

Extending Linux Controllability of Bluetooth Low Energy Devices in the IoT

Sorin Zamfir*, Bogdan-Alexandru Lupu*, George-Alex Stelea* and Dan-Nicolae Robu*

*Transilvania University of Brasov

B-dul Eroilor nr.29, 500036 Brasov, Romania

Telephone: (+40) 268-413000

Emails: {zamfir.sorin, george.stelea, robu.dan}@unitbv.ro,

bogdan.alexandru.lupu@gmail.com

Abstract—The Bluetooth Low Energy technology has become more and more popular among wearable devices due to its low price and reduced power consumption. However, the development industry has oriented itself among mobile platforms neglecting the standard operating systems and providing only a limited set of integration possibilities most often using a smart-phone, restricting in this way the utility of such sensors in more complex services. In this paper we provide a demonstrator capable of running on a standard Linux host, interacting with two Bluetooth enabled devices and also demonstrate a programmatic approach using the Java language. Furthermore we explain in detail our exploratory methodology and the tools used to achieve our proof-of-concept.

I. INTRODUCTION

While the invention of Bluetooth at Ericsson [1] in the early nineties shaped an alternative way to RS-232 [2] for transferring data between devices, the consumer market pushed this technology into new domains, increasing its popularity in the last decade. Devices like car head-units, laptops, fitness trackers and smart-home appliances use Bluetooth to synchronize data in a peer to peer fashion.

Driven by the low power consumption requirement of wearable devices, such as fitness trackers, health monitors and mobile smart-phones, the Bluetooth SIG (Special Interest Group) standardized a so called "low-energy" version of this wireless technology in 2010 [2]. In fact, the governing body of Bluetooth states that it has specifically designed it for the IoT (Internet of Things) ecosystem and that is aimed for "short-bursts of long-range radio connection" [4]. Furthermore, the low-energy version uses only the GFSK (Gaussian Frequency Shift Keying) modulation scheme with 2 MHz wide RF channels [4] in contrast to GFSK and PSK (Phase Shift Keying) used in "classic" Bluetooth.

However, the adoption of this technology has sparked interest mostly among mobile application developers making such software dependent on specific programming languages and limiting its vision to simple use-cases. In our opinion this approach restricts its applicability from its true potential in the IoT scope. The *motivation* of our work was to extend the controllability of Bluetooth Low Energy devices. Our aim was to broaden the possibility of new and much more complex scenarios involving such appliances.

A. Generic Attribute Profile

A fundamental aspect of Bluetooth Low Energy is represented by how data is actually accessed from a peripheral device. Based on the low-level ATT (Attribute Protocol), the communication pattern uses a client-server architecture, and a series of new concepts such as services and characteristics [5]. In most cases the GATT (Generic Attribute Profile) *server* is represented by the sensor providing the actual data (e.g. a smart-watch), whereas the GATT *client* is the consumer of this data, frequently in the form of a smart-phone. A distinguishing particularity of this transfer method states that connections are exclusive [5]. This means that once the data association is established with a peripheral device, no further incoming connections are possible to that node.

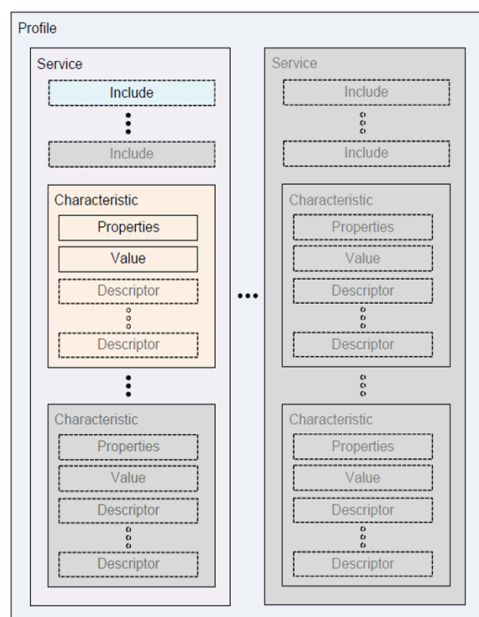


Fig. 1. Generic Attribute Profile Hierarchy [3]

1) *Services*: As stated in the Bluetooth Core Specification [4], a service represents a collection of data and associated behaviors to accomplish a particular function or feature. At a first glance, this approach is somewhat similar to the one used in web services where resources are exposed through an

uniform API (Application Programming Interface) to potential consumers. Additionally, the specification defines two types of services [4] :

- Primary - that present the desired functionality of the device and can be directly accessed
- Secondary - that can be used as a composition item and be referenced from other primary services

The distinction between these two categories is done by the application developer on a higher level.

2) *Characteristics*: A characteristic represents, in a minimalist way, a resource exposed by the actual device that is part of a particular service. In the context of GATT, the definition of a characteristic is achieved using three items [4]:

- Characteristic declaration - represents an attribute that defines the actual resource, identifiable by means of a handle (16 bit or 128 bit Universal Unique Identifier - UUID), and specifies properties such as the address for fetching the actual data and value access permissions.
- Characteristic value declaration - represents the attribute that stores the data of the device (provided by the higher level application) as well as the access permissions.
- Characteristic description declaration - an optional attribute describing the actual contents of the value (e.g. the measurement units, decoding information)

B. Authentication and Encryption

An important aspect in the IoT world refers to security and privacy. One cannot argue that data from smart-devices must be protected to avoid tampering and eaves-dropping, but in most cases practical implementations often ignore the security related features in favor of development speed and faster time-to-market.

Fortunately, Bluetooth LE (Low Energy) provides by design some interesting security features aimed to protect the devices and their owners. Encryption key distribution and authorization of data received on the radio interface represents the main responsibility of the Security Manager. [4].

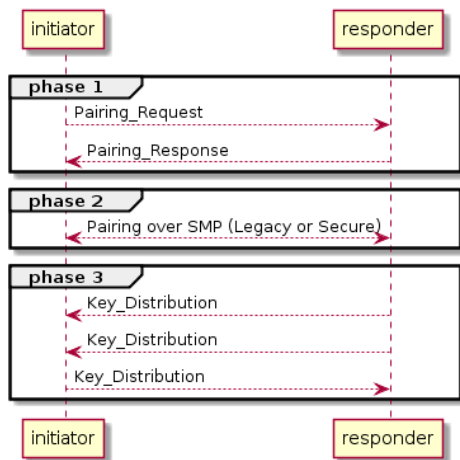


Fig. 2. Pairing phases [4]

Pairing, specific to Bluetooth technology, represents a key process that consists of three phases [4]

- Phase 1 - Feature exchange
- Phase 2 - Short Term Key generation (used in so called Legacy Pairing or Secure Simple Pairing mode - pre-4.2 specification implementations)
- Phase 2' - Long Term Key generation (used in LE secure connections)
- Phase 3 - Transport Specific Key distribution

In the feature exchange phase one of the following methods for authenticating is used :

- Just Works - basically this means no authentication
- Numeric Comparison - request use confirmation from the user on outputs (e.g. display)
- Passkey Entry - similar to a PIN entry requested by one of the devices
- Out Of Band - no user interaction is required

However, even though these mechanisms exist, a series of researches [6], [7] have shown that traffic from Bluetooth LE devices can be "sniffed" using a relatively low cost hardware device [8] thus exposing these systems to a number of passive attack vectors and even the theoretical possibility to inject rogue traffic. Finally, these studies confirm that there are some aspects that are still to be improved when considering low energy devices communicating over Bluetooth.

II. DEVICES OF INTEREST

The DuT (Devices under Test) that were carefully chosen for our demonstrator came from two different vendors (to meet the requirements of two distinct usage scenarios):

Xiaomi MiBand 1S - a fitness tracker manufactured by the Chinese company Xiaomi, one of the cheapest (with a cost of roughly 16 US\$) and most popular health bands on the market. The most important features are the battery-longevity (up to 30 days) and the possibility to wear it under water due to its IP67 (Ingress Protection) rating [10].



Fig. 3. Xiaomi MiBand 1S [9]

L-Tek FireFly FlyTag - a SoC (system on a chip) with a multitude of sensors such as temperature, luminance, humidity, magnetometer, gyroscope and accelerometer produced by the Slovenian company L-TEK in close collaboration with IBM.



Fig. 4. L-Tek FireFly FlyTag [11]

In order to test the extended controllability (above-mentioned among our goals), a standard Linux host was used (development PC) along with the popular Raspberry Pi model 3, both using their embedded Bluetooth adapter. Our efforts towards providing a reusable proof of concept justified this choice of an open system.

While Bluetooth LE technology is well adopted within the mobile smart-phone community, the opportunity of our research was given by the fact that standard operating systems do not benefit from the same amount of popularity among developers. In fact, it would seem that with the notable exception of BlueZ [12], the alternatives for software development are limited to a quite small list of proprietary hardware and software stacks.

As a result, most software written to interact with Bluetooth LE-enabled devices runs on smart-phones and is limited to a one-to-one scenario where the actual mobile device becomes an Internet gateway.

III. IMPLEMENTATION DETAILS

For our software implementation we used the BlueZ software stack in order to access the GATT profiles of the devices under test (DuT). Starting from version 5 [12] onwards, the API is provided using D-Bus [13], allowing the possibility to uncouple the application life-cycle from the user-level libraries offered by this Bluetooth software component.

To be more specific, rather than linking directly the binaries into the application sources and using a native interface, all the APIs are served through a messaging system [13] -important means for our aimed extended controllability, as it allows software developers to interact with the Bluetooth stack by a multitude of programming languages that provide bindings to D-Bus (e.g. C++, Python, Erlang, Perl, TCL, Java).

A. Interacting with MiBand

While Bluetooth SIG provides a set of standardized services and characteristics, vendors prefer to define their own attributes to expose the device sensors and relevant measurement data. As a result, our first step is to explore existing applications and obtain the necessary information to proceed with the implementation.

Fortunately, the GadgetBridge project [14] has already identified and decoded, in their open source Android application, most of the relevant characteristics and services of the MiBand. Our task was to port these under Linux using the BlueZ API and ensure the values are decoded as expected. This provided a back-compatibility of our integration solution at service level.

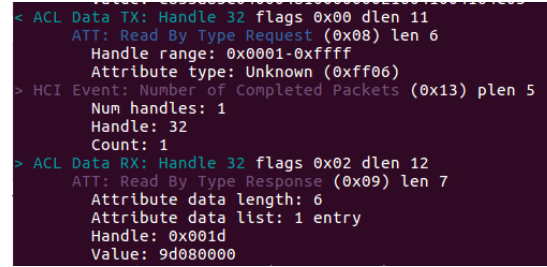
```
$ sudo gatttool -b C8:0F:10:33:A8:3C --primary
... uuid: 00001800-0000-1000-8000-00805f9b34fb
... uuid: 00001801-0000-1000-8000-00805f9b34fb
... uuid: 0000fee0-0000-1000-8000-00805f9b34fb
... uuid: 0000fee1-0000-1000-8000-00805f9b34fb
... uuid: 0000180d-0000-1000-8000-00805f9b34fb
... uuid: 00001802-0000-1000-8000-00805f9b34fb
```

Listing 1. Exploring MiBand primary services with gatttool

Our research started by defining Bash scripts for automating the data acquisition from these devices with the mere purpose of testing. While most of the initial and critical effort was done in this language, in the end we also focused on a more object oriented approach using Java.

We observe that two of the six services (the ones with 0000feeX) do not use standard SIG UUIDs and as a result these represent vendor specific resources. Most notably the 0000fee0 service offers characteristics such as:

- device characteristics (UUID 0000ff01)
- battery information (UUID 0000ff0c)
- device name (UUID 0000ff02)
- real-time steps (UUID 0000ff06)



```
< ACL Data TX: Handle 32 flags 0x00 dlen 11
ATT: Read By Type Request (0x08) len 6
  Handle range: 0x0001-0xffff
  Attribute type: Unknown (0xff06)
> HCI Event: Number of Completed Packets (0x13) plen 5
  Num handles: 1
  Handle: 32
  Count: 1
> ACL Data RX: Handle 32 flags 0x02 dlen 12
ATT: Read By Type Response (0x09) len 7
  Attribute data length: 6
  Attribute data list: 1 entry
  Handle: 0x001d
  Value: 9d080000
```

Fig. 5. Real-time steps Bluetooth trace

Interrogating these characteristics renders binary results that require further decoding into user-understandable details and as result we accomplished a utility script to deal with all the byte processing. *Device name* is encoded as an UTF8 (Unicode Transformation Format) string and a simple hexadecimal to character conversion is sufficient.

Running our utility script renders the following output:

```
$ ./bluetooth-scan.sh
Device name : MI1S
Power level: 78 percent
Status: 4 (0-normal ; 1-low ; 2-charging ; 3-
      full ; 4-charge off)
Number of charges: 13
Hw version: 4
Appearance: 0
Feature: 4
Fw version: 4.16.4.22
Hr version: 3.76.22
Number of steps: 2205
```

Listing 2. Utility script output

Battery information represents a series of 10 bytes where the first byte represents the percentage of battery left, the 7 and 8th bytes represent the number of charges and the 9th byte represents a power status. The *device characteristics* represents a 19 byte long that is decoded as follows: byte 4 represents feature version(distinguishing between multiple versions of MiBands), byte 6 is hardware version, bytes 13 to 16 account for the firmware version and bytes 17-19 represent the heart-rate chip firmware version (specific to MiBand 1s).

Real-time steps specifies the number of steps counted by the device during a 24 hour period. Interrogating this resource

returns a binary encoded number that specifies the actual number of steps detected by the device within this period.

An interesting remark about our approach here, is that these characteristics can be read without any kind of authentication or encryption mechanism. In fact, in our first attempts, we managed to retrieve the above mentioned information using a simple connection mechanism without requiring any kind pairing.

After deeper explorations, we discovered that much more sensitive data (such as activity tracking, heart rate monitoring and notification triggering) cannot be accessed without Bluetooth LE pairing and *application level authentication*. On most mobile devices, low-energy pairing works "out of the box". However, on a standard Linux system, BlueZ requires additional configuration.

Investigation of Bluetooth traces and system logs allowed us to identify the key configurations that need to be made in order to allow pairing with MiBand:

```
root@...# hciconfig hci0 auth
root@...# hciconfig hci0 encrypt
root@...# hciconfig hci0 sspmode 0
root@...# hciconfig hci0
hci0: Type: Primary Bus: USB
BD Address: 18:CF:5E:6F:8C:B6
ACL MTU: 1022:8 SCO MTU: 183:5
UP RUNNING AUTH ENCRYPT
RX bytes:1298 acl:0 sco:0 events:85 errors:0
TX bytes:4101 acl:0 sco:0 commands:85 errors:0
```

Listing 3. BlueZ/adaptor configuration

The authentication and encryption allow the adapter to negotiate a secure connection on Bluetooth level. Disabling ssp (simple pairing mode or legacy pairing) is a necessary step to allow the session negotiation between the fitness tracker and our development platform. After these settings are submitted to the host adapter, pairing can be established successfully, and writing *some* characteristics is possible. Our tests also revealed that not all devices support deactivating sspmode, most notably chip-sets from Atheros and Broadcom seemed to work.

Furthermore, accessing heart-rate sensor data or activity data requires even more privileges that need to be established by using a so called *application level pairing* [14]. This actually means that some characteristics are writable after simple Bluetooth pairing procedure while others require an additional authentication step.

One possible way to identify the sequence used in *application pairing* is by "sniffing" the traffic using an Android phone while the GadgetBridge or even the official vendor application is initializing the bracelet.

The sequence identified for authentication is as follows:

- 1) The first step is to activate vendor notifications on characteristic UUID 0000ff03
- 2) The second step is to setup latency on the vendor LE characteristic UUID 0000ff09
- 3) Third step is to read the device information on characteristic on UUID 0000ff01

- 4) Fourth step is the actual authentication where the user information characteristic UUID 0000ff04 is written with an array of 20 bytes encoding the actual user data.

```
< ACL Data TX: Handle 32 flags 0x00 dlen 27
  ATT: Write Request (0x12) len 22
    Handle: 0x0019
    Data: 42fd06a30018b44b0004004761726669656c6466
> HCI Event: Number of Completed Packets (0x13) plen 5
  Num handles: 1
  Handle: 32
  Count: 1
> ACL Data RX: Handle 32 flags 0x02 dlen 5
  ATT: Write Response (0x13) len 0
> ACL Data RX: Handle 32 flags 0x02 dlen 8
  ATT: Handle Value Notification (0x1b) len 3
    Handle: 0x0016
    Data: 05
```

Fig. 6. MiBand authentication sequence

The connection between steps 3 and 4 is important, since bytes 9 and 10 of the user information depend on specific features that can be identified only after step 3. Also, the last byte of the series represents a CRC8 checksum "xor-ed" with the last two hexadecimal digits of the Bluetooth address of the device [14].

The above procedure renders different behaviors, depending on the state of the fitness tracker and the information sent as user data:

- If the information is completely new, the device will interpret it as part of the bonding process, that requires tapping gently for user confirmation thus registering the wearable band to a new end-user.
- If the information is the one sent before (e.g. the one used when bonding), the MiBand just authenticates the existing user and allows access to the advanced characteristics.

Once the authentication sequence is complete, the application can make use of the sensors by writing specific commands to the characteristics ("capabilities") used to control them. Information is eventually transferred by means of notifications therefore all processing must be done asynchronously.

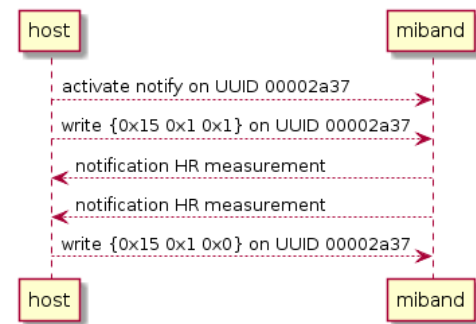


Fig. 7. MiBand heart-rate measurement sequence

B. Interacting with FlyTag

Integrating the FlyTag sensor in our demonstrator was much more straightforward, due to the fact that no special handling for pairing was required. Furthermore, the sensor itself allowed writing characteristics without the need to authenticate, hence

making it more easy to manipulate its behavior through the control characteristic.

Once again the first step is to identify the primary services and characteristics:

```
root@....# gatttool -b CF:98:46:6B:C4:16 --
primary -t random
uuid: 00001800-0000-1000-8000-00805f9b34fb
uuid: 00001801-0000-1000-8000-00805f9b34fb
uuid: 0000180a-0000-1000-8000-00805f9b34fb
uuid: 0000a000-ed6b-11e3-a1be-0002a5d5c51b
uuid: 0000180f-0000-1000-8000-00805f9b34fb
```

Listing 4. Exploring FlyTag primary services with gatttool

We can quickly observe that 0000a000 represents a custom service and after exploring the exposed characteristics we find out that the only characteristic 0000fff0 represents some kind of control point that allows both reading and writing operations. In fact, all the other characteristics are exposed through standard UUIDs which can easily be interpreted based on SIG recommendations:

- manufacturer (UUID 00002a29)
- model number (UUID 00002a24)
- hw version (UUID 00002a27)
- sw version (UUID 00002a26)
- battery level (UUID 00002a19)

The rendered results only return basic information while accessing sensor data such as temperature, luminosity or humidity requires a bit more work. In a similar approach to the MiBand heart-rate measurement the device reacts to commands that are sent from the host and returns the desired information. Reusing the same approach as before we automated the procedure through a helper script that renders the following output:

```
./bluetooth-firefly.sh
Device manufacturer: L-TEK
Device model: FF1502
Device serial nr: CF98466BC416
Device hw revision: FF000007
Device fw version: 1.0
Device sw version: 1.0.12
Device battery lvl: 36[%]
Temp: 20.32
Humidity: 39.55[%]
Luminosity: 80[Lux]
```

Listing 5. Firefly utility script

Although a quite expensive appliance, with apparently a simple API, we had to identify the commands from the manufacturer demo applications [15] in order to ensure correct data decoding and to manage the transfer of control information.

The device expects a binary encoded command comprising the following elements

- 1) the last three digits of the name identifier
- 2) the command sequence string 0001305

After writing the control sequence the device outputs a *snapshot* of the sensors as a sequence of bytes where the

temperature is encoded into bytes 21 and 22, *humidity* in bytes 23 and 24 and *luminosity* into bytes 25 and 26.

```
> ACL Data RX: Handle 32 flags 0x02 dlen 27
ATT: Read Response (0x0b) len 22
Value: 3439371f474747474747014000004140fda4ff2a
< ACL Data TX: Handle 32 flags 0x00 dlen 9
ATT: Read Blob Request (0x0c) len 4
Handle: 0x0018
Offset: 0x0016
> HCI Event: Number of Completed Pack.. (0x13) plen 5
Num handles: 1
Handle: 32
Count: 1
> ACL Data RX: Handle 32 flags 0x02 dlen 23
ATT: Read Blob Response (0x0d) len 18
64 f8 5a 96 00 50 78 78 82 78 78 78 78 78 78 78
78 78
```

Fig. 8. Flytag sensors snapshot trace

Because of this write-request, read-request model for sensor measurements, any application would have to periodically use this sequence to gather data and record it for later processing. This kind of on-demand interrogations allow the logic on the SoC to be very efficient in terms of speed and size thus enabling the device to maintain the battery life for a long period of time.

C. Programmatic access

At this point we have all the information that we need to proceed with the upper level of our integration solution: the programmatic access using Java. Our library of choice was tinyB [16] which exposes the GATT API of BlueZ in a much more friendly manner, making it easier for application developers to build actual use cases with these energy-efficient devices.

Since it is based on the D-Bus API exposed by the Bluetooth stack, communication is achieved through a series of JNI (Java Native Interface) calls and registered callbacks for device notifications. As a result, our demonstrator has a direct dependency on the Java binding and also on the shared object distributed with tinyB.

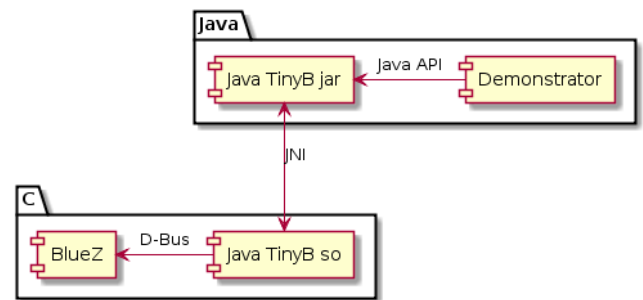


Fig. 9. TinyB setup for Java

One important aspect here is that all the binaries must be built from source since no pre-built binaries exist and as a consequence, setup requires adding this library to the class-path manually and also setting `java.library.path` jvm (Java Virtual Machine) variable to point to the JNI C shared object on the system.

After setting up the project properly the Java API requires life-cycle handling with `Asynchronous Executors` mainly because it relies on blocking calls and asynchronous callbacks especially in case of notifications:

```
(...)
BluetoothManager manager = BluetoothManager.
    getBluetoothManager();
manager.startDiscovery();
(...)
List<BluetoothDevice> devices = manager.
    getDevices();
BluetoothDevice miBand = devices.stream().
    filter(dev -> MI_BAND.equals(dev.
        getAddress())).findAny().orElse(null);
miBand.enableServiceDataNotifications(new
    MiBandServiceDataNotification());
if (miBand.connect()) {
    BluetoothGattService miBandService = miBand.
        find(MiBandConstants.MIBAND_SERVICE);
    BluetoothGattCharacteristic devInfo =
        miBandService.find(MiBandConstants.
            MIBAND_DEVICE_INFO_CHARACTERISTIC);
    byte[] result = devInfo.readValue();
}
(...)
manager.startDiscovery();
```

Listing 6. Accessing MiBand with tinyB

The work-flow used is described more accurately with the following steps:

- 1) The `BluetoothManager` is triggered to start discovery
- 2) After a period of time, we interrogate it and retrieve the discovered devices
- 3) We filter our device of interest based on the physical address
- 4) Next, we select a specific service and obtain handle to it
- 5) Using the service handle we obtain a characteristic handle
- 6) Finally we perform the read or write operation using byte arrays

IV. CONCLUSIONS AND FURTHER WORK

In this paper we have demonstrated an end to end integration of Bluetooth Low-Energy devices in a standard Linux distribution, thus opening a mobile-centric ecosystem to all application developers. We implemented a proof of concept and also evaluated the possibility to access in a programmatic fashion GATT services from high level languages on standard development hardware using open source software.

We hereby provided code snippets and critical configuration information, as well as Bluetooth traces, detailing interaction models based on commodity hardware, offered to any reader interested not only in concepts but also practical aspects. In fact, our future purpose is to use such technologies in combination with state-of-the-art machine learning techniques and develop advanced IoT services that bring together the two parts of the ecosystem : the devices and the Internet.

Data acquisition and preprocessing on a smart-home IoT gateway will leverage the benefits of fully powered computing nodes and most likely simplify the maintenance effort. The necessity of such a gateway is justified by the heterogeneity of the devices where a semantic integration, between multiple communication techniques, is needed [17]. While big data processing still remains a topic for cloud computing technologies, we believe security and privacy must be enforced on a lower level to ensure that personal information does not end up being directly controlled by a handful of corporations.

Finally, our contribution aims to bring closer the development environments (comprising tools, programming languages and performance) from the Linux ecosystem to the utility of more and more IoT devices.

ACKNOWLEDGMENT

Parts of the research presented in this paper were developed in joint effort between the authors and researchers from the EU AAL project "NOAH - NOT Alone at Home", number AAL-2015-2-115.

The authors would like to thank the GadgetBridge developers for their important contribution into identifying characteristics and behaviors of the MiBand bracelet in various scenarios. Without their initial effort our development process would have encountered an additional obstacle to pass.

REFERENCES

- [1] Ericsson, *Bluetooth inventor nominated for top European honor*, https://www.ericsson.com/news/120612_bluetooth_inventor_nominated_for_top_european_honor_244159019_c, 2012.
- [2] Bluetooth SIG, *Fact or Fiction*, <https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-fact-or-fiction>
- [3] Bluetooth SIG, *Generic Attributes and the Generic Attribute Profile*, <https://www.bluetooth.com/specifications/generic-attributes-overview>
- [4] Bluetooth SIG, *Bluetooth Low Energy Specification*, <https://www.bluetooth.com/specifications/bluetooth-core-specification>
- [5] K. Townsend, *Introduction to Bluetooth Low Energy*, <https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt>, 2015.
- [6] M. Ryan, *With Low Energy Comes Low Security*, 7th USENIX Workshop on Offensive Technologies, 2013, Washington D.C. <https://www.usenix.org/system/files/conference/woot13/woot13-ryan.pdf>
- [7] H. O'Sullivan, *Security Vulnerabilities of Bluetooth Low Energy*, Tufts University, 2015 <http://www.cs.tufts.edu/comp/116/archive/fall2015/hosullivan.pdf>
- [8] Project Ubertooth, *Ubertooth One HW Platform*, <https://github.com/greatscottgadgets/ubertooth/wiki/Ubertooth-One>
- [9] Gearbest, *The Xiaomi Band 1s*, http://www.gearbest.com/smart-wristband/pp_263275.html
- [10] M. Brewis, *Mi Band review*, TechAdvisor, 2015, <http://www.pcadvisor.co.uk/review/activity-trackers/xiaomi-mi-band-1s-pulse-best-value-activity-tracker-review-3630972/>
- [11] L-Tek Elektronika, *The FireFly FlyTag*, <https://firefly-iot.com/product/ff1502-sensor-ble/>
- [12] BlueZ Project, <http://www.bluez.org/profiles/>
- [13] D-Bus Message System Bus, <https://www.freedesktop.org/wiki/Software/dbus/>
- [14] GadgetBridge Project, <https://github.com/Freemove/gadgetbridge>
- [15] L-Tek Elektronika, *FireFly FlyTag Control Application*, <https://github.com/FireFly-IoT/FlyTag-FireFly-control>
- [16] Intel, *tinyB Project*, <https://github.com/intel-iot-devkit/tinyb>
- [17] F. Sandu, C. Costache, T. Balan, *Semantic data aggregation in heterogeneous learning environments*, IEEE 21st International Symposium for Design and Technology in Electronic Packaging (SIITME), Brasov, 2015, pp. 409-412.