Verifying Sensor Readings and Event Notifications through Monitoring Co-located IoT Devices

Shiva Sunar

A Thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Information Systems Security) at

Concordia University

Montréal, Québec, Canada

September 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Shiva Sunar**

Entitled: **Verifying Sensor Readings and Event Notifications through Monitoring Co-located IoT Devices**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Arash Mohammadi

_____ External Examiner
Dr. Paria Shirani

_____ Examiner
Dr. Arash Mohammadi

_____ Supervisor
Dr. Suryadipta Majumdar

Approved by _____
Dr. Abdessamad Ben Hamza, Chair
Department of Concordia Institute for Information Systems Engineering

_____ 2023

_____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Verifying Sensor Readings and Event Notifications through Monitoring Co-located IoT Devices

Shiva Sunar

As the number of smart environments is increasing, our reliance on IoT devices and sensors is also increasing. However, the data from sensors may not always be reliable as sensors can report incorrect sensor readings and event notifications due to sensor failure or a compromise by a malicious actor. Although there has been extensive research on sensor data verification, they have their own limitations. Most of them deal with only certain types of sensor data, either only verifying the events notifications or only the sensor readings. They use redundant sensors for verification which might incur additional cost and overhead. As those works are designed to learn from a single smart home data without collaborating with other smart homes, they cannot utilize more diverse data from multiple smart homes to achieve better accuracy.

In this thesis, we present an approach that learns the relationships among the co-located sensors and uses that relationship to verify the reported sensor readings, and detects event notifications (including masked and spoofed events) by monitoring co-located IoT devices and also learns collaboratively from the data of multiple smart homes without sharing the private data to a centralized server using federated learning. We implement our solution in the context of smart homes and evaluate its effectiveness using a public smart home dataset. For sensor reading verification, we achieve an R2 score of 0.98, and for event verification, we achieve an accuracy of up to 100% which is among the best of existing works.

Acknowledgments

I want to express my sincere gratitude to my supervisor, Dr. Suryadipta Majumdar, who has served as my supervisor and been a fantastic mentor to me throughout my graduate studies. His consistent encouragement, sound advice, and original ideas have improved my morale and propelled me to the next level of competence. The financial and logistical support I have received from Dr. Majumdar is greatly appreciated. Dr. Majumdar was always open to new research ideas and welcomed them, which was crucial to the smooth development of this thesis. In a personal sense, he was kind, patient, and motivating.

I would like to express my gratitude to Dr. Paria Shirani for her valuable feedback and suggestions which have significantly contributed to the quality of this work. Her insightful comments and recommendations have been instrumental in refining my ideas and improving the overall structure of this thesis.

I want to express my gratitude to my family and friends for their unflagging love, support, and inspiration along with me on this journey. I have found strength in their unwavering encouragement, and I could not have accomplished this without them.

I am grateful to be a part of the CISR Lab, which is led by Dr. Majumdar. For their assistance and collaboration during various stages of this thesis, I am grateful to my bright, creative, and diligent teammates at the CISR Lab. Their valuable suggestions and constructive criticism have consistently assisted me in raising the quality of my work.

I am also grateful to the department and university for providing me with the resources and opportunities that made this research possible. I appreciate your invaluable contributions to my academic and personal growth.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Symbols and Abbreviations

ADL	Activity of daily living
μ_e	Average value of function f_{ed} for event timestamps
μ_{ne}	Average value of function f_{ed} for non-event timestamps
CSG	Colocated sensor group
CS_{S_T}	Correlated sensors of sensor S_T
f_{ed}	Event Detecting Function
S_{ed}	Event Detecting Sensor
T_e	Event Timestamps
$FFNN$	Feed Forward Neural Network
$LSTM$	Long Short-Term Memory
T_{ne}	Non-Event Timestamps
n	Number of sensors to be used to verify a sensor reading
$\rho(x, y)$	Pearson correlation between sensors x and y
T^+	Post-event Window
E_{pred}	Predicted events
T^-	Pre-event Window
RNN	Recurrent Neural Network
E_{rep}	Reported events
β	Tolerance threshold

Chapter 1

Introduction

This chapter briefly describes the context, motivations, problem statements, and contributions of this thesis.

1.1 Context

With the widespread adoption of smart home networks, sensors and actuators are becoming a part and parcel of our daily lives, since they are the major components of smart home systems. Sensors measure the physical parameters of an environment and actuators are used to change the physical parameters; the outputs of these devices are reported to the users as *sensor readings* and *event notifications*, respectively. For example, a thermometer (i.e., a sensor) is used to measure the temperature whereas a radiator (i.e., an actuator) is used to change the temperature. Prior works [1–5] show that the reported sensor readings and events notifications may not always be correct and reliable. For instance, the reported sensor readings from a smart home device may contain errors for several reasons, such as sensor defects, physical wear and tear of the sensor/device, external interference and noises, or an attacker actively changing the reading through a malicious app [4, 6, 7]. In addition, the reported event notifications may be incorrect due to event spoofing (i.e., reporting fake events) and event masking (i.e., blocking event notifications) attacks [6, 8–10].

Incorrect reporting of sensor readings or events can be inconvenient or even dangerous in a smart home. As events in smart homes are chained with each other via automation and trigger

programming, one false event might trigger another unwanted event [6]. For instance, a spoofed `fire-alarm-on` event may unlock the doors and allow an adversary to unlawfully enter the home; conversely, during an actual fire, the interception or blocking of this event could lead to damage [11–14]. Similarly, a false `presence-on` event from the presence sensor can cause the smart lock app to trigger `door-unlock` event, and a loss or delay of `presence-off` event can leave the door unlocked even after the user leaves home [15]; which can allow unauthorized access to a house. Therefore, it is essential to verify the correctness of the reported sensor readings and event notifications in a smart home network.

There exist several works that focus on verifying either the correctness of sensor readings or the correctness of event notifications from smart home devices. Specifically, the works (e.g., [10, 15–20]) that verify event notifications usually do not verify sensor readings. The other works that verify sensor readings (e.g., [21–26]) rely on either redundant devices (i.e., multiple instances of the same type of sensor) or dedicated devices (e.g., surveillance cameras); these solutions are either costly for regular smart home users due to the necessity of purchasing additional sensors or are limited to specific attacks (e.g., those detected by a camera or image processing [25, 27, 28]). Thus, none of those existing works present a coherent and singular solution to verify the correctness of both sensor readings and event notifications while covering both spoofing and masking attacks.

1.2 Motivating Example

The upper part of Figure 1.1 depicts a typical smart home monitoring scenario where there is a smart home environment (on the left) and a remotely located homeowner (on the right). Under normal operation, the actual status of the smart environment is the same as the reported status to the homeowner. However, due to malfunctions and vulnerabilities in smart devices, hubs, or smart apps, the reported status may not match the actual status. For instance, the reported status of a smart door is `close`, however, its actual status is `open` [29]. Similarly, the smoke detector and fire alarm may incorrectly report false alarms [30] or may not detect an actual fire [31], a motion sensor may fail to detect motion [32], and a temperature sensor may report the incorrect temperature. Due to the potential for error, it is essential for a homeowner to be able to answer the following question:

"*How to remotely verify this discrepancy in a reported sensor status?*". Using the following example use case, we further highlight the limitations of existing works and provide the motivation for our proposed solution.

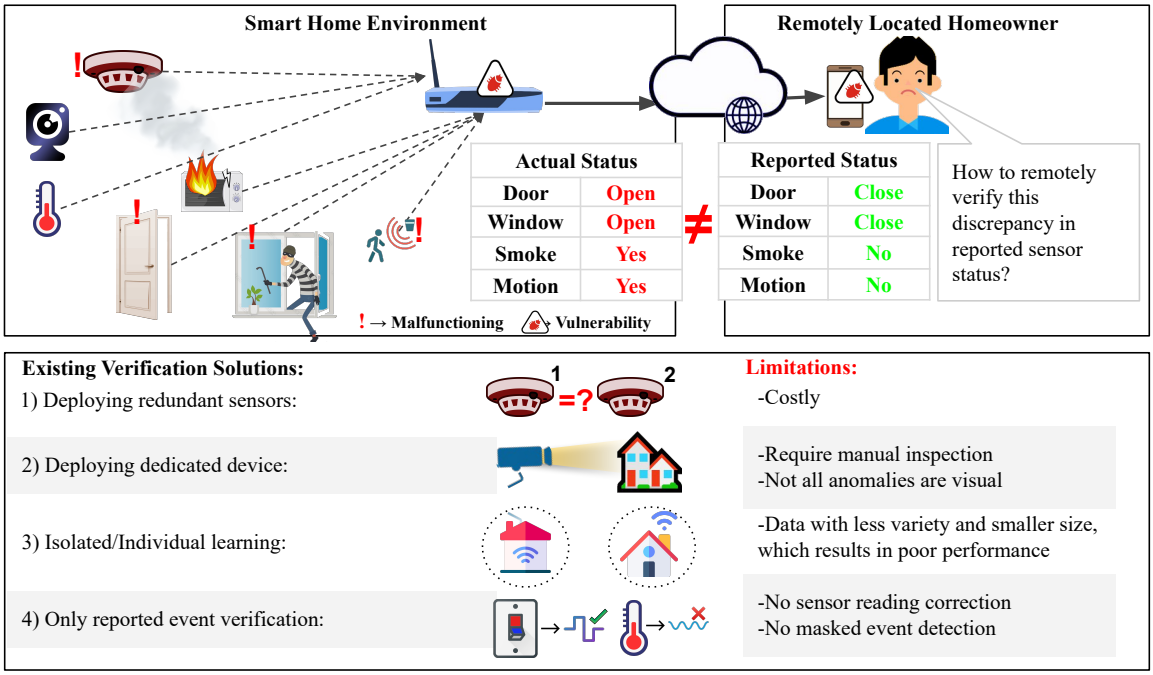


Figure 1.1: The discrepancy between the actual status and reported status of an IoT device and limitations of existing works in remotely verifying them for a smart homeowner.

The lower part of the figure shows the limitations of the existing works for sensor verification in addressing this question, as described below.

- *Deploying redundant sensors:* The verification approach using redundant sensors [21–24] might incur additional costs and overhead to the homeowner and may not be a feasible solution for a typical smart homeowner.

- *Deploying a dedicated device:* The approach using a dedicated monitoring device, such as CCTV [25, 26], requires manual verification from homeowners and is limited to attacks (or malfunctions) that can be verified visually. Also, it can be a single point of failure, for example, the CCTV footage can be replaced [33] or deleted.

- *Proposing isolated/individual learning:* The existing solutions learn to detect incorrect sensor data from the sensor data of a single smart home. They cannot utilize the data of multiple similar

smart homes, i.e., they do not support collaborative learning, which means each of these systems will have lesser data to learn from.

- *Performing event verification only:* Existing event verification solutions [10, 16, 17, 34–37] mostly verify the reported and spoofed events; by design, they do not detect unreported or masked events as well as misreadings from sensors.

1.3 Problem Statement

There are several challenges in verifying sensor data in smart homes.

- Smart homes usually have varieties of devices that can use different communication protocols, different data formats, different sampling rates, different units, etc, therefore it can be challenging to collect and process the data from those devices if, for example, one device uses ZigBee and the other uses Z-wave to communicate, one sensor uses Celsius and the other uses Fahrenheit unit to measure temperature, and one sample at 30Hz and other at 50Hz.

- IoT devices have limited resources with lower power consumption, processing power, memory, and communication bandwidth than traditional computers. They are not able to perform a huge amount of computation, especially training a computationally intensive machine learning model. Thus, our second challenge is to design a verification system addressing those resource constraints.

- Almost all of the existing verification systems are designed to learn from the data of only one home which can be insufficient in some cases. Aggregating the data at a central location enables us to collaboratively learn from the data of multiple homes but doing that gives up the privacy of the user data. Thus thirdly, our challenge is to enable collaborative learning without giving up user data privacy.

- Existing sensor verification systems for smart home verify limited types of sensor data. Some verifies only spoofed events, some only masked events and none of them verifies the reported sensor readings. Thus, our fourth challenge is to design, implement, and evaluate a system that verifies both masked and spoofed events, as well as sensor reading faults.

We will address these challenges in Chapter 4.

1.4 Thesis Contributions

In this thesis, we overcome the above-mentioned challenges of existing works and propose an approach to verify both sensor readings and event notifications by monitoring the (pre-existing) co-located devices in a solution, namely, *SensAudit*. Specifically, SensAudit first learns the relationships among co-located smart home devices. Next, it leverages this learning to predict a sensor reading or event based on the readings from other co-located devices. Finally, it matches the predicted readings or events with the reported (or unreported) readings from a device to verify the correctness of its sensor readings and event notifications. We implement SensAudit and evaluate it with a real smart home dataset [16] to demonstrate its effectiveness.

Our main contributions are as follows:

- To our knowledge, this is the first work to verify the correctness of both sensor readings and event notifications from IoT devices by continuously monitoring other co-located devices without requiring any dedicated or redundant sensors and the presence of any human users, as well as providing an approach for securely sharing sensitive sensor data across different IoT deployments.

- We learn both the sensor-sensor relationship (how a sensor can affect other sensors) and the event-sensor relationship (how an event can affect other sensors) which can potentially be leveraged beyond this work to secure IoT users. This is the first approach to collaboratively learn the relationships among different types of nearby sensors in a set of similar smart environments and to utilize these relationships in order to verify the reported IoT sensor readings and events.

- We implement and evaluate SensAudit in the context of smart home using a real-world dataset and demonstrate its effectiveness by measuring its accuracy and efficiency (e.g., the accuracy of up to 100% for event verification, and an R2 score of 0.98 for sensor reading verification).

1.5 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 reviews the existing works in the related domain, and compares and contrasts them with our work. Chapter 3 provides the necessary background knowledge to understand SensAudit and defines the threat model and its scopes. Chapter 4 describes the methodology and working principles of SensAudit. Chapter 5 describes how SensAudit was implemented. Chapter 6 presents experimental results and performance evaluation of SensAudit. Chapter 7 discusses various aspects of our approach and demonstrates the significance, limitations, and implications of our work. Chapter 8 concludes the thesis with the potential future research directions.

Chapter 2

Related Work

This chapter first compares existing solutions with SensAudit and then discusses different categories of related works.

2.1 Verification in Smart Homes

There exist several works to verify the correctness of event notifications. For instance, *PEEVES* [16] and *Haunted-House* [17] propose a system that verifies the correctness of the reported event notifications using the effects of an event on the nearby sensors. However, unlike ours, they do not provide a remote verification system that can verify both sensor readings and event notifications. Ozmen et al. [10] on the other hand build event fingerprints to facilitate the event verification systems (including ours). However, our work also includes the verification of sensor readings. In *DICE* [34], Choi et al. detect incorrect sensor states and state transitions by learning the possible sets of states of all the sensors in the system and the probability of transition of states from one set to another during the learning phase. In its real-time detection phase, it detects incorrect sensor states and state transitions if the state was not in the learning phase. In [35, 38, 39], the researchers detect sensor faults and incorrect events using the association rules which capture the relationship among sensors during the learning phase. During the detection phase, they detect incorrect sensor states by comparing the current state of sensors with association rules mined during the training. *DETECTIF* [35] and *Idea* [38] detect missing events but these solutions are only applicable to smart homes

which are customized to infer activities of daily living (ADL) and only if the event is part of some ADL, but it is not applicable in general to a smart home scenario.

There are works [21–25, 40] on the verification of the sensor readings, however, all of these solutions are based on either hardware redundancy or dedicated verification devices. For instance, the authors in [21] detect sensor faults by utilizing the data gathered by nearby sensors of similar types. However, those works have not been examined in the context of smart homes. There are several works (e.g., [25–28]) that use cameras to perform various security operations (such as intrusion detection, remote monitoring, etc.). In [25], Keat et al. utilize CCTV cameras and motion sensors are utilized to monitor a home and capture intruder images. In [26], Dash and Choudekar a surveillance system using motion sensors and a surveillance camera is developed. In [28], a cloud-based monitoring framework using an IP camera to implement remote monitoring services of the smart home is proposed. In [27] by Sultana et al., crime events are detected and confirmed in real-time with a camera utilizing an AI and event-driven approach. However, for those works, users need to monitor captured videos to detect the events of interest, which can be challenging [41]. Additionally, those works can only detect attacks that have visual effects (captured by a camera).

2.2 Missing Event Detection in Smart Homes

In the context of smart home, *missing event detection* refers to the ability to detect an event, which was not reported by the primary sensor, through other sensors. For instance, `door` event, which was not reported by a primary faulty contact sensor, but detected by other sensors (such as accelerometer, magnetometer) is an example of missing event detection. Utilizing thresholds in query predicates (e.g., is the sound greater than 10 dB?) to define the events is a common event detection strategy in many works [42–48]. Some works [43, 45, 47] utilize a threshold value to detect events as a common event detection strategy. *SecureHouse* [45] detects `door` events by measuring the vibrations through a smartphone accelerometer mounted near the door and comparing it with a predefined threshold value. Similarly, [43] detects `user-passing` event (i.e., a user passing through the door) primarily by measuring the changes in magnetic intensity using the on-board magnetometer sensor of a smartphone and comparing the changes with a threshold value.

The authors in [47] detect `door` events using the sound impulse generated by a smartphone and analyzing the Doppler shift of the received response caused by the moving door and then comparing it with a threshold Doppler shift. Some other works [49–51] detect events by utilizing spatial and temporal patterns in the sensor data (e.g., slope, oscillation, curvature, jump, and spike). However, all of them only focus on detecting missing events.

2.3 Other Security Solutions for Smart Homes

DÏoT [52] detects compromised IoT devices through network traffic packets while preserving privacy using federated learning by analyzing the packets transmitted by the IoT device. *DÏoT* demonstrates its proof of concept by detecting devices infected with MIRAI and communication initiated by MIRAI. The *SmartThings* system is investigated in [6] where a number of vulnerabilities introduced by privileged smart home applications are discovered. One of those vulnerabilities enables malicious apps running on the smart hub to spoof events by making them appear to be coming from other networked devices, resulting in unwanted behaviors such as mistakenly activating fire alarms. As shown in [53], malicious mobile phone applications can infect home networks leaving smart home devices vulnerable to attacks from remote attackers through the internet. In [54], the authors develop a worm that spreads from house to house throughout a city by leveraging the communication protocol used in Philips Hue smart lighting to compromise smart devices. However, none of those works verify sensor readings and event notifications in a smart home.

2.4 Comparison between Related Works

The existing works on sensor verification in smart homes are compared and contrasted in Table 2.1.

Attack Coverage columns imply the types of attacks that the solution can detect. PEEVES [16], *HoMonit* [7] and *CLEAN* [36] cannot detect the masked events and only verify the events that have been reported by the devices. *Idea* [38] cannot detect the spoofed events, and only *ThingsDND* [55] and *DICE* [34] can detect incorrect sensor readings. We mark a work supporting *data correction* when it can predict the actual events or sensor values to potentially be used in correcting both event

Research Works	Attack Coverage			Features				Performance		
	Masked Events	Spoofed Events	Sensor Reading	Data Correction	Continuous Monitoring	Precise Verification	Collaborative Learning	Attack / Faulty Data Detection Time		Event Detection Accuracy
								Sensor Readings	Events	
PEEVES [16]	×	✓	×	✓	×	✓	×	NA	5-10 sec	A:1.0
Haunted House [17]	✓	✓	×	✓	✓	✓	×	NA	5-10 sec	A:1.0
HAWatcher [15]	✓	✓	×	✓	×	✓	×	NA	60 sec	P:0.97
HoMonit [7]	×	✓	×	×	×	×	×	NA	NA	f1:0.98
ThingsDND [55]	✓	✓	✓	×	×	×	×	66 min	66 min	f1:0.88
DICE [34]	✓	✓	✓	×	×	×	×	30 min	30 min	P:0.94
Idea [38]	✓	×	×	×	×	×	×	NA	18-20 hrs	f1:0.88
CLEAN [36]	×	✓	×	×	×	×	×	NA	NA	P:0.94, R:0.8
SensAudit	✓	✓	✓	✓	✓	✓	✓	1 sec	5-10 sec	A:1.0

Table 2.1: Comparing existing solutions with SensAudit. The symbol ✓ and × means supported and unsupported correspondingly. In performance, A, P, R, and f1 mean accuracy, precision, recall, and f1-score correspondingly.

notifications and sensor readings. Otherwise, we mark them as not supported because they detect their correctness using anomaly detection, without predicting their corresponding correct values. Similarly, the *continuous monitoring* column is marked if the verification is performed continuously, i.e., at the frequency of more than once per second, and does not require any trigger from any event. The *precise verification* column is marked if the solution verifies each of the individual reported values from a device, as opposed to verifying a set of multiple reported values as a single data point. The *attack / faulty data detection time* is the time required for a solution to identify an attack or faulty or incorrect data. SensAudit can detect incorrect sensor readings in less than a second. Faulty sensor reading detection time is marked *NA* for the solution that does not perform sensor reading verification. Faulty event detection time for *PEEVES* and *Haunted House* is estimated based on the event window size as per their methodology (as well as ours), it is equal to the event window size. For others, it is as reported by themselves.

In summary, SensAudit differs from other works as follows. Firstly, SensAudit is the first system that covers all three types of attacks (i.e., incorrect sensor readings, masked events, and spoofed events) that are well-known in the sensor reading and event verification literature. Only SensAudit can identify incorrect sensor readings in less than a second, and SensAudit along with *PEEVES* and *Haunted House* can identify incorrect events in almost real-time (5-10 seconds), so our detection time is one of the lowest among them. SensAudit is the only solution that is able to learn collaboratively from the data of multiple smart homes and it does so without forgoing the data privacy of each home by utilizing the federated learning. SensAudit has an event detection accuracy of up to 100%, higher than most of the other solutions. SensAudit is one of the few solutions that perform precise verification and can correct the reported incorrect data. In the following, we discuss existing works from several related categories.

Chapter 3

Preliminaries

This section provides the necessary background on how IoT devices communicate with each other and send their sensor readings and event notifications to end users, formulates the problem of verifying both sensor readings and event notifications and defines our threat model.

3.1 Background on Sensor Readings and Event Notifications

A smart home is a home that is equipped with devices and appliances that are connected to and can be controlled remotely using a smartphone or computer via the internet. A typical smart home architecture is shown in Figure 3.1, where various smart home transducers are connected to the smart hub. Smart home transducers (which include both sensors and actuators) generate two types of data: sensor readings and event notifications from their actuators. *Sensor readings* are continuous physical parameters from a sensor (e.g., temperature, pressure, light intensity) that can have any decimal number (i.e., floating point) values within a range. *Event notifications* are physical parameters from an actuator, with a fixed set of values, e.g., door contact sensor (open, close), motion sensor (motion, no-motion), proximity sensor (near, not-near). Both sensor readings and event notifications are transmitted from devices to the users through the smart hub (i.e., home automation system, such as Home-Assistant [56], openHAB [57], and OpenMotics [58]).

A hub is connected to the devices either through a third-party cloud or directly using various communication protocols. The functionality of each IoT device might be different and according

to their functionalities, the corresponding requirements of bandwidth, physical ranges, latency, and power consumption can vary. For example, a security camera might need a high bandwidth connection and its power consumption does not matter much, as it can be mains powered. But for a battery-powered device such as a thermometer or a contact sensor, low bandwidth is acceptable but they must be power efficient. Therefore, to cater to these different needs of different devices there are different communication protocols with their own strength and weakness. For low-powered and low-bandwidth use cases, there are protocols such as ZigBee, Z-Wave, Thread, and Matter. Whereas, for more bandwidth, there are protocols such as Bluetooth Low Energy (BLE), and for even more bandwidth at the cost of more power, there are protocols such as Wi-Fi [59]. There are also power-line communication protocols that use existing AC powerlines to communicate between devices such as X-10, UPB, and LonWorks [60]. Some IoT devices, nest thermostats, for example, store their data directly to the cloud platform of their manufacturers [61]. Different IoT devices using different protocols may not be able to communicate directly with each other. Home automation systems, such as Home-Assistant, openHAB, and Open-Motics, can be utilized to integrate all the devices in a single platform which enables them to communicate with each other, as shown in Figure 3.1.

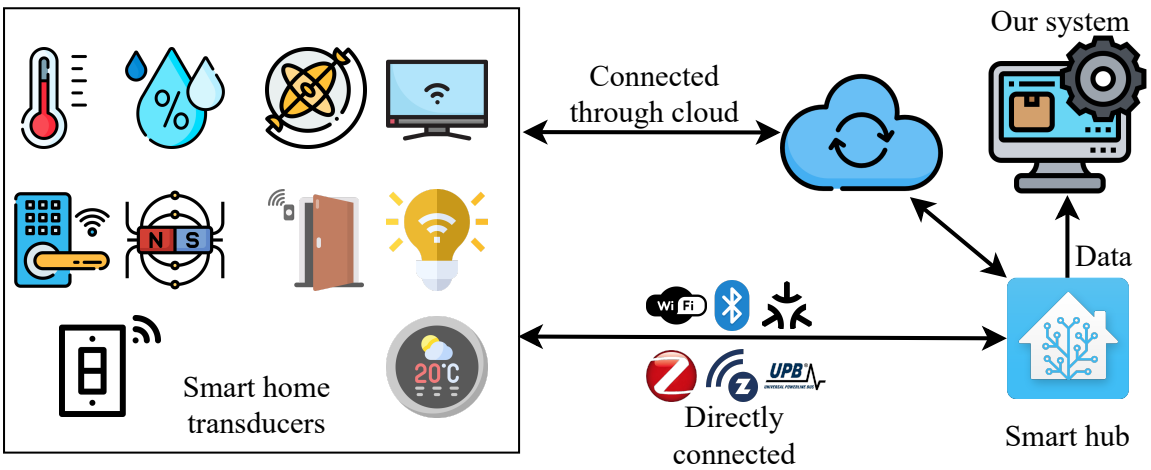


Figure 3.1: An overview of a smart home architecture [62, 63].

3.2 Background on Learning Techniques

This section discusses some of the statistical tests and machine learning techniques that are used in SensAudit.

3.2.1 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test [64] (KS test or K-S test) is a non-parametric test used to quantify how different the distributions of data samples are, based on their one-dimensional probability distributions. Its value ranges from 0 to 1, 0 means that the two samples are from the same distribution, and the higher the value more different the two samples are. There are two types of K-S test; one sample K-S test and two sample K-S test.

One sample K-S test. One sample K-S test takes a sample dataset (X_i) and probability distribution function (F_n) and provides the likelihood of the sample X_i being sampled from F_n.

Two sample K-S test. Two sample K-S test takes two sample datasets X_i and Y_i and calculates the likelihood of those samples being sampled from the same underlying probability distribution function. We use two sample K-S test to identify the best event-detecting function in Section 4.3.3.

3.2.2 Neural Networks

A neural network is a collection of algorithms that simulates how the human brain works to find connections among huge amounts of data. It can refer to a network of artificial neurons or nodes in the case of an artificial neural network, or it can refer to a neural circuit made up of real neurons. Artificial intelligence (AI) problems are solved using neural networks, which simulate the connections between biological neurons and synapses that are found in the brain. Artificial neural networks have three major layers: an input layer, one or more hidden layers, and an output layer. Each layer is made up of one or more nodes, and each node is connected to other nodes and has its associated weights and thresholds. If the output of a node is greater than its threshold then the node is said to be activated, which then can pass the data to its next node otherwise it cannot pass the data. Neural networks depend on training to learn and improve their performance, and once trained they can be used in various machine learning tasks such as classification, regression, clustering, etc.

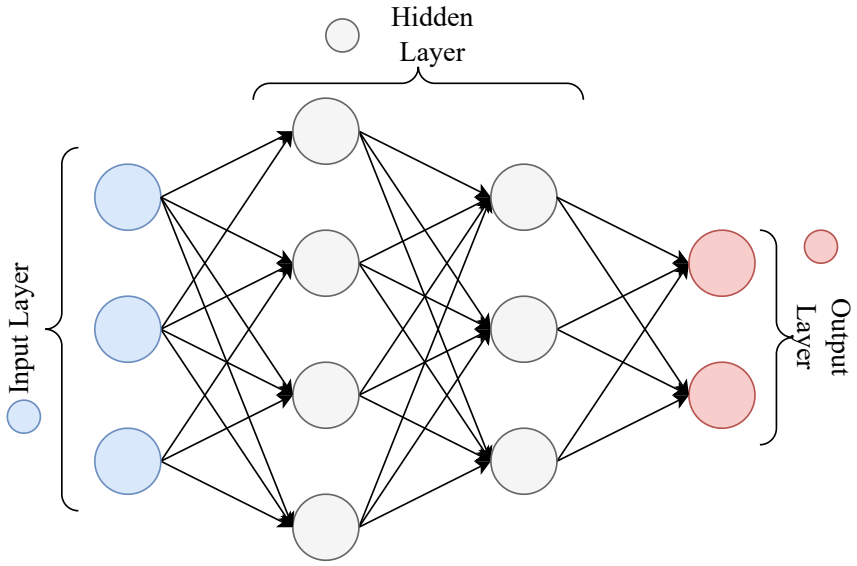
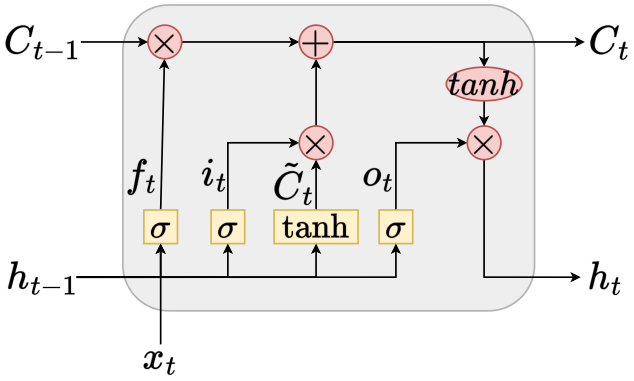


Figure 3.2: A feed-forward neural network with an input layer, 2 hidden layers, and an output layer.

Feed Forward Neural Network. A feed-forward neural network (FFNN) [65] is a type of artificial neural network in which there is no cycle in the connections between the nodes. Since input is only processed in one direction, the feed-forward model is the simplest type of neural network. Although the data may flow via several hidden nodes, it only flows forward and never backward. Each node in a FFNN is called a perceptron. It has four parts: inputs, weights and biases, a summing function, and an activation function. In a perceptron, each input is multiplied by its weight and then summed together to compute a weighted sum. Then activation function is applied to the weighted sum which results in the output of the perceptron, which normalizes the output and also decides whether a neural network is activated or not. It also affects the ability and the speed at which the neural network converges. We use FFNN for event verification as described in Table 5.3.

Long Short-Term Memory Network. Long Short-Term Memory (LSTM) [66] Network is a type of Recurrent Neural Network (RNN) [67] that is designed to handle the vanishing and the exploding gradient problems faced by conventional RNNs by using the gated activation function mechanism. RNNs have cyclic connections, unlike the feed-forward neural networks, where the output from the previous time step is fed to as current input, making them more suitable for modeling sequencing

tasks such as time-series prediction, music composition, video analysis, speech recognition, handwriting recognition etc [68]. It consists of three major parts: the input gate, the forget gate, and the output gate. The input gate decides what information to store in the cell state. The forget gate decides what information to discard from the cell state. The output gate decides what information to output based on the input and the memory of the cell. We use LSTM for sensor reading prediction and verification.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 3.3: Architecture of LSTM cell [69]. C_{t-1} is the previous cell state, C_t is the current cell state, h_{t-1} is the previous hidden state, h_t is the current hidden state, x_t is the current input data. $\bigotimes$ is pointwise multiplication operator, $\bigoplus$ is pointwise addition operator, $tanh$ is tanh activated NN, σ is sigmoid activated NN.

3.2.3 Ensemble Learning

Ensemble learning is a group of algorithms that combines outputs of multiple machine learning models to obtain an output with better prediction performance than the individual constituent models as shown in 3.4. There are three main classes of ensemble learning [70]; bagging, stacking, and boosting.

Bagging. Bagging stands for Bootstrap AGGregatING and it has two parts; bootstrapping and aggregation. It creates multiple diverse datasets from the original dataset, which is called bootstrapping. Then each of those smaller datasets is used to train multiple models of the same type (i.e., the same type of machine learning algorithm) independently and in parallel; these models are called weak or base learners. Then finally, predictions from these models are combined to compute an ensemble prediction as shown in Figure 3.5; this process is called aggregation. To combine those

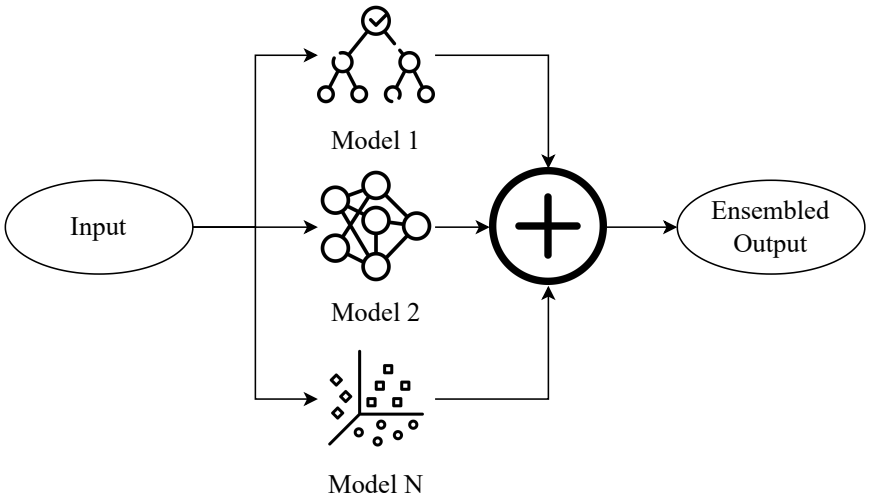


Figure 3.4: Illustration of ensemble learning

predictions various techniques such as averaging, weighted averaging, majority voting, etc are used. Bagging is used to reduce the variance and over-fitting of models.

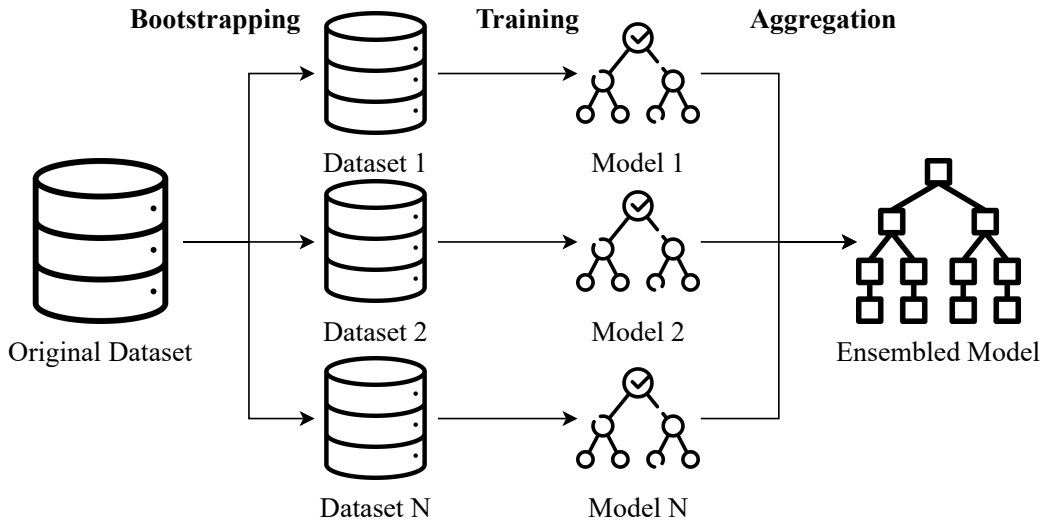


Figure 3.5: Illustration of bagging ensemble learning

Stacking. Stacking is an ensemble learning method in which, the original data is used to train multiple models of different types (i.e., different types of ML algorithms), called weak or base learners, independently and in parallel. Then the predictions of the models are combined to compute a prediction with overall higher accuracy as shown in Figure 3.6. It is used to increase the overall prediction accuracy.

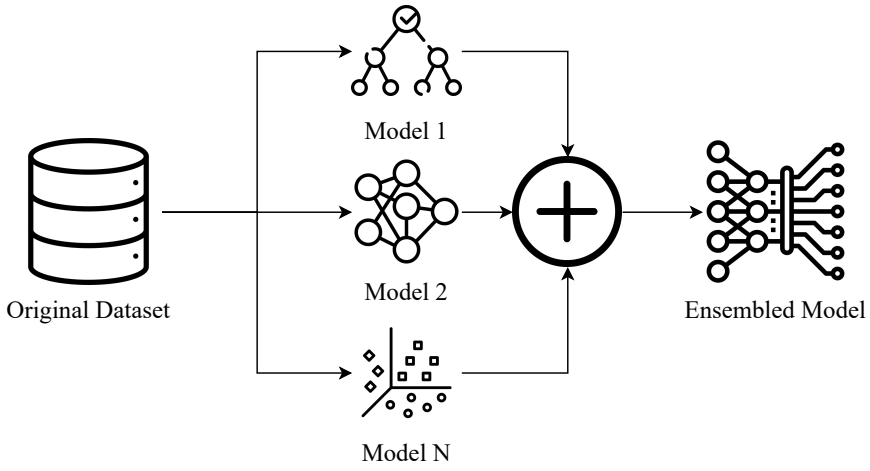


Figure 3.6: Illustration of stacking ensemble learning

Boosting. Boosting is an ensemble learning technique that creates strong models iteratively by adjusting the distribution of the training set. In boosting, the subset creation is not random and depends upon the performance of the previous models. Every new subset contains the elements that were (likely to be) misclassified by previous models. Boosting reduces bias and improves accuracy. It is illustrated in Figure 3.7.

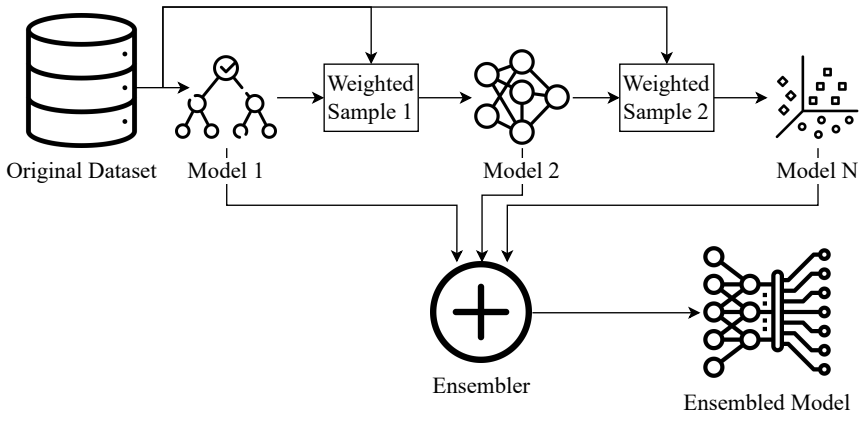


Figure 3.7: Illustration of boosting ensemble learning

3.2.4 Federated Learning

In machine learning, the more and wider the range of training data we feed to train the model, the better performance the model can achieve. One way of achieving and wide range of data is

collaborative learning (i.e., utilizing the data of other similar systems in addition to our own). This may not be feasible in many scenarios as organizations handling sensitive or personal data may not be willing to share the data with a third party or even with other departments of the same organization. Nowadays, due to the increasing awareness and concerns about privacy, end users are also unwilling to share their private data. Also, various privacy protection laws such as General Data Protection Regulation (GDPR) [71] prohibit transmitting user data across countries. Therefore, traditional methods of centralized machine learning, where user data is transmitted to a server to train the model, are becoming less feasible day by day. Consequently, machine learning techniques involving the personal data of users should be done in a way that preserves the privacy of data and follows local data privacy regulations.

Federated learning is a method in which a global model is created using the local models that were trained locally using local data without requiring the data to be shared with others [72], as illustrated in Figure 3.8. As only the model is shared with a central server, multiple organizations can collaborate to increase the performance of the global model without forgoing data privacy.

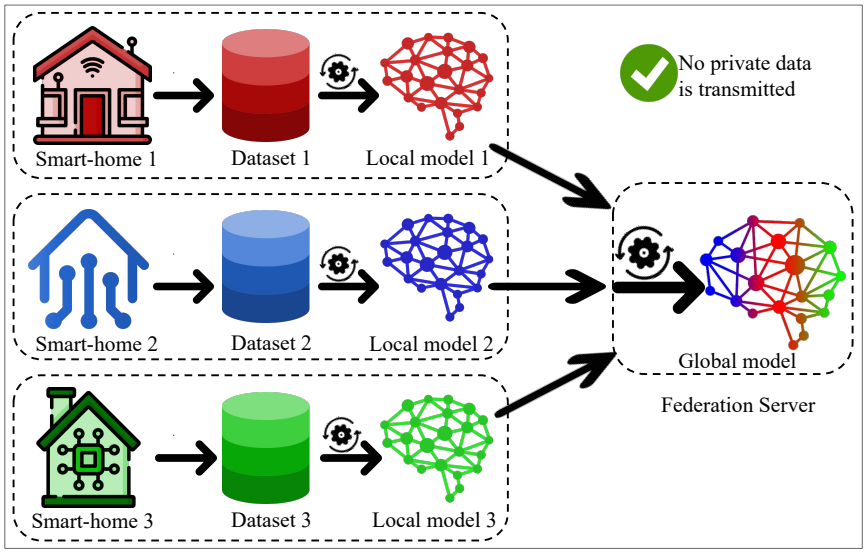


Figure 3.8: Overview of how the global model is created using the locally trained models in federated learning.

3.3 Physical Event Signature on Sensors

Whenever a physical event occurs, the event has an effect on the physical property of the environment for some time and its effect can be visible in nearby sensors for some time. For example, whenever a door opens, it has some effect on its surroundings. For instance, it can produce vibration in the nearby wall, can produce sound due to its slamming, can change the room temperature by letting outside cold air inside, and can lighten the room by letting sunlight inside. These effects can be captured by the nearby sensors as shown in Figure 3.9 which shows an example of such effects of `door-opening` and `door-closing` events on various sensors. These relationships between the events and the sensor reading are what we call *event-sensor relationship*, and our work learns and utilizes these relationships to verify the events.

Similarly, as an event can have an effect on multiple sensors, there can also be a relationship between the readings of two sensors. For instance, in our door opening example, it is likely that the more strongly the door is slammed, the stronger will be the vibration and the sound, i.e., the vibration and the sound have some relationship. This kind of relationship is what we call *sensor-sensor relationship*, and our work learns and utilizes these relationships to verify the sensor readings.

3.4 Threat Model

We primarily consider the following in-scope threats. *(i) Incorrect reporting of sensor readings* covers the threats in which the reported reading of a sensor is different than the actual measurement of the physical condition of an environment. For example, a temperature sensor is reporting 30°C when the actual room temperature is 20°C. A real-world example of damage caused by this kind of threat is the Stuxnet malware [73], which alternated the centrifuge speed to extremely high and low but modified the data from the speed sensor and displayed the normal speed, which eventually broke the centrifuge. Even though this incorrectness can be due to various reasons including sensor defects and attacks, SensAudit is agnostic to its cause. *(ii) Incorrect reporting of event notifications* covers the threats in which a reported event notification from an event sensor is different than the actual event. This can be of two kinds, masking of event notification and spoofing of event notification. *Masking of event notification* covers the threat of the *missing event* where an event occurs physically

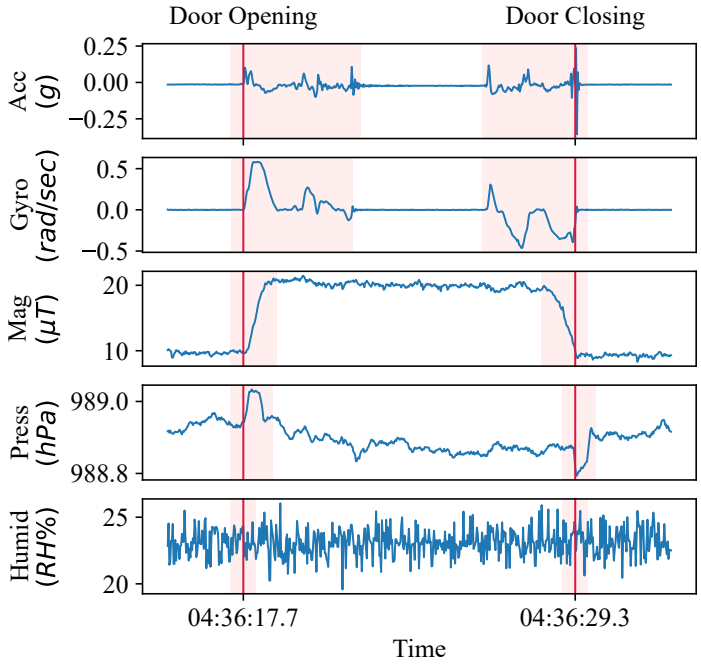


Figure 3.9: The effects of `door-opening` and `door-closing` events. The events have a clearly visible effect on the accelerometer, gyroscope, and magnetometer sensor readings, little in pressure, and no effect on humidity.

but is not reported to the hub either because the corresponding event sensor did not report it due to some fault or the reported event is lost on the way or intercepted by a malicious attacker. *Spoofing of event notification* covers the threat of the *spoofed event* where a fake event is reported to the hub either because of device malfunction or by a malicious attacker.

Our out-of-scope threats include the threats that might affect the relationships among all co-located devices, as in such cases SensAudit can no longer rely on their relationships. Any data-poisoning attacker is beyond the scope of this work (and will be covered in our future work). Like most learning-based approaches, our learning phase relies on attack-free data. Furthermore, we assume that there is a relationship between the co-located sensors in the smart home, and all the participating smart homes are similar in terms of their sensors, devices, and configurations. We will tackle the negative or opposite impacts of dissimilar smart homes in the learning step during our future work.

Chapter 4

Methodology

This chapter first gives an overview of how SensAudit works and then describes the major steps with a set of running examples.

4.1 Overview

Figure 4.1 shows an overview of our methodology that contains three major steps. (i) The *data collection and pre-processing* step (detailed in Section 4.2) collects sensor readings and event notifications from all the IoT devices in a smart home, identifies a set of correlated sensors for each sensor/actuator, and extracts statistical features for each event reported by a device using the sensor readings of its co-located devices. (ii) The *learning* step (detailed in Section 4.3) learns two types of relationships between co-located smart devices: a) the *sensor-sensor relationship* that captures how one sensor reading impacts the other co-located sensors, and b) the *event-sensor relationship* that captures how a device event impacts the other co-located sensors. (iii) The *verification* step (detailed in Section 4.4) continuously monitors co-located devices, and verifies both sensor readings and event notifications of a device to detect the impacts of various threats, such as incorrect sensor readings, incorrectly reported events (i.e., spoofed events) and missing reported events (i.e., masked events).

For example, in a smart home, there is a door event sensor (i.e., a contact sensor) and two co-located sensors; sound and light sensor. In the data collection and pre-processing, the events from

the contact sensor and the sensor readings from the sound and light sensors are collected, the required pre-processing of the data is performed, and for each event of the contact sensor statistical features (such as minimum, maximum, mean, etc.) are computed using the sound and the light sensor readings. In the learning phase, for each sensor, a machine learning model is trained with the data of its co-located sensors to learn the relationships between them. Here relationships like the sensor-sensor relationships (i.e., between sound and light sensors), and the event-event relationships (i.e., between contact and sound sensor, or contact and light sensor) are learned. Parameters required to detect the masked events of the door using sound or light sensors are computed. In the verification phase, events and sensor readings from all three sensors are monitored continuously and their reported data is verified using the models that were trained in the learning phase. Masked events of the door are also detected using the parameters computed in the learning phase.

4.2 Data Collection and Pre-processing

This section discusses how SensAudit performs the collection and pre-processing of the sensor readings and event notifications.

4.2.1 Co-located Sensor Reading Extraction

There can be many IoT devices and sensors in a smart home; however, as the effects of an event on its co-located sensors decrease with distance from the event source/device, its effect can only be seen in nearby sensors [10, 16, 17]; for instance, when a `door-closing` event occurs, the intensity of the generated sound is higher near the door. Therefore, for each event source, we manually group sensors whose readings might be affected by that event source based on their relative distance and call them the *co-located sensor group* (CSG) of that event source. The impact of distance between two devices while measuring their mutual effect is further discussed in Section 7. Some sensors might be affected by multiple event sources, in such cases, those sensors can be a part of multiple CSGs. To later verify the events of an event source, we collect sensor readings of all the members of its CSG (from a setup similar to as shown in Figure 3.1). As each CSG contains all the sensors that can be affected by the events of the corresponding event source, we can view each CSG as isolated

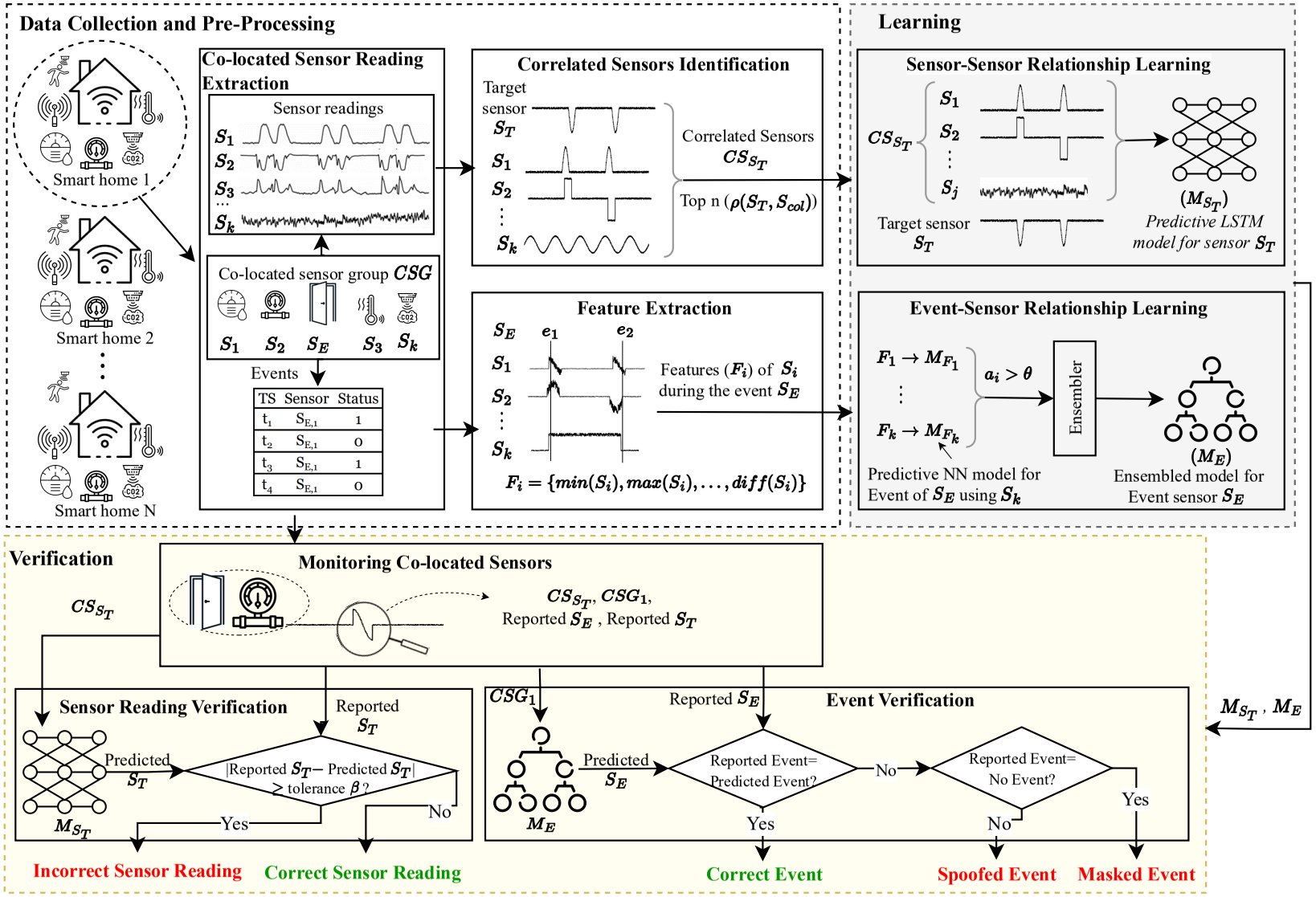


Figure 4.1: Overview of the working methodology of SensAudit.

from each other, which simplifies our system model.

Example 1. Figure 4.2 shows an example where a door is surrounded by *accelerometer* (S_{acc1}), *magnetometer* (S_{mag}), *gyroscope* (S_{gyro}), *door-contact* (S_{door}), and *sound* (S_{snd}) sensors; and a window is also surrounded by several sensors, *accelerometer* (S_{acc2}), *temperature* sensor (S_{temp}), and *window-contact* sensor (S_{win}), as well as a *sound* sensor (S_{snd}) that is not connected to either the door or the window.

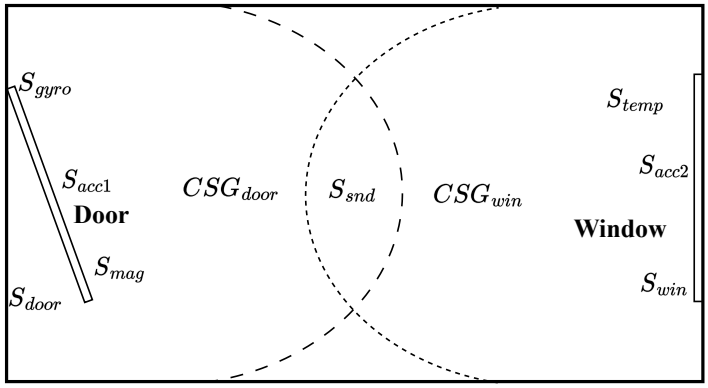


Figure 4.2: Co-located sensors grouping.

For both door and window, we create two CSGs, CSG_{door} and CSG_{win}, containing sensors that are close to those devices and might be affected by their events. As the sound of opening and closing of both door and window is likely to be affected by the *sound* sensor due to their proximity, S_{snd} will be in both CSGs. Therefore, CSG_{door}=$\{S_{acc1}, S_{mag}, S_{gyro}, S_{door}, S_{snd}\}$ and CSG_{win}=$\{S_{acc2}, S_{temp}, S_{win}, S_{snd}\}$, which shows that the S_{snd} is a part of both CSGs in this example. We confirm the effect of an event on the member of its CSG using the following steps.

4.2.2 Correlated Sensor Identification

This step is to identify the set of the most correlated sensors of each sensor in a CSG, that can be later used to predict the value of a sensor (in Sections 4.3.1 and 4.4.3). First, we compute the correlation between each sensor in a CSG using the absolute Pearson correlation coefficient [74]. Then, for each target sensor (S_T) in a CSG, a set of n sensors having the highest correlation with the S_T are identified as *correlated sensors* (CS_{S_T}). The procedure to choose the best value of n is discussed in Section 5.2.3.

4.2.3 Feature Extraction

The effects of different events on a co-located sensor can be different. For example, the reading of a magnetometer increases during the `door-opening` event, whereas it decreases during the `door-closing` event, as shown in Figure 4.3. This kind of difference in effect can be used to differentiate and identify various event types. To capture and quantify the effects of an event type on its co-located sensors, we use various statistical features/metrics (e.g., *minimum*, *skewness*). To extract the features of an event reported by an event sensor (S_E) at time t_e for its co-located sensor (S_i), we first select two time periods of length w_i: (i) *pre-event window* (T^-) of duration $[t_e - w_i,\ t_e]$, the period before the event timestamp, and (ii) *post-event window* (T^+) of duration $[t_e,\ t_e + w_i]$, the period after the event timestamp, where w_i is called the *event window length* of sensor S_i and it is selected as described in Section 6.4.1. For instance, the magnetometer sensor readings during two events occurring at timestamps, t_o and t_c, are shown in Figure 4.3, where the *pre-event window* is indicated by the area with light-blue shade, while the *post-event window* is shown by the area with light-green shade. Then, for both event windows, T^- and T^+, we extract eight statistical features (F), *minimum*, *maximum*, *mean*, *median*, *standard deviation*, *skewness*, *kurtosis* and *difference* (i.e., differences between the first and last data points of the event window). The feature extractor is shown in Figure 4.4.

4.3 Learning

This section describes how SensAudit learns the relationships between sensors and events, and detects the masked events.

4.3.1 Sensor-Sensor Relationship Learning

In sensor-sensor relationship learning, the relationship between each of the sensors within a CSG is learned using an ML model, so that the model can later be used to predict sensor value for verification. To create a model M_{S_T}, which predicts the value of a target sensor S_T, a long short-term memory (LSTM) model is trained using its correlated sensors (CS_{S_T}) obtained in Section 4.2.2.

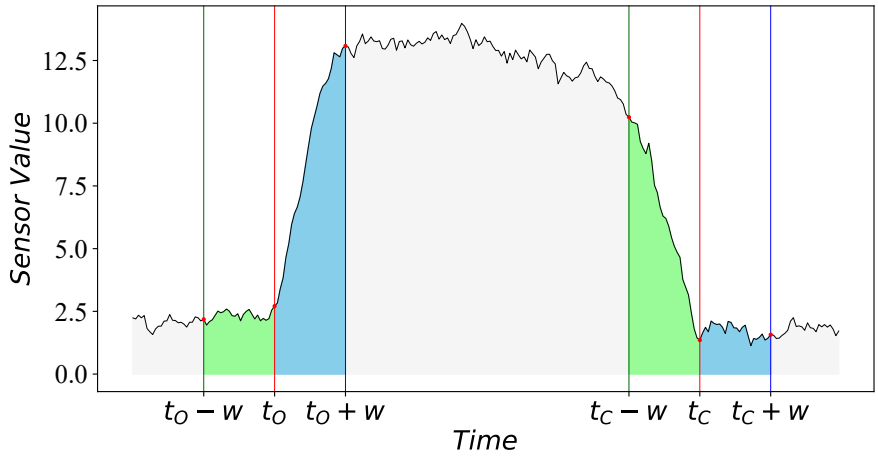


Figure 4.3: The pattern of sensor reading of a co-located magnetometer during `door-opening` (t_o) and `door-closing` (t_c) events is different; the sensor reading increases during `door-opening` (i.e., "difference" is positive, "skewness" is negative) whereas it decreases during `door-closing` (i.e., "difference" is negative, "skewness" is positive). This fact can be used to differentiate the events.

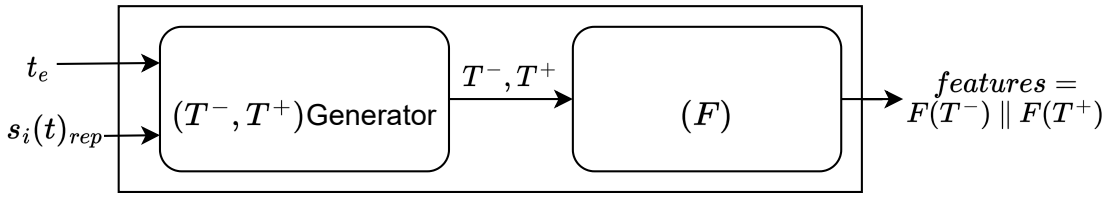


Figure 4.4: Feature extractor. The real-time reported data $S_i(t)_{rep}$ of sensor S_i is continuously fed into the feature generator. Once an event at timestamp t_e is reported, the "Event Windows Generator" selects windows T^- and T^+. Then, feature functions will be used to generate the features of those windows.

4.3.2 Event-Sensor Relationship Learning

In event-sensor relationship learning, the relationship between the events (S_E) and each of its co-located sensors is learned through a feed-forward neural network (FFNN) model, which can be used to predict the events of S_E. To learn the relationships between the events (e) of the event source S_E and each of its co-located sensor S_i, a model M_i is created for each $S_i \in S_N$. Model M_i is trained to predict the events of S_E using the features generated with the data of sensor S_i for that event and its accuracy (a_i) is measured. Afterward, the models with an accuracy of more than a threshold value ($a_\theta > 0.5$) are selected for the ensemble learning. On these selected models, we apply the weighted majority voting ensemble (WMVE) method from [75] in order to combine them

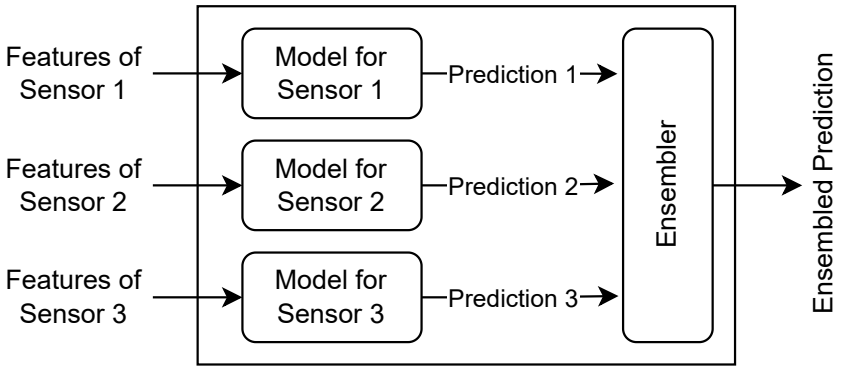


Figure 4.5: Event prediction. The model for each sensor predicts an event using the features, and the ensembler ensemble all these predictions and generates an ensembled prediction.

and generate an ensemble model (M_E) as illustrated in Figure 4.5. As such, the weight (W_i) of each model M_i is calculated as $W_i = \log[\frac{a_i}{1-a_i}]$ [75]. Then, the ensemble prediction (p_{ens}) is calculated as follow:

$$p_{ens} = \begin{cases} 1\ , if\ \ (\sum W_{pred=1} - \sum W_{pred=0}) > 0 \\ \\ 0\ , if\ \ (\sum W_{pred=1} - \sum W_{pred=0}) \leq 0 \end{cases}$$

where $\sum W_{pred=x}$ is the sum of weights of all the models that predicted $x \in \{0, 1\}$ for the event. The ensemble learning is described in Algorithm 1.

The learning steps described above can potentially be implemented using different methods such as isolated, centralized, and federated learning (as described and evaluated in Section 6).

4.3.3 Learning to Detect Unreported Events

This step is to formulate our function and select the values of parameters that are used to detect unreported events in Section 4.4.1, that includes the *event detection function* (f_{ed}), which is a function to detect the missing events, and the *boundary value* of the event detection function, which is used to differentiate between event and non-event timestamps. Whenever an event occurs at the timestamp, t_e, the readings of the sensor S_i in the CSG changes noticeably within the period of $[t_e - w_i,\ t_e + w_i]$, which we call *event detection window*, where w_i is the *event window length* of S_i (shown with w in Figure 4.6).

To detect an event at a certain timestamp (t), we need a function that takes the event detection

Algorithm 1 Compute ensemble prediction

Input:

$\mathcal{A} \leftarrow$ *set of accuracy for each model*,

$\mathcal{P} \leftarrow$ *set of prediction of each model*

Output:

$p_{ens} \leftarrow$ *ensemble prediction*

$sum \leftarrow 0$ `# Sum of weights`
$p_{ens} \leftarrow 0$
for all (p_i, a_i) s.t. $p_i \in \mathcal{P}$ and $a_i \in \mathcal{A}$, **do**
  $W_i \leftarrow \log[a_i/(1 - a_i)]$ `# Weight of each model`
  if $p_i = 1$ **then**
    $sum \leftarrow sum + W_i$
  else
    $sum \leftarrow sum - W_i$
  end if
end for
if $sum > 0$ **then**
  $p_{ens} \leftarrow 1$
else
  $p_{ens} \leftarrow 0$
end if
return p_{ens}

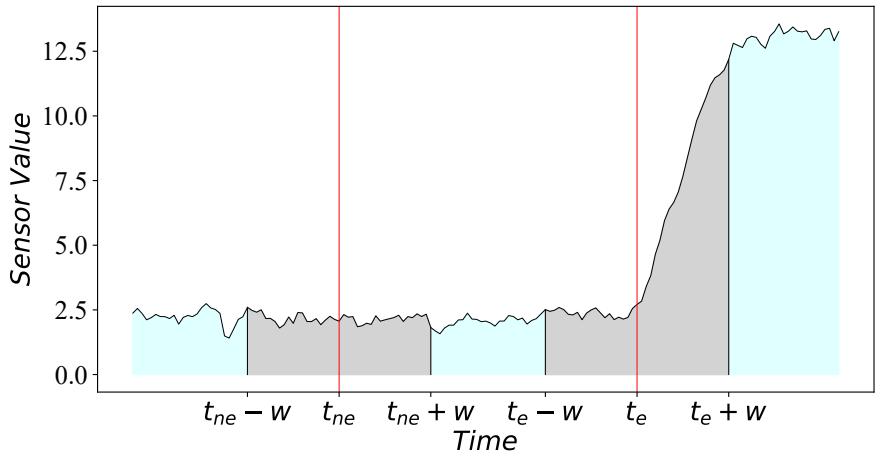


Figure 4.6: Sensor readings of a co-located sensor S_i at two timestamps (indicated by the red vertical lines), non-event timestamp (t_{ne}) and event timestamp (t_e), as well as their respective event detection windows $[t_{ne} - w,\ t_{ne} + w]$ and $[t_e - w,\ t_e + w]$ (shaded in gray).

window of a co-located sensor at timestamp t and detects if t is an *event* or *non-event* timestamp. This should use as few computations as possible to keep it lightweight, as this step is performed

Algorithm 2 Algorithm to select the best event/non-event timestamp discriminator function

Input:

$\mathcal{L} \leftarrow$ list of event detection window length

$\mathcal{F} \leftarrow$ list of candidate event detecting functions

$\mathcal{T}_e \leftarrow$ list of event timestamp

$\mathcal{T}_{ne} \leftarrow$ list of non-event timestamp

Output:

$f_h \leftarrow f \in \mathcal{F}$, best event/non-event discriminator function

1: $KL - D_h \leftarrow 0$ `# highest KL-Divergence till now`
2: $f_h \leftarrow \mathcal{F}[0]$ `# function with the highest KL-Divergence`
3: **for all** f in $\mathcal{F}$ **do**
4: $KL - D \leftarrow D_{KL}(f(T_e), f(T_{ne}))$
5: **if** $KL - D > KL - D_h$ **then**
6: $f_h \leftarrow f$
7: $KL - D_h \leftarrow KL - D$
8: **end if**
9: **end for**
10: **return** $KL - D_h$

very frequently. We call this function the *event detection function* (f_{ed}). The timestamp of an event of a target event sensor (S_E) is detected using *event detecting sensor* (S_{ed}), which is a co-located sensor of S_E with the highest accuracy to predict its event. We examine the effects of different window sizes to choose the best value as explained in Section 6.4.1.

(i) Selecting Event Detection Function: The objective of this step is to select the best event detection function (f_{ed}) from several candidate functions. To that end, we use the features identified in the previous step *minimum*, *maximum*, *standard deviation*, *skewness*, *kurtosis*, and *difference* (i.e., the absolute difference between first and last values) as the candidate functions. We then evaluate the kernel density estimation [76, 77] for each of the candidate functions for the sets of event timestamps (T_e) and non-event timestamps (T_{ne}) using event detecting sensor (S_{ed}) as shown in Chapter 6. We find that the *standard deviation* and *difference* are the most discriminating functions between the event and non-event timestamps. This is also confirmed by their high Kolmogorov-Smirnov Score [78] of 0.96, which determines the difference or similarity of distributions of two random variables. The results of our experiments are shown in Chapter 6. Finally, we select the *difference* as the *event detection function* (f_{ed}) due to its lower time complexity, i.e., $\mathcal{O}(1)$ for the difference

and $\mathcal{O}(n)$ for the standard deviation where n is the number of data-points in the *event detecting window*.

(ii) Selecting Boundary Value for Event Detection Function: We select a boundary value for the *event detection function* to classify event and non-event timestamps. We observe that when there is no event occurring in the event source (S_T), the readings of the nearby event detecting sensor (S_{ed}) are uniform, which means that the average of the *difference* function for non-event timestamps (μ_{ne}) will be close to zero. However, the value of sensor data changes abruptly when an event occurs, which means that the average of the *difference* function for event timestamps (μ_e) is much higher than zero. These observations can be found in Section 6.5.1. Thus, we can use the *mean* of μ_{ne} and μ_e, i.e., $\mu = \frac{\mu_{ne}+\mu_e}{2}$, to separate the event and non-event timestamps, i.e., t is an event-timestamp, if $f_{ed}(s_{ed}(t)) > \mu$; otherwise it is a non-event timestamp.

4.4 Verification

This section describes how SensAudit uses the relationships learned in the earlier section to verify the reported sensor data.

4.4.1 Monitoring Co-located Sensors

To detect unreported events of an event sensor (S_E), we continuously monitor its event-detecting sensor (S_{ed}). At each timestamp (t) the event detection function (f_{ed}) is computed for S_{ed} (i.e., $f_{ed}(S_{ed}(t))$. If its value is greater than μ (i.e.,$f_{ed}(S_{ed}(t) > \mu$) and no event is reported by the event sensor S_E within period of $[t-w, t+w]$, then t is considered as a probable masked event timestamp. Then, further event verification is performed for this timestamp.

4.4.2 Event Verification

We describe the steps for event verification as follows.

(i) Predicting Events from Timestamp. This step is to predict the occurrence of an event at a given

timestamp. The timestamp (t) is either obtained from a reported event or from the previous monitoring step for unreported events. Each of the features ($F_i(T^-) + F_i(T^+)$) are fed into their corresponding models (M_i) which in turn produces their corresponding predictions (p_i). These prediction results are combined as described in Section 4.3.2 to produce an ensemble prediction (p_{ens}), as detailed in the example below.

Example 1. Suppose there are three sensors, *accelerometer*, *magnetometer* and *gyroscope*, with their respective models $\{M_{acc}, M_{mag}, M_{gyro}\}$ to predict the `door` event (e) with accuracies $\{a_{acc} = 0.7, a_{mag} = 0.8, a_{gyro} = 0.9\}$ and their predictions $\{p_{acc} = 1, p_{mag} = 0, p_{gyro} = 1\}$ for the `door` event. The weight of each model is calculated by $W_i = \log[\frac{a_i}{1-a_i}]$: $W_{acc} = \log[\frac{0.7}{(1-0.7)}] = 0.847$, $W_{mag} = \log[\frac{0.8}{1-0.8)}] = 1.38$ and $W_{gyro} = \log[\frac{0.9}{(1-0.9)}] = 2.19$. $W_{acc} = 0.847$, $W_{mag} = 1.38$ and $W_{gyro} = 2.19$. We then calculate the sum of weights, $\sum W_{pred=1} - \sum W_{pred=0} = 0.847 - 1.386 + 2.197 = 1.658 > 0$. The ensemble prediction (p_{ens}) is one, meaning that the predicted event is `door-open`.

(ii) Matching Reported and Predicted Events. The predicted event (E_{pred}) is compared with the reported event (E_{rep}). If both indicate the same event, the reported event is considered as *correct*, otherwise, it is flagged as *incorrect*. To further decide whether an incorrect event is a spoofed or a masked event, we utilize our defined event/non-event timestamps. Thus, if $E_{rep} = no\_event$ but $E_{pred} = some\_event$ then the predicted event (E_{pred}) is considered to be *masked* event, whereas if $E_{rep} = some\_event$ but $E_{pred} = no\_event$ then the E_{rep} is considered to be a spoofed event.

4.4.3 Sensor Reading Verification

In this step, the value of the sensor readings of a given target sensor (S_T) is predicted using the sensor readings of its correlated sensors (CS_{S_T}) as input as shown in Figure 4.7. The predicted value ($S_{T,pred}$) and the reported value ($S_{T,rep}$) of the target sensor (S_T) are compared and if their absolute difference is more than a threshold value (β) (i.e., $|S_{T,pred} - S_{T,rep}| > \beta$) then the reported value is considered to be incorrect. For each sensor, S_i, the value of threshold β_i can be adjusted on the basis of how much error a user may tolerate and should be tuned as per the individual use cases as described in Section 6.3.1.

Example 2. Suppose there are four co-located sensors namely, *magnetometer*, *light* sensor, *temperature* sensor and *pressure* sensor, i.e., CSG=$\{mag, light, temp, press\}$ as shown in Figure 4.7 and we aim to predict the value of the *pressure* sensor (i.e., $S_T = S_{press}$). First, we calculate the correlation between *pressure* and each of the remaining co-located sensors as follows: $\rho(press, mag) = 0.6$, $\rho(press, light) = 0.7$, and $\rho(press, temp) = 0.8$. Suppose that the maximum number of correlated sensors to be used for training (n) is two. Then, the set of its correlated sensors are *light* and *temperature*: $CS_{press} = \{light, temp\}$. Thus, the ML model is trained using correlated sensors $CS_{press} = \{light, temp\}$ as input, and then it is used to predict $press$ as output ($S_{T,pred}$). If $|S_{T,pred} - S_{T,rep}| > \beta$) then the reported value ($S_{T,rep}$) is considered to be incorrect.

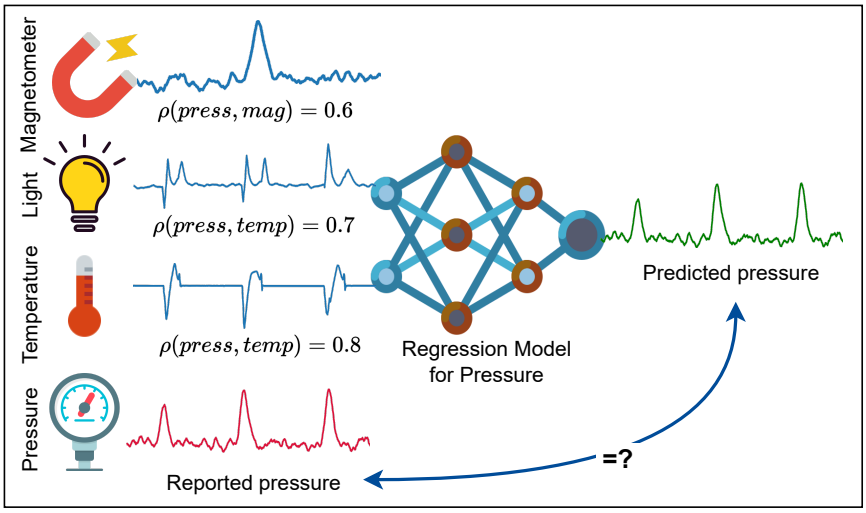


Figure 4.7: An example for the prediction of *pressure* sensor (S_T) readings utilizing the regression model and its three co-located sensors. Note that only correlated *light* and *temperature* sensors are used for regression, as $n = 2$.

Chapter 5

Implementations

This section describes the architecture of SensAudit, its experimental setups, and how it is actually implemented.

5.1 System Architecture

The high-level architecture of SensAudit is shown in Figure 5.1. SensAudit consists of four major components: data collection and preprocessing engine, learning engine, verification engine, and dashboard and reporting engine. The *data collection and preprocessing engine* collects and preprocesses the sensor data and it has three parts; *data collector* collects the data from the sensors, *data preprocessor* does the required preprocessing of the data and the *feature extractor* extracts features for each reported events using the co-located sensors of the event sensor. The *learning engine* learns the relationships between the sensors in a machine learning model. It has three parts; *sensor-sensor relationship learner* learns the relationship between a sensor and its co-located sensors in an LSTM neural network model, *event-sensor relationship learner* learns the relationship between an event sensor and its co-located sensors in a feed-forward neural network (FFNN) model, and the *event detection learner* learns the parameters to detect the event of an event sensor using the sensor reading of its co-located sensor. The *verification engine* utilizes the relationships learned by the learning engine to verify the reported events and sensor readings. It has three parts; *sensor reading verifier* verifies the reported sensor readings, *reported event verifier* verifies the reported events

and the *masked event detector* detects the unreported or masked events of an event sensor using its co-located sensors. The *dashboard and reporting engine* acts as an interface for SensAudit through which the user can interact with SensAudit. It provides a dashboard, reports, and alerts to the user.

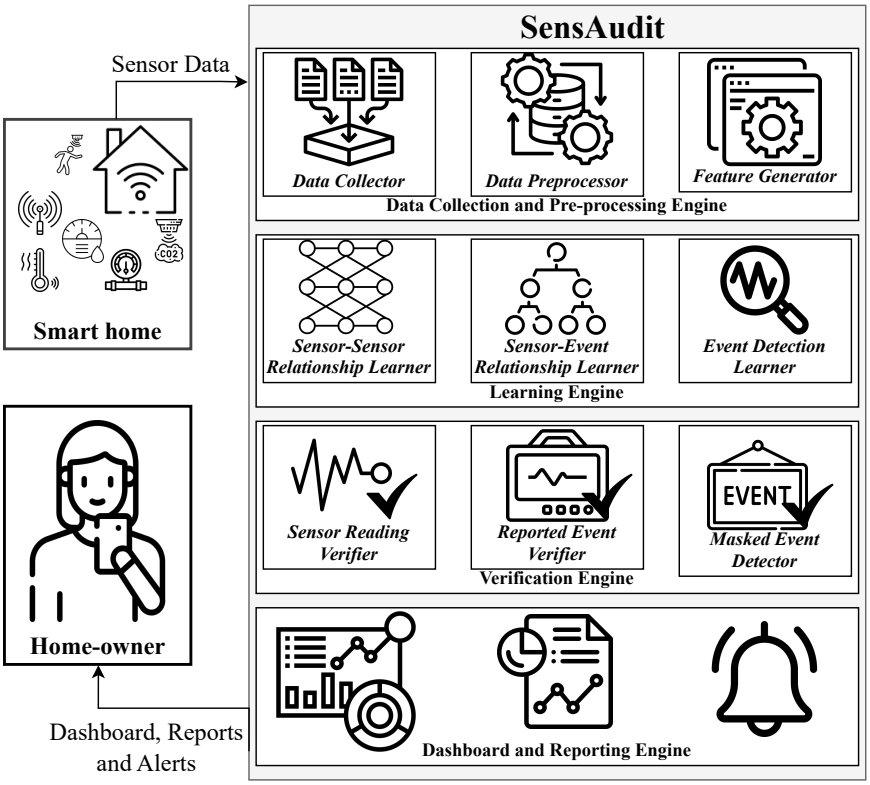


Figure 5.1: High-level architecture of SensAudit.

5.2 Data Collection and Preprocessing Engine

This section discusses how we implemented the data collection and preprocessing engine of SensAudit.

5.2.1 Co-located Sensor Reading Extraction

To conduct the experiments, we utilize the PEEVES real-world dataset* obtained from PEEVES research [16] containing *sensor data* from 48 sensors and event data from 12 different *event sources* and 12 Raspberry Pis collected over the period of 12 days in an office environment. Physically

*PEEVES dataset homepage

Pi	Event Source (# of Events)	Sensor Module	Sensors								
			A	G	P	T	L	H	M	Q	S
Pi3	Window (88)	MPU6050	•	•							
		BME680			•	•		•			
		TSL2560					•				
		SGP30								•	
		USB microphone									•
Pi5	Fan (748)	MPU6050	•	•							
		TSL2560					•				
		BMP280			•	•					
		USB microphone									•
Pi7	Light (102)	MPU6050	•	•							
		BME680			•	•		•			
		TSL2560					•				
		SGP30								•	
		USB microphone									•
Pi8	Door (680)	ST Micro LPS25H			•	•					
		ST Micro HTS221				•		•			
		ST Micro LSM9DS1	•	•					•		
Pi10	Fridge Door (112)	MPU6050	•	•							
		BME680			•	•		•			
		TSL2560					•				

Table 5.1: PEEVES Dataset including the Raspberry Pi boards, event sources, sensor modules connected to the board, and sensors. In the 'Sensor' column, A, G, P, T, L, H, M, Q, and S refer to the accelerometer, gyroscope, pressure, temperature, light, humidity, magnetometer, air quality, and sound sensors, respectively.

closely-located sensors are connected to the same Raspberry Pi board; hence, the sensors on the same Raspberry Pi board are considered to be *co-located sensors*. A subset of the PEEVES dataset (which includes event notifications and their corresponding co-located sensors) used in our experiments including the target event sensor which is to be verified, the Raspberry Pi board that it is connected to, and the sensor modules that are connected to the same Pi board are shown in Table 5.1. The Raspberry Pi_8 senor data also includes three derived values (*yaw*, *pitch*, *roll*) that are utilized in our experiments. We exclude those Raspberry Pis that do not have event sensor data from our experiments.

The data of a *sensor module* is split into hourly segments and saved as *csv* files in one folder, with filenames in *"YYYYMMDDhhmmss.csv"*. For each *.csv* file, the first column is the timestamp in *"YYYY-MM-DD hh:mm:ss.mmm"* format and the remaining columns are the sensor data (as shown

in Figure 5.2). Events are saved in a separate *.csv* file with the timestamp and event status column with binary event status.

timestamp	temperature	pressure	magnetometer	accelerometer	gyroscope
20190404 04:36:02.728	36.2705	988.9197	-32.4472	-0.9856	-0.0012
20190404 04:36:02.805	36.2538	988.9167	-32.4419	-0.9839	0.0006
20190404 04:36:02.881	36.3041	988.9214	-32.2801	-0.9841	-0.0024
20190404 04:36:02.958	36.2034	988.9194	-32.3792	-0.9844	0.0015
20190404 04:36:03.034	36.2538	988.9141	-32.5463	-0.9854	-0.0003
20190404 04:36:03.111	36.2538	988.9114	-32.2808	-0.9849	0.0018
20190404 04:36:03.187	36.1866	988.9124	-32.3443	-0.9841	-0.0012
20190404 04:36:03.263	36.1866	988.9128	-32.2794	-0.9834	0.0028
20190404 04:36:03.340	36.2705	988.9146	-32.2055	-0.9844	0.0006
20190404 04:36:03.416	36.2538	988.9153	-32.0628	-0.9856	0.0012
20190404 04:36:03.493	36.1363	988.9155	-32.4782	-0.9846	0.0006
20190404 04:36:03.647	36.1866	988.9111	-32.523	-0.9841	0
20190404 04:36:03.723	36.1866	988.9138	-32.0263	-0.9859	-0.0024
20190404 04:36:03.803	36.2538	988.9106	-32.3409	-0.9859	-0.003
20190404 04:36:03.878	36.1866	988.9231	-32.7413	-0.9841	0
20190404 04:36:03.955	36.2034	988.9136	-32.7651	-0.9849	0.0006
20190404 04:36:04.031	36.2202	988.915	-32.8935	-0.9854	0.0009
20190404 04:36:04.107	36.237	988.9175	-32.8639	-0.9846	-0.0003

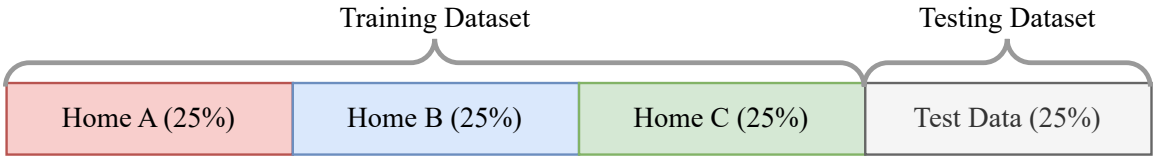
Table 5.2: Example of different sensors, their data, and timestamps in a *csv* file.

Although we use PEEVES dataset to conduct our experiments, to demonstrate how we can collect the data from multiple devices in a real smart home scenario, we created a setup using several sensor/actuator devices (such as a Google home speaker, TP-Link smart-plugs and a smartphone), Home-Assistant [56] and its InfluxDB add-on [79] to collect data from multiple devices (similar to as shown in Figure 3.1) and store them in *csv* format.

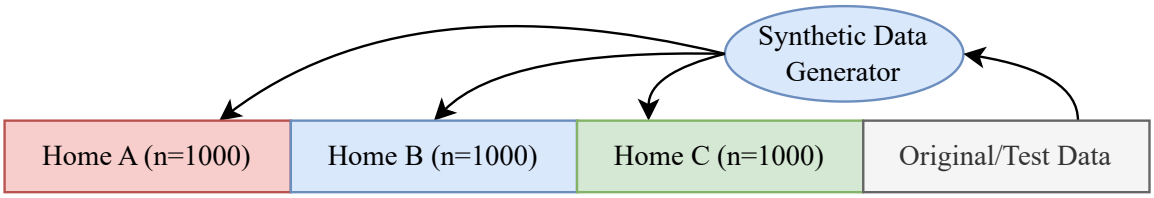
5.2.2 Dataset Division

To perform training and testing of models for sensor readings verification, the data collected during the learning phase is divided into 75% of train data and 25% of test data. The train data is further partitioned equally into three train sets for three smart homes (A, B, and C). For the event data verification, as the number of events or data points in our dataset is already low (as presented in Table 5.1), if we divide that data into three smart home training data and a test data, then the number of data-points per home would not be sufficient for the training or testing purposes. Thus, we leverage the Synthetic Data Vault (SDV) framework and generate a synthetic training dataset for

smart homes A, B, and C using the original data as described in Figure 5.2. The code snippet for generating the synthetic data is shown in Appendix A.1.3. We then use the original data as the test set.



a) Data division for sensor readings verification: Splitting the original data into four partitions representing three smart homes and one test data



b) Dataset division for event verification: Generating a synthetic training dataset with n datapoints for each of the three smart homes from the original data and using the original data as the test set

Figure 5.2: Dataset divided for the experiment during the learning phase.

5.2.3 Correlated Sensor Identification

Choosing the Number of Correlated Sensors (n). To select the number of correlated sensors (n) to be used as input to predict the sensor reading of a target sensor, we predict its value by iterating n from 1 to the total co-located sensors and compute the R2 score of prediction for each n. The minimal n after which R2-score begins to saturate is selected as the final n. An example of selecting n for the sensors in Pi_8 is shown in Figure 5.3. As seen, after $n = 5$, the R2-score begins to saturate, and adding another sensor does not increase the average performance (the mean value) of our model. Thus, we select the value of n as 5.

5.3 Learning Engine

This section presents the implementation details of the learning engine of SensAudit.

Description of Used Models. For event notification verification, we use a typical feed-forward neural network (FFNN) with fully connected layers and the *Cross_Entropy* loss function. FFNN was

chosen for event notification verification as FFNN is simple and lightweight, its learning process is supervised, and the input for the model (i.e., the features of events) is neither sequential nor time-dependent [80]. For the sensor readings verification, we utilize an LSTM neural network with the *MSE* loss function. LSTM was chosen for the sensor reading verification, as the sensor reading of a sensor is sequential and time-dependent and LSTM is better for predicting time-series data [81]. In both models, the *Adam* optimizer with *lr=0.01*, *betas=(0.9,0.999)*, *eps=1e-8*, *weight_decay=0*, *amsgrad=False* is used. The details of both models are listed in Table 5.3 and the Python source code of the models are presented in Code Snippets in Appendices A.1.1 and A.1.2.

FFNN Model		
#	Layers	Options
1	Fully Connected	#Output=32, Activation=ReLU
2	Fully Connected	#Output=32, Activation=ReLU
3	Linear Output Layer	#Output=2, Activation=Linear

LSTM Model		
#	Layers	Options
1	LSTM	#Output=16
2	LSTM	#Output=16
3	Linear Output Layer	#Output=10

Table 5.3: Details of the FFNN and LSTM Models

5.4 Verification Engine

This section presents the implementation details of the verification engine of SensAudit.

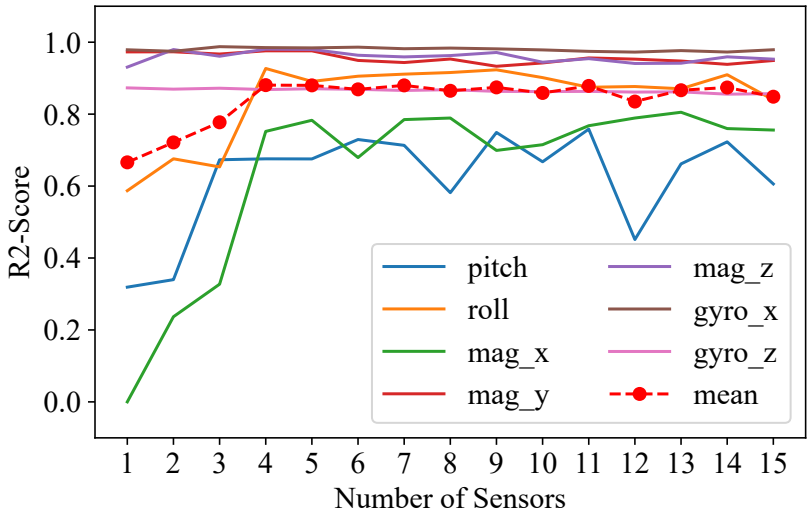


Figure 5.3: Selecting the number of correlated sensors to use for sensor-sensor relationship learning for Pi_8 raspberry pi.

Evaluation Metrics. To evaluate the performance of event detection and event verification, we use the accuracy, precision, recall (True Positive Rate), and F1-Score, and to evaluate the performance of sensor data verification we use R2-score metrics.

$$precision = \frac{TP}{TP+FP} \qquad recall = \frac{TP}{TP+FN}$$
$$accuracy = \frac{TP+TN}{TP+TN+FP+FN} \qquad f_1 = \frac{2 \cdot precision \cdot recall}{precision+recall}$$

where TP indicates *true positive*, i.e., the number of correctly identified events, TN indicates *true negative*, i.e., the number of correctly identified non-events, FP indicates *false positive*, i.e., the number of incorrectly identified events, and FN indicates *false negative*, i.e., the number of incorrectly identified non-evets.

Chapter 6

Experimental Result

This chapter presents our experimental results for sensor readings verification, event verification, and event detection.

6.1 Experimental Setup

SensAudit is implemented on a Windows 10 host machine with an Intel i7 10700 CPU and 32 GB of memory. The *Docker* 4.17 is used to containerize the development environment. *Python* 3.8.10 is used for coding. *PyTorch* 1.13.1 is used to implement machine learning. *Flower* 1.3.0 library is used for federated learning. *sdv* 0.18.0 is used to generate synthetic data. *matplotlib 3.7.1* and *seaborn 0.12.2* are used to plot figures. *scikit-learn 1.2.2* is used to compute performance metrics. *Pandas 1.5.3* and *Numpy 1.24.2* for data manipulation. We implemented our learning steps (as discussed in Section 4.3) leveraging three methods: isolated, centralized, and federated. *Isolated learning* is where the model for each smart home is trained individually. *Centralized learning* is where one model is built by combining historical sensor data from multiple smart homes. *Federated learning* is where the model for each smart home is built by aggregating the learning results from multiple smart homes.

6.2 Evaluation Objectives

This section describes research questions whose answer is used to evaluate the performance of SensAudit. The research questions are as follows:

- RQ1: How well does our solution predict and verify incorrect sensor readings, and detect masked and spoofed events? How does it compare to existing solutions in this regard?
- RQ2: How does the performance of isolated, centralized, and federated learning methods compare with each other for incorrect sensor reading verification, and masked and spoofed event detection?
- RQ3: What are the additional benefits of federated learning in terms of accuracy? How much extra overhead does it introduce?

6.3 Sensor Reading Verification

This section discusses the experimental results of the sensor reading verification.

6.3.1 Choosing the Value of Tolerance Threshold (β)

To choose the value of the tolerance threshold (β), we plot the β vs. $accuracy$ plot as shown in Figure 6.1. If the chosen β is too high there will be too many false negatives (i.e., largely incorrect sensor reading identified as correct), whereas if it is too low there will be too many false positives (i.e., almost correct sensor reading identified as incorrect). Therefore, we need to select the minimum value of β without getting many false positives or giving up the accuracy of the detection.

6.3.2 Sensor Reading Verification Results

To measure the performance of the regression model in predicting sensor readings, we measure the R2 score between the reported and predicted values, as it is more informative than other metrics for regression analysis evaluation [82]. Figures 6.2 and 6.3 show the R2 scores between the reported

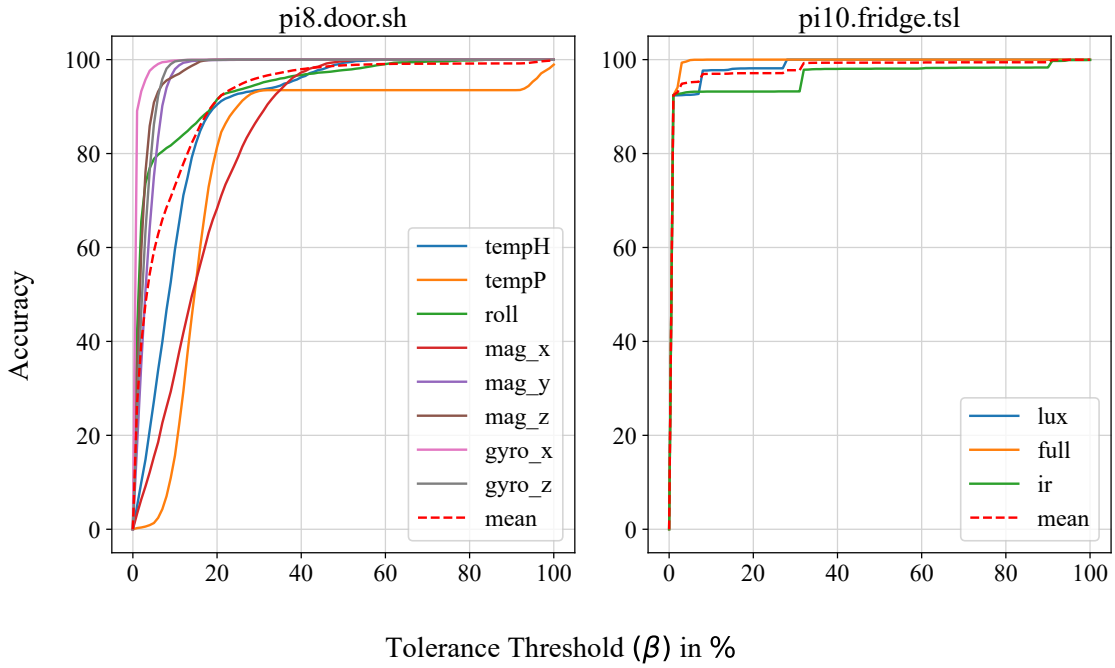


Figure 6.1: The accuracy of the sensor readings verification module with varying β

and the predicted values for various co-located sensors attached to the Pi_8 and Pi_{10} Raspberry Pis, respectively. We observe the highest R2 score of 0.98 in Figure 6.2 and 1.0 in Figure 6.3; which demonstrates the benefit of using these sensors in our verification.

6.4 Event Verification

This section discusses the experimental results of the event verification.

6.4.1 Event Window Length Selection (w)

To examine the effects of event window length (w) for event verification, we vary the length of windows starting from 0.5 to 5 seconds (using incremental steps of $0.5s$) and measure the accuracy for each of the values of the window length. An example of it for the co-located sensors on the Raspberry Pi_8 is shown in Figure 6.4. The window length with the highest accuracy is chosen as the final event window length; if multiple window lengths have the same highest accuracy, the longest window length is chosen for w as the final event window length. For instance, a window size of 3 is

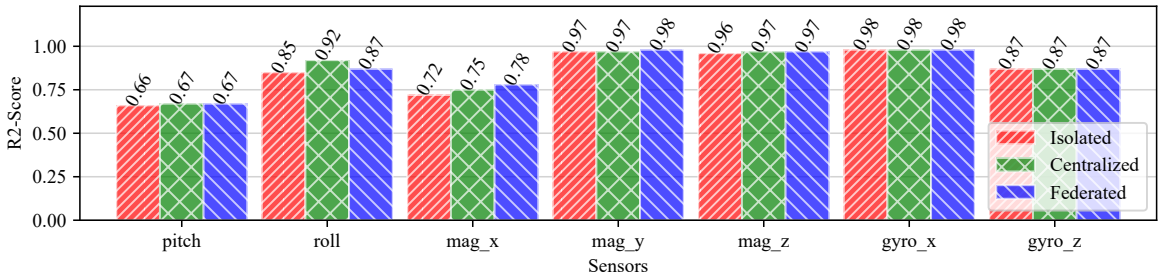


Figure 6.2: R2 score of reported data and the predicted values for various sensors of Pi_8.

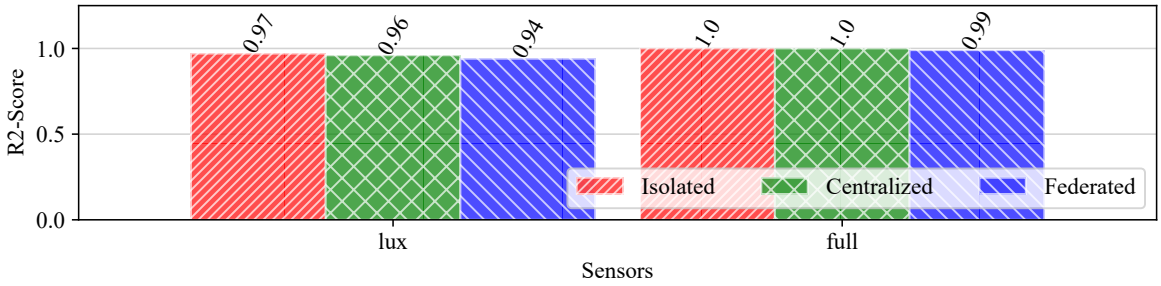


Figure 6.3: R2 score of reported data and the predicted values for various sensors of Pi_{10}.

chosen for the *yaw* sensor, while the window size of 1 is selected for the *acc_y* sensor.

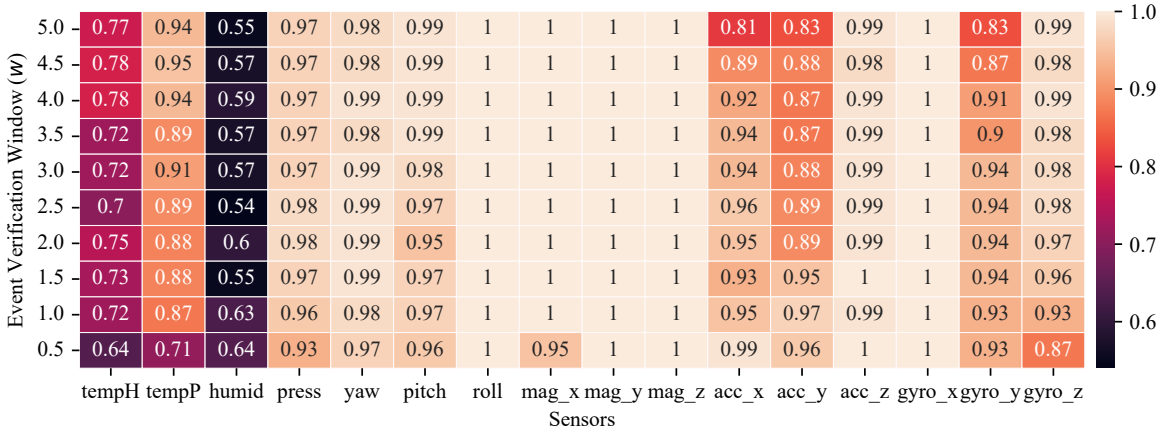


Figure 6.4: Impact of different event window length (w) on the accuracy.

6.4.2 Event Verification Results

To evaluate the accuracy of our event verification (i.e., spoofed event detection) solution, we use `door`, `window`, `fan`, `fridge`, and `light` events as our target events. For an event source

in a CSG, we first predict its events using the trained model of each co-located sensor and then, we perform ensemble learning using all the models having an accuracy of more than 0.5. This process is performed for isolated, centralized, and federated learning models. We run each of the experiments ten times and the average accuracy of each individual co-located verification sensor in predicting the value of the target sensor and the accuracy of the ensemble prediction is calculated. Obtained results for both individual and ensemble learning are presented in Table 6.1. As seen, in most cases, isolated learning shows lower accuracy than the other two methods and the centralized method obtains slightly higher accuracy than federated. Moreover, in some cases, one sensor data is sufficient to detect events (e.g., *mag_y* sensor to detect `door` event). We further measure the f1-score and obtain similar results. In what follows we provide our insights on the obtained results.

Door Event. The *accelerometer*, *gyroscope*, *magnetometer* and rotation sensors (*yaw*, *pitch*, *roll*) can detect `door` events, as shown in Figure 6.5. This is expected as these sensors are attached to the door itself; whenever the door is opened or closed, all these sensors capture any related effects resulting from the physical movements and orientation changes of the door. *Pressure* sensor can also predict the `door` event which implies that opening and closing the door changes the pressure inside the room. *Temperature* sensor predicts the `door` event with lower accuracy than aforementioned sensors. Opening the door brings in outside air which might be of different temperatures; therefore it can predict `door` events. However, as the changes in temperature take some time (whereas for other sensors changes are seen instantly), and the outside-inside temperature difference can be high sometimes but low other times, therefore the accuracy to predict `door` events might have been lower. Some sensors such as *humidity* sensor were not able to predict `door` events. This implies that opening or closing the door does not change the humidity of the room instantly, which might be due to the fact that the humidity outside the room was the same as inside, or even if it was different it would take some time much longer than our event window.

Window Event. The *sound* sensor (*rms* and *rms_db*), as shown in Figure 6.6), is the most reliable predictor of the `window` events with an accuracy of around 99%, as shown in Figure 6.6. It may have been the case that the window produced sound while being opened and closed, or it can also be the case that the outside was much noisier than inside the room. The *light* sensor was also able

Door	tempH	tempP	press	yaw	pitch	roll	mag_x	mag_y	mag_z	acc_x	acc_y	acc_z	gyro_x	gyro_y	gyro_z	Ensem
Isolated	0.5	0.5	0.5	0.74	0.51	0.65	0.69	1	**0.99**	0.75	**0.81**	**0.92**	0.99	0.9	0.78	0.98
Centralized	**0.75**	**0.84**	**0.96**	0.87	**0.94**	**0.99**	1	1	**0.99**	0.73	0.8	**0.92**	0.99	0.81	0.76	1
Federated	0.65	0.74	0.91	**0.94**	0.86	**0.99**	0.99	1	**0.99**	**0.8**	0.79	0.9	1	**0.91**	**0.85**	1
Window	temp	humidity	acc_x	acc_y	acc_z	gyro_x	gyro_y	gyro_z	temp	co2	voc	rms	rms_db	lux	full	Ensem
Isolated	**0.94**	**0.81**	0.89	0.89	**0.93**	0.74	**0.94**	**0.8**	0.55	**0.81**	0.8	0.97	0.98	0.69	**0.82**	0.99
Centralized	0.89	0.76	**0.9**	0.89	0.85	**0.82**	0.92	0.76	**0.77**	0.72	**0.82**	**0.99**	**0.99**	**0.81**	0.73	1
Federated	0.93	0.74	0.89	**0.9**	0.88	**0.82**	0.92	0.78	0.62	0.76	0.81	**0.99**	**0.99**	0.76	0.8	0.99
Fridge	temp	press	humidity	acc_z	gyro_x	gyro_y	gyro_z	lux	full	ir	Ensem					
Isolated	0.91	1	**0.97**	0.96	0.6	0.9	**0.98**	**0.99**	**0.99**	0.96	1					
Centralized	**0.98**	1	0.96	**0.97**	0.74	**0.95**	0.95	**0.99**	**0.99**	**0.97**	1					
Federated	0.96	1	**0.97**	0.96	**0.76**	0.9	0.97	**0.99**	**0.99**	**0.97**	1					
Fan	temp	rms	rms_db	Ensem	**Light**	lux	full	Ensem								
Isolated	0.47	0.84	0.88	0.85	Isolated	0.77	0.82	0.64								
Centralized	**0.72**	**0.88**	0.9	**0.89**	Centralized	0.87	**0.89**	0.54								
Federated	0.61	0.86	0.88	0.87	Federated	0.91	0.84	**0.75**								

Table 6.1: Comparing the accuracy results of different event detection models for each event and their co-located sensors. Column "Ensem" means ensembled. The values in bold show the highest accuracy per sensor. The cells shaded in grey represent the highest accuracy for each event.

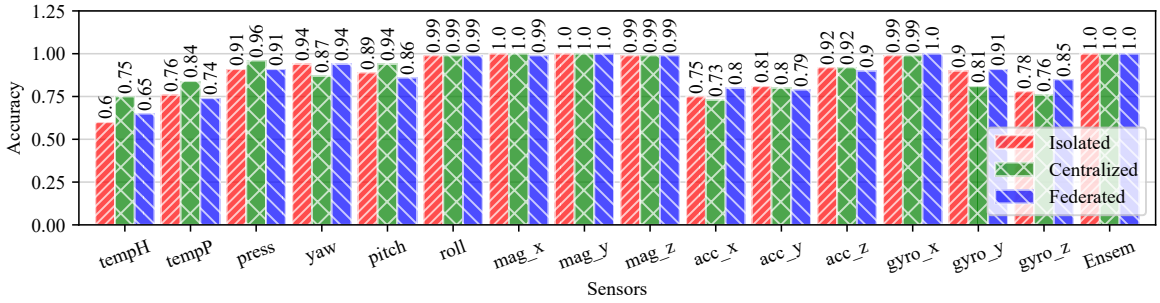


Figure 6.5: Comparing the accuracy of predicting `door` events using isolated, centralized and federated learning methods for each of the co-located sensors and the ensemble methods. In most cases, isolated learning shows significantly lower accuracy than the other two methods, and the centralized method obtains slightly higher accuracy than the federated.

to predict the `window` event with around 70%-80% accuracy. It might be the case that opening the window let more light come in from outside. The carbon dioxide (CO_2) and the volatile organic compound (*voc*) sensors were able to predict the `window` event with around 70% and 80% accuracy, respectively. One possible explanation could be the difference in the amount of those gases inside and outside the room. *Temperature* sensor in BME sensor module is able to predict the `window` events with higher accuracy ($\sim$90%) than that of temperature sensor in MPU ($\sim$70%). One possible explanation is that the BME sensor module might have been closer to the window than the MPU. Other sensors such as *pressure* and *humidity* were not able to predict the `window` event as *humidity* takes time to change a lot longer than our event window. The pressure inside and outside the room might have been the same.

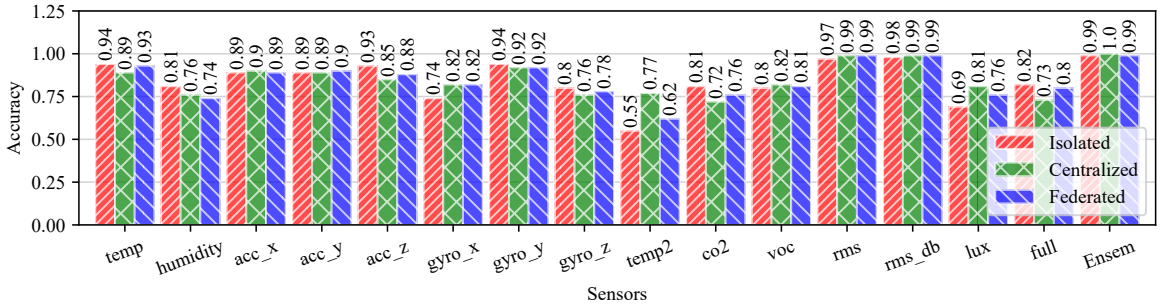


Figure 6.6: Comparing the accuracy results of predicting `window` event using each learning method for each of the co-located sensors and the ensemble method.

Fridge Door Event. Most of the sensors can predict `fridge` events with high accuracy, as shown

in Figure 6.7. For *pressure*, *temperature*, *accelerometer*, and *gyroscope* sensors, the results can be explained with reasoning the same as the `door` events. *Light* sensor (placed inside the fridge) might have been able to predict it as opening the fridge door lets lights get in from outside, and also from the light bulb inside the fridge whereas closing the door will make inside completely dark. Inside the fridge is very humid compared to outside, therefore the *humidity* sensor might have been able to pick that up while opening and closing the fridge door, which explains its high accuracy.

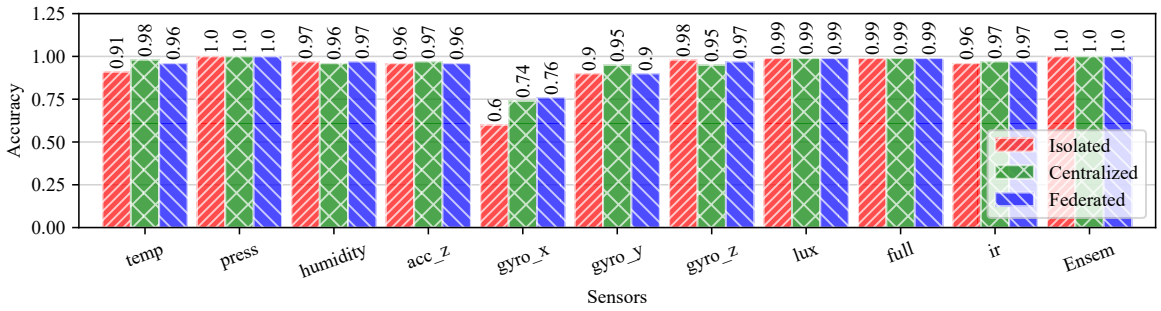


Figure 6.7: Comparing the accuracy results of predicting `fridge-door` events using each learning method for each of the co-located sensors and the ensemble method.

Fan Event. Only the *pressure* sensor was able to predict the `fan` events with higher accuracy of 90%, while the *temperature* sensor could predict `fan` events with lower accuracy, as shown in Figure 6.8. Other sensors were not able to detect the `fan` events. This is expected as a fan produces sound and does not have an effect on other sensors.

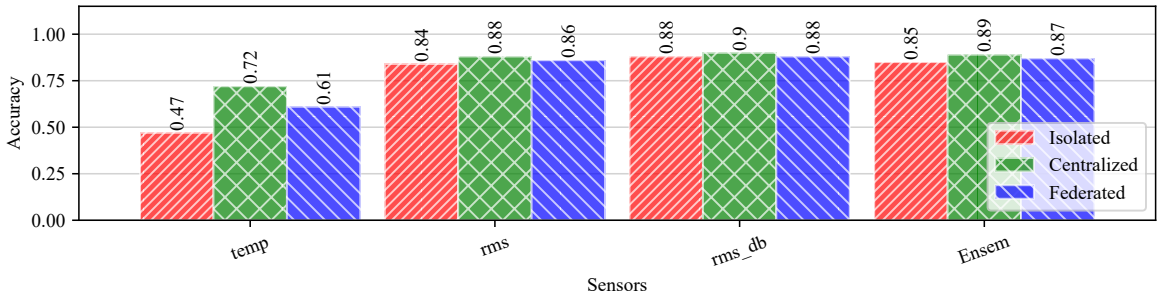


Figure 6.8: Comparing the accuracy results of predicting `fan` event using each learning method for each of the co-located sensors and the ensemble method.

Light Event. As expected, `light-on/light-off` events were reliably predicted with the light sensor with an accuracy of up to 99%, as shown in Figure 6.9.

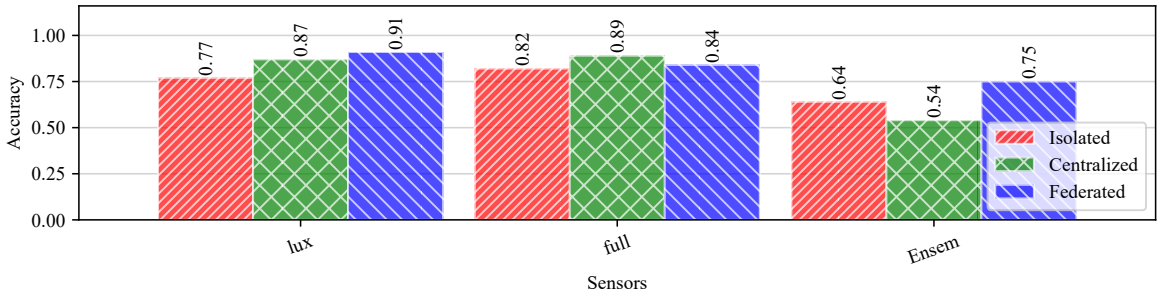


Figure 6.9: Figure comparing the accuracy of predicting `light` event using each learning method for each of the co-located sensors and the ensemble method.

6.4.3 Resource Overhead

We further compare the CPU and memory overhead of federated and non-federated (Centralized) methods for training a model for door event verification. Obtained results shown in Figure 6.10 suggest that the federated learning uses around 1.59 (for *datapoints = 100*) to 2.02 (for *datapoints = 600*) times more CPU time and both of utilizing almost the same amount of memory. as shown in Figure 6.10. With respect to the non-federated method, we use centralized because federated and centralized have the same number of data points whereas isolated has less number of data points.

6.5 Event Detection

This section discusses the experimental results of the event detection.

6.5.1 Kernel Density Estimations for Event Detection

We collect the data from the Pi_8 and calculate the KDE of each candidate function for the sets of event timestamps and non-event timestamps. As shown in Figure 6.11, the *standard deviation* and *difference* are the most discriminating functions between the event and non-event timestamps. This is also confirmed by their high Kolmogorov-Smirnov Score of 0.96. We obtain similar results for the other data in our dataset.

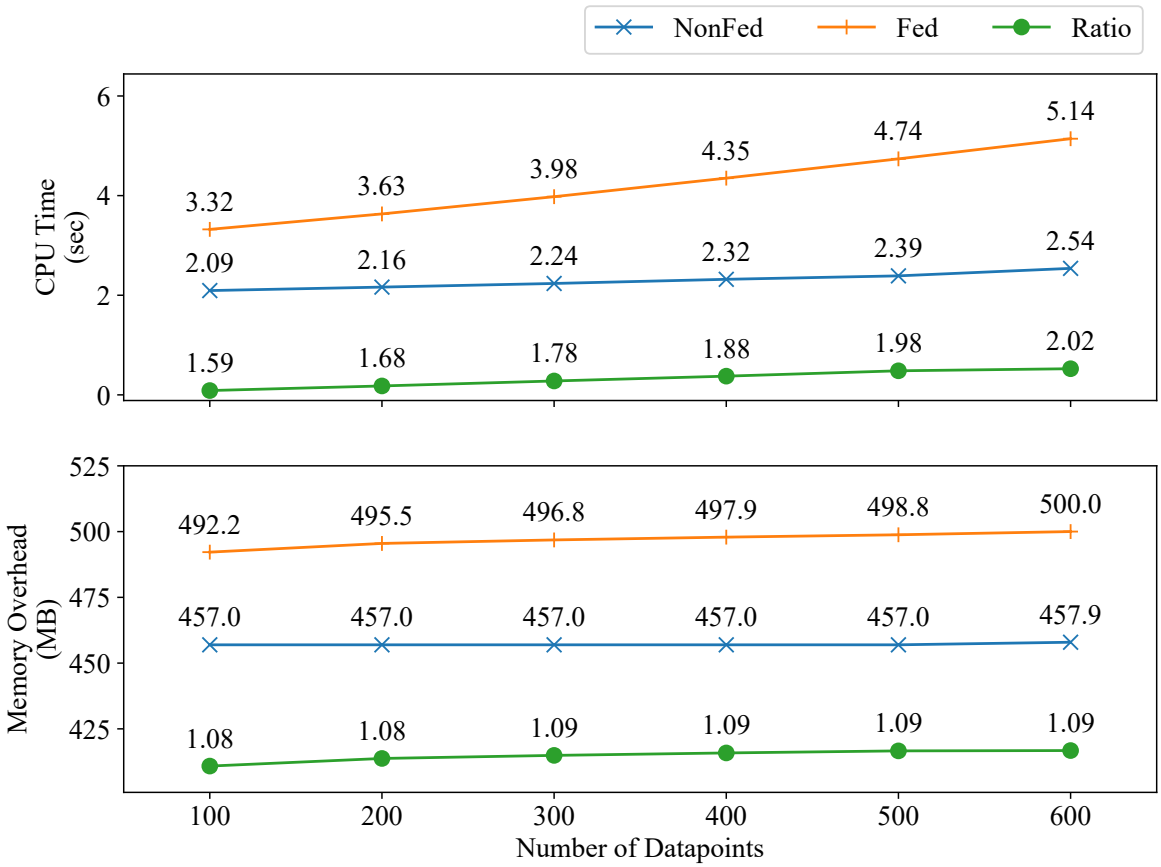


Figure 6.10: Comparing the CPU and memory overhead of federated and non-federated (centralized) methods for event verification.

6.5.2 Event Detection Results

To evaluate the performance of the event detection for an event source, we select a list of timestamps with the same number of event and non-event timestamps for that event source. For each of those timestamps, we predicted whether it is an event or a non-event timestamp using the method described in Section 4.3.3. Based on the true events and predicted events for the timestamps, the confusion matrix was created for `door`, `window`, `fan`, `light` and `fridge-door` events detection as shown in Figure 6.12. In our experiments, the timestamp when an event occurred is represented with 1, whereas a timestamp when an event did not occur is represented with 0.

We evaluate various performance metrics (accuracy, precision, recall, and f1-score). Obtained results for `door`, `window`, `light` and `fridge-door` event detection are presented in Figure 6.13 per event. As seen, we achieve a high score in all these performance metrics except for the `fan`

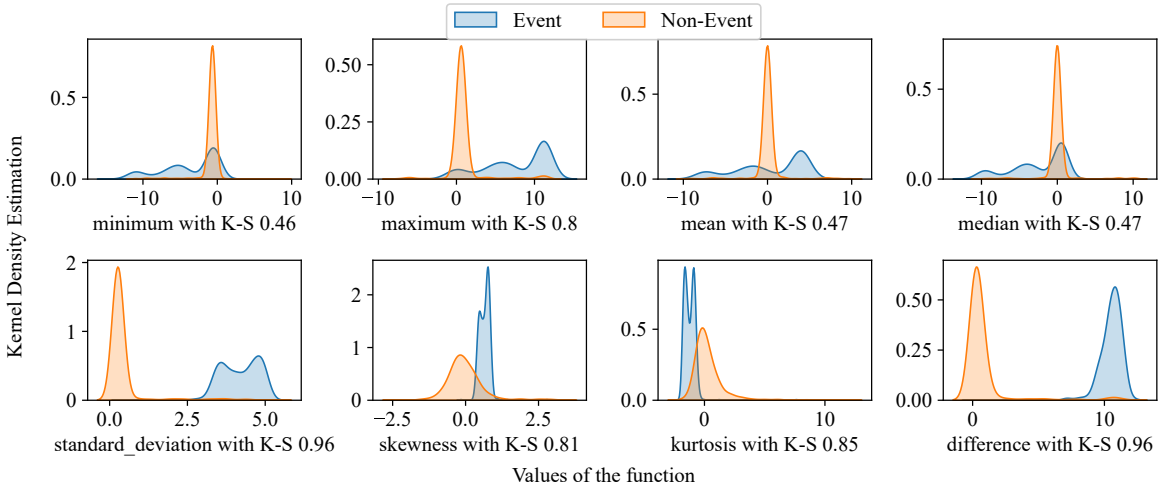


Figure 6.11: Kernel density estimation of each candidate metric for "event" and "non-event" timestamps and their corresponding Kolmogorov-Smirnov Score.

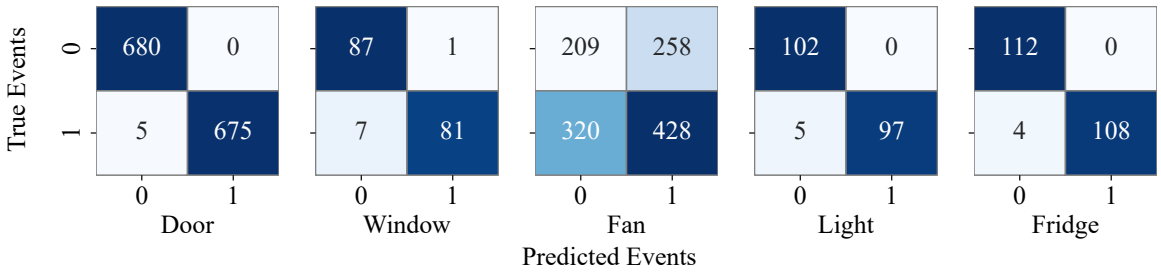


Figure 6.12: Confusion matrix for event detection for various event sources

event which achieved relatively lower accuracy due to the less effect of fan events on its co-located sensors.

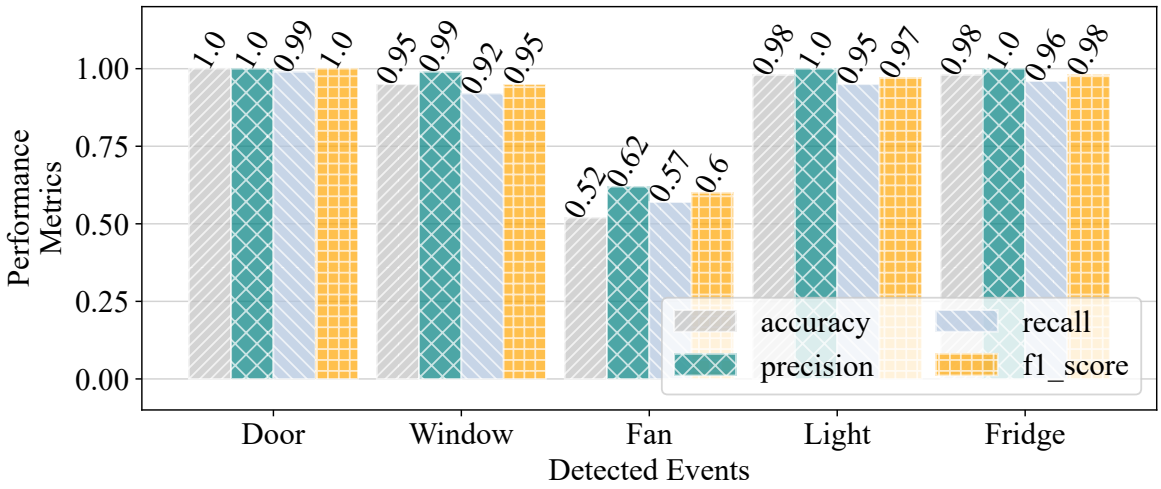


Figure 6.13: Performance metrics for the event detection of various events.

Chapter 7

Discussion

The Impact of Distance between Devices. Like most event verification systems [16, 17], our event verification step also relies on how an event affects the verification sensors physically. This effect is typically dependent on the distance between an event source and co-located sensor as the effect. The effectiveness of SensAudit (like most event verification systems [16, 17]) relies on how an event affects the verification sensors physically. If the verification sensors are far away from the event source, the effect of an event on them is reduced, and the system may not be able to verify the data. However, if two event sources are nearby, then the system can have other problems as described later. Therefore, in our future work, we intend to explore the effect of the distance between devices on SensAudit.

The Impact of Outliers and Poisonous Data. We assume that our training data is free of incorrect, poisonous, and outlier data. If the training data is poisoned by an adversary or includes outliers (e.g., opposite nature data from a smart home) or contaminated with those sorts of data, then our learning step might be affected and it can have an adverse effect on the performance of the system. One possible way to reduce this effect could be to assign weights to models based on their performance and then perform weighted aggregation of the models during the federation step, which is an area of our future work.

The Impact of Majority of Devices being Flawed or Malicious. Each co-located sensor group (CSG) acts as an independent group, which is unaffected by the sensor failures outside of that

group. For sensor readings verification, SensAudit can verify the target sensor since the majority of the correlated sensors (a smaller set of co-located sensors) are functioning properly. If more than one correlated sensor fails, then SensAudit can detect sensor failure without identifying specific sensors; in such cases, a manual inspection can be used for sensor identification. For event verification, SensAudit can tolerate the failure of multiple correlated sensors depending on the weights (as discussed in Section 4) of the failed sensors and the performance will decrease accordingly.

The Impact of Simultaneous Occurrence of Events. If two event sources are nearby then the event of one source might result in a wrong unreported event detection for the other event source, which might also allow an attacker to perform a masked attack as described in [10]. If two event sources are nearby then an attacker may also spoof an event without being detected, by spoofing the event in one event source only when there is a real event in another event source. For example, if a door and window are nearby and opening both of them produce a similar effect on a sound sensor, then a masked `door-opening` event can go undetected if SensAudit would only be using sound sensor data. Therefore, using multiple sensor readings (as long as not affected by the other event(s)) and their ensemble predictions (as discussed in Section 4.3) might help to reduce this effect.

Chapter 8

Conclusion

In this thesis, we proposed an approach to verify both sensor readings and event notifications by monitoring co-located IoT devices. To that end, we first learned the relationships among co-located devices based on various events and sensor readings. Then, we leveraged this relationship to verify the correctness of reported sensor readings and event notifications as well as detect unreported events. We implemented SensAudit in the context of smart homes and evaluated its effectiveness using the publicly available PEEVES dataset. Obtained results showed that SensAudit can detect masked and spoofed events with up to 100% accuracy and detect incorrect sensor readings with an R2 score of up to 0.99. However our approach has some limitations, for instance, it is only able to detect incorrect verification sensor readings from one sensor at a time, it is vulnerable to data poisoning during training, and its performance can be affected by the simultaneous occurrence of multiple events. In our future work, we plan to apply our method to other IoT applications such as autonomous vehicles and smart cities, where co-located sensors are more common and verification of sensor readings and event notifications is needed. Also, we plan to automate the identification of some of the parameters (e.g., window size, verification threshold), which are currently manually identified.

Appendix

A.1 Code Snippets

7/4/23, 8:21 PM printCode.py

A.1.1 FFNN Model

printCode.py

```python
class ModelC(nn.Module):
    def __init__(self, input_dim=16):
        super(ModelC, self).__init__()
        a = 32
        self.layer1 = nn.Linear(input_dim, a)
        self.layer2 = nn.Linear(a, a)
        self.layer3 = nn.Linear(a, 2)

    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = self.layer3(x)
        return x

```

Code Snippet CS.1: Python source code for the FFNN model that is used in event verification.

A.1.2 LSTM Model

printCode.py

```python
class LSTMRegModel(nn.Module):
    def __init__(self, num_sensors, hidden_units):
        super().__init__()
        self.num_sensors = num_sensors
        self.hidden_units = hidden_units
        self.num_layers = 2
        self.lstm = nn.LSTM(
            input_size=num_sensors,
            hidden_size=hidden_units,
            batch_first=True,
            num_layers=self.num_layers,
        )
        self.linear = nn.Linear(
            in_features=self.hidden_units, out_features=1
        )

    def forward(self, x):
        batch_size = x.shape[0]
        h0 = torch.zeros(
            self.num_layers, batch_size, self.hidden_units
        ).requires_grad_()
        c0 = torch.zeros(
            self.num_layers, batch_size, self.hidden_units
        ).requires_grad_()
        _, (hn, _) = self.lstm(x, (h0, c0))
        out = self.linear(hn[0]).flatten()
        return out

```

Code Snippet CS.2: Python source code for the LSTM model that is used in sensor readings verification.

A.1.3 Synthetic Data Generation

```
from sdv.single_table import GaussianCopulaSynthesizer
from sdv.metadata import SingleTableMetadata

def getSyntheticData(dataDF, num_rows=500):
    metadata = SingleTableMetadata()
    metadata.detect_from_dataframe(data=dataDF)
    metadata.update_column(column_name="timeIndex", sdtype="id")
    metadata.set_primary_key(column_name="timeIndex")
    model = GaussianCopulaSynthesizer(metadata)
    model.fit(dataDF)
    return model.sample(num_rows)

```

Code Snippet CS.3: Python source code to generate the synthetic data using sample data.

A.1.4 Training FFNN Model

```python
def train(
    model: ModelC, train_loader: torch.utils.data.DataLoader, epochs: int
) -> None:
    optimizer = torch.optim.Adam(model.parameters(), lr=Arguments.lr)
    model.to(Arguments.device)
    model.train()
    for i in range(epochs):
        for batch_idx, (data, target) in enumerate(train_loader):
            data = data.float()
            target = target.long()
            data, target = data.to(Arguments.device), target.to(
                Arguments.device
            )
            optimizer.zero_grad()
            output = model(data)
            loss = Arguments.loss_function(output, target)
            loss.backward()
            optimizer.step()

```

Code Snippet CS.4: Python source code to train the FFNN Model for the event verification.

A.1.5 Testing FFNN Model

```python
def test(
    model: ModelC, test_loader: torch.utils.data.DataLoader, currentDict
) -> Tuple[float, float]:
    model.to(Arguments.device)
    model.eval()
    combinedPrediction = torch.Tensor()
    combinedTarget = torch.Tensor()
    with torch.no_grad():
        for data, target in test_loader:
            data = data.float()
            target = target.long()
            data, target = data.to(Arguments.device), target.to(
                Arguments.device
            )
            output = model(data)
            _, prediction = torch.max(output.data, 1)
            combinedTarget = torch.cat((combinedTarget, target))
            combinedPrediction = torch.cat(
                (combinedPrediction, prediction)
            )
    correct = (combinedPrediction == combinedTarget).sum().item()
    accuracy = correct / len(combinedTarget)
    epochLoss = Arguments.loss_function(
        combinedTarget, combinedPrediction
    ).item()
    tplmDict = {
        "featureSet": currentDict[
            "featureSetName"
        ],  # this is just sensorname
        "target": combinedTarget,
        "predicted": combinedPrediction,
        "test_loss": epochLoss,
        "model": model,
    }
    saveTPLMDict(tplmDict, currentDict)
    return epochLoss, accuracy

```

Code Snippet CS.5: Python source code to test the FFNN Model for the event verification.

A.1.6 Training LSTM Model

```python
def trainLSTMReg(model: LSTMRegModel, train_loader: torch.utils.data.DataLoader, epochs: int):
    optimizer = torch.optim.Adam(model.parameters(), lr=Arguments.lr)
    model.to(Arguments.device)
    model.train()
    for epoch in range(epochs):
        combinedTarget = torch.Tensor()
        combinedPrediction = torch.Tensor()
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = Arguments.loss_function(output, target)
            loss.backward()
            optimizer.step()
            combinedTarget = torch.cat((combinedTarget, target))
            combinedPrediction = torch.cat((combinedPrediction, output))
        epochLoss = Arguments.loss_function(combinedTarget, combinedPrediction).item()

```

Code Snippet CS.6: Python source code to train the LSTM Model for the sensor reading verification.

A.1.7 Testing LSTM Model

```python
def test(
    model: ModelC, test_loader: torch.utils.data.DataLoader, currentDict
) -> Tuple[float, float]:
    model.to(Arguments.device)
    model.eval()
    combinedPrediction = torch.Tensor()
    combinedTarget = torch.Tensor()
    with torch.no_grad():
        for data, target in test_loader:
            data = data.float()
            target = target.long()
            data, target = data.to(Arguments.device), target.to(Arguments.device)
            output = model(data)
            _, prediction = torch.max(output.data, 1)
            combinedTarget = torch.cat((combinedTarget, target))
            combinedPrediction = torch.cat((combinedPrediction, prediction))
    correct = (combinedPrediction == combinedTarget).sum().item()
    accuracy = correct / len(combinedTarget)
    epochLoss = Arguments.loss_function(combinedTarget, combinedPrediction).item()
    tplmDict = {
        "featureSet": currentDict["featureSetName"],  # this is just sensorname
        "target": combinedTarget,
        "predicted": combinedPrediction,
        "test_loss": epochLoss,
        "model": model,
    }
    saveTPLMDict(tplmDict, currentDict)
    return epochLoss, accuracy

```

Code Snippet CS.7: Python source code to test the LSTM Model for the sensor reading verification.

Bibliography

[1] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. SoK: Security evaluation of home-based IoT deployments. In *IEEE Symposium on Security and Privacy*, 2019.

[2] Nancy E ElHady and Julien Provost. A systematic survey on sensor failure detection and fault-tolerance in ambient assisted living. *Sensors*, 18(7):1991, 2018.

[3] Timothy W Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The Hitchhiker's guide to successful residential sensing deployments. In *ACM Conference on Embedded Networked Sensor Systems*, 2011.

[4] Haotian Chi, Chenglong Fu, Qiang Zeng, and Xiaojiang Du. Delay wreaks Havoc on your smart home: Delay-based automation interference attacks. In *IEEE Symposium on Security and Privacy*, 2022.

[5] Dorothy N Monekosso and Paolo Remagnino. Data reconciliation in a smart home sensor network. *Expert Systems with Applications*, 40(8):3248–3255, 2013.

[6] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *IEEE symposium on Security and Privacy*. IEEE, 2016.

[7] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[8] Bin Yuan, Yuhan Wu, Maogen Yang, Luyi Xing, Xuchang Wang, Deqing Zou, and Hai Jin.

Smartpatch: Verifying the authenticity of the trigger-event in the IoT platform. *IEEE Transactions on Dependable and Secure Computing*, 20(2), 2022.

[9] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. *arXiv preprint arXiv:1811.03241*, 2018.

[10] Muslum Ozgur Ozmen, Ruoyu Song, Habiba Farrukh, and Z Berkay Celik. Evasion attacks and defenses on smart home physical event verification. In *NDSS 2023*. Network and Distributed System Security, 2023. doi: 10.14722/ndss.2023.23070.

[11] Shin-Juh Chen, David C Hovde, Kristen A Peterson, and André W Marshall. Fire detection using smoke and gas sensors. *Fire Safety Journal*, 42(8), 2007.

[12] W Ross Stone. A Tale of False Smoke Alarms [From the Screen of Stone]. *IEEE Antennas and Propagation Magazine*, 64(6):133–134, 2022.

[13] Marty Ahrens. "False Alarms and Unwanted Activations" from US Experience with Smoke Alarms and Other Fire Detection/Alarm Equipment. *National Fire Protection Association*, 2004.

[14] Marty Ahrens. *Smoke alarms in US home fires*. National Fire Protection Association, Fire Analysis and Research Division, 2009.

[15] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. HAWatcher: Semantics-aware anomaly detection for appified smart homes. In *USENIX Security Symposium*, 2021.

[16] Simon Birnbach, Simon Eberz, and Ivan Martinovic. Peeves: Physical Event Verification in Smart Homes. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2019.

[17] Simon Birnbach, Simon Eberz, and Ivan Martinovic. Haunted house: physical smart home event verification in the presence of compromised sensors. *ACM Transactions on Internet of Things*, 3(3):1–28, 2022.

[18] Amit Kumar Sikder, Leonardo Babun, Hidayet Aksu, and A Selcuk Uluagac. Aegis: A context-aware security framework for smart home systems. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[19] Amit Kumar Sikder, Leonardo Babun, and A Selcuk Uluagac. Aegis+: a context-aware platform-independent security framework for smart home systems. *Digital Threats: Research and Practice*, 2(1):1–33, 2021.

[20] Rozhin Yasaei, Felix Hernandez, and Mohammad Abdullah Al Faruque. IoT-CAD: Context-aware adaptive anomaly detection in IoT systems through sensor association. In *International Conference on Computer-Aided Design*, pages 1–9, 2020.

[21] Saurabh Ganeriwal, Laura K Balzano, and Mani B Srivastava. Reputation-based framework for high integrity sensor networks. *ACM Transactions on Sensor Networks*, 4(3):1–37, 2008.

[22] David S Bayard and Scott R Ploen. High accuracy inertial sensors from inexpensive components, April 19 2005. US Patent 6,882,964.

[23] Mahdi Jafari and Jafar Roshanian. Inertial navigation accuracy increasing using redundant sensors. *Journal of Science and Engineering*, 1(1):55–66, 2013.

[24] Thomas George Thuruthel, Josie Hughes, Antonia Georgopoulou, Frank Clemens, and Fumiya Iida. Using redundant and disjoint time-variant soft robotic sensors for accurate static state estimation. *IEEE Robotics and Automation Letters*, 6(2), 2021.

[25] Lee Han Keat and Chuah Chai Wen. Smart indoor home surveillance monitoring system using Raspberry Pi. *International Journal on Informatics Visualization*, 2(4-2):299–308, 2018.

[26] Shruti Dash and Pallavi Choudekar. IoT-Based Smart Home Surveillance System. In *Applied Information Processing Systems*, pages 417–427. Springer, 2022.

[27] Tanin Sultana and Khan A Wahid. IoTGuard: Event-driven fog-based video surveillance system for real-time security management. *IEEE Access*, 7, 2019.

[28] Lingshan Xu, Xianghan Zheng, Wenzhong Guo, and Guolong Chen. A Cloud-based monitoring framework for smart home. In *4th IEEE International Conference On Cloud Computing Technology And Science*, pages 805–810. IEEE, 2012.

[29] Ring LLC. Door Sensor reports open status when it appears to be closed, May 2020. URL https://community.ring.com/t/door-sensor-reports-open-status-when-it-appears-to-be-closed/7863.

[30] Mike Howell. Vancouver housing provider paid $63,760 in false alarm fines for one SRO. *Vancouver Is Awesome*, 2023. URL https://www.vancouverisawesome.com/local-news/vancouver-housing-provider-paid-63760-in-false-alarm-fines-for-one-sro-6334541.

[31] Leslie Rojas. Smoke detectors not working inside William Bell Apartments where fire left 2 dead. *WLOX*, 2023. URL https://www.wlox.com/2023/01/26/smoke-detectors-inside-william-bell-apartments-that-killed-2-kids-were-not-working/.

[32] Anurag Chawake. Wyze users say motion detection feature is not working after latest update, 2021. URL http://surl.li/hpswn.

[33] Forbes. Replacing CCTV Camera Footage Is Easier Than You Think, Oct 20 2017. URL https://www.forbes.com/video/5620287208001/replacing-cctv-camera-footage-is-easier-than-you-think/. [Online; accessed 2022-11-06].

[34] Jiwon Choi, Hayoung Jeoung, Jihun Kim, Youngjoo Ko, Wonup Jung, Hanjun Kim, and Jong Kim. Detecting and identifying faulty IoT devices in smart home with context extraction. In *48th Annual IEEE/IFIP International Conference on DSN*, pages 610–621. IEEE, 2018.

[35] Madhumita Mallick, Archan Misra, Niloy Ganguly, and Youngki Lee. DETECTIF: Unified detection & correction of IoT faults in smart homes. In *IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks*, pages 78–87. IEEE, 2020.

[36] Juan Ye, Graeme Stevenson, and Simon Dobson. Fault detection for binary sensors in smart home environments. In *International Conference on Pervasive Computing and Communications*. IEEE, 2015.

[37] Sirajum Munir and John A Stankovic. FailureSense: Detecting sensor failure using electrical appliances in the home. In *International Conference on Mobile Ad-Hoc and Smart Systems*. IEEE, 2014.

[38] Palanivel A Kodeswaran, Ravi Kokku, Sayandeep Sen, and Mudhakar Srivatsa. Idea: A system for efficient failure management in smart IoT environments. In *ACM International Conference on Mobile Systems, Applications, and Services*, 2016.

[39] Nancy E ElHady, Stephan Jonas, Julien Provost, and Veit Senner. Sensor failure detection in ambient assisted living using association rule mining. *Sensors*, 20(23):6760, 2020.

[40] Nithya Ramanathan, Thomas Schoellhammer, Eddie Kohler, Kamin Whitehouse, Thomas Harmon, and Deborah Estrin. Suelo: human-assisted sensing for exploratory soil monitoring studies. In *ACM Conference on Embedded Networked Sensor Systems*, pages 197–210, 2009.

[41] N Malarvizhi, Arun Kumar Dash, V Manikanta, and Athreayasa Kalyan. AI-Based Tracking System from Real-Time CCTV Captures. In *Artificial Intelligence and Sustainable Computing*, pages 739–747. Springer, 2022.

[42] Daniel J Abadi and Samuel R Madden. Reed: Robust, efficient filtering and event detection in sensor networks. *Computer Science and Artificial Intelligence Laboratory Technical Report*, 2004.

[43] Liangyi Gong, Yiyang Zhao, Chaocan Xiang, Zhenhua Li, Chen Qian, and Panlong Yang. Robust light-weight magnetic-based door event detection with smartphones. *IEEE Transactions on Mobile Computing*, 18(11):2631–2646, 2018.

[44] Joseph M Hellerstein, Wei Hong, Samuel Madden, and Kyle Stanek. Beyond average: Toward

sophisticated sensing with queries. In *International Workshop on International Conference on Information Processing in Sensor Networks*, pages 63–79. Springer, 2003.

[45] Michael A Mahler, Qinghua Li, and Ang Li. SecureHouse: A home security system based on smartphone sensors. In *International Conference on Pervasive Computing and Communications*, pages 11–20. IEEE, 2017.

[46] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[47] Thilina Dissanayake, Takuya Maekawa, Daichi Amagata, and Takahiro Hara. Detecting door events using a smartphone via active sound sensing. *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(4), 2018.

[48] Xiaoyan Yang, Hock Beng Lim, Tamer M Özsu, and Kian Lee Tan. In-network execution of monitoring queries in sensor networks. In *ACM Special Interest Group on Management of Data*, pages 521–532, 2007.

[49] Wenwei Xue, Qiong Luo, and Hejun Wu. Pattern-based event detection in sensor networks. *Distributed and Parallel Databases*, 30(1):27–62, 2012.

[50] Wenwei Xue, Qiong Luo, Lei Chen, and Yunhao Liu. Contour map matching for event detection in sensor networks. In *ACM Special Interest Group on Management of Data*, pages 145–156, 2006.

[51] Mo Li, Yunhao Liu, and Lei Chen. Nonthreshold-based event detection for 3D environment monitoring in sensor networks. *IEEE Transactions on Knowledge and Data Engineering*, 20 (12):1699–1711, 2008.

[52] Thien Duc Nguyen, Samuel Marchal, Markus Miettinen, Hossein Fereidooni, N Asokan, and Ahmad-Reza Sadeghi. DÏoT: A federated self-learning anomaly detection system for IoT. In *IEEE International Conference on Distributed Computing Systems*, pages 756–767, 2019.

[53] Vijay Sivaraman, Dominic Chan, Dylan Earl, and Roksana Boreli. Smart-phones attacking smart-homes. In *ACM Conference on Wireless Network Security*, pages 195–200, 2016.

[54] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. IoT goes nuclear: Creating a ZigBee chain reaction. In *IEEE Symposium on Security and Privacy*, pages 195–212, 2017.

[55] Alireza Borhani and Hamid R Zarandi. Thingsdnd: Iot device failure detection and diagnosis for multi-user smart homes. In *2022 18th European Dependable Computing Conference (EDCC)*, pages 113–116. IEEE, 2022.

[56] Home Assistant, 2022. URL http://www.home-assistant.io/.

[57] openHAB, Mar 2023. URL https://www.openhab.org/.

[58] OpenMotics makes building automation relevant, 2021. URL https://www.openmotics.com/en/.

[59] Shadi Al-Sarawi, Mohammed Anbar, Kamal Alieyan, and Mahmood Alzubaidi. Internet of Things (IoT) communication protocols. In *2017 8th International conference on information technology (ICIT)*, pages 685–690. IEEE, 2017.

[60] Álvaro San-Salvador and Álvaro Herrero. Contacting the devices: a review of communication protocols. In *Ambient Intelligence-Software and Applications: 3rd International Symposium on Ambient Intelligence (ISAmI 2012)*, pages 3–10. Springer, 2012.

[61] Grant Hernandez, Orlando Arias, Daniel Buentello, and Yier Jin. Smart nest thermostat: A smart spy in your home. *Black Hat USA*, 2015, 2014.

[62] Xiaojing Ye and Junwei Huang. A framework for cloud-based smart home. In *International Conference on Communication Systems and Network Technologies*, volume 2, pages 894–897. IEEE, 2011.

[63] Abhay Kumar Ray and Ashish Bagwari. IoT based smart home: Security aspects and security architecture. In *International Conference on Communication Systems and Network Technologies*. IEEE, 2020.

[64] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[65] Miroslav Kubat. Neural networks: a comprehensive foundation by simon haykin, macmillan, 1994, isbn 0-02-352781-7. *The Knowledge Engineering Review*, 13(4):409–412, 1999.

[66] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

[67] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[68] Hasim Sak, Andrew W Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. *INTERSPEECH*, 2014.

[69] Christopher Olah. Understanding LSTM networks, Aug 2015. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[70] Thomas G. Dietterich. Ensemble methods in machine learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems*, MCS '00, page 1–15, Berlin, Heidelberg, 2000. Springer-Verlag. ISBN 3540677046.

[71] European Commission. 2018 reform of EU data protection rules. https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679, 2018. Accessed on 2019-06-17.

[72] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. Proceedings of Machine Learning Research, 2017.

[73] Oliver Buxton. Stuxnet: What is it & how does it work? *Avast*, Jul 14 2022. URL https://www.avast.com/c-stuxnet. [Online; accessed 2022-11-06].

[74] K Pearson. Notes on regression and inheritance in the case of two parents proceedings of the royal society of London, Vol. 58, 1895.

[75] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2014.

[76] Murray Rosenblatt. Remarks on some nonparametric estimates of a density function. *The annals of mathematical statistics*, pages 832–837, 1956.

[77] scikit learn. sklearn.neighbors.kerneldensity, 2023. URL https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KernelDensity.html.

[78] scipy.stats.ks_2samp — scipy manual, 2023. URL http://surl.li/hqovl.

[79] InfluxDB, 2023. URL https://www.home-assistant.io/integrations/influxdb/.

[80] Brilliant.org. Feedforward neural networks, Sep 2023. URL https://brilliant.org/wiki/feedforward-neural-networks/.

[81] Jason Brownlee. A gentle introduction to long short-term memory networks by the experts, May 2017. URL https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/.

[82] Davide Chicco, Matthijs J Warrens, and Giuseppe Jurman. The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation. *PeerJ Computer Science*, 7:e623, 2021.