND are used to identify a bonded device's LTK in the look-up table that stores the LTKs for all the bonded devices.

Bonding: Bonding is the process of storing the keys created during pairing for use in subsequent connections in order to form a trusted device pair.

Privacy: BLE can use *Random Device Addressing* to help increase the privacy [19], [20] of connections and prevent 'tracking' based on the assumption that eavesdropping did not occur during the pairing process.

If the advertising device has been previously discovered and has returned to an advertising state, the device must be identifiable by trusted devices in future connections without going through the discovery procedure again. The *IRK* stored in the trusted device is used to identify the advertiser as a trusted device.

4 Challenges to BLE Connection Migration

Although BLE connection migration for IoT devices seems similar to that of traditional networks, e.g., WiFi or TCP connection migration, it poses some unique challenges specific to IoT.

1) *Unmodified IoT Devices*: Since billions of IoT devices are already in use, it is not feasible to demand changes to their BLE implementation. Hence, we are constrained to only change the BLE implementation of the gateways for seamless connection migration.
2) *Identifying State Variables*: We need to identify the set of variables that uniquely define the state of a BLE connection. Manually browsing a large code base (up to 100K lines of code) to find state variables is an impractical, error-prone, and time-consuming process. Hence, automatically finding state variables for connection is essential.
3) *Gateway Selection for Connection Transfer:* While transferring the pairing-bonding information, not all the gateways in a network should be a receiver of this information. For example, syncing all the gateways for every pairing between a gateway and an IoT device would require a substantial amount of time and bandwidth and incur significant communication overhead. Hence, it is necessary to select a subset of gateways for sharing the pairing-bonding information.
4) *Secure and Fast Connection Transfer:* IoT gateways need to distribute the pairing-bonding and connection state information to the candidate gateways (or to a central authority) in a secure manner so that an adversary cannot obtain this information and impersonate a legitimate gateway. The connection transfer should be fast enough so that the services are not disrupted. In addition, if a gateway is not part of the trusted cluster of IoT devices, the current gateway may need to establish a trust relationship with that untrusted gateway[2] before seamless connection migration begins.

5 *SeamBlue* Overview

This section provides an overview of *SeamBlue* by briefly describing the sequence of steps for connection migration, which comprises of identifying bonding/connection related state variables, establishing a connection with a gateway, transferring the bonding/connection information, and establishing a connection with a subsequent gateway.

5.1 Basic Workflow

SeamBlue ensures that an IoT device is always connected to a gateway, as long as it is within the range of any gateway. Fig. 3 depicts the basic workflow of the connection migration process which we describe next.

2. We use the terms 'untrusted gateways' as 'the gateways that do not belong to a group of already trusted gateways' interchangeably. Trusted gateways are of two types: (1) gateways sharing the same group key, (2) previously untrusted gateways whose public key certificates have been validated

1) In an off-line, one-time step, we analyze the BLE source code statically to identify the set of variables required for both pairing-bonding and connection information transfers. The *BLE Static Analysis (BSA)* module first performs this static analysis on the BLE source code; and then the identified set of variables (that are required for connection transfers) are stored in each of the gateways. This step is described in Section 6.1.
2) An IoT device advertises its information and a nearby gateway establishes a connection by pairing-bonding. We name the gateway to which the IoT device is currently connected the *current* gateway. The current gateway extracts a set of pairing-bonding information from the connection in order to share it with other nearby gateways. Section 6.2 describes this information extraction step.
3) The current gateway disseminates the pairing-bonding information to a set of gateways that are candidates for the subsequent gateway (the next gateway to which the IoT device may connect). This pairing-bonding information consists of both the bonded device's information as well as a subset of state variables. Section 6.3 describes this step in details.
4) Upon reception of this information, the set of candidate gateways add the IoT device as a bonded device, and initialize the subset of state variables for bonding, but do not initiate a connection.
5) Perceiving the fact that the IoT device is moving out of its range, according to the movement pattern of the IoT device the current gateway selects one of the candidate gateways (found from step 3) as the subsequent gateway to which the connection is going to migrate. This step is described in Section 6.4.
6) The current gateway identifies the current state (or snapshot) of the connection and transfers the required state variables to the subsequent gateway so that the subsequent gateway can reconstruct the connection state with the same peripheral. Section 6.2 describes the extraction of these variables, and Section 6.5 describes how these transfers are done securely.
7) Upon reception of the connection state information, the subsequent gateway creates required objects related to connection, updates the connection state variables, and stores the connection information into gateway's non-volatile memory (NVRAM). As a result, the peripheral gets seamlessly connected to the subsequent gateway, without interrupting ongoing services.

5.2 BLE Stack Cloning

Depending on the accessibility of the state variables at the controller part (e.g., link layer) of BLE stack (shown in Fig. 2), *SeamBlue* provides two modes for connection state extraction: *full stack cloning*, and *partial stack cloning*. Full stack cloning refers to cloning states of all the layers of Bluetooth stack starting from the application layer down to the link layer whereas partial stack cloning refers to the cloning of Bluetooth stack starting from the application layer down to the L2CAP layer. Details of these modes are presented in Section 6.7.

5.3 Information Dissemination

SeamBlue supports two different strategies for the dissemination of both pairing-bonding and connection state information. The the initial gateway either (1) pushes this information to the cloud from where the candidate gateway (s) can sync this information periodically, or - (2) transfers this information directly to the selected subsequent gateway(s). Independently of the dissemination strategy, the current gateway transfers pairing-bonding and connection state information in a secure manner.

5.4 Adversary Model

We consider a strong adversary model where the adversary has the capability of injecting unauthenticated packets, or modifying legitimate packets, or sniffing end-to-end messages. We also assume that the adversary may compromise the IoT gateways and IoT devices at any point of the operation of *SeamBlue* system. We assume that *SeamBlue* can detect such compromised devices and gateways by leveraging existing anomaly detection mechanisms [21], [22] in the context of IoT. Finally, in this paper, we do not consider the denial-of-service attacks through jamming the BLE channels.

6 *SeamBlue* Design Details

This section describes how *SeamBlue* addresses the challenges in BLE connection migration by adding new functionalities to the BLE stack of only the IoT gateways.

6.1 BLE Static Analysis Module

Like other network protocols (e.g., TCP), the BLE implementation follows an event-driven programming paradigm that is centered on executing appropriate protocol logic (known as event handlers) in response to the occurred network events. Such implementation usually contains an initialization part, packet receiving or event loop, and a packet processing part. The initialization code bootstraps the BLE module of the device by loading the configuration files. The packet receiving loop receives a packet from the kernel space, identifies the type of network events (e.g., PAIRING_COMPLETE), and forwards the packet to the corresponding event handler, i.e., packet processing part.

State variables: While processing a packet, some of the variables may get updated depending on the content of the packet and the current state of the protocol. Therefore, we define the variables of a program as state variables whose values are different in different connection states, get updated as the state changes during processing of an incoming packet, and whose scope lasts throughout the lifetime of a connection.

Why do we require systematic analysis?: Fig. 4 shows the simple skeleton of an abstract BLE implementation. However, through our manual analysis we found that the open source BLE implementations [23], [24] are fairly complex. We also observed that the internal states of the BLE protocol consist of many different program variables and objects that are spread over different layers of the BLE stack. This manual analysis required more than 80 man-hours. However, the resulting set of connection state variables obtained by the manual analysis was not complete since a number of variables were left unrecognized. As a result, connection migration was not successful. Therefore, finding internal states (i.e., variables and objects that define the connection state) through manual analysis is not a viable solution. Furthermore, variants of a BLE implementation adopted for different IoT devices may have different sets of variables and objects for defining protocols' internal states. In order to address

```
1  void pair_device (packet_t* rcv_pkt){
2      /* Sanity checks on the received packet */
3      ...
4      /* Packet is accepted after all sanity checks
       . Now process the packet */
5      ...
6      btd_cb.state -> paired = true; // btd_cb
       refers to bluetooth device control block
7      ...
8  }
9  void bond_device (packet_t* rcv_pkt){
10     /* Sanity checks on the received packet */
11     ...
12     /* Packet is accepted after all sanity checks
       . Now process the packet */
13     ...
14     btd_cb.state ->  bonded = true;
15     ...
16 }
17 void connect_device (packet_t* rcv_pkt){
18     /* sanity checks on the received packet */
19     ...
20     /* Packet is accepted after all sanity checks
       . Now process the packet */
21     ...
22     btd_cb.state ->  connected = true;
23     ...
24 }
25 bool disconnect_device (packet_t* rcv_pkt){
26     /* Sanity checks on the received packet */
27     ...
28     /* Packet is accepted after all sanity checks
       . Now process the packet */
29     ...
30     btd_cb.state ->  connected = false;
31     ...
32 }
33 void handle_receive_packet(packet_t* rcv_pkt){
34     pthread_t thread;
35   void (*event_handler)(packet_t* pkt);
36     /* Checks on the received packet */
37     ...
38     /* Received packet passes the sanity checks.
       Now dispatch the packet to appropriate
       handler */
39   if(rcv_pkt->type == EVENT_PAIR)
40     event_handler = pair_device;
41   else if(rcv_pkt->type == EVENT_BOND)
42     event_handler = bond_device;
43   else if(rcv_pkt->type == EVENT_CONNECT)
44     event_handler = connect_device;
45   else if(rcv_pkt->type == EVENT_DISCONNECT)
46     event_handler = disconnect_device;
47   /* Handle the packet in a new thread */
48     pthread_create(&thread, event_handler);
49 }
50 void run_main_event_loop(){
51   while(1){
52     wait_if_no_packet_to_process(); // i.e., the
       queue is empty
53     packet_t *rcv_pkt = dequeue();
54         handle_receive_packet(rcv_pkt);
55     }
56 }
57 int main(int argc, char *argv[]){
58   /* Initialize the Bluetooth adapter */
59   adapter_initialize();
60     ...
61   run_main_event_loop();
62   ...
63
64     adapter_cleanup(); // before disabling/
       removing the Bluetooth adapater
65   ...
66   return 0;
67 }
```

Fig. 4. Skeleton of a simplified BLE stack implementation.

these challenges, we develop a systematic approach to find the minimal state information for BLE connection migration.

Why static analysis?: Dynamic taint analysis [25], [26] may be used to find the state variables while processing the input. However, processing input may not cover all the execution paths where state variables may get updated. Therefore, dynamic taint analysis is not the ideal candidate for systematically finding the state variables. Symbolic execution [27], [28] addresses this challenge by representing the packet content and the runtime state as the symbolic input for exploring all possible execution paths along which state variables may get updated. However, symbolic execution quickly falls into state explosion problem because of loops (if symbolic input is used in loop termination conditions) and pointers. To address the state explosion problem, we adopt static analysis using control flow and data flow analysis that precisely captures the state variables. Static analysis [29], [30] have also been used to find states for virtual machine (VM) migration in network function virtualization (NFV) [31]. However, our proposed mechanism differs from those techniques due to the resource constrained nature of IoT and also results in a more precise set of state variables.

Pre-processing: To support static analysis, we perform the following pre-processing of the BLE source code: (1) we first convert the BLE source code into intermediate representation (IR) using the LLVM compiler [32]; (2) we then build the control flow graph (CFG) [33] from the IR [34]. Each node in the control flow graph represents a basic block which is a straight-line piece of code with no branching except at the end of sequence, i.e., without any jumps or jump targets. Jump targets start a block, and jumps end a block. Edges represent possible flow of control from the end of one block to the beginning of the other. There may be multiple incoming/outgoing edges for each block.

Finding the slice of code that handles state variables: BLE implementations contain code for handling nitty gritty details of all the aspects of BLE protocol. Since our focus is only on the packet processing logic in which the variables that are the members of the global variable list may get updated during the packet processing, we need to filter out the other portions of the code that handles other aspect of the BLE protocol and also update the global variables. We solve this problem using a forward control-flow slicing technique [35] and thus find the required portion of the source code that processes the incoming packets. To find that slice of code, we use the control flow graphs (CFGs) of all the functions of the program. By analyzing all the direct and indirect calls found in those CFGs, we find the flow of the protocol execution starting from receiving of the packet to the functions that implements the packet processing logic.

Call graph construction: To find all possible flows of the execution, we find all possible chains/sequences of functions that process the packets. Hence, we need to construct the call graph [36] of the program that represents the caller-callee relationships in which a node represents a function/procedure and an edge denoted with (x, y) represents the calling of function y from function x. We build such a call graph for the BLE implementation starting from the entry point of the BLE stack. Since the event driven BLE implementations rely on indirect calls, the resulting call graph should include both the direct calls and the indirect calls. The direct function calls can be easily identified from the CFG. However, identifying indirect calls poses additional challenges because the values of the function pointers used for indirect calls are decided at runtime depending on the program input. For example, in the listing in Fig. 4, the function pointer `event_handler` at runtime may take one of the four possible target values, i.e., `pair_device`, `bond_device`, `connect_device`, or `disconnect_device`. Depending on the runtime value of the `event_handler`, the corresponding event handler will be executed.

Points to (alias) analysis: To address the challenge of indirect calls, we use a context, flow, and field sensitive points-to analysis [37] that finds the all possible targets of the indirect calls in the program. For the listing in Fig. 4, the points-

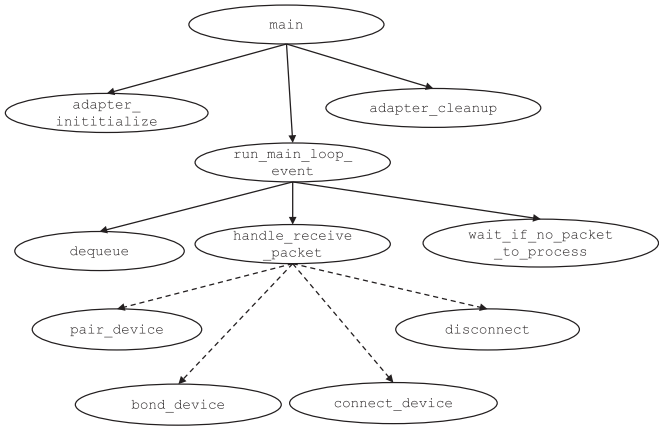

Fig. 5. Call graph for the program in Fig. 4.

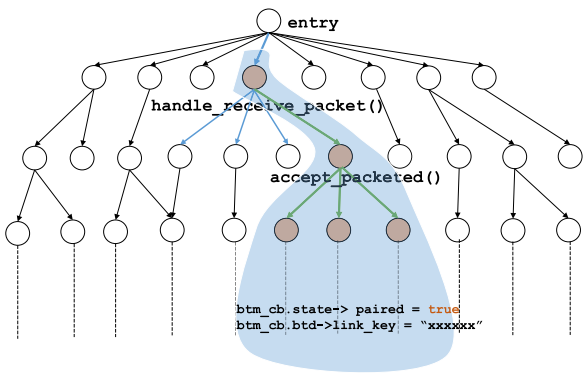

Fig. 6. Slice of the BLE source code that handles packet processing logic.

to analysis identifies the possible targets of `event_handler` function pointer. Apart from function pointers, points-to analysis is also used for resolving the pointer alias problem. Pointer alias is frequently used in BLE implementations when updating the state variables.

Using the result of points-to analysis, we complete the construction of the call graph by incorporating the indirect edges. Fig. 5 shows the call graph of the listing in Fig. 4 generated using our technique. The direct calls are represented with direct arrow lines whereas the indirect calls are represented with the indirect arrow lines. Since the `event_handler` function pointer may point to four possible target functions (i.e., `pair_device`, `bond_device`, `connect_device`, `disconnect_device`) based on the type of an event, we add these targets in the call graph and add indirect arrows from the `handle_receive_packet` function to those target functions. Note that, we use a bottom-up approach for constructing the call graph that represents the interprocedural control flow of the program.

Extracting the slice: Once we have the call graph of the program, we extract the slice of the call graph sub-rooted at the `handle_receive_packet` function. This slice includes all the successor functions that implement the logics of accepting an incoming packet and then processing the packet based on the packet content and the protocol state. Fig. 6 shows an example of an abstract slice extracted from the BLE source code.

Identifying connection state variables: Once a packet is accepted after running the checks on the packet contents, the gateway starts processing the packet and takes actions (i.e., generates events) depending on the type and content of the packet. The gateway initializes and updates some of the program variables and objects specific to that connection. We need to identify those program variables whose liveness lasts throughout a BLE connection. In order to find those variables and objects, we traverse each statement of the functions found in the sliced call graph and identify the variables that are defined (i.e., assigned to some values) within that functions. This includes both local and global variables which are considered as possible connection state variables. Generally the scope of a local variable is only within the function unless it is an alias of a global variable and, therefore, it should not be considered as a connection state variable. However, if a local variable is an alias of a global variable, we consider that local variable as one of the state variables. We compare the points-to set (the list of target values of a pointer obtained through the points to analysis) of the local variables defined in a function with the points-to set of the global variables. If the points-to sets are identical, we infer that the local variable is an alias of the corresponding global variable.

Identifying the connection state variables is done offline and it is a one-time cost operation. *SeamBlue* does not need to compute the state variables every time it needs to migrate the connection. We present an excerpt of the resulting set of connection state variables for the BLE implementation in Table 1.

TABLE 1
An Excerpt of the Set of Connection State Variables of the BLE Stack

Items	Layers where used
Device Type	All layers
Device Address Type	All layers
Bluetooth device pseudo address	All layers
Long-Term Key (LTK)	SMP
Identity resolving key (IRK)	SMP
Connection Signature Resolving Key (CSRK)	SMP
EDiv	SMP
RAND	SMP
Access Address	Link Layer
Hop Interval	L2CAP
Hop Increment	L2CAP
CRCInit	Link Layer
Slave Latency	L2CAP, Link Layer
WinOffset	Link Layer
Channel Map	Link Layer
UUID	GAP, GATT
Characteristics Info	GAP, GATT

6.2 Extracting Pairing-Bonding/Connection State Variables

We instrument the BLE implementation so that we obtain the runtime values of the pairing-bonding/connection state variables for a connected IoT device (as shown in steps (2) and (6) in Fig. 3). The current gateway stores the extracted information into memory and sends it to subsequent gateways. The runtime for this extraction module is distributed across different layers of BLE protocol stack. For the runtime implementation, we add *SeamBlue* APIs so that the different layers can interact among themselves. Note that this instrumentation is performed only at the BLE gateways. Thus our proposed system does not modify the BLE implementation of IoT devices which allows the already deployed billions of IoT devices to integrate to the *SeamBlue* system without further modification.

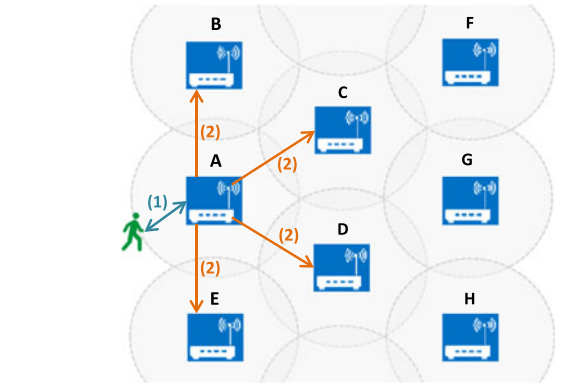

Fig. 7. (1) Pairing and bonding with gateway A, (2) gateway A shares pairing-bonding information with the gateways B, C, D, and E.

6.3 Sharing Pairing-Bonding Information with Candidate Gateways

BLE central serves as a gateway and scans for peripheral devices so that it can connect with them and receive the desired GATT services. In order to ensure secure data transfer to the server through gateways, the current gateway initiates pairing and bonding procedures as shown with (1) in Fig. 7. After creating a connection through pairing and bonding, the current gateway extracts the pairing-bonding related information for transferring to the possible subsequent gateways. In Fig. 7, the gateway A finds B, C, D, and E as the candidate gateways because of their proximity to A and sends the pairing-bonding information to them.

Upon receiving the pairing-bonding information, the candidate gateways B, C, D, and E store this information mapped with the Bluetooth device address of that IoT device so that whenever that device needs service from these gateways, they do not have to execute the pairing-bonding procedures. Note that the candidate gateways do not initiate connection at this stage since they do not have connection state information.

6.4 Selecting Subsequent Gateway for Connection Transfer

If an IoT device moves during or after connection establishment, the current gateway or the IoT service providing cloud system is able to estimate the device's moving direction [14]. *SeamBlue* uses this mechanism and examines a device's locations at recent timestamps to infer the moving direction. Location information of IoT devices can also be obtained using existing indoor and outdoor localization techniques [38]. By analyzing the movement direction and speed of the IoT devices, the current gateway or the service provider selects the subsequent gateway among the candidate gateways to whom the connection information will be transferred. As shown in Fig. 8a, the current gateway A transfers connection information to the subsequent gateway D.

Ping-Pong Effect: An IoT device may move back and forth in a region shared by multiple gateways. As shown in Fig. 8b, the gateways A and D share a common region which is partitioned using a line. The bronze markers denote the area where the signal strength (RSSI) of gateway A is greater than that of gateway D. Conversely, the blue markers denote the opposite case. According to *SeamBlue*, the current gateway A initiates a connection transfer as soon as the IoT device moves out of the A dominant area. However, if an IoT device moves back and forth

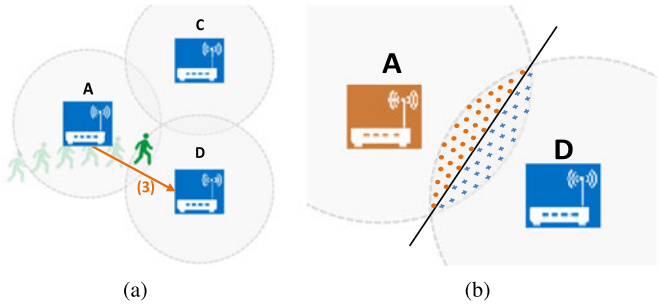

Fig. 8. (a) Subsequent gateway selection and connection transfer as the user with IoT device moves from gateway A to gateway D. (b) Gateways selection and ping-pong effect.

in a shared region, the BLE connection might also switch between the corresponding gateways. To reduce this effect, *SeamBlue* uses a motion prediction mechanism [14] that leverages statistical data (i.e., previous movement patterns) of the IoT device's movements. If the device predominantly moves back and forth, *SeamBlue* gateways do not transfer the connection as soon as it goes beyond the half of the shared region. *SeamBlue* uses a delay tolerant approach to see if the device moves in the direction of the current gateway. If not, the current gateway transfers connection to the subsequent gateway.

6.5 Secure Sharing of Pairing-Bonding/Connection Information

The gateway with which the peripheral is currently connected needs to distribute the bonding and connection information to candidate gateways and the subsequent gateway, respectively. Such sharing can be obtained either by pushing the bonding and connection information to the cloud from where all other gateways can fetch this information or by directly disseminating the bonding and connection information to the appropriate set of gateways. With respect to sharing of this information, the receiver gateways can be categorized into two groups: *trusted* and *untrusted* gateways.

6.5.1 Trusted Gateway

If the receiver gateways belong to the same cluster of gateways as the current gateway, they already share a secret group key. Using this group key they first authenticate themselves and then derive a new session keys (CK and IK) as shown in Fig. 9. The current gateway then shares the data (i.e., pairing-bonding and connection information (PCI)) with the subsequent gateways in encrypted form using the derived session keys. The secret group key can be shared between gateways through WiFi or 4G LTE communication network and thus no change is required in the existing Bluetooth protocol.

The current gateway appends a $nonce$ to the PCI so that the resultant ciphertext of the pairing-bonding and connection information is different for the same plaintext data and thus prevents against the replay attacks. The subsequent gateway obtains the PCI by decrypting the ciphertext and then verifies the MAC of the received message. The current gateway invalidates the session keys and $nonce$ when they are used once for sharing PCI with the subsequent gateway securely. The current gateway establishes new session keys and $nonce$ if it requires to transfer the PCI again.

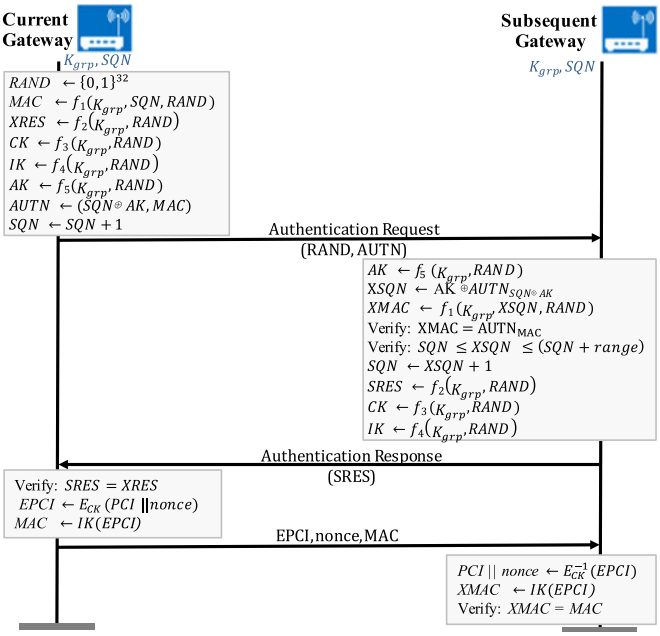

Fig. 9. Authentication, and securely sharing pairing-bonding and connection information with trusted subsequent gateway.

6.5.2 Untrusted Gateway

If the subsequent gateway to which the BLE connection is going to be migrated is not already trusted, e.g., in the case of wide area networks (WANs) when a user moves from one organization (e.g., hospital) to another, the current gateway needs to verify the public key certificate [39] of the subsequent gateway. Upon certificate validation, the current gateway may establish a shared secret key with the receiver gateway using the Diffie-Hellman key exchange protocol [40] or a public key cryptography protocol [41] to share the bonding and connection information securely. Thus *SeamBlue* allows the BLE connection to migrate to a gateway that may be in a different LAN or WAN where the gateways are initially untrusted. Since the Diffie-Hellman protocol requires multiple message exchanges for establishing a shared secret key between two gateways, it might not be energy and time efficient for such scenarios. Therefore, *SeamBlue* adopts public key cryptography which requires less amount of time for mutual authentication. An example of data transfer with public key cryptography is following:

$$\psi \leftarrow E_{PK_{rcv}}(PCI||nonce) \tag{1}$$

$$PCI||nonce \leftarrow D_{SK_{rcv}}(\psi) \tag{2}$$

where PCI is the data to transfer, and PK_{rcv} is the public key of the receiver gateway. In Eqn. (1), the current gateway encrypts the data using receiver's public key, PK_{rcv} and transfers ψ to the receiver gateway. The receiver gateway has knowledge of the corresponding secret key, SK_{rcv}, and thus decrypts ψ to $PCI||nonce$ as shown in Eqn. (2). We use $nonce$ to create different ciphertexts for the same plaintext and also to prevent the replay attacks.

Though public key cryptography may require less amount of time for transferring pairing-bonding/connection information, it does not ensure the perfect forward secrecy. In other words, if the adversary learns the private key of a compromised BLE gateway, it can decrypt the traffic of past and future sessions. Therefore, in order to protect the encrypted communications and sessions recorded in the past, *SeamBlue* may configure the system to adopt the Diffie-Hellman key exchange protocol for creating a shared secret key. This session key is then used by the current gateway to encrypt the pairing-bonding/connection information for transfer securely to the subsequent gateway. More efficient approaches, e.g., the pairing-free certificateless hybrid signcryption (pCL-HSC) [42] scheme, can also be used in the case of untrusted gateways which combines pCLSC-TKEM with a data encryption mechanism (DEM).

6.6 Protecting Previous and Future BLE Packets from Untrusted Gateways

If an untrusted gateway obtains the same pairing-bonding/connection information (e.g., LTK, CSRK, etc.) used by the trusted gateway, the untrusted gateway may decrypt the previously exchanged BLE packets between a peripheral device and the trusted gateways. Besides, if the same pairing/bonding and connection information is still used after the device moves from an untrusted gateway to a trusted gateway, the untrusted gateway may sniff the BLE packets and decrypt them using the keys obtained earlier. Therefore, in order to protect the previous and future BLE packets from untrusted gateways, the trusted gateway changes the keys (i.e., LTK, IRK, CSRK) and parameters (resulted from the initial pairing-bonding procedure and connection setup) when a connection is transferred from a trusted gateway to an untrusted gateway. Similarly, the trusted gateway changes the keys and connection parameters when a connection is migrated from an untrusted gateway to a trusted gateway.

6.7 Cloning Connection Information to Subsequent Gateways

Upon reception of the connection information, the subsequent gateway does not initiate the pairing-bonding procedure, since the IoT device is already added as a bonded device into the subsequent gateway's NVRAM. The *SeamBlue* module running on the gateway updates the connection related parameters for communicating with the IoT device.

Since the subsequent gateway does not scan, discover, or create a new pairing and bonding with the IoT device, the BLE peripheral does not add the subsequent gateway as a bonded central device. Therefore, the IoT device does not need to replace the current gateway's device address with that of the next gateway in its memory. As a result, to send/receive packets to/from the peripheral, the subsequent gateway impersonates the current gateway using the device address. We instrument the BLE implementation on the gateway only so that the communication between the subsequent gateway and the IoT device is executed with the current gateway's device address. Along with the device address, the subsequent gateway uses other connection related information which are already shared with the IoT device. The connection related information is spread across both the host and controller parts of the BLE stack (shown in Fig. 2). Since some gateway devices, e.g., Android smartphones, use proprietary Bluetooth device drivers, they do not allow one to change any variables located at the Link Layer in the controller part. Due to this limitation, we propose two approaches for cloning connection information to the subsequent gateway: *partial* and *full stack cloning*.

Partial Stack Cloning: In *partial stack cloning*, the subsequent gateway impersonates the current gateway using the shared

connection information that are spread across the application layer to L2CAP layer of the BLE stack. The subsequent gateway adds the peripheral as a bonded device and impersonates the current gateway's device address. However, due to the proprietary nature of some Bluetooth device drivers, the subsequent gateway cannot change the values of `Access Address`, `connection interval`, `slave latency`, `channel map`, and `CRCInit` in the link layer of the BLE stack for completely impersonating the current gateway. We address this challenge by using an additional connection request from the subsequent gateway to the IoT device. In this procedure, the current gateway disconnects the connection with the IoT device and provides a control signal to the next gateway for sending connection request(s) to the peripheral. Since the subsequent gateway is already stored as the bonded device in IoT device's NVRAM, according to the BLE protocol, upon reception of the connection request the IoT device just updates the connection related parameters (i.e., `Access Address`, `connection interval`, `slave latency`, `channel map`, and `CRCInit`) for that bonded device. Hence, *partial stack cloning* requires an extra connection request for connection migration to the next gateway without modifying the IoT devices.

Full Stack Cloning: The *full stack cloning* approach allows the subsequent gateway to impersonate the current gateway using the shared connection information that spread from the application layer to the link layer of the BLE stack (i.e., it requires the controller part of the BLE stack to be open source). Therefore, the current gateway's `Access Address`, `connection interval`, `slave latency`, `channel map`, `CRCInit` values of the link layer are used by the subsequent gateway to impersonate the current gateway. Note that channel maps can be updated to handle collisions. Since the current and the next gateways use the same `AA` for sending and receiving packets, the full stack cloning does not require any new connection request message from the next gateway to the peripheral device.

6.8 Managing Storage Requirements

In order to make sure that memory exhaustion does not occur as a result of storing pairing-bonding information from neighboring gateways, we design a technique that enables gateways to remove the unnecessary device data from the bonded devices' list. This technique comprises of the following two principles that define the conditions for a BLE gateway to receive information regarding adding/removing devices to/from its bonded devices' list, respectively:

- *Adding a device to the bonded devices' list:* A BLE gateway shares the pairing-bonding information of a BLE device with a neighboring gateway only if the neighboring gateway is not also a neighbor of the antecedent gateway with which the BLE device was previously connected. In the case that a neighbor is common to both the current and antecedent gateways, that particular neighboring gateway already has the pairing-bonding information for the BLE device stored in list (since the information has already been shared with this gateway by the antecedent gateway). Therefore, given the set of neighbors of current and antecedent gateways, i.e., N_{Cur} and N_{Ant}, respectively, the pairing-bonding information is shared with a set of gateways $N_{add} \in N_{Cur}$ where-

$$N_{add} = N_{Cur} - N_{Ant} - \{Ant\} \tag{3}$$

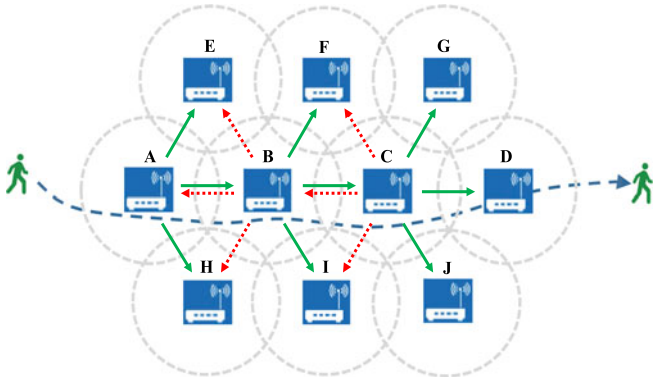

Fig. 10. Memory optimization for pairing-bonding information.

- *Removing a device from the bonded devices' list:* Once a BLE device moves to the subsequent gateway ('current' at this moment), the antecedent gateway notifies a subset of its neighbors regarding this event. This subset comprises those gateways that are neighbor of this antecedent gateway but are not neighbors of the current gateway. In the case that a neighbor is common to both the antecedent and current gateways, the particular neighboring gateway may establish a connection with that BLE device in future for which it should keep the pairing-bonding information (i.e., no action necessary). However, in the case that a neighbor is not also a neighbor of the device's current gateway, the antecedent gateway requests that particular neighbor to safely remove the pairing-bonding information from its bonded devices' list. Therefore, given the set of neighbors of current and antecedent gateways, i.e., N_{Cur} and N_{Ant}, respectively, the antecedent gateway requests a set of its neighbors $N_{rem} \in N_{Ant}$ for pairing-bonding information removal where

$$N_{rem} = N_{Ant} - N_{Cur} - \{Cur\} \tag{4}$$

Consider the arrangement of gateways shown in Fig. 10. The movement of the user is shown with a dotted line which crosses over the range of gateways `A`, `B`, `C`, and `D`, i.e., the user's BLE device gets service from these gateways in order while moving through the area.

The following describes how we optimize the memory consumption for pairing-bonding information using the above principles. At the time the BLE device gets connected with gateway `A`, gateway `A` shares the device's pairing bonding information with its neighbor gateways `E`, `B`, and `H` (shown with green arrows). With the device's movement, when the connection is migrated from gateway `A` to gateway `B`, gateway `B` shares the pairing-bonding information with only the gateways `F`, `C`, and `I` (shown with green arrows). Here, the antecedent gateway is `A`, the current gateway is `B`, $N_{Ant} = \{E, B, H\}$, and $N_{Cur} = \{E, A, H, F, C, I\}$. Therefore, $N_{add} = N_{Cur} - N_{Ant}$ - $\{Ant\} = \{F, C, I\}$. Similarly, with the device's movement, when the connection is migrated from gateway `B` to gateway `C`, gateway `C` shares the pairing-bonding information with only gateways `G`, `D`, and `J` (shown with green arrows). Additionally, since the device's connection has been migrated to gateway `C`, the gateway `B` sends a request to gateways `E`, `A`, and `H` to remove the pairing-bonding information of the device.

Here, the antecedent gateway is B, the current gateway is C, $N_{Ant} = \{E, A, H, F, C, I\}$, and $N_{Cur} = \{F, B, I, G, D, J\}$. Therefore, $N_{rem} = N_{Ant} - N_{Cur}$ - $\{Cur\} = \{E, A, H\}$. Similarly, when the connection is migrated from gateway C to gateway D, gateway C sends a request to gateways F, B, and I to remove the pairing-bonding information of the device.

6.9 Handling Compromised Gateways

If an initially trusted gateway gets compromised, the pairing-bonding information about the bonded devices in that gateway should no longer be used by other gateways. We call these devices *to be removed/unpaired devices*. The bonded devices' pairing-bonding keys stored in the compromised gateway are also shared with other gateways. As a result, other gateways use the same pairing-bonding keys for their communication with the *to be removed devices* and thus the compromised gateway may listen to those BLE communication channels. In order to prevent this situation, *SeamBlue* allows the compromised gateway's neighbors (i.e., the one-hop neighboring gateways) to remove the *to be removed devices* from their memory. The neighboring gateways then propagate this removal event to their neighbors (i.e., the two-hop neighbors of the compromised gateway) so that those bonded devices are also removed from the two-hop neighboring gateways. Note that the removal event is propagated up to only two-hop neighbors because the *to be removed devices* are not stored as bonded devices in the gateways beyond this region (according to Section 6.8). The trusted gateways have to execute the pairing-bonding procedure again to connect with those devices.

6.10 Handling Compromised Devices

If an initially trusted BLE device gets compromised, the gateway with which the device is connected disconnects and unpairs/removes that compromised device from its memory. This removal event is then propagated up to only the one-hop neighboring gateways so that the compromised device is removed. In this way, *SeamBlue* ensures that no compromised BLE devices are present in the system. *SeamBlue* also enlists the compromised devices in the *blacklist* of every trusted gateway so that the compromised devices cannot connect with any of the gateways in the system.

6.11 Implementation Notes

We briefly discuss some of the key implementation issues.

6.11.1 Static Analysis

We implemented the static analysis for finding state variables using the LLVM 3.8 compiler infrastructure [32] by directly following the design from Section 6.1. The LLVM passes operate on the LLVM intermediate representation, which is a low level strongly typed language-independent program representation tailored for static analyses and optimization purposes. The LLVM IR is generated from the C source code by clang. We used Bluedroid [24] for Android smartphones and BlueZ [23], an open source implementation, for many other BLE devices as the BLE protocol implementation.

6.11.2 Runtime Value Extraction

We instrument the Bluedroid (Android 4.2 and later) and BlueZ (Android 4.1 and before) for extracting the runtime values of the bonding and connection related state variables.

TABLE 2
Configurations of Nexus 5 and Nexus 6 Smartphones and Alcatel Onetouch Pixi Tablet

Features	Nexus 5	Nexus 6	Alcatel Onetouch	Moto 360 (1st Gen.)
Processor	Qaud-core Krait CPU	Quad-core Krait 450 CPU	Quad-Core CPU	TI OMAP 3 Single-Core
Processor Speed	2.3 GHz	2.7 GHz	1.2 GHz	1GHz
RAM	2 GB RAM	3 GB RAM	1 GB RAM	512 RAM
WiFi	Yes	Yes	Yes	Yes
Celullar	4G/LTE	4G/LTE	4G/LTE	No
BLE Central	Yes	Yes	Yes	No
BLE Peripheral	No	Yes	Yes	Yes
OS	Android	Android	Android	Android
OS	Lollipop	Marshmallow	Lollipop	Wear

6.11.3 SeamBlue *App*

Our custom written *SeamBlue* application with Java (J2SE) implements the gateway selection algorithm and uses OpenSSL libraries [43] for performing the cryptographic operations. The *SeamBlue* application running on the gateways uses TCP connections to transfer bonding and connection related information and to exchange the control signals.

7 Evaluation

This section starts with the experimental setup followed by two sets of evaluations. First, the success rate of BLE connection migration is measured and the ping-pong effect is evaluated. Second, the overhead of *SeamBlue* is measured with the test bed we build.

7.1 Experimental Setup

7.1.1 Devices

We use five Nexus 5 phones as gateways (i.e., BLE centrals), one Nexus 6 phone, one Alcatel Onetouch tablet, and one Moto 360 watch (1st generation) as IoT devices. The Nexus 5 phones have only the BLE central feature whereas the Nexus 6 and the Alcatel Onetouch tablet have both the BLE central and the BLE peripheral capabilities. The configuration of Nexus 5, Nexus 6, Alcatel Onetouch Pixi tablet, and Moto 360 watch is summarized in Table 2.

7.1.2 Testbed

We have built a testbed (as shown in Fig. 11) by hanging the Nexus 5 smartphones as IoT gateways on the walls along the hallways of our department. To evaluate the amount of time required and the number of data packets lost for a connection migration, we first organize the gateways in a linear topology (where any physical space is shared by at most two gateways) and move a BLE device (e.g., Nexus 6 phone or Alcatel Onetouch Tablet or Moto 360 smartwatch) along the hallway from the first gateway to the fifth. Our testbed setup resembles the *SeamBlue* system that can be deployed in hospitals, airports, and modern theme parks as discussed in Section 2. To evaluate the ping-pong effect, we make changes to the topology and arrange the gateways in a manner so that some areas fall within the ranges of more than two gateways.

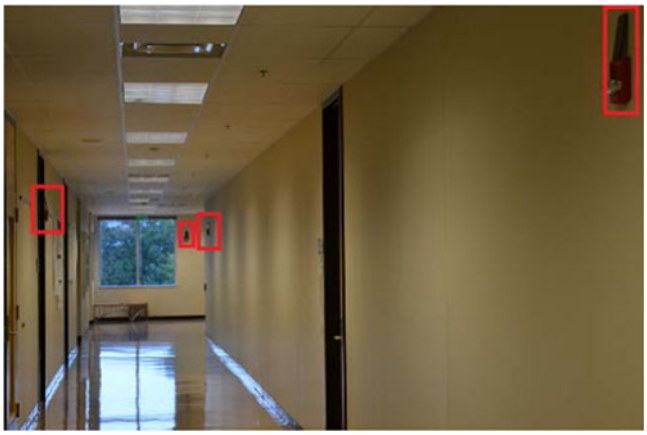

Fig. 11. A part of the testbed showing smartphones (hanging near the fire alarms) that served as gateways.

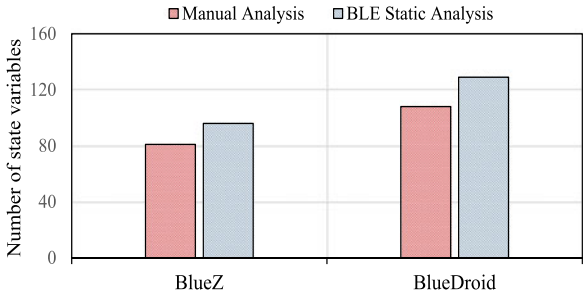

Fig. 12. Number of connection state variables found through manual and static analysis.

7.1.3 Applications

We use the *nRF Connect* application [44] downloaded from the Google Play store for BLE peripherals. For the gateways, we have developed a custom application. We use the heart rate monitoring service that periodically sends heart rate measurement in a single BLE packet of size 20-bytes every second from a BLE peripheral device to the BLE gateway. The heart rate monitoring service is one of the representative applications of *SeamBlue* where patients wearing BLE-enabled heart rate monitoring devices may move indoor or outdoor and may require to migrate the BLE connection from one gateway to another for continuous connectivity. Each data point reported in the experiment is obtained by taking the average of at least five runs.

7.2 BLE Static Analysis

Fig. 12 shows the number of state variables found through manual analysis and using static analysis on both BlueZ [23] and Bluedroid [24] open source BLE implementations. Our manual analysis resulted in some missing state variables because of which we could not successfully migrate the BLE connection from one gateway to other.

7.3 BLE Connection Migration Success Rate

A connection gets migrated from the current gateway to subsequent gateways based on the location and movement direction of the user carrying the BLE peripheral enabled IoT devices/smartphones. We found that every connection migration request was successful both in case of static IoT devices and in the case of IoT devices moving at different speeds. Hence, the success rate we observe for *SeamBlue* in our testbed is 100 percent.

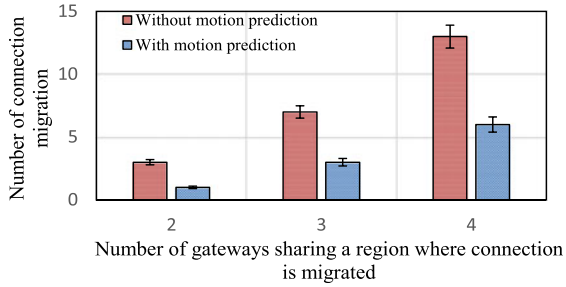

Fig. 13. *SeamBlue* handles ping-pong effect by performing motion prediction during connection migration.

7.3.1 Ping-Pong Effect

To evaluate the ping-pong effect we arrange the gateways in a formation so that the ranges of more than two gateways overlap. Users equipped with IoT devices pass through those shared regions while moving from one gateway to another.

Fig. 13 shows that if users move randomly every after two seconds, the number of times the connection is switched among gateways increases almost at 2X rate with the number of gateways sharing common regions. With the *SeamBlue*'s motion prediction mechanism, the number of connection switches among gateways reduces almost half times than the number without using motion prediction. In the case of four overlapping gateways, the users random movement direction causes a higher number of connection migrations using our simple motion prediction mechanism. This can be improved by using sophisticated motion prediction techniques that leverage additional information about users previous movements, geographical map, and applications.

7.4 BLE Connection Migration Cost

7.4.1 Extra Bytes Required for Connection Migration

In both *partial stack cloning* and *full stack cloning*, the current gateway sends a 512-bytes blob containing the bonding related information to each of its neighbors. However, for *full stack cloning*, the current gateways sends a 2048-bytes blob containing values of all the connection related variables to the next gateway where the connection is going to be migrated.

7.4.2 Time Required for Adding a Peripheral as a Bonded Device

As part of the connection migration procedure, the current gateway sends bonding related information to its neighboring gateways. Upon reception of this information they add the BLE peripheral as a bonded device into their NVRAM when they receive the bonding related information from the current gateway. Adding the peripheral as a bonded device requires the IoT gateway to load the device information, e.g., device address, device type, address type, and keys from the main memory, and then store this information into the gateway's NVRAM for use in future communications. Table 3 shows the mean time required by IoT gateways of different device types to load a peripheral.

Table 3 shows that the Nexus 6 smartphone requires the least amount of time to add a peripheral as a bonded device due to its higher CPU speed and memory capacity compared to the other devices as listed in Table 2.

7.4.3 Time Required for Transferring State Variables

Fig. 14 shows that the time required to transfer the state variables to a trusted and an untrusted gateway over WiFi for

TABLE 3
Time Required for Adding a Peripheral as a Bonded Device

Gateway	Loading Time (ms)	Storing Time (ms)	Total Time (ms)
Nexus 5	40.5	19.1	60.4
Nexus 6	36.7	17.4	54.1
Alcatel OneTouch	43.2	20.3	63.5

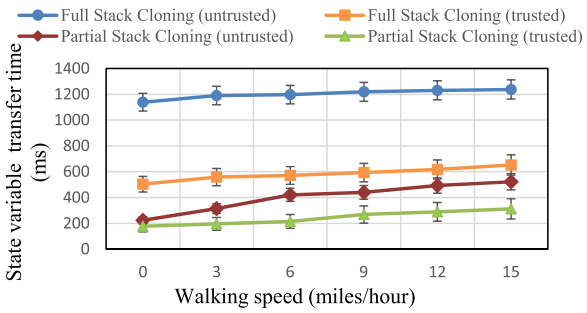

Fig. 14. Time required for transferring the state variables.

both *partial stack cloning* and *full stack cloning* approaches. Because of the proprietary drivers of the BLE gateways used in our testbed, we could not use a open source controller implementation (we could used only the host part) of the BLE stack. Therefore, the results of the *full stack cloning* are obtained by simulating the access of the link layer state variables and using the results of *partial stack cloning*. We assume that the symmetric group keys of trusted gateways and public keys of untrusted gateways are already known by the respective gateways beforehand. Fig. 14 shows that the time required for transferring the state variables for untrusted gateways is almost 2X of the time required for the trusted gateways for different speeds of the user. This is because the current gateway first establishes a session key by performing the Diffie-Hellman key exchange algorithm (which is a time consuming operation) using public keys and then shares the state variables securely with the untrusted gateway whereas for the trusted subsequent gateway the current gateway derives the session key from the shared group key in a much faster way using one-way cryptographic hash function.

Note that the *partial stack cloning* sends only the bonding related information and requires the next gateway to send connection request to the IoT devices for connection migration. Therefore, the *partial stack cloning* always requires less time for transferring state variables than *full stack cloning*. Also, since the control plane packet losses increase with users mobility, the time required for all the scenarios (shown in Fig. 14) increases slightly with the increase of the users moving speed.

7.4.4 Time Required for Connection Migration

The time required for connection migration in *full stack cloning* is computed by considering the time required to (1) extract the values of connection related variables, (2) send this information to the next gateways securely, (3) decrypt the received information, and (4) update the connection related state variables. On the contrary, the time required for connection migration in *partial stack cloning* is only the time to establish a new connection without further creating any pairing and bonding between the subsequent gateways and the IoT devices.

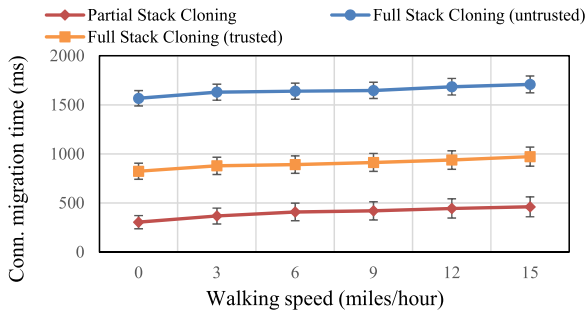

Fig. 15. Time required for connection migration.

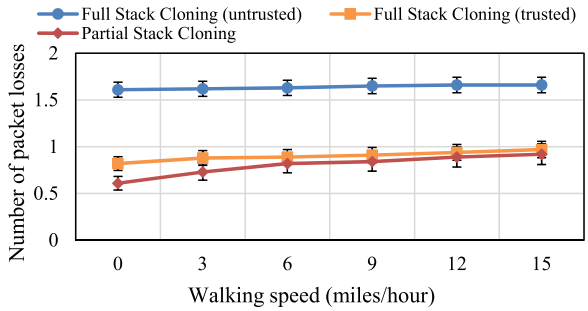

Fig. 16. Number of packets lost when data packets are sent with 1 second time interval.

Fig. 15 shows that the connection migration time increases with the increase of users speed as there are more packet losses associated with increased mobility of users. Also, the connection migration time for the *partial stack cloning* is smaller than that of the *full stack cloning* because creating a new connection between the subsequent gateway and the IoT device does not require any cryptographic operation in the case of the *partial stack cloning*. However, we observed that establishing a BLE connection between a device and a gateway sometimes require multiple connection attempts because of a known implementation issue [45] of the Android Lollipop OS used by the Nexus 5 phones (i.e., the BLE gateways). Due to this reason, a few times, the *partial stack cloning* required multiple connection requests for a single connection migration.

There is a trade-off between the full stack cloning and the partial stack cloning. The *SeamBlue*'s connection migration mechanism with *full stack cloning* does not require a new connection request from the target/next BLE gateway as opposed to the *partial stack cloning*. Thus *full stack cloning* requires a smaller number of message transmissions than the *partial stack cloning*. As a result, seamless migration with *full stack cloning* has lower power consumption compared to the *partial stack cloning*. However, since the *full stack cloning* requires a higher number of instrumented instructions as it needs to extract a higher number of connection state variables, it results in higher delays than the *partial stack cloning* for transferring connection. Since the *partial stack cloning* requires further connection request(s) from the subsequent gateway to an IoT device for connection migration, it incurs higher power consumption for sending/receiving more number of messages than the *full stack cloning*.

7.4.5 Data Packet Loss

Fig. 16 shows the number of data packets lost as an impact of BLE connection migration when the heart rate monitoring application running on an IoT device sends data to the

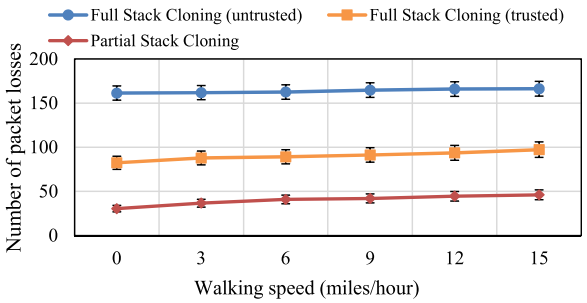

Fig. 17. Number of packets lost when data packets are sent with 20 ms time interval (minimum connection interval for BLE connection).

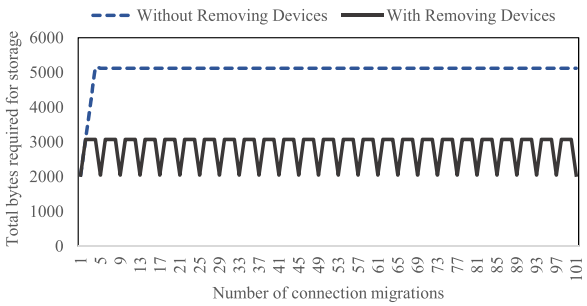

Fig. 18. Memory required to store a BLE peripheral device to the five Nexus 5 gateways arranged in a linear topology.

gateway after every 1 second. *Full stack cloning* with untrusted gateway causes a loss of at most 2 packets which are about 2X of the other scenarios. With our implementation of the *SeamBlue* system, the number of heart-rate monitoring packets lost has a direct relationship with the connection migration time. This is because the packets from a BLE device are not processed by either the current gateway or the subsequent gateway during a connection migration. The current gateway stops processing the BLE packets at the start of a connection migration, and the subsequent gateway resumes processing the BLE packets once the connection is transferred. As a result, packets sent during the connection migration time are not processed and and thus are lost. Therefore, the trend for the packet losses is similar to the connection migration time as shown in Fig. 16. However, the number of packet losses can be further minimized if the current gateway disconnects the BLE connection with the peripheral device after transferring the pairing-bonding and connection information and immediately establishes a new connection with the subsequent gateway.

Fig. 17 shows a stress testing of packet losses when the heart rate data is sent every after 20 ms intervals which is the minimum connection interval for BLE devices. For *full stack cloning*, around 160 data packets were lost during connection migration which span the user's heart rate information for only about 1.5 minutes.

7.4.6 Storage Requirements

Fig. 18 shows the number of bytes required to store a BLE peripheral device in our system where five BLE gateways are arranged in a linear topology. We move the peripheral device from the one end to the other end of the system and repeat this movement a number of times. As the peripheral device visits the subsequent gateways, soon the device is stored as a bonded device in all the five gateways of the system if we do not remove the device

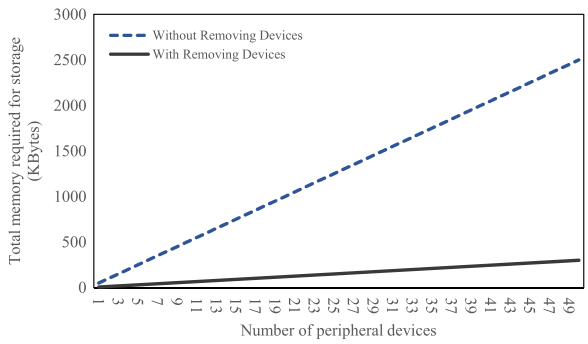

Fig. 19. Memory required to store different number of BLE devices at the *SeamBlue* system consisting of fifty BLE gateways where each gateway has maximum six neighboring gateways.

from any of the gateways. However, with our enhanced storage management component we limit the number of gateways storing the peripheral device to three at any point of the operation. Note that, the number of such gateways varies between two and three as the peripheral device moves from one end to the other.

Fig. 19 shows the maximum number of bytes required to store all the peripheral devices in a system. The system is assumed to have $G = 50$ BLE gateways where each gateway has maximum $N = 6$ neighboring gateways. Let M be the total number of BLE peripheral devices currently present in the system. If no bonded device is ever removed from any gateway of the system, the maximum number of cloned/duplicate instances of the M peripheral devices is $M * G$ (stored at G gateways). However, if we remove the BLE peripheral device according to our storage management strategy, the maximum number of cloned/duplicate instances of the M devices is $M * N$ (stored at G gateways). As the number of peripheral devices increases in the system, Fig. 19 shows that the number of bytes required to store the peripheral devices without our storage management component increases with a much higher rate than that of with our storage management component.

7.4.7 Energy Consumption

If a BLE device (in the absence of *SeamBlue* system) wants to connect to a new gateway, the device has to disconnect the existing BLE connection with the current gateway, broadcast the advertisement packets to the surrounding gateways, perform the pairing-bonding procedure with the subsequent (i.e., the new) gateway, and then establish the BLE connection. However, in *SeamBlue*, a BLE device does not require to create a new pairing-bonding while the connection is migrated from the current gateway to the subsequent gateway. With *partial stack cloning*, the BLE device only disconnects the existing BLE connection with the current gateway and connects immediately with the subsequent gateway without sending any BLE advertisement packets. Thus *SeamBlue* enables the BLE devices to save the energy required for the paring-bonding procedure, and also for the advertisement packets. We leverage BLE's energy consumption models proposed by Treurniet et al. [46] and Siekkinen et al. [47], and find 4.96 mJ energy consumed by a BLE device for the pairing-bonding procedure and 0.1 mJ energy for sending an advertisement packet in a BLE channel. Note that the BLE device may send more than one advertisement packets until it is connected with the subsequent gateway. Therefore, the BLE devices in *SeamBlue*

TABLE 4
Energy Required by a BLE Gateway to Transfer the Pairing-Bonding and Connection Information

Scenario	Key Generation	Key Exchange	Encryption	Signature Generation	Data Exchange	Total
Trusted	5.318 mJ	150.89 mJ	0.62739 mJ	0.38912 mJ	152 mJ	309.22451 mJ
Untrusted	875.96 mJ	1046.5 mJ	0.82944 mJ	0.38912 mJ	152 mJ	2075.67856 mJ

system save at least 5.06 mJ amount of energy compared to that of the non-*SeamBlue* system.

We use the energy models proposed by Castiglione et al. [48], [49] and Potlapally et al. [50] to compute the energy required (as shown in Table 4) by the current gateway in the *SeamBlue* system for encrypting the paring-bonding and connection information. For an untrusted subsequent gateway, the current gateway requires to generate a shared session key using Diffie-Hellman key exchange protocol which results in higher energy consumption than that for a trusted subsequent gateway. *SeamBlue* uses AES-128-CBC for symmetric key encryption and SHA1 for signature generation. By using the Android phone's energy model for wireless communications proposed by Nika et al. [51], we find the energy consumed for transferring the encrypted pairing-bonding and connection information over WiFi to be 152 mJ.

8 Security Analysis

In this section, we analyze the security and privacy of our *SeamBlue* system with respect to the adversary model discussed in Section 5.4. We show that the adversary with the given capability cannot compromise the secure communication in *SeamBlue*.

- *The adversary cannot inject or modify or replay BLE packets*. *SeamBlue* ensures that the gateways always perform secure communication with BLE peripheral devices using the keys established through pairing-bonding procedure and later on shared securely with the subsequent gateways during connection migration. *SeamBlue* uses the LTK (long term key) along with AES-128 encryption mechanism, which is already proven to be secure, for encrypting the protocol data unit (PDU). For signing the PDU, *SeamBlue* uses the CSRK (Connection Signature Resolving Key) and generates the signature which is placed after the PDU. Upon reception of a BLE packet, the receiving device verifies the signature using the CSRK. If the signature is correct, the PDU is assumed to come from the trusted device. The signature is composed of a message authentication code (MAC) generated by the signing algorithm and a counter. The counter is used to protect against replay attacks and is incremented on each signed data PDU sent. Thus *SeamBlue* ensures that every packet sent or received by client and server is encrypted, authenticated, and integrity-protected. Thus maliciously injected or modified or replayed packets by adversaries are always identified.

  We also use a cryptographic protocol verification tool, *ProVerif* [52], to formally verify the authenticity and secrecy of the messages exchanged between the BLE device and the gateway. *ProVerif* assumes that the adversary has complete control over the network, i.e., it can overhear, intercept, and synthesize any message and is only limited by the constraints of the cryptographic methods used. Using applied pi calculus, we first model the communication protocol between the BLE device and the gateway using the LTK for encryption and CSRK for signature generation. We then formally verify that the messages are authenticated, i.e., messages are issued from a legitimate endpoint. In addition, we verify that the adversary cannot learn the BLE messages sent over the public communication channel. The model and the corresponding verification code are given in Fig. 20.
- *Untrusted gateways cannot learn the previous and future BLE packets*. When a BLE connection is required to transfer from a trusted gateway to a not-already trusted gateway, the trusted gateway first changes the pairing-bonding keys (e.g., LTK and CSRK) and connection parameters, and then provides the new set of connection information to the not-already

```
type ltk.
type csrk.

fun senc(bitstring, bitstring, ltk):bitstring.
reduc forall m:bitstring, r:bitstring, k:ltk;
      sdec(senc(m, r, k), k) = m.

fun sign(bitstring, csrk): bitstring.
reduc forall m:bitstring, k:csrk; getmessage(sign
      (m, k)) = m.
reduc forall m:bitstring, k:csrk; checksign(sign(
      m, k), k) = m.

free c:channel.
free message:bitstring [private].

query attacker(message).

event acceptBleDevice(bitstring).
event endBleGateway(bitstring).

query x:bitstring; event(endBleGateway(x))==>
      event(acceptBleDevice(x)).

let ble_device(enc_key:ltk, sign_key:csrk) =
new counter:bitstring;
let ciphertext = senc(message, counter, enc_key)
      in
      event acceptBleDevice(message);
      out (c, (ciphertext, sign(ciphertext,
      sign_key), counter)).

let ble_gateway(enc_key:ltk, sign_key:csrk) =
      in (c, (ciphertext:bitstring, signature:
      bitstring, xcounter: bitstring));
      if checksign(signature, sign_key) =
      ciphertext then
      let (plaintext:bitstring, counter:bitstring)
      = sdec(ciphertext, enc_key) in
      if counter = xcounter then
      event endBleGateway(plaintext).

process
      new enc_key: ltk;
      new sign_key: csrk;
      ((!ble_device(enc_key, sign_key)) | (!
      ble_gateway(enc_key, sign_key)))
```

Fig. 20. Proof of authentication, secrecy, and replay protection of the BLE data packets.

trusted gateway. As a result, the not-already trusted gateway cannot obtain the previous set of keys used for exchanging BLE packets securely between the trusted gateways and BLE peripheral devices. Similarly, when the connection is transferred from the untrusted zone to trusted zone, the trusted current gateway changes the connection information so that the untrusted gateway cannot learn the future BLE packets.

- *The Adversary cannot derive pairing-bonding and connection information.* While migrating a BLE connection, the current gateway first generates a session key either using the shared group key (in the case of trusted gateways scenario) or the Diffie-Hellman key exchange protocol (in the case of untrusted gateway scenario). The current gateway also appends a new *nonce* to the pairing-bonding/connection information (in both the trusted and untrusted gateway scenarios) every time it sends the information to the subsequent gateway. Thus, use of different session keys and *nonces* result in different ciphertexts for the same pairing-bonding/connection information sent by a gateway. As a result, the adversary cannot infer or derive any correlation in the pairing-bonding/connection information from ciphertexts.
- *The adversary cannot impersonate legitimate gateways.* During connection migration the adversary may try to impersonate a trusted subsequent gateway with the subsequent gateway's Bluetooth device address. If the spoofed gateway is already trusted, the impersonating gateway is short of the secret group key which is shared only after mutual authentication. If the spoofed gateway is not already trusted, *SeamBlue* allows both the current and the subsequent gateways to authenticate themselves through certificate validation. After that the subsequent gateway can obtain the pairing-bonding keys and connection parameters secretly through the public key cryptography or through establishing a shared session key using the Diffie-Hellman key exchange protocol. Since, a malicious gateway impersonating a legitimate one will not be validated during mutual authentication, the malicious gateway cannot obtain the sensitive pairing-bonding and connection information. Even if the malicious gateway impersonates an already trusted gateway, the adversary cannot decrypt the pairing-bonding or connection related information since it does not have the secret group key. In the case of impersonating a legitimate (i.e., authenticated) subsequent gateway for which the public key certificate is already validated, the adversary cannot decrypt the pairing-bonding or connection related information since it does not know the private key of the legitimate gateway.
- *The adversary cannot retrieve pairing-bonding/connection information even if a gateway is compromised. SeamBlue* ensures perfect forward secrecy for sharing the pairing-bonding/connection information using Diffie-Hellman key exchange protocol. Therefore, if a gateway is compromised, there is no key material on the gateway to help the adversary decrypt previously exchanged ciphertext. Since the adversary cannot decrypt the ciphertexts, it cannot extract the pairing-bonding keys and connection parameters used earlier.
- *The adversary cannot identify the gateway to which an IoT device is connected.* For an IoT device, each subsequent gateway impersonates the initial gateway (which the IoT device initially connects to). Therefore, every BLE data packet received/sent from/to the IoT device includes the same Bluetooth device address for all the subsequent BLE gateways. Since the adversary observes the same Bluetooth device address for different BLE gateways for an IoT device, it cannot locate the particular BLE gateway to which the IoT device is currently connected.

9 Related Work

Despite the heavy use of BLE for numerous smart applications, few research efforts [20], [53], [54] have been devoted to enable seamless connectivity for IoT devices. Zachariah et al. [6] address the problem of running different applications on a single gateway for different IoT services (e.g., heart rate monitoring, activity monitoring, smart home appliance monitoring, etc.) and envision an application-agnostic connectivity for worldwide deployment of IoT gateways. In contrast, *SeamBlue* addresses the existing limitation of seamless connectivity and proposes a framework for seamless connection migration for unmodified IoT devices. Kodeswaran et al. [55] identify timely maintenance of failed sensors as a critical task to ensure minimal disruption to monitoring services, and propose an approach to optimize maintenance scheduling. However, their approach does not consider the case when sensor devices move out of the communication range of gateways or when the gateways suddenly fail.

Michalevsky et al. [10] have developed a mobile application that enables members of a secret community to discover other affiliates that are in proximity to their mobile devices. Das et al. [20] have carried out a measurement-driven study of privacy leakage from communication between wearable fitness trackers and smart phones. These fitness trackers mostly use BLE for communicating and syncing the data with the user's smart phone. Albazrqaoe et al. [54] propose a Bluetooth traffic sniffer, *BlueEar*, by which two Bluetooth-compliant radios can coordinate to learn the hopping sequence of indiscoverable Bluetooth network, to predict adaptive hopping behavior, and mitigate the impacts of RF interference.

As opposed to the cellular-handovers [11], [12], [13], *SeamBlue* does not require modifications to the IoT devices for BLE connection migration. Like cellular-handovers, *SeamBlue* reallocates BLE channels in the *partial stack cloning* through new connection establishment. However, in the *full stack cloning*, *SeamBlue* transfers the BLE channels without creating a new connection.

Compared to our previous paper [1], the current paper has the following novel contributions. We have extended the BLE static analysis that obviates the necessity of finding the state variables in the path constraints and thus the analysis results in a more precise set of state variables. We have also enhanced the security features of connection migration by ensuring the perfect forward secrecy while sharing connection information with a not-already trusted gateway. Our proposed enhance solution prevents—(i) not an already trusted gateway from learning the previous and future BLE packets (ii) the already compromised gateways and the compromised BLE devices from connecting with the benign gateways/BLE devices. We

have also designed a memory/storage management solution that allows BLE gateways to better manage the storing and removing of bonded BLE devices. Finally, we have provided a detailed security analysis of our *SeamBlue* system with a cryptographic protocol verifier,*ProVerif*.

10 Conclusion and Future Work

In this paper, we focus on the problem of IoT devices being unable to connect to multiple gateways seamlessly and thus propose a framework that ensures seamless communication between a mobile IoT device and a remote service in a network of BLE gateway environments. Our framework consists of a static analysis module for the automatic extraction of the state variables required during connection transfer. Moreover, we design a gateway selection mechanism that transfers connection related information to an optimal set of gateways and thus reduces both communication overhead and latency. In future, we would like to evaluate our testbed in a real environment, e.g., airports or shopping malls with a large number of BLE gateways and IoT devices. Also, we would like to investigate if the same connection migration technique can be applied to other communication protocols, e.g., ZigBee [4], used by the IoT devices.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments which helped us to improve the quality and the presentation of this paper. The second author has been partially supported for this work by the Schlumberger Foundation under the Faculty For the Future (FFTF) Fellowship.

References

[1] S. R. Hussain, S. Mehnaz, S. Nirjon, and S. Bertino, "Seamblue: Seamless bluetooth low energy connection migration for unmodified iot devices," in *Proc. Int. Conf. Embedded Wireless Syst. Netw.*, 2017, pp. 132–143.

[2] *How far is the hype surrounding claims of up to 50B IoT and machine-to-machine devices by 2020 away from reality?*. [Online]. Available: http://www.rcrwireless.com/20160628/opinion/reality-check-50b-iot-devices-connected-2020-beyond-hype-reality-tag10

[3] *Bluetooth Low Energy*. [Online]. Available: https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy

[4] *Zigbee*. [Online]. Available: http://www.zigbee.org/what-is-zigbee/

[5] *Near Field Communication*. [Online]. Available: http://nearfieldcommunication.org/

[6] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta, "The internet of things has a gateway problem," in *Proc. 16th Int. Workshop Mobile Comput. Syst. Appl.* 2015, pp. 27–32. [Online]. Available: http://doi.acm.org/10.1145/2699343.2699344

[7] *Moto 360 1st Generation* [Online]. Available: https://www.motorola.ca/products/moto-360-gen-1

[8] *Samsung Gear* [Online]. Available: http://www.samsung.com/us/explore/gear-s3/?cid=ppc-

[9] A. A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein, "Beetle: Flexible communication for bluetooth low energy," in *Proc. 14th Annu. Int. Conf. Mobile Syst. Appl. Services.*, 2016, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906414

[10] Y. Michalevsky, S. Nath, and J. Liu, "Mashable: Mobile applications of secret handshakes over bluetooth low energy," in *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.*, Jul. 2016. [Online]. Available: https://www.microsoft.com/en-us/research/publication/mashable-mobile-applications-of-secret-handshakes-over-bluetooth-le/

[11] A. Sgora and D. D. Vergados, "Handoff prioritization and decision schemes in wireless cellular networks: A survey," *IEEE Commun. Surveys Tut.*, vol. 11, no. 4, pp. 57–77, Oct.–Dec. 2009.

[12] A. Rath and S. Panwar, "Fast handover in cellular networks with femtocells," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2012, pp. 2752–2757.

[13] D. Wong and T. J. Lim, "Soft handoffs in CDMA mobile systems," *IEEE Personal Commun.*, vol. 4, no. 6, pp. 6–17, Dec. 1997.

[14] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu, "Prediction and indexing of moving objects with unknown motion patterns," in *Proc. ACM SIGMOD Int. Conf. Manag. Data.*, 2004, pp. 611–622. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007637

[15] *The future of IoT in airports Lessons from London City Airport* [Online]. Available: http://www.totalbluesky.com/2015/03/10/future-iot-airports-lessons-london-city-airport/

[16] *Disneys $1 Billion Bet on a Magical Wristband* [Online]. Available: https://www.wired.com/2015/03/disney-magicband/

[17] *Bluetooth 4.2 Core Specification* [Online]. Available: https://www.bluetooth.com/specifications/bluetooth-core-specification/technical-considerations

[18] *Bluetooth Low Energy* [Online]. Available: http://groups.inf.ed.ac.uk/teaching/slipb13–14/Ewan/

[19] K. Fawaz, K.-H. Kim, and K. G. Shin, "Protecting privacy of BLE device users," in *Proc. 25th USENIX Security Symp.* Aug. 2016, pp. 1205–1221. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/fawaz

[20] A. K. Das, P. H. Pathak, C.-N. Chuah, and P. Mohapatra, "Uncovering privacy leakage in BLE network traffic of wearable fitness trackers," in *Proc. 17th Int. Workshop Mobile Comput. Syst. Appl.*, 2016, pp. 99–104. [Online]. Available: http://doi.acm.org/10.1145/2873587.2873594

[21] D. Midi, A. Rullo, A. Mudgerikar, and E. Bertino, "Kalis - a system for knowledge-driven adaptable intrusion detection for the internet of things," in *Proc. 37th IEEE Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 656–666.

[22] *Rogue Device Detection* [Online]. Available: https://www.pwnieexpress.com/solutions/rogue-device-detection

[23] *Official Linux Bluetooth Protocol Stack* [Online]. Available: http://www.bluez.org/

[24] *Bluetooth - Android Open Source Project* [Online]. Available: http://llvm.org/

[25] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE Symp. Security Privacy*, May 2010, pp. 317–331.

[26] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," Internet Society, 2005.

[27] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proc. 18th Int. Conf. Static Anal.*, 2011, pp. 95–111. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041552.2041563

[28] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation.*, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[29] F. A. Teixeira, G. V. Machado, F. M. Q. Pereira, H. C. Wong, J. M. S. Nogueira, and L. B. Oliveira, "Siot: Securing the internet of things through distributed system analysis," in *Proc. 14th Int. Conf. Inf. Process. Sensor Netw.*, 2015, pp. 310–321. [Online]. Available: http://doi.acm.org/10.1145/2737095.2737097

[30] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao, "Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 388–400. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813643

[31] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using statealyzr," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, Mar. 2016, pp. 239–253. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khalid

[32] *LLVM* [Online]. Available: http://llvm.org/

[33] *Control Flow Graph* [Online]. Available: https://en.wikipedia.org/wiki/Control_flow_graph

[34] *LLVM Language Reference Manual* [Online]. Available: http://llvm.org/docs/MIRLangRef.html

[35] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.*, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557
[36] Call Graph. [Online]. Available: https://en.wikipedia.org/wiki/Call_graph
[37] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proc. 28th ACM SIGPLAN Conf. Programming Language Des. Implementation*, 2007, pp. 278–289. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250766
[38] R. Faragher and R. Harle, "An analysis of the accuracy of bluetooth low energy for indoor positioning applications," in *Proc. 27th Int. Tech. Meeting. Satellite Div.e Inst. Navigation*, 2014, pp. 201–210.
[39] *Public Key Certificate* [Online]. Available: https://en.wikipedia.org/wiki/Public_key_certificate
[40] *DiffieHellman key exchange* [Online]. Available: https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange
[41] *Public Key Cryptography* [Online]. Available: https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange
[42] S.-H. Seo, M. Nabeel, X. Ding, and E. Bertino, "An efficient certificateless cryptography scheme without pairing," in *Proc. 3rd ACM Conf. Data Appl. Security Privacy*, 2013, pp. 181–184. [Online]. Available: http://doi.acm.org/10.1145/2435349.2435375
[43] Openssl Tech. Rep. [Online]. Available: https://www.openssl.org/
[44] *nRF Connect for Mobile* [Online]. Available: https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp&hl=en
[45] *Android 5.0: Connect() extremely unreliable but works fine on Android 4.4.x* [Online]. Available: https://www.pwnieexpress.com/solutions/rogue-device-detection
[46] J. J. Treurniet, C. Sarkar, R. V. Prasad, and W. d. Boer, "Energy consumption and latency in BLE devices under mutual interference: An experimental study," in *Proc. 3rd Int. Conf. Future Internet Things Cloud*, Aug 2015, pp. 333–340.
[47] M. Siekkinen, M. Hiienkari, J. K. Nurminen, and J. Nieminen, "How low energy is bluetooth low energy? comparative measurements with zigbee/802.15.4," in *Proc. IEEE Wireless Commun. Netw. Conf. Workshops*, April 2012, pp. 232–237.
[48] A. Castiglione, F. Palmieri, U. Fiore, A. Castiglione, and A. D. Santis, "Modeling energy-efficient secure communications in multi-mode wireless mobile devices," *J. Comput. Syst. Sci.*, vol. 81, no. 8, pp. 1464–1478, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000200001400186X
[49] A. Castiglione, A. D. Santis, A. Castiglione, F. Palmieri, and U. Fiore, "An energy-aware framework for reliable and secure end-to-end ubiquitous data communications," in *Proc. 5th Int. Conf. Intell. Netw. Collaborative Syst.*, Sep. 2013, pp. 157–165.
[50] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha, "Analyzing the energy consumption of security protocols," in *Proc. Int. Symp. Low Power Electron. Des.*, Aug 2003, pp. 30–35.
[51] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: A measurement study and implications for network applications," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas.*, 2009, pp. 280–293. [Online]. Available: http://doi.acm.org/10.1145/1644893.1644927
[52] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proc. 14th IEEE Workshop Comput. Security Foundations*, 2001, Art. no. 82. [Online]. Available: http://dl.acm.org/citation.cfm?id=872752.873511
[53] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology," *Sensors*, vol. 12, no. 9, p. 11734, 2012. [Online]. Available: http://www.mdpi.com/1424–8220/12/9/11734
[54] W. Albazrqaoe, J. Huang, and G. Xing, "Practical bluetooth traffic sniffing: Systems and privacy implications," in *Proc. 14th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2016, pp. 333–345. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906403
[55] P. A. Kodeswaran, R. Kokku, S. Sen, and M. Srivatsa, "Idea: A system for efficient failure management in smart iot environments," in *Proc. 14th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2016, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906406

Syed Rafiul Hussain is working toward the PhD degree in the Department of Computer Science, Purdue University. He has worked in the Networking and Mobility Lab, Hewlett-Packard Labs, in Palo Alto, CA (2015–2017), as a research associate intern. His work on seamless secure BLE connection migration has received a Best Paper Nomination Award at EWSN'17. His research interests include network and IoT security. He is a member of the ACM and the IEEE.

Shagufta Mehnaz is working toward the PhD degree in computer science at Purdue University. Her work has applications in the area of privacy preserving analytics on IoT data. She won the Best Paper Award from the 2017 ACM Conference on Data and Applications Security and Privacy. She is a member of the Center for Education and Research in Information Assurance and Security (CERIAS). Her research interests include information privacy and security. She is also a member of the ACM and the IEEE.

Shahriar Nirjon received the PhD degree from the University of Virginia, Charlottesville, in 2014. He is an assistant professor in the Department of Computer Science, the University of North Carolina, Chapel Hill. His research challenges that he deals with include practical issues in physical world sensing, user-contexts and mobility, real-time issues, and resource constraints of the embedded platform. His work has applications in the area of remote health and wellness monitoring, and mobile health. He has won a number of awards, including two Best Paper Awards, at the Mobile Systems, Applications and Services (MOBISYS 2014), and the Real-Time and Embedded Technology and Applications Symposium (RTAS 2012). He has worked as a research scientist in the Networking and Mobility Lab, the Hewlett-Packard Labs, Palo Alto, CA (2014-2015), and as a research intern at Microsoft Research, Redmond, WA (Summer 2013) and at Deutsche Telekom Lab, Los Altos, CA (Summer 2010). Several of his work have been highlighted in the electronic and print media, including the *Economist*, the *New Scientist*, and the BBC. He is interested in building practical cyber-physical systems that involve embedded sensors and mobile devices, mobility and connectivity, and mobile data analytics. He is a member of the IEEE.

Elisa Bertino is a professor of computer science with Purdue University, and serves as director of the CyberSpace Security Lab (Cyber2SLab). She is also an adjunct professor of computer science & info tech, RMIT, in Melbourne. Prior to joining Purdue in 2004, she was a professor and department head in the Department of Computer Science and Communication at the University of Milan. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, CA, in the Microelectronics and Computer Technology Corporation, at Rutgers University, and at Telcordia Technologies. Her recent research focuses on database security, digital identity management, policy systems, and security for web services. She received the IEEE Computer Society 2002 Technical Achievement Award, the IEEE Computer Society 2005 Kanai Award, and the ACM SIGSAC Outstanding Contributions Award. She is currently serving as EiC of the *IEEE Transactions on Dependable and Secure Computing*. She is a fellow of the ACM, of the IEEE, and the AAAS.