JOHANNES KEPLER
UNIVERSITY LINZ

Stefan Kempinger
Institute of
Networks and Security

@ kempinger@ins.jku.at
https://www.digidow.eu/

April 2024

Implementing a Digidow-compatible Sensor for UWB Indoor Positioning

Technical Report

Christian Doppler Laboratory for
Private Digital Authentication in the Physical World

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World. We gratefully acknowledge financial support by the Austrian Federal Ministry of Labour and Economy, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, and Österreichische Staatsdruckerei GmbH.

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Contents

Abstract

The Digidow project aims to research solutions for privacy-preserving decentralized digital identity authentication in the real world. The project uses a Personal Identity Agent (PIA) to manage the user's identity and credentials, and a sensor to determine the user movement and intentions. These sensors register real world events and send the data to the PIA, which then processes the data and sends it to a verifier. There was an existing sensor implementation for a facial recognition sensor, a generic sensor library that can be used to implement other sensors, and somewhat working UWB anchors. This report describes the implementation of a UWB sensor that detects the door a user is standing in front of, and which integrates seamlessly with the exiting components.

1. Introduction

The Digidow project aims to research solutions for privacy-preserving decentralized digital identity authentication in the real world.

In the scope of this project multiple UWB anchors were acquired for research into indoor positioning. This report describes the implementation of a UWB sensor that is compatible with the Digidow system.

The sensor is responsible for detecting the door a user is standing in front of, and for sending the data to the PIA.

2. Background

2.1 Ultra Wideband

Ultra Wideband (UWB) is a wireless communication technology that uses a large portion of the radio spectrum. In the context of indoor positioning, UWB is used to measure the time of flight of a signal between two devices.

UWB is defined by the standard IEEE 802.15.4. The standard defines a set of channels, each with a specific frequency and always a 500MHz bandwidth per channel [4]. Since channel 9 (at 7987.2 MHz) is the most commonly supported channel in UWB devices, it is the channel we use for our UWB anchors [1].

2.1.1 UWB Anchors by MobileKnowledge

The UWB anchors used in this project are from MobileKnowledge and are based on the NXP UWB Trimension SR150 chip. The anchors are capable of measuring the time of flight of a signal between two devices, and can be used to determine the distance between the two devices. The accuracy of the anchors for distance is around $\pm$10 cm, which is sufficient for our use case, but the azimuth and elevation angles are too inaccurate with too strong fluctuations for our use case. The anchors are connected to a Nvidia Jetson Nano via USB, and this connection is used to power the anchors and to receive the UWB data from the anchors.

2.1.2 UWB Tags/Android App

Since the official UWB tags need specific hardware to flash a custom firmware, we decided to use an Android app to simulate the UWB tags. The app uses a modified version of an existing implementation of an out-of-band setup protocol, in this case using Bluetooth as medium, and the *androidx.core.uwb* [5] library to do the ranging with the anchors, which both were provided by MobileKnowledge in their *MK UWB Kit Mobile edition 2.0* [2].

2.2 W3C Verifiable Credentials

The W3C Verifiable Credentials standard is a standard for creating and exchanging credentials in a privacy-preserving way [8].

A Verifiable Credential is a tamper-evident credential that has a set of claims about a subject, which are signed by the issuer. The credential can then be presented to a verifier, who can verify the signature and the claims.

In our case, the PIA sends a registration request to the sensor, which contains a Verifiable Credential with the necessary data to identify the user. If the sensor gets a reading that matches the data in the registration credential, it sends whatever data it has about that user back to the PIA in a Verifiable Presentation, which might contain multiple Verifiable Credentials.

2.3 TOR

The Onion Router (TOR) is a privacy network that allows for anonymous communication over the internet [6].

TOR onion services, formerly known as hidden services, are services that can only be accessed through the TOR network. They are used to provide a secure and private way to communicate with a service, as the service is only accessible through the TOR network and the traffic is end-to-end encrypted. In practice, this means the server providing the service maps an onion address to a specific local network socket, and the client connects to the onion address using their own TOR SOCKS proxy, which then forwards the traffic to the server.

In our case, both the sensor and the PIA use TOR onion services to communicate with each other. The sensor uses an onion service to receive registration requests from the PIA, and the PIA uses an onion service as a callback to receive data from the sensor.

2.4 Digidow

Digidow is a project that aims to research solutions for privacy-preserving decentralized digital identity authentication in the real world [3].

In the scope of this implementation the Digidow system uses a Personal Identity Agent (PIA) to manage the user's identity and credentials, and sensors to determine the users movement and intentions. These two components communicate with each other using a custom protocol, which is based on the W3C Verifiable Credentials standard.

There is an existing sensor implementation for a facial recognition sensor, and a generic sensor library that can be used to implement other sensors.

3. Implementation

3.1 Hardware Setup

The hardware setup consists of a Jetson Nano and two MobileKnowledge UWB anchors, both utilizing NXP's UWB Trimension SR150 chip, as seen in Figure 1.

The Jetson Nano is connected to the UWB anchors via USB.

The important part of the hardware setup is the anchor placement, as the anchors need to be sufficiently far apart to allow for accurate positioning (Figure 3). The calculation of the necessary distance is done by adding up the maximum error of the anchors, in our case 20 cm each, and then multiplying it by

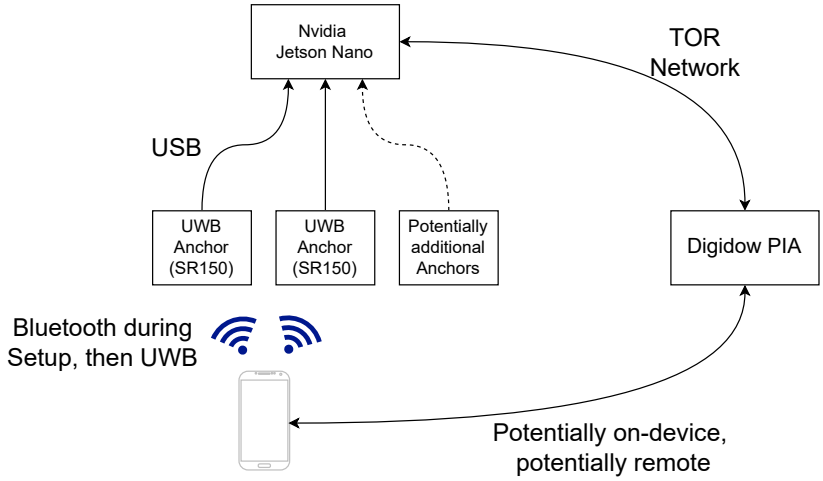

Figure 1: Setup

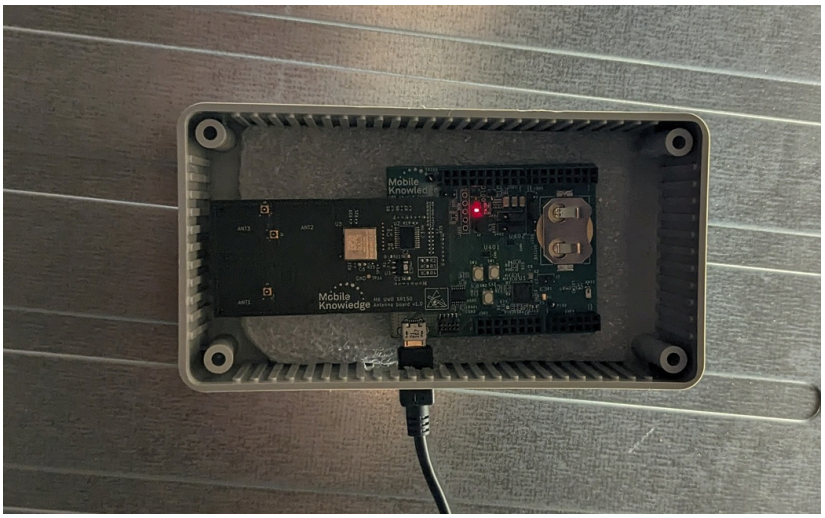

Figure 2: Anchor installation

a factor of 2 for eventualities. This results in a minimum distance of 80 cm between the two anchors. This calculation comes down to the fact that we have to calculate the error as the volume of the hull of two intersecting spheres. The thickness of the hull is the maximum error of the anchors, and the distance between each anchor and the UWB tag is the radius of each sphere. The volume should be as small as possible to minimize the error, so the distance between the anchors should be as large as possible.

Note: At some point the anchors will always generate almost-fully overlapping hulls, but since we assume the ranging is only viable up to 50 m, we can somewhat safely ignore this case.

Note: A moving 3D model to represent the hull of the spheres intersecting would be a good addition to this report, but movement is classically hard to print on paper.

Each anchor is mounted in its own plastic case, which is magnetically held to the metal cable tray on the ceiling, as seen in Figure 2.

The anchors are placed in the hallway of the Institute of Networks and Security at the Johannes Kepler University Linz, as seen in Figure 3.

Figure 3: Anchor placement

3.2 Communication Protocols

There are three communication protocols used in this setup:

- UWB tags $\leftrightarrow$ UWB anchors
- UWB anchors $\rightarrow$ Jetson Nano
- Jetson Nano $\leftrightarrow$ PIA

3.2.1 Between UWB tags and UWB anchors

The UWB tags set up their UWB connection to the UWB anchors via Bluetooth. This is done through an out-of-band protocol provided by MobileKnowledge that mimics the *NINearbyAccessoryConfiguration* from Apple, as the implementation on Apple devices does not require any custom code. It creates a standard GATT connection using the Nordic UART Service protocol [7] from the mobile phone to the UWB Anchor, wherein a 50 byte payload is sent to the anchor, which then uses the payload to set up the UWB connection. In its original state, this 50 byte payload only contained the UWB parameters, but we modified it to also contain the onion address public key of the users PIA. This is done to identify the user in the further process. More information on this communication can be found in Apple's developer resources. After this setup, plain UWB ranging is used to get the distance between the UWB tags and the UWB anchors.

3.2.2 From UWB anchors to Jetson Nano

The UWB anchors communicate with the Jetson Nano via USB. The UWB data is sent to the Jetson Nano via a serial connection. On this connection, there is debugging information from the anchors, as well as unformatted UWB metadata, and certain lines injected by us to get parsable data with the anchor's onion address. It is a simple one-way communication, and all of the relevant lines start with the string "[Parse This]" and are parsable with the following regex:

```
\[(([^\[]+?)\s*:\s*(.*?))\]
```

The logs are formatted according to the following format string:

```
[Parse This] [MAC: %02X:%02X] [Session: %u] @ [Distance: %d cm], [Azimuth: %d],
[Elevation: %d], [Line of Sight: %d] [Onion: %s]\n
```

The regex is split into multiple capture groups, which are then used to extract the data. The interesting captures are 1 and 2, which contain the key and value of the data.

3.2.3 Between Jetson Nano and PIA

The Jetson Nano uses a TOR onion service to receive registration requests from the PIA, and the PIA uses a TOR onion service as a callback to receive data from the sensor. This callback address is sent to the sensor in the Bluetooth connection. This communication uses W3C Verifiable Credentials to send data, and is based on the Digidow protocol.

3.3 Software Setup

The software setup consists of native C code for the anchor firmware and a Rust program for the Jetson Nano.

The first iteration of the sensor was written in Python. It was later decided to rewrite the sensor in Rust, as it is more performant and easier to integrate with the existing codebases.

3.3.1 Anchor Firmware

The anchor firmware is based on the MobileKnowledge UWB Anchor firmware and was modified to send formatted UWB data to the Jetson Nano via USB. Modifications include the hijacking of the Bluetooth pairing process to send a byte sequence containing an onion address public key from the connecting device to the anchor, which is then used to identify the device in the further process. The anchor logs all UWB events to the serial console, and if the event is a ranging event, it is formatted properly to be distinguishable from other events. This formatted logging is also a modification to the original firmware, as the original firmware already logs the UWB events, but was missing information to identify the producer of the event.

3.3.2 UWB Sensor

The UWB sensor is a Rust program that runs on the Jetson Nano and is responsible for receiving the UWB data from the anchors and sending it to the responsible Digidow PIA.

The program is split into multiple units, each responsible for a specific task.

Anchor Communication To read the UWB data from the anchors, the program uses the *serialport* crate.

Since the anchors are connected via USB, and we do not know the right port(s) beforehand, the function *find_fitting_serial_ports* iterates over all available ports and tries to connect to them with specific serial port settings. If the connection does not fail, the function stores the port as potential anchor port and at the end of the iteration returns all ports that did not fail to connect. This way closed or busy ports are filtered out. In a future version, the function will also check if the anchor is actually an anchor via the USB metadata, and not just any device that is connected via USB.

The function *open_and_listen_on_port* is then used to read the serial communication from the anchor and listen for incoming data. It is called for each port returned by *find_fitting_serial_ports* and promptly spawns a new thread for each port. The function then reads the data from the port and filters out all lines that do not contain the string "[Parse This]". The remaining lines are then parsed using a regex (??) and the resulting data is stored in a HashMap. Also present in the HashMap is the port the data was received on, which is used to identify the anchor the data came from, and the timestamp of the received data. Furthermore, the onion address public key, which was sent during the Bluetooth pairing process, is converted to a proper onion address and also stored in the HashMap.

The HashMap is then sent to the *ranging_event_handling* module, which processes the data further.

UWB Data Processing The processing of the UWB data is done in the *ranging_event_handling.rs* file.

The processing currently contains the following steps:

- **Buffering**: The data is stored in a queue, which is *UWB_RANGING_BUFFER_SIZE*, currently 4 elements, long.
  - If the queue is uninitialized, it is initialized with the current reading.
  - If there is at least one reading in the queue, the new reading is averaged with the last entry in the queue and the result is added to the back of the queue. This is done to smooth out the readings and reduce the noise. This operation has never shown to cause any issues, it just slows down the reaction time of the sensor.
  - If the queue is full, the oldest reading is removed from the front of the queue and the new reading is added to the back.
- **Door Detection**: Now the sensor checks if the user is standing in front of a door.
  - The configuration file contains the distances from the UWB anchors to the doors in the hallway.
  - A simple range check is done to see if a user is within the upper and lower range bounds of the door.
  - If a matching door is found that fulfills the range check for all sensors, the door name is stored.
- **Movement Speed Calculation**: The movement speed is calculated by comparing the current reading with the previous readings.
  - Currently, we use the latest *UWB_RANGING_BUFFER_SIZE* - 1 readings, if that many are available.

- We use as little readings as possible to get a good reaction time, as many as possible to get a good average.

The movement speed is calculated as follows:

1. We subtract each queue entry from the previous entry.
2. We sum up all the differences for a single sensor and divide by the number of readings.
3. We sum up all the absolute differences for all sensors and divide by the number of sensors.

- **Is Moving In Front Of Door Detection**: The movement speed is then compared to a threshold, which is currently set to 0.5 m/reading.
  - If the movement speed is above the threshold, the sensor assumes that the user is moving in the hallway, so only the current location data with the movement vector is sent to the PIA.
  - If the movement speed is below the threshold, the sensor assumes that the user is standing. If the user is standing in front of a door, the door name is sent as an extra credential to the PIA.

    Note: There is an added check to see if the user is standing in front of a door for a longer time, in our case 5 seconds, to prevent the sensor from sending the door name multiple times. This means a new door name is sent at most once every 5 seconds. Since we cannot expect the system to always be able to immediately recognize when a user is moving away from a door, we cannot just send an indication that the user is not standing in front of a door anymore.

Digidow Sensor Library The interaction with the Digidow PIA is done via the *digidow_sensor_library* module. Using that library, the sensor doesn't have to worry about the specifics of the Digidow protocol, but can instead use the provided functions to send the UWB data to the PIA.

The only needed implementations are in the *digidow_sensor_code.rs* file, which defines the Verifiable Credential that will be sent, and the registration object that is sent from the PIA and matched against the embedding of the user, in our case just the onion address.

The *UwbSensorRegistration* struct looks like the following:

```
#[derive(Clone, Deserialize, Serialize, Debug, PartialEq)]
pub struct UwbSensorRegistration {
pub identity: String,
}
```

This registration struct can be this simple because the sensor only needs to match the onion address of the user to the onion address in the registration object, and all member variables of the struct (so just the String identity) implement the required traits for the required functionality.

There is another struct, *UwbSensorDataPush*, which contains the data that is sent to the PIA. It has to implement a trait that allows it to be implicitly converted to a *Vec<VerifiableCredential>*, which is packed up by the *digidow_sensor_library* into a neat *VerifiablePresentation*, signed and sent to the PIA. In the case of the UWB sensor, the data is converted to a VerifiableCredential containing the UWB data and movement vector, and optionally an additional VerifiableCredential containing the door name a person is in front of.

The credentials contain the following data:

uwb-tracking-data:

- **anchor_distances**: A BTreeMap containing the latest distances from the anchors to the user.
- **anchor_az_angles**: A BTreeMap containing the latest azimuth angles from the anchors to the user.
- **anchor_el_angles**: A BTreeMap containing the latest elevation angles from the anchors to the user.
- **anchor_mac_address**: The MAC address of the latest anchor that updated data.
- **movement_vector**: The movement vector of the user.
- **datetime**: The timestamp of the data.
- **proposed_verifiers**: The onion addresses of the proposed verifiers.
- **identity**: The onion address of the user.

uwb-door-detection-data (only sent if the user is in front of a door):

- **has_stopped_in_front_of_door**: A boolean that is true if the user has stopped in front of a door.
- **identified_door**: The door name the user has stopped in front of.
- **movement_vector**: The movement vector of the user.
- **datetime**: The timestamp of the data.
- **proposed_verifiers**: The onion addresses of the proposed verifiers.
- **identity**: The onion address of the user.

Settings The settings module is responsible for loading the configuration from the config file and providing it to the other modules.

It reads the config file and parses it into a struct, with missing values being replaced by default values. The struct is then returned as a variable, which is used by the other modules to access the configuration.

The config file is a JSON file and contains the following values:

- **ia_public_keys**: The public key(s) of the Issuing Authority, used to verify the signature of the data sent by the PIA.
- **door_distances**: A HashMap containing the distances from the UWB anchors to the doors in the building.
- **bbs_secret_key**: [Optional] The secret key of the BBS+ signature scheme, used to sign the data sent by the sensor. If not present the sensor will create a new keypair and store it in the config file.
- **bbs_public_key**: [Optional] The public key of the BBS+ signature scheme, used to verify the signature of the data sent by the sensor. If not present the sensor will create a new keypair and store it in the config file.
- **onion_key**: [Optional] The onion address private key, used to create the onion address of the sensor. If not present the sensor will create a new keypair using the *zwuevi* crate and store it in the config file.

- **tor_control_port**: [Optional] The port the Tor control port is listening on. If not present the sensor will use the default port 9051.
- **proposed_verifiers**: [Optional] A list of onion addresses of the proposed verifiers. If not present the sensor will not propose any verifiers.
- **serial_port_settings**: [Optional] The serial port settings used to communicate with the anchors. If not present the sensor will use the default settings. The default settings are: 115200 baud, 8 data bits, 1 stop bit, no parity, 1 second timeout.

Main Function The main function ties all the other functions together.

It initially loads the configuration from the config file. Then it calls *find_fitting_serial_ports* to find the ports anchors might be connected to. If there are no ports, the program will exit with an error, as it cannot function without anchors. If there are ports, the program will continue running by constructing an object *Sensor* of the sensor library using the provided *SensorBuilder*. The *SensorBuilder* takes several arguments, such as the trusted issuing authority public keys, the onion service private key and the BBS+ keypair. We then wrap the *Sensor* object in an *Arc<Mutex<Sensor»* to allow for concurrent access to the sensor object. This is necessary because the *Sensor* object is accessed by multiple threads, specifically one thread for each used USB port (each anchor). These threads are then spawned and the main program enters a sleeping state, where it *awaits* all threads to finish, which technically should never happen, as the threads of *open_and_listen_on_port* are infinite loops.

4. Conclusion

The UWB sensor is now fully functional and can be used to send UWB data to the Digidow PIA.

The implementation of the sensor library allows for easy integration of the sensor into the Digidow system by abstracting away the specifics of the Digidow protocol. The only exposed parts are for initializing the sensor, the structs that contain the relevant data, and the function that does the matching and sending the data to the PIA.

This allows the implementer to focus on the specifics of the sensor and not worry about the specifics of the Digidow system. In the case of the UWB sensor, this meant dealing with the receival of UWB data from the anchors and processing it in exactly the way we needed it.

References

[1] FiRa Consortium, Inc. 2023. How UWB Works. Retrieved 02/08/2024 from https://www.firaconsortium.org/discover/how-uwb-works.

[2] MobileKnowledge. 2024. MK UWB Kit Mobile edition 2.0. Retrieved 02/08/2024 from https://www.themobileknowledge.com/product/mk-uwb-kit-mobile-edition-2-0/.

[3] Institute of Networks and Security. 2024. Private Digital Authentication in the Physical World. Retrieved 02/08/2024 from https://digidow.eu/.

[4] Hans-Juergen Pirch and Frank Leong. 2020. Introduction to Impulse Radio UWB Seamless Access Systems. White Paper. FiRa Consortium. https://www.firaconsortium.org/sites/default/files/2020-10/introduction-to-impulse-radio-uwb-seamless-access-systems-102820.pdf.

[5] Android Open Source Project. 2024. androidx.core.uwb. Retrieved 02/08/2024 from https://developer.android.com/reference/kotlin/androidx/core/uwb/package-summary.

[6] The Tor Project. 2024. Browse Privately. Explore Freely. Retrieved 02/08/2024 from https://www.torproject.org/.

[7] Nordic Semiconductor. 2024. UART/Serial Port Emulation over BLE. Retrieved 04/17/2024 from https://infocenter.nordicsemi.com/topic/com.nordic.infocenter.sdk5.v14.0.0/ble_sdk_app_nus_eval.html.

[8] W3C. 2022. Verifiable Credentials Data Model 1.1. Retrieved 02/08/2024 from https://www.w3.org/TR/vc-data-model/.