Smarthome 2 - Ein lokal betriebenes Sicherheitssystem für das Smart Home

Hendrik van Essen, Arndt Tigges, Robert Munteanu

Abstract

Dieses Projekt beschäftigt sich mit dem immer aktueller werdenden Thema des *Smart Homes*. Die großen Anbieter dieser Systeme bieten jedoch häufig Lösungen an, die den Nutzer an bestimmte Vorgaben binden, sei es die ausschließliche Unterstützung von Geräten bestimmter Hersteller oder der Zwang zur Datenverarbeitung über die Cloud.
Ein System zu entwickeln, welches diese Beschränkungen nicht aufweist war Ziel dieses Projektes.
Um die Sicherheit des Systems und der zwischen den Systemkomponenten übertragenen Informationen zu gewährleisten, verläuft die Kommunikation und Datenverarbeitung nicht über einen nicht identifizierbaren Server, sondern findet stattdessen verschlüsselt im lokalen Netz statt.
Mit dem Ziel, ein offenes, erweiterbares, kostengünstiges, benutzerfreundliches und sicheres System zu entwickeln, lag das Hauptaugenmerk darauf, möglichst bereits existierende Lösungen, von Kommunikationsprotokollen, über Hardwarekomponenten bis hin zu Softwarelösungen, mit einer großen User-Community in dem Gesamtsystem zum Einsatz zu bringen, um ein zukunftsorientiertes System zusammenzustellen, das die Philosophie des Open Source Entwicklungskonzepts symbolisiert.

Abkürzungsverzeichnis

CA	Certificate Authorithy
ESP-IDF	Espressif IoT Development Framework
openHAB	Open Home Automation Bus
OTA	Over-the-Air
MQTT	Message Queue Telemetry Transport
TLS	Transport Layer Security
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
RAM	Random Access Memory

1 Motivation

Derzeit gibt es auf dem Markt eine Vielzahl an Lösungen für Home Automation. In den meisten Fällen handelt es sich jedoch um Systeme, welche zwar innerhalb ihres Ökosystems gut funktionieren, aber Probleme erzeugen, sobald unterschiedliche Geräte mehrerer Hersteller genutzt werden sollen. Weiterer Nachteil der Verwendung eines geschlossenen Systems ist die Abhängigkeit vom Hersteller. Sowohl fehlende Sicherheitsupdates als auch die Einstellung der Produktion können für den Kunden zu einem großen Ärgernis werden und hohe Kosten verursachen. Eine Variante zu finden, die unabhängig von einem Hersteller ist, also frei durch den Nutzer erweiterbar und konfigurierbar ist und im lokalen Netzwerk (d. h. nicht über einen fremdverwalteten Server) läuft, ist wiederum schwer zu finden. Bei Abschaltung der Server werden geschlossene Systeme unter Umständen nutzlos. Auch die Frage des Datenschutzes ist offen. Unser Ziel ist also eine Open Source Lösung, um eigene Daten nicht aus der Hand zu geben und in der Auswahl von Hardware flexibel zu sein.

2 Anwendungsbeispiel

Wir möchten ein Sicherheitssystem erstellen, das mögliche Einbrüche in der Nacht oder bei Abwesenheit der Besitzer erkennt. Dazu werden Mikrocontroller mit Sensoren ausgestattet, die an Fenster und Türen angebracht werden. Diese können dann Bewegungen feststellen und an die Smart Home Zentrale weiterleiten, welche dann einen entsprechenden Alarm sendet, falls das System aktiv geschaltet ist. Falls dies geschieht, wird auch ein akustisches Signal gestartet, damit Einbrecher abgeschreckt werden. Das Sicherheitssystem soll gängige Angriffe, wie die Störung des Funkkanals, oder den Ausfall von einem Knoten detektieren können, um ein angemessenes Sicherheitsniveau zu halten. Einige kommerzielle Produkte haben solche Funktionen nicht.

Da bereits entsprechende Hardware in größerem Maße verbaut wird, können diese mit weiteren nützlichen Sensoren erweitert werden, um zum Beispiel eine Temperaturüberwachung hinzufügen und so das System attraktiver zu gestalten. So könnten Bequemlichkeitsfunktionen wie „Gewitterwarnung, Fenster im 1. OG schließen“oder „Es ist ein guter Zeitpunkt zu Lüften, da es draußen abgekühlt ist“. Dabei ist zu beachten, dass die Mikrocontroller mit einem Akku und über Funk verbunden sind. So ist eine bequeme und gering-invasive Installation der Geräte möglich. Die Akkulaufzeit soll mit Stromsparmaßnahmen erweitert werden, um eine intensive Wartung zu vermeiden.

3 Anforderungen

Das Sicherheitssystem arbeitet über Funk, daher kann die Kommunikation von einem Angreifer mitgehört werden. Deswegen muss der komplette Funkverkehr verschlüsselt über aktuelle Standards erfolgen. Der Funkkanal ist unzuverlässig, daher müssen übertragene Nachrichten bestätigt werden, um einen Verlust zu vermeiden. Außerdem kann er gestört werden, weshalb starkes Rauschen von der Zentrale erkannt werden muss. Die Mikrocontroller werden als Knoten mit der

Zentrale verbunden und laufen dann autark ohne Verkabelung. Da die Stromversorgung ausfallen kann, weil der Akku leer defekt ist, müssen diese Aspekte betrachtet werden und ausgefallene oder gestörte Knotenpunkte detektiert werden.
Die Knoten sollen über Funk verbunden werden, um eine möglichst einfache Installation und Bedienung des Systems zu ermöglichen. Kommerzielle Smart Home Sicherheitslösungen lassen sich ebenfalls ohne Fachpersonal installieren. Unser System soll genauso einfach sein und durch den Open Source Ansatz auch widerstandsfähiger und zukunftssicherer sein.
Wir möchten uns auf keine Hardwarelösung fixieren, um so das System nach Ausfall von Geräten auch in mehreren Jahren weiterhin betreiben und erweitern zu können. Deswegen soll auf verbreitete Hardware und gängige Protokolle gesetzt werden.
Es ist uns wichtig, dass die Komponenten preiswert sind, damit wir ein möglichst großes Publikum ansprechen. Das System kann dann auch von Leuten betrieben werden, die sich eine kommerzielle Lösung nicht leisten können. Deswegen verzichten wir auf teure Spezialkomponenten, die uns mehr Datendurchsatz oder eine bessere Energieeffizienz versprechen.

4 Aufbau des Systems

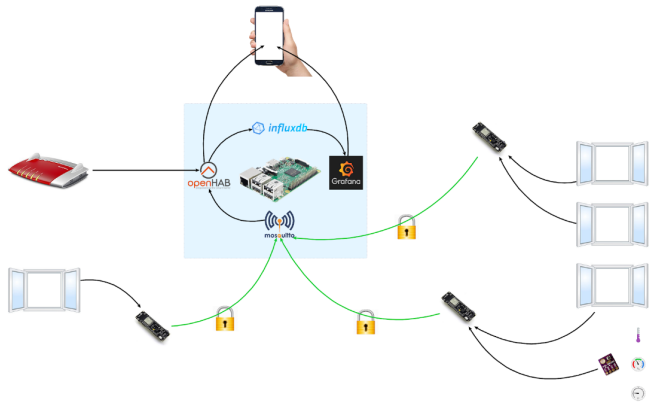

Abbildung 1: Systemübersicht

Die Anforderungen als Grundlage nehmend, ist die Ausarbeitung des Gesamtsystems und Ermittlung der zur Umsetzung nötigen Komponenten der nächste Schritt. Folgende Aspekte werden dazu berücksichtigt:

- Hardware
- Software
- Kommunikation der Systemkomponenten
- Datensicherung
- Funktionsablauf
- User-Schnittstelle

Das zentrale Element des Systems ist eine Smart Home Zentrale, welche mit mehreren Knoten verschlüsselt über Message Queue Telemetry Transport (MQTT) kommuniziert. Ein Knoten ist Schnittstelle für alle Sensoren, die zum Einsatz kommen sollen. Die Knoten senden die Messdaten über das MQTT-Protokoll an die Smart Home Zentrale, wo sie dann alle in einer Datenbank persistiert werden. Diese Messdaten werden dem User über eine Webschnittstelle angezeigt. Über diese ist ebenfalls die Aktivierung des Systems steuerbar.

Im konkreten Anwendungsfall sendet der Knoten eine Nachricht, wenn ein Fenster geöffnet wurde, daraufhin authentifiziert der Message Bus (MQTT) den Knoten und akzeptiert die Nachricht. Die Zentrale nimmt dann die Nachricht an, verarbeitet sie und sendet eine Nachricht, wenn die Bedingungen *Security aktiv* und *Fenster wird geöffnet* erfüllt wurden.

Systemkomponenten

- Smart Home Zentrale
- Sensor-Knoten
- Sensoren

5 Wahl der Systemkomponenten

Zur Umsetzung des Systems in die Praxis musste nun eine Wahl getroffen werden, welche (Hardware und Software) Komponenten in der Praxis zum Einsatz kommen sollen und aus welchen Gründen. Hierzu werden mehrere alternative Lösungen gegenübergestellt und abgewägt, mit welcher Lösung sich die Anforderungen am besten und nachhaltigsten erfüllen lassen.
Bei der Wahl der Soft- und Hardware muss die Kompatibilität zwischen den beiden Bereichen berücksichtigt werden. Trotzdem werden im Folgenden die Komponenten der beiden Bereiche in separaten Kapiteln betrachtet.

5.1 Hardware

Auf der untersten Schicht des Systems ist die Hardware, auf der die zur Umsetzung des Systems nötige Software installiert wird.
Diese gliedert sich in folgende Untergebiete:

- Gerät zum Einsatz als Smart Home Zentrale (1 x)
- Gerät zum Einsatz als Sensor-Knoten (mehrfach)
- Sensoren (mehrfach)
  - Fenster-Sensor
  - optional weitere Sensoren (z. B. Umweltsensor: Temperatur, Luftfeuchtigkeit, Luftdruck, Gas)

5.1.1 Smart Home Zentrale Board

Softwareseitig soll openHAB als Smart Home Sever eingesetzt werden (siehe Kapitel Software - openHAB). Im Folgenden wird die Entscheidung für die Wahl der Hardwarelösung zur Umsetzung der Smart Home Zentrale begründet.

PC oder Microcontroller OpenHAB kann auch auf einem PC laufen, jedoch muss dieser zur Aufrechterhaltung des Sicherheitssystems ständig in Betrieb sein. Einen PC eigens für diese Aufgabe einzusetzen ist nicht ressourcensparend und beinhaltet unnötig viele Funktionalitäten, welche die Kosten des gesamten Smart Home Systems (Stromverbrauch, Hardware-Anschaffung) unnötig in die Höhe treiben würden. Daher muss ein Gerät gewählt werden, welches folgenden Grundanforderungen entspricht.

- unterstützt ein OS, welches die Installation von openHAB ermöglicht
- WLAN Modul
- ein Gerät, welches eine höhere Leistungsfähigkeit bietet, um als Server für mehrere Sensor-Knoten eingesetzt zu werden

Aufgrund der allgemeinen Bekanntheit und Benutzerfreundlichkeit des *Raspberry Pi*[1] haben wir uns dafür entschieden, diesen als Smart Home Zentrale einzusetzen.

5.1.2 Sensor-Knoten Boards

Die Anforderungen an ein Sensor-Knoten Board sind die folgenden:

- **WLAN (802.11n) Modul integriert:** zur Kommunikation über größere Entfernungen; zur Verbindung mehrerer Geräte mit der Smart Home Zentrale (bei Bluetooth gibt es eine Beschränkung bis 8 Geräte); daher nicht Bluetooth
- **Große User-Community:** große, internationale Bekanntheit des Boards; Durch eine große User-Community kann man von der Erfahrung vieler Nutzer profitieren. Es stehen viele Erfahrungsberichte und Tutorials (frei) zur Verfügung.
- **Günstiger Preis:** Mehrere Boards können gekauft werden, wenn der Bedarf (z. B. bei vielen Fenstern in einem Haus) besteht
- **Programmierbare Hardware:** Nutzer sind nicht an eine Technologie gebunden, sondern können ein eigenes System ausarbeiten und ausführen.
- **Akku-Betrieb:** Keine kabelgebundene Stromversorgung mit dem Zwang, Kabel verlegen zu müssen notwendig
- **Stromsparend:** die Boards sollen so lange wie möglich ohne Wiederaufladung der Akkus betrieben werden
- **Platzsparend, möglichst kleine Abmessungen:** Da dieses Gerät am oder in der Nähe eines Fensters platziert werden soll, ist es wünschenswert, dass es optisch nicht zu stark auffällt.

[1] Alle Raspberry Pi Modelle ab Version 2 sind für den Einsatz als Smart Home Zentrale zu empfehlen. Es bietet sich jedoch an, mindestens die Version 3 zu wählen, da dort ein WLAN Modul bereits fest integriert ist und nicht separat erworben werden muss.

Der Mikrocontroller *ESP32* der chinesischen Firma *Espressif Systems*[2] hat am besten den oben genannten Anforderungen entsprochen und wurde daher für den Einsatz als Sensor-Knoten ausgewählt.

Hardwaremerkmale des ESP32 Boards 4 MB Flash, Dual Core, Ladetechnik und 18650 Akku Anschluss, Eingebaute Funkmodule (WLAN + Bluetooth)

Vorteile gegenüber anderen Board Alternativen Im Vergleich zu anderen Boards, die *RIOT OS* als Betriebssystem unterstützen[3], sind ESP32 Boards **kostengünstig** erhältlich, was bedeutet, dass es eine **große User-Community** gibt, sodass viele Tutorials und eine ausführliche Dokumentation der Hardwarefunktonalität zur Verfügung stehen, welche die Einarbeitung in die Handhabung des ESP32 erheblich erleichtern.
Bei möglichen Alternativen zum ESP32 Board[4], die RIOT OS unterstützen würden, wird die Entwicklung und damit der Support eingestellt, was für die Entwicklung eines zukunftsorientierten Systems nicht annehmbar ist.

- geringer Preis
- große User-Community
- low-power Mikrocontroller

Der Vorgänger des ESP32, ESP8266 (von Espressif Systems) ist ebenfalls weit verbreitet, ihm fehlen jedoch folgende Eigenschaften, die für unser System von Nutzen sind:

- Verschlüsselungskoprozessor
- bessere Antenne
- mehr Speicher
- Deep Sleep mit 5 μA Stromverbrauch[5]

Wahl des Betriebssystems Aus den oben genannten Kosten- und Kompatibilitätsgründen wurde für dieses Projekt nicht RIOT OS eingesetzt. Stattdessen wird *FreeRT OS*[6] eingesetzt, das von Espressif zur gemeinsamen Benutzung mit Espressif IoT Development Framework (ESP-IDF) empfohlen wird.[7] Die Wahl des Betriebssystems folgte der weiten Verbreitung und der großen Entwicklergemeinschaft. Wir wollen auch in Zukunft Patches ausliefern können und da ist ein weit verbreitetes Betriebssystem ideal. Zusammen mit der ESP-IDF gibt es den Arduino Core[8] für den ESP32. Durch Verwendung dieser Bibliothek ist es möglich, Arduinokompatiblen Code zu schreiben, der auch auf anderen von Arduino unterstützten Geräten benutzt werden kann. Lediglich native Aufrufe, wie es beispielsweise beim Aufruf des Deep

[2] `https://www.espressif.com`
[3] Unterstützung von RIOT OS für ESP32 Boards ist bereits in der Entwicklung und soll noch folgen: `https://github.com/gschorcht/RIOT-Xtensa-ESP/wiki`
[4] Alternativen zum ESP32: z. B. der Vorgänger ESP8266; weitere Alternativen: `https://github.com/RIOT-OS/RIOT/tree/master/boards`
[5] Nur der Chip allein, ohne Peripherie
[6] FreeRTOS: https://www.freertos.org/
[7] https://esp-idf.readthedocs.io/en/v2.0/build-system.html
[8] `https://github.com/espressif/arduino-esp32`

Sleep der Fall ist, müssten angepasst werden. Ein weiterer Vorteil neben der erhöhten Portabilität ist die Tatsache, dass man alle für Arduino entwickelten Dritt-Bibliotheken einbinden und nutzen kann. So können wir z.B. auch die MQTT Implementierung *arduino-mqtt*[9] nutzen.

5.1.3 Sensoren

Die Boards sind relativ leistungsstark und unterstützen diverse Sensoren. Wir haben für unseren Use-Case Magnetsensoren für Türen und Fenster integriert, welche lediglich einen Stromkreis öffnen und schließen. Es können bis zu 2 Sensoren pro Knoten angeschlossen werden. Für eine mögliche Erweiterung des Systems wurden auch Umgebungssensoren (Temperatur/ Luftdruck/ Luftfeuchtigkeit) integriert. So können wir die Raumluft überwachen und das System mit zusätzlichen Information interessanter gestalten. Wir unterstützen die Bosch Sensoren BME280[10] und BME680[11]. Die Wahl fiel auf diese Sensoren, da die Qualität gegenüber den fast gleich teueren DHT22 wesentlich besser ist. (siehe Abb. 2, S. 4)

5.2 Kommunikationsmechanismus

5.2.1 (Drahtlose) Kommunikationstechnologie

Für dieses Projekt standen mehrere Funkstandards zur Verfügung. Die drei am besten geeigneten sind:

802.11 (Wifi) ist ein sehr beliebter Funkstandard und in nahezu jedem modernen Zuhause bereits verfügbar.[12] Die moderneren Standards ermöglichen einen sehr hohen Datensatz, aber der Funkstandard ist nicht sehr energieeffizient. Ein WLAN-Netzwerk kann sehr viele Geräte verwalten.

802.15.1 (Bluetooth) ist ein Nahfeldfunkstandard. Er wurde entwickelt um Daten zwischen Geräten in kurzer Distanz zu übertragen. Die Reichweite der meisten verfügbaren Geräte beträgt circa 10 Meter und seit Version 4.0 gibt es einen Energiesparmodus, der den Stromverbrauch deutlich reduzieren kann. Es lassen sich bei vielen Geräten nicht beliebig viele pairen (verbinden) und erst mit Version 5.0 können zwei Verbindungen gleichzeitig gehalten werden.

802.15.4 (6LoWPAN) ist ein Funkstandard, der speziell für stromsparende Netzwerke erstellt wurde und wird bei IoT Projekten gerne verwendet. Er erfüllt alle Kriterien, die wir an ihn stellen. Lediglich die Auswahl der Mikrocontroller ist begrenzt und erfüllt unsere Erwartungen momentan nicht.

Bluetooth ist schnell aus unserer Betrachtung gefallen, weil wir keine Geräte mit Protokollversion 5 günstig erwerben konnten und auch die Erweiterbarkeit des Netzwerkes schien uns zu aufwendig für unser Projekt[13]. Zudem spricht die verfügbare Reichweite gegen Bluetooth. Für

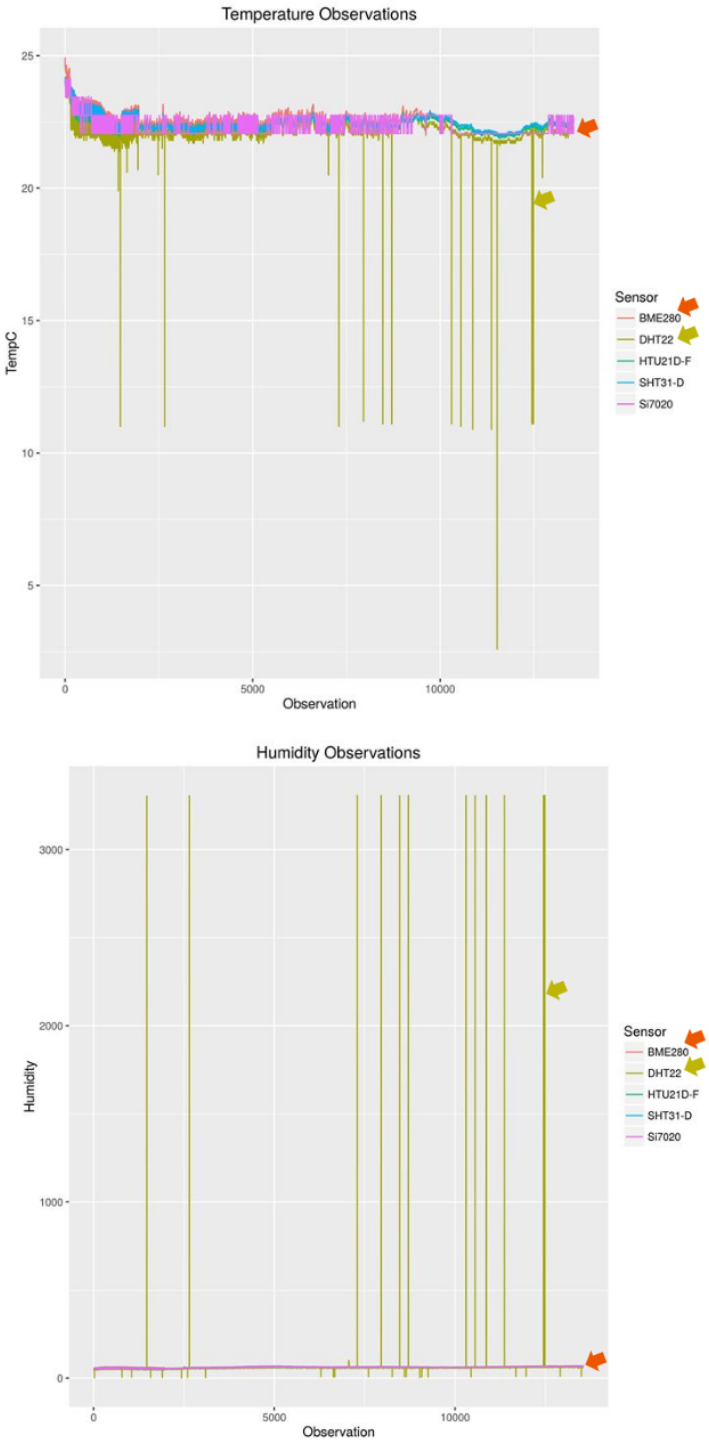

Abbildung 2: Vergleich der Stabilität der Messwerte (Temperatur und Luftfeuchtigkeit) für die Sensoren BME280, DHT22, HTU21D-F, SHT31-D, Si7020; Der gewählte BME280 gehört zu den Sensoren mit einer guten Messstabilität, im Vergleich dazu treten beim DHT22 häufig starke Schwankungen in der Messgenauigkeit auf.

[9] `https://github.com/256dpi/arduino-mqtt`

[10] `https://www.bosch-sensortec.com/bst/products/all_products/bme280`

[11] `https://www.bosch-sensortec.com/bst/products/all_products/bme680`

[12] Bei SmartHome interessierten setzen wir ein verfügbares WLAN voraus.

[13] Wir wollen eine Installation leicht machen und wenn wir eigene Repeater oder Access Points bauen müssen, wäre das zu umfangreich geworden.

WLAN spricht die weite Verfügbarkeit und einfache Erweiterung des Netzwerkes mit Repeatern etc. Es ist sogar recht Preiswert alle Smart-Home Geräte über ein seperates Funknetzwerk komplett vom Rest abzukapseln. Was den Zuspruch für WLAN brachte waren die niedrigpreisigen und weit verfügbaren Mikrokontroller. Die Kosten des Projekts auf Basis von 6LoWPAN Kommunikation würden derzeit zu hoch sein. Durch die gängigen Kommunikationsprotokolle sollte es aber kein Problem sein, in Zukunft auch solche Geräte einbinden zu können, wenn man einen Access Point in das System integriert.

5.2.2 Kommunikationsprotokolle

Wir haben uns beim Übertragen der Daten vom Mikrokontroller zu der Smarthome-Zentrale für MQTT entschieden. MQTT steht für "Message Queue Telemetry Transportünd ist ein offenes Nachrichtenprotokoll für Machine-to-Machine-Communication (M2M). Dabei schreiben die Knoten ihre Nachrichten in ein Topic, das von der SmartHome-Zentrale abbonniert wird. Es konsumiert die Nachrichten und reagiert entsprechend darauf. Der Vorteil, den das Protokoll bietet ist, dass die Nachrichten asynchron und mit Störungen klarkommt, sodass die Nachrichten ggf. nochmals gesendet werden. Zuverlässigkeit ist für eine Sicherheitsanwendung essentiell.

Als Broker haben wir uns für Mosquitto entschieden, da die Software Open Source und sehr leichtgewichtig ist. Sie läuft problemlos parallel auf einem Raspberry Pi mit openHAB und allen anderen genutzten Diensten zusammen.

Die Topics können frei gewählt werden, wobei wir uns für folgendes Muster entschieden haben:

```
room/office/nodeID/WindowSensor1
room/livingroom/nodeID/humidity
```

Zu Beginn beschreiben wir die Wohneinheit. Derzeit gibt es nur Räume, aber eine Erweiterung auf den Garten oder Haus (Gesamtstromverbrauch des Hauses etc.) ist denkbar. Als nächste Einheit wird der Raum näher spezifiziert. Danach folgt eine Knoten-ID zur Identifizierung des Knotens und dann die einzelnen Sensoren, die angeschlossen wurden.

5.3 Software

5.3.1 Smart Home Zentrale

Die Smart Home Zentrale stellt die zentrale Komponente in unserem Projekt dar. Sie führt alle Sensordaten zusammen, speichert deren Verläufe, sendet Benachrichtigungen und bereitet die Daten für die Nutzer grafisch auf. Sie muss erweiterbar sein und sich in das bestehende Smart Home integrieren. Nicht zu vergessen, es ist für uns wichtig, dass die Zentrale mit Open Source Software mit Standardprotokollen läuft, damit wir das System in Zukunft warten und austauschen können. Der vielversprechendste und eingesetzte Kandidat ist Open Home Automation Bus (openHAB).

OpenHAB wird seit Jahren von der Eclipse Foundation entwickelt. Sie ist sehr finanzkräftig und die große Nutzercommunity lässt uns hoffen, dass das System noch mehrere Jahre entwickelt und gewartet wird. Es wurde in Java geschrieben und läuft daher auf allen gängigen Plattformen, ganz egal ob auf einem Server, Notebook oder Raspberry Pi. Für unsere Tests haben wir verschiedene Varianten des Raspberry Pi (2b,3b und 3b+) verwendet. Sie dienten als untere Leistungsschranke, für die wir das System optimieren wollten. Außerdem wird ein fertiges Image auf Raspbian-Basis von openHAB bereitgestellt, was uns für eine vereinfachte Bedienung entgegenkommt.

OpenHAB hat eine Plugin Architektur, was es erlaubt verschiedene Systeme und Technologien anzubinden. Derzeit sind es über 200 inklusive den Marktführern wie Philips Hue. Es können Befehle gesendet und empfangen werden, um somit Aktionen zu steuern. Dies ist ein wichtiger Punkt. Das Installieren der Bindings über die Webschnittstelle ist sehr komfortabel. Wir haben das MQTT, Influx, OpenHAB-CloudConnector und das Fritzbox TR064 Binding verwendet.

- MQTT Binding erlaubt uns die Sensordaten von MQTT auszulesen und zu benutzen.
- Influx Binding persistiert die Sensordaten für Grafana für die späteren Auswertungen.
- OpenHAB-CloudConnector lässt uns die Verbindung zu myopenhab.org aufbauen, damit wir Push-Notifications auf unsere Smartphones senden lassen können.
- Fritzbox TR064 Binding ist optional und erlaubt es uns das Sicherheitssystem automatisch zu aktivieren, wenn die Besitzer mit dem Smartphone das Haus verlassen.

Wie haben wir unsere Sensoren integriert? Dazu haben wir für openHAB Items angelegt. Ein Item hat einen Typ und einen Zustand. Es kann ein Schalter sein, der An oder Aus ist oder eine Temperaturanzeige, welche die Raumtemperatur zeigt. Zum Anlegen neuer Items editiert man eine Konfigurationsdatei auf dem Server, indem man den Typ, Namen, Icon, Gruppe und die Binding-Konfiguration angibt. In unseren Fall war die Binding-Konfiguration das MQTT-Topic, an welches die Knoten senden. Anschließend lädt openHAB die Konfiguration sofort neu und das Item ist dem System von da an bekannt.

User-Schnittstelle Nutzer haben auf verschiedene Weise Zugriff auf die Smart Home Daten. Zum Einen gibt es folgende Möglichkeiten auf openHAB zuzugreifen:

- über die openHAB App auf dem Smartphone (siehe Abb. 3, S. 6) oder der Smart Watch
- über das Web-Interface von openHAB (erreichbar (beim Einsatz eines Raspberry Pi) über `openhabianpi:8080`)

OpenHAB kann so eingestellt werden, dass es die Sensordaten in InfluxDB persistieren, also speichern soll. So können ggf. die Änderungen am System nachverfolgt und Alarme ausgeben werden, falls eine längere Zeit keine Daten mehr kommen. Die Aufbereitung der Daten übernimmt Grafana,

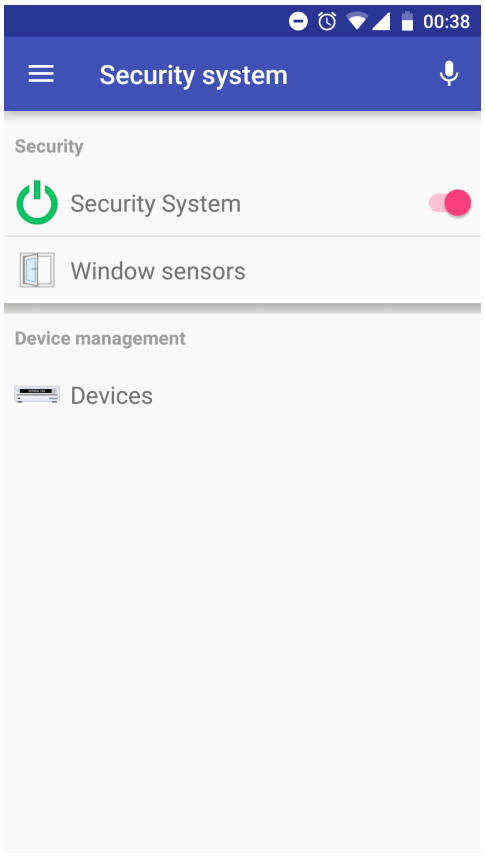

Abbildung 3: openHAB User-Schnittstelle über die Smartphone App

welches ein Analyse- und Monitoring-Werkzeug ist (siehe Abb. 4, S. 6).

OpenHAB liest aus MQTT alle Sensordaten, stellt sie in der GUI dar, löst Alarme aus und speichert die Daten in einer Datenbank, aus welcher Grafana Graphen über den Verlauf erstellt und ebenfalls Warnungen erstellt, falls Daten ausbleiben.

All diese Funktionen und der MQTT-Broker laufen auf einem Raspberry Pi ohne die Ressourcen übermäßig zu beanspruchen. Deswegen rechnen wir nicht mit Problemen im Dauerbetrieb mit den Geräten.

6 Umsetzung

6.1 Sicherheitsmerkmale

6.1.1 Angriffserkennung

Akustisches Signal Mithilfe des Anschlusses eines Lautsprechers an die Smart Home Zentrale kann, sobald ein Alarm ausgelöst wird, ein Signalton abgespielt werden. Dieser kann nur wieder über die Webschnittstelle von openHAB deaktiviert werden.

Störung des Funkkanals Die Smart Home Zentrale (Raspberry Pi 3/3+) hat ein WLAN-Modul. Dieses überprüft per Cron-Job die WLAN-Qualität. Sobald die Qualität des

Abbildung 4: Grafische Darstellung der Sensormesswerte in Grafana

WLAN unter einen festgelegten Schwellenwert fällt, wird in der Standardkonfiguration ein stiller Alarm ausgelöst. Das bedeutet, dass der Nutzer zwar eine Benachrichtigung über das schlechte WLAN-Signal erhält, allerdings kein akustischer Alarm ausgelöst wird. Ansonsten könnte es unter Umständen zu einer Vielzahl an Fehlalarmen führen. Der Server sollte für den tatsächlichen Betrieb über LAN an den Router angeschlossen sein, sodass er im Falle einer Funkstörung weiterhin von außen erreichbar ist und auch die Benachrichtigung über den Alarm an den Nutzer absetzen kann.

6.1.2 Lost Node Detection

Da wir durch Stromsparmaßnahmen die Knoten nicht von der Zentrale aus ansprechen können, können wir nur das Fehlen von Nachrichten als ausgefallenen Knoten betrachten. Deswegen wird die Eigenschaft unserer Anwendung ausgenutzt, dass jeder Knoten in einem konfigurierten Zeitintervall eine Nachricht mit den aktuellen Sensordaten sendet. Da alle Änderungen in Influx persistiert werden, kann Grafana dem Nutzer Benachrichtigungen über fehlende Nachrichten von einzelnen Knoten zusenden. Grafana unterstützt verschiedene Benachrichtigungsformen, getestet haben wir die Benachrichtigung in Form einer Email.
Grafana bietet einen Webhook an, der anstatt oder zusätzlich zu einer Email benutzt werden kann. Wir wollten dies mit der Rest-API von openHAB verbinden, was allerdings nicht gelang, weil der Webhook den Mediatype nicht mitsendet und openHAB dabei wenig tolerant ist. So müssen wir mehrere Benachrichtungskanäle einrichten, anstatt alles über die Push-Notifications von openHAB laufen zu lassen.

6.1.3 Physischer Zugriffsschutz

Zusätzlich neben dem Schutz der Kommunikationskanäle, den verschiedenen Servern und deren dort gespeicherten Daten können auch die Knoten selbst vor physischem Zugriff geschützt werden. Espressif hat dazu in den ESP32 Unterstützung für Flash Encryption[1] und Secure Boot[2] implementiert. Mithilfe von Flash Encryption ist es beispielsweise nicht mehr möglich, in dem Chip abgespeicherte Daten wie etwa das WLAN-Passwort auszulesen. Durch Secure Boot wird verhindert, dass andere Programme auf

dem Chip installiert werden, die keine korrekte Signatur aufweisen.

Da es sich bei diesem System um ein Alarmsystem für ein Haus oder eine Wohnung handelt, haben wir aufgrund der Tatsache, dass bei einem physischen Zugriff auf das Gerät im Haus durch Dritte bereits andere Probleme bestehen, diese Möglichkeiten nicht weiter erforscht. In anderen Anwendungen, wo das Gerät öffentlich zugänglich ist, ist eine Verwendung dieser Funktionen sinnvoller.

6.1.4 Mesh Netzwerk

Durch Verwendung eines Mesh-Netzwerkes können die Daten eines Geräts über mehrere andere Geräte hinweg weitergeleitet werden, ehe sie ihr Ziel erreichen. Einer der größten Vorteil eines solchen Netzwerkes ist die verbesserte Reichweite, die ansonsten nur mit größerer Infrastruktur, wie etwa Einsatz von WLAN-Repeatern, erreicht werden könnte. Ebenfalls von Vorteil ist es, dass das Netzwerk eben nicht auf besagte Repeater angewiesen ist. Sollte ein Repeater ausfallen, so wären unter Umständen mehrere Knoten vom Netzwerk getrennt. Sollte bei einem Mesh-Netzwerk ein Knoten ausfallen, könnte sich die Netzwerkstruktur neu aufbauen und anpassen, sodass die übrigen Knoten weiterhin alle erreichbar sind, was die Ausfallsicherheit der einzelnen Knoten erhöht.

Wir haben uns gegen eine Nutzung entschieden, da ein derartiges Netzwerk aus unserer Sicht nicht mit unseren Stromsparzielen (Kap. 6.2.1, S. 7), welche einen hohen Stellenwert in den Anforderungen einnehmen, zu vereinbaren ist. Eine eventuell angestrebte Reichweitenverbesserung des Netzwerkes kann stattdessen durch den Einsatz von WLAN Repeatern erreicht werden.

6.1.5 Absicherung der Kommunikation über Mosquitto

Eine Absicherung der Kommunikation über Mosquitto ist essentiell, da der Transport der Nachrichten über den Fensterstatus sicherheitskritisch ist.
Dazu wird Transport Layer Security (TLS) im Mosquitto MQTT Broker eingerichtet, wobei die Smart Home Zentrale als Certificate Authorithy (CA) dient. Eine sichere Authentifikation wird über Client Zertifikate realisiert. Die Ausstellung dieser ist ebenfalls Aufgabe der CA.

6.2 Stromverbrauch

Die ESP32 Boards haben eine Akkuhalterung, wodurch die Boards mit Lithium-Ionen vom Typ 18650 betrieben werden können.
Das Laden der Akkus über das Board ist möglich, sodass die Akkus nicht zwangsläufig herausgenommen und über ein externes Ladegerät wieder aufgeladen werden müssen.

6.2.1 Stromsparmaßnahmen

Schlafmodus Die Idee hinter dem Schlafmodus ist es, das Board nur dann wirklich aktiv zu betreiben, wenn tatsächlich Arbeit anfällt. Dazu können entweder der Light Sleep oder der Deep Sleep verwendet werden, die vom ESP32 hardwareseitig unterstützt werden. Aktiv ist das Gerät nur nach folgenden Ereignissen

- Weckruf nach einem Intervall, bei dem Sensordaten übermittelt werden sollen
- Interrupt nachdem ein Fenster geöffnet oder geschlossen wurde

Der *Deep Sleep* [3] führt im Gegensatz zum Light Sleep zum Reset des ESP32. Das bedeutet, dass nach dem Aufwachen alle Prozeduren neu ausgeführt werden müssen, unter anderem also auch Verbindungen neu hergestellt werden müssen. Der ebenfalls verfügbare *Light Sleep* [4] ist hingegen weitaus schneller, da die CPU die Ausführung dort fortsetzt, wo der Light Sleep initiiert worden ist. Nachteil hierbei ist der erhöhte Stromverbrauch, da der Ausführungszustand im Random Access Memory (RAM) gespeichert werden muss. Beim Light Sleep muss beispielsweise der RAM also zwangsläufig weiterhin mit Strom versorgt werden, während dieser beim Deep Sleep abgeschaltet werden kann.
Um den Chip wieder aufzuwecken gibt es verschiedene Möglichkeiten. Entsprechend der oben genannten Ereignisse wird ein Timer-Interrupt benötigt. Hierfür wird die integrierte RTC im ESP32 verwendet. Für externe Interrupts, die für die Fenstersensoren benötigt werden, stehen 2 Quellen zur Verfügung: *ext0* und *ext1*. Die beiden Quellen unterscheiden sich stark. An *ext0* kann immer nur ein einziger GPIO Pin gleichzeitig gebunden werden. Ein einzelner Fenstersensor belegt damit bereits den ersten externen Interrupt. An die zweite Quelle *ext1* können zwar mehrere GPIO Pins gebunden werden, allerdings werden diese untereinander in einem analogen Logikgatter verbunden. Die möglichen Konfigurationen von *ext1* sind entweder *ANY_HIGH* oder *ALL_LOW*. Während *ANY_HIGH* unproblematisch ist, beschränkt *ALL_LOW* die Verwendung von *ext1* wieder auf einen einzigen Fenstersensor. Würden mehrere Fenstersensoren an *ext1* angebunden werden, müssten alle angebundenen Fenster geöffnet werden, bevor der Chip aufgeweckt wird. Das ist offensichtlich nicht praktikabel, daher ist mit dieser Architektur lediglich die Verwendung von 2 Fenstersensoren pro Gerät möglich.

Anwendungslogik Weiterhin ist es möglich mit erweiterter Anwendungslogik noch mehr Strom zu sparen, z.B. indem der Grund für das Aufwecken betrachtet wird. So besteht beispielsweise nicht bei jedem Öffnen des Fensters die Notwendigkeit nach einer möglichen Aktualisierung zu suchen. Glücklicherweise lässt sich der Grund für das Aufwecken des ESP32 auslesen, jedoch nur für den Deep Sleep. Für den Light Sleep scheint die entsprechende Methode nicht geeignet zu sein. Wird ein Fenster geöffnet, wird lediglich der Fensterstatus an die Zentrale gesendet. Wird das Gerät allerdings durch den Timer aufgeweckt, so werden sowohl der Fensterstatus, als auch die Daten des Umgebungssensors übermittelt. Außerdem findet eine Prüfung nach einer eventuellen Aktualisierung statt.

Konfiguration Ebenfalls möglich sind weitere Energieeinsparungen mittels der Konfiguration des ESP32. Dort sollte

der zweite CPU-Kern deaktiviert werden, da dieser für unsere Anwendung nicht benötigt wird. Eine Senkung der Taktfrequenz ist nicht zu empfehlen, da dadurch die Latenz bis zum Alarm erhöht wird. Zwischen der niedrigsten (80 MHz) und der höchsten (240 MHz) Taktfrequenz macht das im Durchschnitt etwa 4.5 Sekunden Unterschied[14] aus.

Deaktivierung von Funktionalität Es ist davon auszugehen, dass durch Verzicht auf die implementierten Sicherheitsmaßnahmen messbar Strom eingespart werden könnte. Liegt der Fokus ausschließlich auf dem Alarmsystem, könnte ebenfalls auf zusätzliche Sensorik verzichtet werden. Im Zuge dessen könnte auch auf den Intervall-Weckruf verzichtet werden, wenn kein Over-the-Air (OTA)-Update bereitgestellt werden soll. Dann würde der Knoten nur noch beim Öffnen und Schließen von Fenstern aufgeweckt werden.

6.2.2 Akkulaufzeit

Das erklärte Ziel in den Anforderungen ist, dass die verbauten Lithium-Ionen Akkus vom Typ 18650 *mindestens ein Jahr* ohne erneutes nachladen genutzt werden können. Für genauere Werte wird zwar ein langer Testzeitraum benötigt, jedoch kann die Laufzeit mithilfe der erstellten Rechentabelle[15] abgeschätzt werden.

Zur Berechnung von Beispiellaufzeiten wird die folgende Konfiguration angenommen:

- Stromverbrauch aktiv: 120 mA
- Stromverbrauch im Deep Sleep: 55 μA
- Akkukapazität: 3400 mAh
- Aktive Zeit pro Weckruf: 9 s
- Selbstentladung von 4% pro Monat [6, S. 20]

In der folgenden Tabelle ist das Weckintervall variabel, da es der treibende Faktor für den Stromverbrauch ist. Vernachlässigt wird in der Berechnung die Tatsache, dass das mit dem Akku betriebene Gerät schon vor vollständiger Entladung des Akkus nicht mehr arbeiten wird, da eine Betriebsspannung von mindestens 3.3 V gehalten werden muss und die anliegende Spannung im Laufe der Entladung abfällt.

		Akkulaufzeit
Aufwecken alle 15 Minuten	:	99 Tage
Aufwecken alle 60 Minuten	:	262 Tage
Aufwecken zweimal pro Tag	:	532 Tage

Unter Einberechnung der Selbstentladung und der zusätzlichen Beanspruchung des Akkus durch das Weckintervall zur Sensormessung sowie manuelles Fenster öffnen ist eine Akkulaufzeit von mindestens einem Jahr leider nicht realistisch. Wenn man allerdings einen zweiten Akku hinzunimmt und diese zur Kapazitätssteigerung auf 6800 mAh parallel schaltet, erzielt man bei gleichen Bedingungen bei einem Weckintervall von 60 Minuten eine Laufzeit von 389 Tagen. Dieses Ergebnis genügt uns vorerst.

[14] siehe Messungen im Repository[5] unter misc/measurements/runtimes/

[15] Im Repository[5]: misc/power_consumption_calculation.ods

6.3 Usability

6.3.1 Over-the-Air (OTA) Updates

Sind die einzelnen Geräte erst montiert, ist es von Vorteil einen Mechanismus anzubieten, der es ermöglicht, Updates leicht durchführen zu können, ohne sie wieder abbauen zu müssen. OTA-Updates kommen daher zum Einsatz.

Nginx wird als Hypertext Transfer Protocol (HTTP)-Server betrieben. Eine Aktualisierung wird mittels simpler HTTP-GET-Requests auf die auf dem Server liegenden Binärdateien abgefragt. Die URL für die Abfrage eines Boards mit der Geräte-ID *esp32-1* und einer laufenden Anwendung auf Versionscode 2 lautet beispielsweise *https://local_ota_server/esp32-1/3/app.bin*. Über dieses Verfahren aktualisiert sich die Version n immer zuerst auf n+1, bevor sie sich auf n+2 aktualisiert. Wird keine solche Datei gefunden, gibt der Webserver HTTP 404[16] zurück womit das Gerät implizit annimmt, dass es bereits mit der aktuellsten Version betrieben wird. In der Praxis sollte dies keine großen Nachteile haben, da eine Aktualisierung bei jedem Timer-Interrupt abgefragt wird und die Geräte somit im Rahmen des Weckintervalls stets aktuell sein sollten. In der Zukunft besteht hier trotzdem Optimierungsbedarf, da mit diesem Verfahren beispielsweise kein Versionscode ausgelassen werden darf.

Die Sicherheit des Updateprozesses ist besonders wichtig. Ein Aspekt ist der Schutz der Binärdateien auf dem Server selbst. In den Binärdateien sind alle Konfigurationen enthalten, welche das Gerät für den Betrieb benötigt. Dies umfasst auch sensible Daten wie das Passwort für das Heimnetzwerk und Client-Zertifikate zur Authentifikation des Gerätes am MQTT-Broker. Eingesetzt werden deshalb, wie beim MQTT-Broker, Client-Zertifikate, um sich gegenüber dem Webserver zu authentifizieren. Ohne entsprechende Authentifizierung wird der OTA-Server lediglich eine Fehlermeldung zurückgeben. Weiterhin ist ein Passwort für den Zugriff auf die Binärdateien erforderlich. Ein anderer Aspekt ist der Schutz der Übertragung der Binärdateien. Um ein Abfangen der Daten während der Übermittlung zu verhindern werden die Daten ausschließlich über Hypertext Transfer Protocol Secure (HTTPS) übertragen. Der ganze Prozess läuft zudem innerhalb des Heimnetzwerkes ab und ist dadurch noch stärker geschützt, vorausgesetzt der Nutzer hat dieses mit einem geheimen Passwort und starker Verschlüsselung abgesichert.

6.3.2 Einfache und schnelle Einrichtung

Für die Einrichtung des Systems sind zwei Komponenten zu betrachten. Einerseits die Smart Home Zentrale, andererseits die einzelnen Knoten.
Die Smart Home Zentrale samt all ihrer Unterkomponenten wird mithilfe von Fabric eingerichtet und verwaltet [17]. Fabric ist eine Python Bibliothek, mit deren Hilfe Kommandos über SSH auf einem anderen Computer, in diesem Fall der Smart Home Zentrale, ausgeführt werden können. Auch

[16] Datei nicht gefunden

[17] Im Repository[5]: openHABian/setup_openhab_pi.py

Up- und Downloads von Dateien sind möglich. Die Skripte basieren hierbei alle auf der Annahme, dass die Zentrale mit openHABian[18] betrieben wird. Ist diese Grundvoraussetzung erfüllt, kann mit einem einzigen Aufruf des Skriptes ein neuer UNIX-Benutzer und die CA erstellt, sowie InfluxDB, Grafana und Mosquitto installiert und eingerichtet werden. Außerdem werden alle notwendigen Konfigurationsdateien des Alarmsystems für openHAB auf die Zentrale kopiert. Auch die Einrichtung des OTA-Servers und des öffentlichen Zugriffs auf Grafana mittels Dynamic DNS unter Verwendung von *Let's Encrypt* für eine HTTPS-Verbindung wird von dem Skript abgedeckt. Es ist möglich Aufgaben separat auszuführen. Das ist beispielsweise dann sinnvoll, wenn kein OTA-Server eingerichtet werden soll.
Um die Anwendung zu kompilieren und auf die Knoten zu übertragen ist eine entsprechende Toolchain notwendig, im Falle des ESP32 ist es die Toolchain von Espressif. Die Einrichtung dieser Toolchain wird ebenfalls mithilfe von Skripten[19] automatisiert.
Mehrere Kompilier-, Flash- und/ oder Uploadprozesse für OTA-Updates können mittels eines weiteren Skriptes[20] automatisiert werden. Dieses Skript umgeht dazu die Konfiguration durch das Buildsystems für den ESP32 und automatisiert im Grunde die Konfiguration partiell für die notwendigen Daten. Momentan sind das die Geräte-ID und der Raum, in dem sich das Gerät befinden soll.

6.3.3 Gehäuse für die Knoten

Nach intensiver Suche nach universellen Gehäusen konnten wir kein Passendes finden. Der nötige Formfaktor für unseren ESP32, der lang und schmal ist, scheint zu selten zu sein, als dass es dafür viele Angebote geben würde. Deshalb haben wir uns dazu entschieden, selbst ein Gehäuse mittels 3D-Drucker herzustellen (siehe Abb. 5, S. 9), welches besser auf die Hardware zugeschnitten ist, als es ein Universalgehäuse je sein könnte und dazu noch deutlich billiger ist, wenn bereits ein 3D-Drucker vorhanden ist. Die Materialkosten bei einem Druck unseres Modells belaufen sich auf etwa 70 Cent bis zu einem Euro.

Das Gehäuse bietet Löcher im Boden für eine Befestigung der Platine des Boards mittels Schrauben sowie eine separate Kammer innerhalb des Gehäuses mit Luftschlitzen für den Umgebungssensor. Der Deckel ist eine Schiebekonstruktion, da sich diese nach mehreren Iterationen bewährt hat und die eventuelle Abnutzung nicht dazu führen sollte, dass der Deckel nicht mehr auf dem Gehäuse hält. An den beiden Seiten befinden sich jeweils eine Bohrlasche zur Befestigung des Gehäuses an einer Wand. Es gibt ein Loch für die Führung von Kabeln nach außen, sowie eine Aussparung um den USB-Port des Boards.

[18] `https://www.openhab.org/docs/installation/openhabian.html`
[19] Im Repository[5]: ESP32/window_alert/setup_esp32_environment.sh
[20] Im Repository[5]: openHABian/tools/batch.py

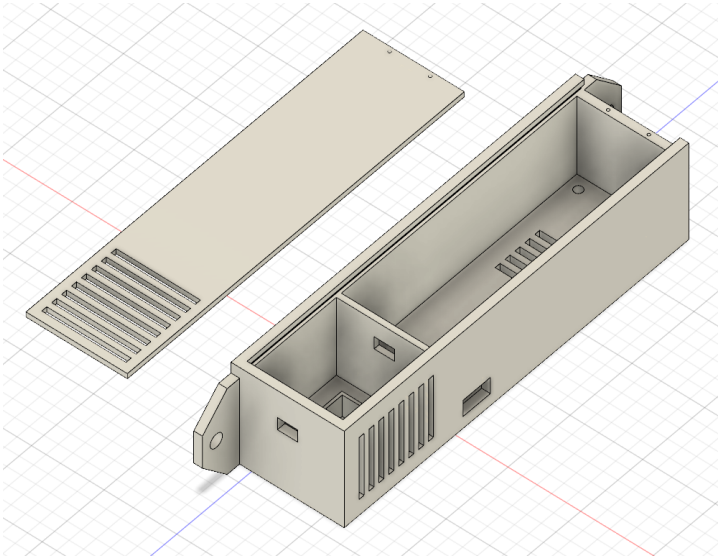

Abbildung 5: 3D Modell unseres Gehäuses

7 Ergebnisse

7.1 Funktionsumfang

Wir haben funktionierende Prototypen, die mit bis zu 2 Fenstersensoren und einem Umweltsensor Daten an die Smart Home Zentrale senden. Die Knoten warten im Deep-Sleep auf einen Timer-Interrupt oder einen Interrupt von den Fenstersensoren. Die Smart Home Zentrale unterstützt beliebig viele Knoten mit Sensoren und stellt keine Beschränkung dar. Die Lost Node Detection sendet Benachrichtungen bei einem Ausfall und das Sicherheitssystem bei einem Ereignis im Alarmzustand. Ebenfalls funktioniert das automatische Aktivieren des Alarmzustandes beim Verlassen des Hauses. Das Stören des Funkkanals wurde nicht im realen Betrieb getestet. Bei entsprechenden Testdaten reagiert das System aber wie gewünscht.

Die Konfiguration des Systems konnte leider nicht so einfach gestaltet werden, wie wir uns das gewünscht hätten, aber durch die Shell und Fabric-Skripte ist der größte Teil abgefangen worden.

7.2 Langzeittest

Bisher wurde unser System nur auf die Korrektheit seiner Funktionalität getestet. Einem umfassenden Langzeittest wurde das System bisher aus zeitlichen Gründen noch nicht unterzogen. Lediglich ein durchgängiger Zeitraum von einer Woche wurde getestet. Bei diesem Test ist aufgefallen, dass das verwendete Board weitaus mehr Energie im Schlafmodus verbraucht, als geplant war und der Akku deshalb bereits nach einer Woche nicht mehr genug Energie zum Betrieb des Gerätes gespeichert hatte. Anstatt von 55 μA verbraucht es etwa 15 mA im Deep Sleep[21]. Das wurde von uns keines-

[21] Hier genannte Verbrauchswerte wurden nicht von uns selbst gemessen, sondern einem Video von Andreas Spiess[7] entnommen. Allerdings deckt sich unsere Berechnung der Akkulaufzeit von einer Woche mit dem Verbrauch von 15 mA. Unser Board ist in der ersten Zeile der Tabelle

wegs erwartet, vor allem da auf dem Board eine Halterung für einen Akku verbaut ist und von uns angenommen worden ist, dass es somit auf Akkubetrieb ausgelegt sein sollte. Unsere Vermutung ist es, dass der verbaute Spannungsregler für diesen Stromverbrauch verantwortlich ist.

7.3 Gesamtkosten für das System

Für das Gesamtsystem fallen geschätzt folgende Kosten[22] an:

Systemkomponente	Stückzahl	Preis
Raspberry Pi 3 (inkl. WLAN, Bluetooth, Gehäuse, SD-Karte, Netzteil)		60,00€
Wemos ESP32 Board (inkl. Li-Ion Batterie)	4	60,00€
Sensoren		30,00€
Magnetsensoren	8	
BME280	4	
oder BME680	4	
Gesamt:		**150,00€**

Im Vergleich dazu kommen bei existierenden Smart Home Angeboten (hier: von der Firma *innogy*) in etwa folgende Kosten zustande, die mehr als das doppelte der Gesamtkosten unseres Systems ausmachen:[23]

Systemkomponente	Stückzahl	Preis
SmartHome Zentrale	1	99,95€
Tür- und Fenstersensoren	8	239,60€
	(1)	(29,95€)
Gesamt:		**339,55€**

7.4 Fazit

Insgesamt sind wir zufrieden mit dem Funktionsumfang und den Einstellungsmöglichkeiten unseres Projektes. Wir haben viel Zeit in Recherche gesetzt und waren teilweise sehr unzufrieden mit den am Markt angebotenen Lösungen. Wir sind aber überzeugt, dass unser Projekt die Anforderungen gut umgesetzt hat. Wir haben die komplette Kommunikation verschlüsselt und mit Zertifikaten abgesichert.
Die eingesetzten Kryptografieverfahren sind auf dem aktuellen Stand und bieten ausreichend Schutz. Wir haben das System vor gängigen Angriffen abgesichert und können bei Bedarf Benachrichtigungen senden. Die eingesetzte Software ist Open Source und unser Quellcode ist unter entsprechender Lizenz veröffentlicht.
Die Auswahl der Komponenten entspricht im Endeffekt unseren Wünschen, wobei wir noch nicht vollständig zufrieden mit den Boards sind. Unter anderem sind wir noch auf der Suche nach einer stromsparenderen Alternative zu unserem aktuellen ESP32 Board. Bei den Anschaffungs- und Erweiterungskosten sind wir unterhalb den kommerziellen Lösungen. Dabei sind unsere Knoten vielseitiger einsetzbar. Lediglich im Design und Kompaktheit können wir nicht mithalten, aber mit verbesserten und langzeiterprobten 3D-Druckvorlagen könnten wir aufschließen.

in den ersten Sekunden des Videos zu finden.

[22] Die Kosten können abhängig von der Wahl des Bestellortes und evtl. Vergünstigungen durch den Erwerb größerer Mengen abweichen.

[23] https://www.innogy.com/web/cms/de/3749570/home/vorteile/sicher-wohnen/

8 Ausblick

Es wird ein Langzeittest des Systems folgen, anhand dessen das System optimiert werden kann und eventuelle Probleme behoben werden können.
Aktuell wird an einer Portierung[24] der Anwendung für den ESP32 auf RIOT gearbeitet, was allerdings nicht mehr Teil dieses Softwareprojektes ist. Es bleibt abzuwarten, in welchem Umfang die Anwendung portiert werden kann, da die Portierung des ESP32 auf RIOT selbst noch nicht abgeschlossen ist[25]. Eine der wichtigsten fehlenden Features des RIOT-Ports ist derzeit die Implementierung des Deep Sleep. Ohne dessen Implementierung ist der Betrieb über einen Akku aussichtslos und nur über feste Stromversorgung möglich.

Es ist naheliegend, dass wir in Zukunft auch weitere Sicherheitsfunktionen hinzufügen, um auf kommerzielle Produkte aufzuschließen. Dazu käme die Integration von Kameras und Bewegungsmeldern, um einen detaillierteren Eindruck über das Geschehen im Haus zu erlangen. Zur Sicherheit gehört nicht nur der Schutz vor Einbrüchen, sondern auch vor Gefahren für das Haus und dessen Bewohner. Wir haben mit den Umweltsensoren bereits angefangen, wobei der BME680 sogar in der Lage ist, flüchtige Gase zu erkennen, aber weitere Sensoren, wie etwa Brandmelder oder Wassersensoren, die eine auslaufende Waschmaschine, Heizung oder Geschirrspüler erkennen, fehlen derzeit noch.

Literaturverzeichnis

[1] Espressif. Flash encryption. `https://esp-idf.readthedocs.io/en/latest/security/flash-encryption.html`. Letzter Zugriff: 21.08.2018.

[2] Espressif. Secure boot. `https://esp-idf.readthedocs.io/en/latest/security/secure-boot.html`. Letzter Zugriff: 21.08.2018.

[3] Espressif. Esp-idf programming guide: Deep sleep. `https://esp-idf.readthedocs.io/en/v2.0/api/system/deep_sleep.html`. Letzter Zugriff: 07.08.2018.

[4] Espressif. Esp-idf programming guide: Sleep modes. `http://esp-idf.readthedocs.io/en/latest/api-reference/system/sleep_modes.html`. Letzter Zugriff: 07.08.2018.

[5] Hendrik van Essen, Arndt Tigges, and Robert Munteanu. `https://github.com/HendrikVE/smarthome2`.

[6] Umweltbundesamt. Batterien und akkus - ihre fragen, unsere antworten zu batterien, akkus und umwelt. `https://www.umweltbundesamt.de/sites/default/`

[24] Im Repository[5] unter dem Branch *riot_port*

[25] `https://github.com/gschorcht/RIOT-Xtensa-ESP/wiki/ESP32-port`. Stand vom 26.07.2018

files/medien/publikation/long/4414.pdf. Letzter Zugriff: 21.08.2018.

[7] Andreas Spiess. 193 comparison of 10 esp32 battery powered boards without display (incl. deep-sleep). https://www.youtube.com/watch?v=-769_YIeGmI. Letzter Zugriff: 21.08.2018.

[8] openHAB Community and openHAB Foundation e.V. openhab documentation. https://www.openhab.org/docs/. Letzter Zugriff: 07.08.2018.