Secureloc Documentation

1. Introduction

SecureLoc is an open security testbed for 802.15.4 Impulse Radio Ultra-wideband (IR-UWB) indoor positioning.
IR-UWB is currently the leading radio solution for indoor positioning. Global Navigation Satellite Systems (GNSS) have very limited performances in indoor environments: the Line-of-Sight (LoS) with the satellites is obstructed inside buildings, and the presence of walls and furniture create multipath and fading effects, which together induce significant accuracy degradations. As a consequence, multiple positioning solutions based on popular radio standards, including Wireless Fidelity (WiFi) or Bluetooth, have been developed over the last two decades. Radio Indoor Positioning Systems (IPS) estimate the position from a triangulation or multilateration process.
In a triangulation process, the Angle of Arrival (AoA) between a mobile radio transceiver and a set of reference stations placed in the building is estimated based on the signal phase. The position is then extracted as the intersection of the beams between the radio transceiver and each reference station.
In a multilateration process, the distance between the mobile radio transceiver and the reference stations estimated, which is a process known as ranging. In that case, different methods allow calculating geometrically the transceiver's position. Typically, distances are extracted from either signal attenuation or time-of-flight.
Most radio solutions can reach meter-level accuracy with either method. However, IR-UWB is the only technology that supports time-of-flight ranging. Indeed, electromagnetic signals travel at the speed-of-light in the air. To give an order of magnitude, that means that a radio signal travels 1 meter in about 3 nanoseconds. The modulation techniques and bandwidth used on popular radio standards (e.g., 802.11, Bluetooth, Zigbee, etc.) do not allow to reach such granularity on time-of-flight estimations. Yet, IR-UWB is based on narrow pulses (2 ns-width), featuring a bandwidth of at least 500 MHz, which allow reaching an accuracy of about 10 cm. Considering that, IR-UWB largely outperforms other radio technologies today when it comes to positioning, and is commonly found in exigent industrial applications. A few examples are inventory tracking in warehouses, drone indoor navigation, or manufacturing chain supervision in factories.

IR-UWB is also known for its inherent security. Indeed, radio positioning solutions based on signal attenuation or angle-of-arrival are vulnerable to relay and replay attacks, which are common attacks against proximity-based technology. In a replay attack, an attacker Eve impersonates a device Alice by repeating one of A's previous messages to another device, Bob. The goal of Eve is to fool Alice to into thinking she is Bob. Relay attacks are based on the same principle, except that in that case Alice and Bob are out of range of each other (hence Eve, is *relaying* the messages between Alice and Bob, without them being aware), which means that an illicit link between them is created. A common example is the proximity verification with car keys. Several car thefts mounted with relay attacks have been reported: usually, the thief relays the signal of a car key that is inside a house to the car parked outside. Since these attacks do not require from the relay to decipher or modify the messages exchanged, cryptographic solutions cannot do anything against them.
Regarding that issue, one of the biggest assets provided by time-of-flight ranging is the capability to detect frauds based on replay or relay attacks. Since the attacker should not have the capacity to make a signal travel faster than the speed-of-light, the delay induced by the replaying or relaying process will be clearly noticeable as the time-of-flight will be absurdly high. This property has promoted IR-UWB to one of the most secure proximity-based technology, and has led in 2019 to the development of secure UWB car keys by Volkswagen and NXP [1]. In parallel, smartphone manufacturers have demonstrated an increasing interest in IR-UWB over the first months of 2020, especially with the lack of proper social distancing solutions for Covid-19 on smartphones, Bluetooth having limited ranging performances. The iPhone 11 Pro, released by Apple in September 2019, was the first smartphone integrating an UWB chip. It is now expected that the next generation of smartphones will integrate UWB chips by the next two or three years.

Despite being praised for its security, IR-UWB is not vulnerability free. Several security flaws have been identified in IEEE 802.15.4, including notably unsecure acknowledgments, which can lead to distance tampering attacks against ranging protocols [2]. It has been also shown that the physical layer defined in 802.15.4-2011 is potentially vulnerable to Early-Detection/Late-commit (ED-LC, [3]) attacks, which are distance decreasing attacks based on a "relay backward in time" mechanism. Also, none of the security mechanisms defined in 802.15.4 prevents a node from lying on its position by cheating in the ranging protocols, which is also known as *internal attack* [4]. Yet, off-the-shelf IR-UWB IPS

solutions on the market are not open-source, and the access to the multiple layers involved in the positioning process is limited. As a consequence, commercial IR-UWB positioning solutions are a limited tool for security research, which typically needs a complete access to the layers.
Considering that issue, the platform SecureLoc has been developed. SecureLoc is an open, testbed for security research on 802.15.4 IR-UWB. This testbed features notably:

- A complete open-source implementation of an IR-UWB IPS, including several state-of-the art ranging protocols and multilateration algorithms.
- Real-time graphical rendering; log and replay functionalities, which enable easy benchmarking and data collect for machine learning; dedicated calibration modes for performances evaluation
- Fast deployment functionalities, allowing to deploy and test security scenarios within a few minutes
- Implementations of several attacks based on spoofed acknowledgment or internal attacks, for security testing
- Implementations of several countermeasures against the aforementioned attacks
- Cyber-physical simulation features, in which real data can be mixed with simulated data

This documentation provides all the information needed to run SecureLoc project. The software documentations for the Python (localization engine) and C++ code (DecaWino firmware) is provided independently.
We first describe the hardware and software architecture of SecureLoc. Then, an installation guide is provided. This is followed by a user guide regarding the localization engine, control menu and deployment tool usage. Finally, a description of the firmware available (for the different attacks and countermeasures implemented) is given.

For further academic references, the following papers introduce different results obtained on SecureLoc:

- Regarding the platform architecture, ranging protocols and localization algorithm performances, refer to [2, 5]
- Regarding attacks based on the acknowledgment vulnerability, refer to [2]
- Regarding physical authentication on IR-UWB devices, refer to [6]

Outline

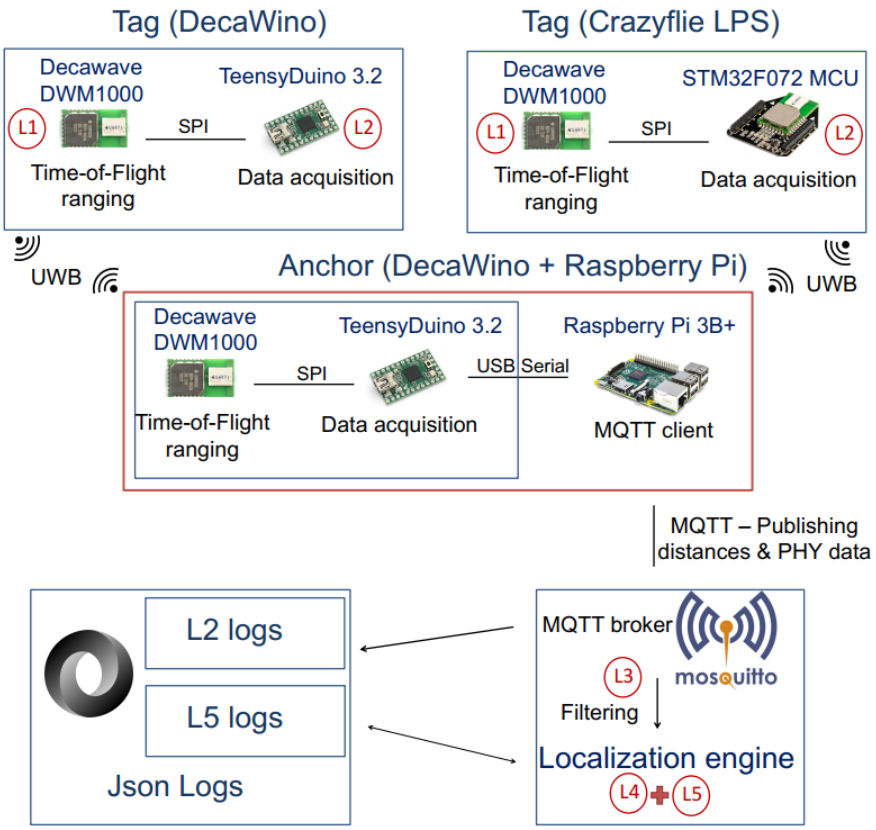
2. Hardware Description

Platform architecture

SecureLoc's hardware is based on DecaWino chips, which have been originally developed by IRIT Toulouse as part of the OpenWino project [7, 8]. DecaWino chips are based on the IR-UWB transceiver DWM1000, manufactured by Decawave [9]. The DWM1000 transceivers implements the physical layer defined in IEEE 802.15.4-2011 [10]. They are meant to be used a slave devices and require a host microcontroller. On DecaWino chips, the DWM1000 is driven by a Teensyduino 3.2, which implements the Medium Access Control (MAC) protocols defined in the standard.

Indoor Positioning Systems are usually built around two types of nodes: fixed reference stations, denoted as *anchors*, and mobile transceivers, denoted as *tags*. Tags are mounted on the objects of interest tracked by the IPS (e.g., drones, forklifts, factory workers…).

Figure 1- SecureLoc hardware architecture



On SecureLoc, tag simply consists of a DecaWino tag powered by a battery. On the other hand, anchors consist of a DecaWino chip wired to a Raspberry (RPI). The RPI and DecaWino communicate through a USB-serial link; RPIs are connected to a local network, on which they transmit the data collected by the UWB transceivers with MQTT protocol [11]. MQTT is an efficient and lightweight messaging protocol for sensors networks, in which data can be organized in different topic; the devices on the network can subscribe to their topics of interest and receive all the data published on these topics.
SecureLoc architecture is organized in 5 layers, shown in Figure 1. A description of each layer is provided below in Table 1.

Table 1- SecureLoc Layers

L1	*Physical layer*	Impulse radio transceiver, high-resolution clock
L2	*MAC layer*	Ranging protocols to estimate the distance between two nodes.
L3	*Ranging filtering*	Sliding Window filter; dynamic correction
L4	*Multilateration*	Localization algorithms: Gauss-Newton, Weighted Centroid…
L5	*Position filtering*	Saturation filter

Regarding the physical layer, considering that it is entirely implemented in the DWM1000 transceiver, we recommend referring to DWM1000's user manual for technical details [12]. The DWM1000 transceiver is piloted through the DecaDuino library, which has been developed specifically for DecaWino, and allows exploiting effortlessly DWM1000's functionalities. DecaDuino is an open source library also developed at IRIT Toulouse, and can be found at [13][1].

[1] Note that SecureLoc source code embeds a slightly modified version of Decaduino. Thus, installing Decaduino is no required to run this project.

3. Software

Teensyduino

Development with Arduino IDE: Note: the .ino files in this project are deprecated. The platform has been switched to a custom compilation tool documented in the later sections, based on cpp files. Nevertheless, if you wish to use the ino files:

- Download Arduino IDE: https://www.arduino.cc/en/main/software
- Download the teensyduino utilities: https://www.pjrc.com/teensy/td_download.html
- Copy/Paste the following files in your Arduino libraries directory:
  - SecureLoc/DecaWino/Deployment/teensy3/src/cpp/Decaduino.cpp
  - SecureLoc/DecaWino/Deployment/teensy3/src/header/Decaduino.h

If you do not use Arduino IDE, you do not need to install external libraries to compile the DecaWino projects. Use whatever IDE is convenient for text editing. See section 'Compialtion' for further information on how to compile and deploy DecaWino projects.

Raspberry

Server/Broker

The project is based on MQTT protocol. A broker (*server)* should be run on a host machine to allow communications.
MQTT Broker and 3D engine can be run on the same machine if the CPU can handle it.
Refer to MQTT section for detailed information.

3D engine

Download Panda 3D with embedded Python 3.6. Note that regular versions of Panda 3D use Python 2.7, which will not be able to run the codes since it includes Python 3 libraries. Panda 3D with embedded python 3.6 can be found there: https://www.panda3d.org/download.php?platform=windows&version=devel&sdk
Anchors positions are set in anchors.tab. The multilateration function should be set in the localize method of the World class in World.py.
Important: use the embedded Panda 3D python interpreter rather than interpreters found in regular python installation. On Windows, most convenient way is to add the path to Panda 3D python interpreter before any instance of other interpreters (*typically C:\Panda3D-1.10.0-x64\python*). Add also C:\Panda3D-1.10.0-x64\python\scripts to get pip in your path.

MQTT

Nodes data are dumped through MQTT protocol .
MQTT client: paho (*pip install paho-mqtt* on the Raspberry Pi & server)
MQTT broker: mosquitto (download here https://mosquitto.org/download/). To run (on a Windows laptop): go to Program Files/mosquito directory and run mosquito.exe. *Note: this broker is relatively resources-consuming so you may want to shut it off when not in use to avoid overheating.*

4. Installation

Proceed first to clone the github if it is not already done. SecureLoc can be found at https://github.com/Hedwyn/SecureLoc.

```
> git clone https://github.com/Hedwyn/SecureLoc
```

First, you should to try running the localization engine. The Python 3 code is contained in a package called *ips*. The *ips* directory can be found at the root of the repository. You need a Python 3 interpreter to run the localization engine, preferably Python 3.6 [1] . You can download Python 3.6 at this URL: https://www.python.org/downloads/release/python-360/
Do not forget to add the python installation directory to your path, otherwise, you will not be able to call the interpreter from a terminal with the *python* command.

It is usually recommended to work with virtual environments in Python projects. In a virtual environment, the python interpreter and required libraries are defined only locally: as consequence, it is possible to use different version of Python (and different versions of a given package) across different projects on the same machine if each project is installed in a virtual environment. To create a virtual environment *SecureLoc*, run:

```
> python –m venv SecureLoc
```

The command will take a short while to complete. After completion, a directory *SecureLoc* should be created. Go to that directory: it should contain several sub-directories, including one called Scripts and run:

```
> activate
(SecureLoc)>
```

After running *activate* you should see the directory name displayed in parenthesis, which means that the virtual environment is enabled. You can call *deactivate* at any time to disable the virtual environment. Note that you do not need to call it from the *Scripts* directory, the *deactivate* command will work anywhere.

[1] Intepreters above 3.6 have not been tested, but it should run fine.

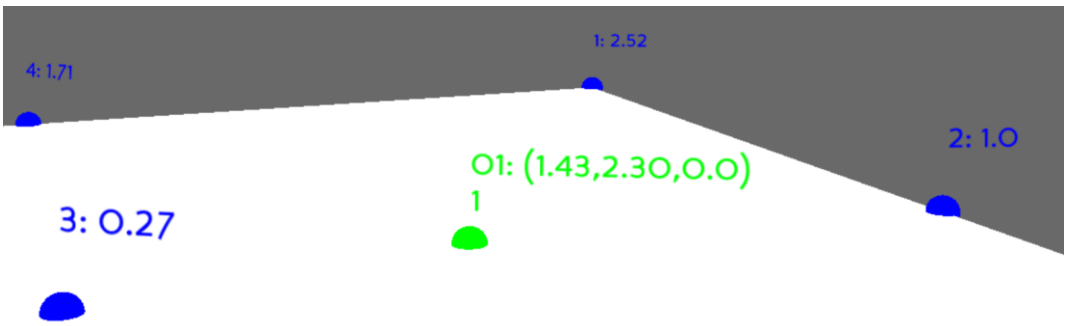
Now that you have a working Python interpreter and a virtual environment, you can proceed to install SecureLoc dependencies in your virtual environment. Do not forget to activate the virtual environment before starting that step. At the root of the repository, run:

```
> pip install -r ips/requirements.txt
```

The file *requirements.txt* contains all the dependencies required for the package *ips*: pip will parse this file and proceed to install one by one all the required libraries for you.
At this point, you should have a working installation. By default, the localization engine is configured to the *playback* mode. This mode allows replaying measurement logs, and can display recorded trajectories. SecureLoc contains already some example logs; if you run the localization engine, you should see the replay of one of these examples, by calling:

```
> python run.py
```

The 3D engine should now start:



You can leave the replay by pressing 'q'. The blue points represent the *anchors,* and the green one the *mobile tags* (there was only on in the example above). The distance measured by the anchors is shown at the right of their ID (=anchor number), while the 3D coordinates are shown for the mobile tags.
The provided example has not much interest and is enough to check that your installation is working. You should see the mobile tag (green point) moving if everything is working correctly.

At that point, if you do not have the required hardware at your disposal (defined in the Hardware section), you can only run the localization engine in *playback* mode You can check the playback section for instructions on how to select which log you want to replay. You can also check the localization engine user manual to learn about changing the calculation parameters (e.g., localization algorithm, filters…) used for the replay.

If you do have the required hardware, the next section provides further instructions on how to use the platform.

5. User Manual - running the platform in real-time

We give step-by-step instructions to properly use the platform in this section. There are four main steps to set up everything:

1. **Raspberry Pi (RPI) installation**: if you are using mounting a new SecureLoc-based platform, you need to install the required software on your RPIs. If you have access to an already existing platform, skip this step.
2. **Network configuration**: your machine and the platform's RPIs should be on a local network. You should first set up physically your local network, then, proceed to configure the proper IP addresses for your RPIs and main machine.
3. **Deploying Decawino's firmware:** SecureLoc features a deployment mode which allows deploying easily the firmware without having to flash individually each node. A graphical user interface (GUI) let you do that in a few clicks.
4. **Running the localization engine in real-time**: one everything is set up, you should be able to start the localization engine with a few commands. You will then be able to observe the mobile tag's motion in real-time.

These four steps are detailed below in the following sub-sections.

Step 1: Raspberry Pi installation

You should install the lastest version of Raspbian Operating System. Instructions can be found here:
https://www.raspberrypi.org/documentation/installation/installing-images/
When using the deployment feature available on SecureLoc, the RPIs are prompted via ssh to flash the Teensyduinos. To allow a RPI to flash a Teensyduino, it is required to add a rules file in */etc/udev/rules.d/49-teensy.rules* . Follow the steps described at this URL:
https://www.pjrc.com/teensy/loader_cli.html
The last thing you need to do is to copy the client code on the RPIs. The client code *clientMQTT.py* can be found in the *Raspberry* directory at the root of the repository. You can place it in any directory, we simply recommend being consistent and using the same emplacement on each RPI to later facilitate the

management of multiple RPIs at the same time. It is better to default to the Desktop directory.
Raspbian is typically shipped with a Python 3 interpreter. Any Python 3 interpreter should run the client code fine. If you notice any problem, you can try to install a Python 3.6 interpreter as detailed in the previous section.
You will need to install the serial and MQTT library to run the client code. Connect the RPIs to the internet and run:

```
> pip install serial paho-mqtt
```

At that point, the RPIs are ready for use. To run the localization engine, you will probably need a ssh session with each RPI, to run the client code and also for debug purposes. You can use any ssh software, but we recommend the free software MobXTerm, which features a useful multi execution mode:
https://mobaxterm.mobatek.net/
In the next section, we discuss the network configuration. You should be able to create a ssh session with all the RPIs in MobaXterm after completing the following steps.

Step 2: Network configuration

Network layout

You will probably need a switch for your local network. The local network consists of the RPis plus the machine on which you did the installation. Plug an ethernet cable between each RPI and your switch; do the same on your machine. You do **not** need an internet access, just keep the network local.
By default, all the devices on the local network should take an IP address starting with *169.254*. Setting static IP addresses on your machine and the RPIs is recommended to avoid future troubles. On RPIs, this can be configured in /etc/network/interfaces. Check the following resources if you do not know how to proceed:
https://www.raspberrypi.org/documentation/configuration/tcpip/
On a windows machine, check the instructions provided at this URL:
https://pureinfotech.com/set-static-ip-address-windows-10/

Do not forget to notate down the IP addresses of the RPIs. You should now be able to connect the the RPIs through SSH. Create one ssh session for each RPI on MobaXterm (or you preferred ssh software). Check that you can successfully connect to each of your RPIs.

MQTT broker

The data are exchanged between the RPIs and your machine through MQTT protocol, which has been presented in the introduction. You need to run a MQTT broker on your machine, which is basically the server for the TCP/IP links.

You can now proceed to configure the proper IP addresses in the localization engine.
We recommend installing mosquito, a popular MQTT broker:
https://mosquitto.org/download/

Client side

In the client code for the RPIs *clientMQTT.py*, you should replace the value of the parameter HOST by the IP address of your machine.

Server side

Second, you need to provide the IP addresses of your RPIs in the configuration file used by the Deployment tool. From the root of the repository, go the deployment configuration directory:

```
> cd Decawino\Deployment\Config
```

Then, edit the configuration file *config.txt* in this directory. This configuration file contains the following entries:

```
169.254.108.111 pi raspberry A
169.254.35.79   pi raspberry B
169.254.73.145  pi raspberry C
```

Each line contains the details of one RPI. A line is organized as:
{IP_address} {hostname] {password} {anchor_ID}

Each field should be separated by one or multiple spaces. Depending on how many Decawino nodes and RPIs you have at your disposal, two different configuration are possible.
If you have enough RPIs, having one Decawino plugged to each RPI is the most convenient solution. This the case in the example configuration above.

If you have more Decawinos than RPIs, then we will need to plug multiple Decawino nodes to some of your RPIs. In that case, your configuration file should look like this:

```
169.254.108.111 pi raspberry A
169.254.35.79   pi raspberry B
169.254.35.79   pi raspberry F
169.254.73.145  pi raspberry C
169.254.142.238 pi raspberry D
```

In the example, you can see that line 2 and 3 correspond to the same RPI (same IP address). That means that two Decawino nodes are plugged to that RPI; they will take the anchor IDs B and C. The same thing applies for anchors D and E. Note that for RPIs plugged to two Decawino nodes or more, the platform user will have to manually press the Decawino's reset button when deploying the firmware as the RPI cannot identify by itself which of the two nodes it should flash. We discuss that in the following section.

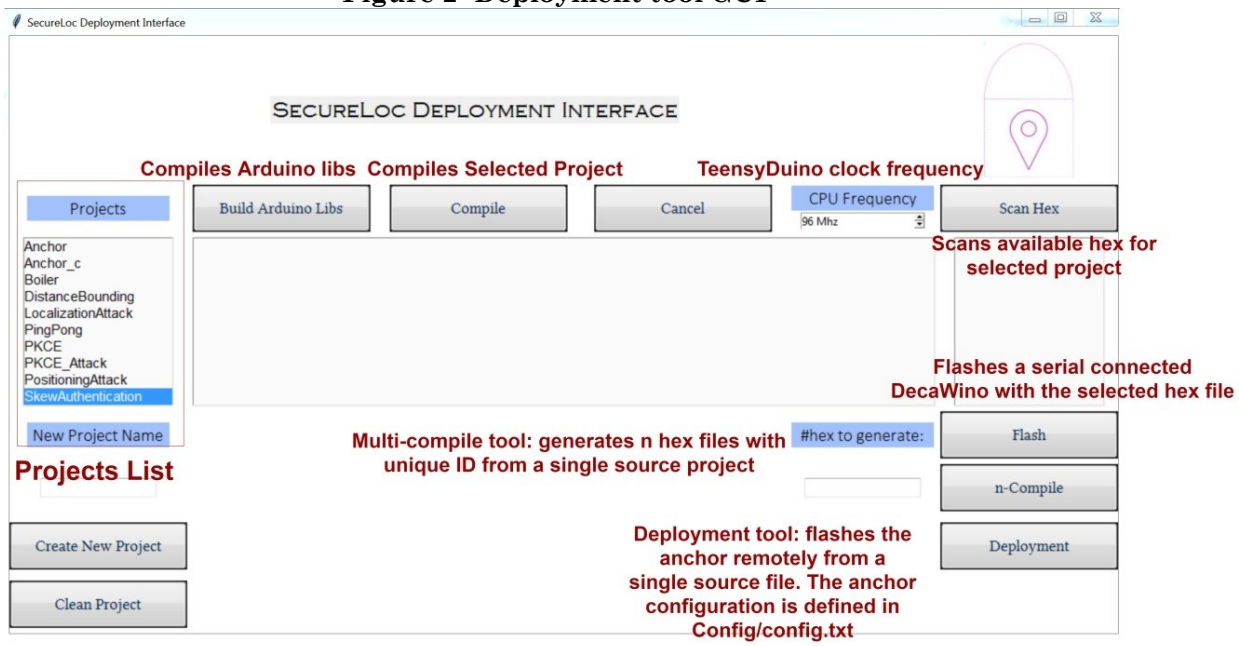
Step 3: Deploying Decawino's firmware

SecureLoc features a dedicated GUI for project compilation and deployment.
From the repository's root, go to the deployment tool directory and start the GUI with:

```
> cd Decawino\Deployment
Decawino\Deployment> python Compilation.py
```

Below is displayed a screenshot of the GUI with comments.

Figure 2- Deployment tool GUI



The list of projects available is shown in the left column. A detailed description of each is provided in the firmware section of this document. For now, we will mostly focus on the two most important ones which are the anchors firmware and the tag firmware.

Pressing the *Compile* button will trigger a regular compilation of the selected project. If you want to setup quickly the platform, you should use the *n-Compile* and *Deployment* buttons.

Anchors

The anchor firmware is contained in *Anchor_c*. If you press *Deployment,* the deployment tool proceeds to parse the configuration file presented in the previous section. It generates one binary for each anchor, and takes care of generating a unique ID per binary. Then, it proceeds to send the binaries to the RPIs, and triggers Decawino's flashing by the RPIs via SSH. If your configuration is correct, you should be able to deploy the anchors firmware in one click.

Tags

As tags are not wired to a RPI, it is unfortunately not possible to flash them all at once. However, the compilation tool can take care of generating unique IDs for each tag without having to run *Compilation* multiple times. You need to use the *n-Compile* command for that. The entry at the right of the button allows to

indicate the number of binaries that you need. For example, if you want to run the platform with 3 tags, select the Tag project on the left, type '3' in the entry box, press *n-Compile* and wait for completion. When it is over, you should see three hex files, *Tag1.hex*, *Tag2.hex*, and *Tag3.hex* in the right column (note: you can list the hex files available for a given project by clicking on *Scan Hex*). Then, you can flash manually each tag by selecting the desired hex file (e.g., T*ag1.hex* for the first tag), and clicking on *Flash*: you will then be prompted to press the mechanical reset button on the Decawino node to complete the operation.

Configuring the number of tags and anchors

If you want to add more tags and/or anchors than defined in the fault settings (four anchors and one tag), you need to modify a few parameters.
The number of tags and anchors is defined in both the anchor firmware and the python code for the localization engine.
For the anchors configuration, you do not need to do anything in the code. The compilation tool will take care of that for you, you simply need to ensure that the configuration file with RPIs addresses and the anchors' IDs is correct (see Step 2). Regarding the localization engine, it automatically detects the number of anchors.
You need to inform the localization engine of the anchors coordinates. The coordinates of the anchors are indicated in a configuration fil. Go to the *ips* directory and open anchors.tab, which should like this:

```
0    0    0   1   blue
0    2.5  0   2   blue
2.5  2.5  0   3   blue
2.5  0    0   4   blue
```

Each entry correspond to one anchors. A line is parsed as:
{x_coordinate} {y_coordinate} {z_coordinate} {anchor_ID} {rendered_color}

Each field should be separated by one or multiple spaces. Note that the last field (color) is deprecated and will have no impact whatsoever as the anchors now default to blue.
Make sure to have the proper number of anchors in this configuration file and the right coordinates.

Concerning tags, you need to manually inform the indoor positioning system of their number.

On the firmware side- you need to edit the anchor header file first:

```
> cd Decawino/Deployment/Projects/Anchor_c/src/header
```

Then, open *Anchor_c.h* and find the following parameter:

```
> #define NB_ROBOTS 1
```

Modify the value to the desired number of tags, and you are done.

On the localization engine side- you need to edit *parameters.py* of the localization engine in:

```
> cd ips/core/
```

Look for the NB_BOTS parameters and set it to the desired value.

Step 4: Running the localization engine in real-time

You should now be able to run the localization in real conditions.
First, go to the directory in which you installed mosquito and run it (*./mosquito* on Linux, *mosquito.exe* on Windows).
Then, open the deployment tool and proceed to deploy the anchor and tag firmware if it is not already done.
You need a power supply for your tags (either batteries or USB charger).
Connect now to the RPis via ssh and start the client code:

```
> cd Desktop
Desktop> python3 clientMQTT.py
```

You can use the multi-exec mode of MobaXterm to do that on all your RPIs at once.
You should see the debug output on the anchors in the RPIs' terminals.

On your machine, in *ips/parameters.p,* disable the playback mode by setting PLAYBACK to False. Go to the respository's root and start the localization engine:

```
> python run.py
```

You should now see the position of your tags in real-time. Refer to the next section for a more detailed description on the localization engine functionalities.

6. 3D engine- user manual

Playback

This mode allows to replay a scenario that has been recorded previously. The ranging values dumped by the anchors in the json file are replayed, the replay speed can be set from the tkinter menu (See Tkinter section). The positions logged are **NOT** reused and are recomputed, which means that different localization algorithms can be used and compared on the same dataset.
Copy the log file to be replayed in the playback directory and enable PLAYBACK in parameters.py before running.
Note: if several files are in the playback directory, the most recent one will be replayed.

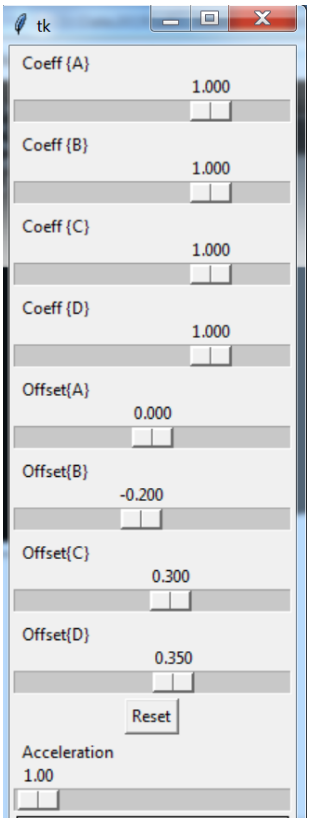
Measurements:

This mode allows characterizing the accuracy of the localization systems on set of reference points with known locations.
First enter the studied reference point in rp.tab. Follow the current formatting of the file. See map section for more information on how to calculate coordinates.
Then, set NB_MEASUREMENTS and NB_REST in parameters.py. The first one defines how much measurements will be done for each reference point. Note that a single ranging made by one anchor is considered as measurement, which means that a full turn of rangings with N anchors equals N measurements. NB_REST defines the number of 'blank' (aka, unlogged) measurements between two reference points, and needs to be set high enough to have the time to move anchor from one point to another.
Every time a measurement for one reference point is a starting/ending a musical signal will notify you, sound should be turned on.
A class for measurements analysis is provided in readMeasurements.py. Check code documentation for further information.

Tkinter Menu

A tkinter Menu allows setting a few parameters in real-time. Corrections can be applied on the ranging from that menu (either as an offset or a coefficient), and in playback mode the playback speed can be increased from that menu as well. Dynamic functionalities should be implemented in the Tkinter. Reset button brings the parameters back to their original values.

Figure 2- Tkinter Menu



Logs

When logs are enabled and/or measurement mode is on, the datasets are logged into json file.

4 sub-directories are used for json logs (in json directory):

- **MQTT:** logs for ranging values in normal mode
- **Pos:** logs for positions in both normal & measurement mode
- **Measurements:** logs for rangings in measurement mode.
- **Playback:** logs for positions in playback mode.

Logs file names are based on the current date & time.

7. Firmware Deployment

Project architecture

The directories in SecureLoc are organized as displayed below:

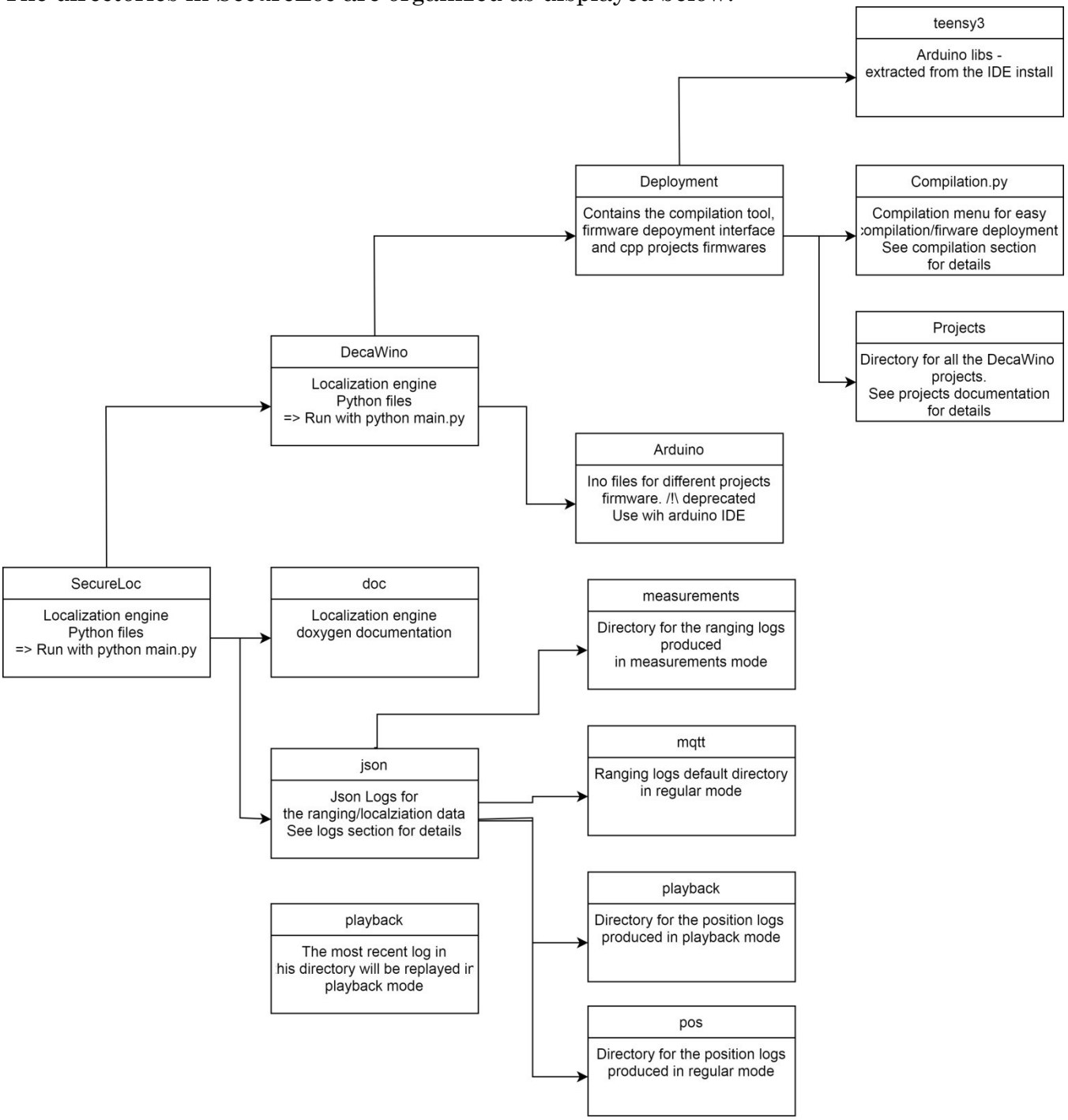


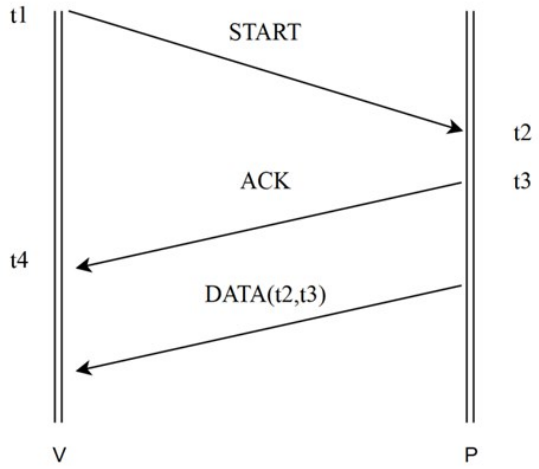
Figure 2- SecureLoc Directory Architecture

8. Background

Rangings

SecureLoc system is based on UWB Time-of-Flight localization. The distance between the tag and the anchor is calculated through Two-Way Rangin (TWR) protocol as following:

Figure 3- TWR protocol



The time-of-flight is given by :

$$d(TWR) = \frac{(t4 - t1) - (t3 - t2)}{2} * c$$

The tag is not informed of the distance in the current model.

Ranging filtering

A sliding window is applied on the ranging. This window is a combination of median and mean filtering. Two parameters are given: the **window size (S)** and the **eliminations numbers (E).**
Basically, the S distance values in the SW filter are sorted and the E maximum values eliminated. The output of the filter defined as the mean of the S – E values remaining.

Localization algorithms

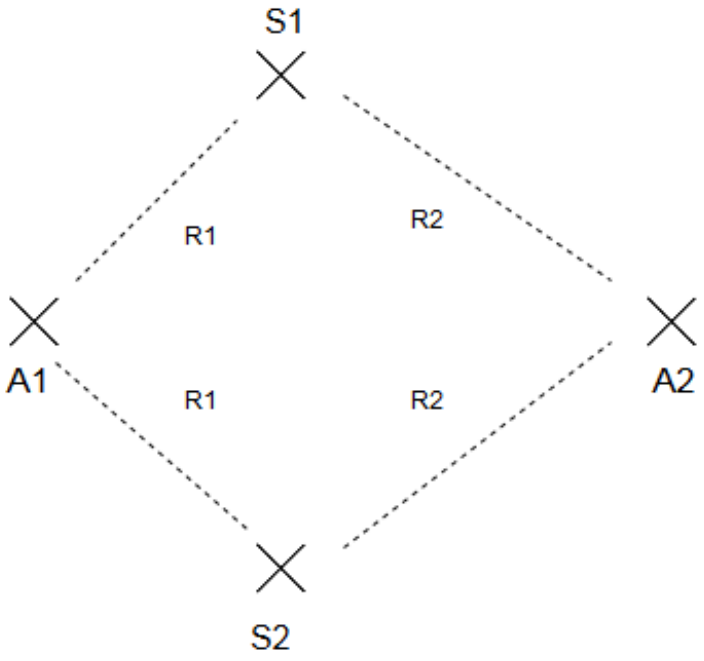
Several methods are available to compute the tag's position:

- **Weighted centroid**
- **Gauss-Newton**
- **Weighted-centroid + Iterative MSE reduction**

Weighted centroid

For 2D positioning, two anchors are enough to get a solution.
From a mathematical perspective, applying Pythagora's Threorem with two ranging values lead to two solutions:

Figure 4- Trilateration



If only two anchors are available, the map should be reduced to a single side of the anchors (i.e., either the right of left side to the line (A1,A2)) such as excluding one of the solutions. If other anchors are available, both solutions are compared to the rangings obtained to the other anchors and the closer one is kept.

For each set of 2 anchors among the N anchors available, a position is calculated by trilateration as described above. Then, the Mean-Squared Error of each solution is computed. The MSE of each anchor for a given position P is defined as

the square of the difference between the distance measured and the distance between the anchor and P. The MSE of a position P is defined as the sum of the anchors MSE for P. Each trilateration solution receives a coefficient proportional to its MSE and the final solution is defined as the weighted centroid of all trilateration solutions.

Gauss-Newton

Standard Gauss-Newton with a magnitude reduction after each iteration. See for example https://en.wikipedia.org/wiki/Gauss%E2%80%93Newton_algorithm
The starting point can be either the solution of weighted centroid or an arbitrary point (RANDOM_SEARCH).

Weighted-centroid + Iterative MSE reduction

This method first computes the weighted centroid solution, then iterates by small steps from this position to find neighbor point with a lower MSE. The number of iterations and the magnitude of the steps can be set. This method can slightly improve the accuracy of the weighted centroid.