ENFORCING SECURITY POLICIES WITH DATA PROVENANCE TO ENRICH THE SECURITY OF IOT/ SMART BUILDING SYSTEM

by

Abdullah Al Farooq

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Software & Information System

Charlotte

2020

Approved by:

_____
Dr. Thomas Moyer

_____
Dr. Weichao Wang

_____
Dr. Heather Lipford

_____
Dr. Dong Dai

_____
Dr. Bojan Cukic

ABSTRACT

ABDULLAH AL FAROOQ. Enforcing Security Policies with Data Provenance to Enrich the Security of IoT/ Smart Building System. (Under the direction of DR. THOMAS MOYER)

Smart Building Management Systems is rapidly growing worldwide mainly to reduce energy consumption and carbon footprints. Some other benefits that can be achieved through this system include lowering operational costs and increasing occupant's comfort, safety, and increased productivity. Programmable Logic Controllers (PLCs) are used widely for automating smart buildings. The vulnerabilities and attack surfaces of PLCs enable an attacker to control the smart building in order to cause more energy usage, target specific people, and destroy assets. This thesis summarizes the vulnerabilities, threats, and attacks for PLC-based systems. Moreover, the current state of the art of static and dynamic analysis, threat and attack detection, automation, and conflict analysis of smart buildings are discussed. This thesis aims at detecting, analyzing, and mitigating the attack surfaces that are possible for smart buildings. A formal methods approach is proposed for detecting safety and security property violations for smart buildings. Then, a rule-based method is developed to detect violations leveraging provenance data collected from the system. These safety violating incidences are mapped to corresponding variables in a PLC source program. Finally, we implement defeasible reasoning that enforces safety properties in the system. We verify that not only does the new PLC program adhere to the safety properties that have been instrumented, but also it does not trigger any new safety property violations. Finally, we outline new directions to be investigated in the future.

DEDICATION

This dissertation is dedicated to my family for their endless support and encouragement.

ACKNOWLEDGEMENTS

I want to express my sincere gratitude to my advisor Dr. Thomas Moyer. Without his tremendous support, this dissertation would not have been possible. I also have to thank my wife, Synthia Tagar, my parents, and my parents in-law for their formidable support throughout my PhD journey. Finally, I would like to mention my daughter, Arya Tawshi Farooq. Ever since we knew her existence, I was blessed with limitless luck.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CPS Cyber Physical System

DCS Distributed Control System

HMI Human Machine Interfaces

ICS Industrial Control System

IED Intelligent Electronic Device

IoT Internet of Things

MTU Master Terminal Unit

PID proportional-integral-derivative

PLC Programmable Logic Controller

Prov Provenance

RPI3 Raspberry PI 3

RTU Remote Terminal Unit

RTU Structured Text

SCADA Supervisory Control and Data Acquistion

ST Structured Text

CHAPTER 1: Introduction

The significance of smart building management systems has become widespread over the past decade. People have different thoughts about the architecture, functionality, and impact of an intelligent building for commercial and social purposes. In defining smart buildings, we would like to quote from Clements-Croome et al. [4],

> "An intelligent building is one that is responsive to the requirements of occupants, organizations, and society. It is sustainable in terms of energy and water consumption besides being lowly polluting in terms of emissions and waste: healthy in terms of well-being for the people living and working within it; and functional according to the user needs. "

The emergence and growth of highly integrated and intelligent buildings can be seen as achieving several goals; such as reducing energy consumption [5], [6], improving worker productivity, and achieving maximum business profitability [7]. Additionally, due to the increase in energy cost, energy efficient smart building management is considered one of the most viable options for both energy independence and sustainability. It is worth noting that the smart building market is expected to achieve a 20% reduction in primary energy use by 2020 with an aim to keep carbon dioxide emissions under control [8]. Furthermore, smart buildings have the potential to cut greenhouse gas emissions by 80-95% by 2050, as set by European Union [8]. At this point, we note that people often confuse smart buildings with smart homes. Smart buildings cover a wide domain with numerous devices, rules, and requirements as compared to smart homes. The controller, Programmable Logic Controller (PLC), is the most distinguishing component for smart building automation. PLCs are digital computers to automate electromechanical processes and industrial appliances. They

contain input ports to receive signals from the physical world and output ports to control devices with the help of stored programs. Therefore, the safe and secure operation of a smart building depends largely on the characteristics and features that a PLC-controlled system offers. It is worth noting that the PLC-controlled system was expected to grow to $10.33 billion in different controlled system domains by 2018 [9]. Because PLCs have been used successfully for large scale substantial automation, it is regarded as a reliable component for most control systems. This emergence cannot be abolished overnight by the likes of smart home devices for smart building automation.

In the past two decades, control systems have been exposed to a number of cyber attacks. One of the first attacks was the disruption of Worcester air traffic communication where an attacker knocked out phone service at the control tower, airport security, airport fire department, weather service, and over 600 homes and businesses [10]. Another attack that took place in Australia in 2000 released 264,000 gallons of raw sewage into nearby rivers and parks [11]. This attack interfered with normal life as well as introduced potential health issues. Furthermore, another group of attackers managed to infect the train signaling system in Jacksonville, FL in 2003 [12]. This attack affected ten Amtrak trains on routes serving the northeast corridor of the USA. While these incidents caused essential service interruption and property damage, they did not cause any deaths. However, an incident in Bellingham, WA in 1999 caused a pipline failure which leaked 237,000 gallons of gasoline and an ignition 1.5 hours later [13]. Eventually, this incident caused three deaths and eight injuries along with extensive property damage . Recently, another type of cyber attack affected control systems that has geo-political motives and impacts. The Stuxnet malware which uploaded malicious code into a PLC is an example of such an attack. This malware was believed to be jointly built by American-Israeli cyber intelligence to sabotage Iran's nuclear program. The uploaded code in the PLC resulted in damage to the machines of nuclear plants for years without being detected [14]. Although no attacks have

been recorded targeting smart buildings to date, they are still vulnerable because the same/similar PLCs are used for the automation.

1.1 Smart Building and PLC Security

With the advancement of technology, a smart building is no longer limited to heating, ventilation, and air condition (HVAC) automation functionalities. The likes of smart lights, locks, window blinds, speakers, IP cameras, carbon monoxide detectors, alarm devices, motion and water sensors have become integral components of a smart building. Most of these devices are resource constrained and do not include security features in themselves. These devices are infrequently patched, leaving them with known vulnerabilities, making it easy for attackers to gain access. Furthermore, the way they are distributed with insecure communication protocols, leaves the communication channels exposed to attackers. An attacker can trigger an event that leads to conflicting actions on the same device or feature of a smart building. For example, an attacker can create multiple events that trigger a thermostat to increase and decrease the temperature of a room at the same time. Sending two different commands to the thermostat at the same time continuously can damage it, by artificially shortening the device's lifespan. In this way, the attacker not only damages an asset but also may drive the occupants out of the room to leave due to fluctuations in the comfort level of the room. As we see from the above-mentioned example, only two rules are enough to create a conflict. Recent research indicates that there can be as many as 30,000 rules in a smart building [15]. Moreover, misconfigurations are possible as there are numerous rules or policies for taking actions by the controllers after events have occurred.

The network that handles the distributed control application of a smart building is generally organized in a two-tiered model; a field network and a backbone network [16]. Sensors, actuators, and controllers (SACs) comprise the field networks where predominantly non-IP field protocols are used. On the other hand, the back-

bone network is comprised of the management and operating workstations where the programming of initial points, changing set points and controller logic takes place. Interconnection devices (ICDs), e.g., routers or gateways, connect these two networks to a wide area network (WAN). An attacker can get access to the data passing through an ICD by attacking the applications running there. Moreover, because an ICD can be connected to the internet directly, this can be used as an access point to initiate further attacks on a smart building.

Weak authentication and poor integrity checks are regarded as the two most significant weaknesses of the communication protocols used in PLC-based systems [17]. These flaws can be used to initiate a man of the middle (MITM) attack, denial of service (DoS) attack, and memory corruption attacks (e.g., array, stack, and heap overflow, integer overflows, pointer corruption). The lack of proper authentication methods enables an attacker to replay, modify, or spoof data as well as spoof devices. Moreover, most of the PLC-based system use protocols that transmit clear text while communicating, thus making the system susceptible to eavesdropping and manipulation. The reconnaissance for an attack on a smart building is straightforward. Additionally, these protocols have very few or no security capabilities and their documentation is available for free. It is not uncommon to have a single network to handle both control and non-control traffic of a smart building system. It is possible for an attacker to flood a network with unsolicited messages (non-control traffic) to make a particular device unavailable. They can impact the availability of critical sensor data, leading to a denial of service attack.

1.2 Summary of the Background

There have been several attempts to deploy security policies with static verification, dynamic verification, and the combination of these two for cyber-physical systems where PLCs are used. Static verification (model checking) is proposed in TSV [18] where a middleware, sitting between a PLC and the devices, verifies that

the safety properties are maintained. The verification is performed before the commands reach the devices from the PLC. The safety properties are written in temporal logic and are verified using model checking. While this work is mainly on verifying the system's behavior, other research focuses on the verification from the PLCs' source program [19, 20]. Furthermore, some works propose to automatically generate formal models from PLC programs [21, 22, 23, 24, 25]. While static analysis performs the verification before a PLC program is operational, dynamic analysis, on the other hand, ensures that policies are not violated at run-time. C^2 [26] is one of the most prominent methods to enforce safety policies in PLC-based system. When a PLC issues a command to an actuator, the current state of the system is checked and decisions are made through C^2 to determine if the command should be issued. In this work, concerns about the size of the trusted computing base (TCB) and state explosion in the model checking are expressed. Another approach reduces the size of the TCB considerably and combines the static and dynamic analysis of TSV and C^2 [27]. The works by McLaughlin, et al. focus specifically on safety properties. This was subsequently extended in [28] with an effort to find malicious PLC programs. Another approach to dynamic analysis of PLC-based systems is proposed in [29] using Interval Temporal Logic (ITL) and the Tempura framework, which aims to provide early alerts when the system output does not match the safety properties. Later, this work was extended in [30] where an ITL/Tempura definition of a Siemens S7-1200 PLC ladder logic was presented. Their developed monitoring methodology captures a snapshot of the current state (with values for markers, input, output, counters, and timers) of the PLC. Tempura was implemented to execute on an Arduino Uno connected to the PLC, ensuring that the PLC does not need a powerful computing node to perform the computations. While static analysis has proven promising, the wide range of possible inputs and outputs for automation can lead to state explosion. Therefore, the completeness of the verification approach is not assured. Similarly, dy-

namic verification suffers from a coverage problem, where only executed code paths are verified. Symbolic execution helps in this regard by minimizing the state space. It cannot guarantee complete verification of outputs (actuation command) from input sets (sensor measurement).

Apart from verification and real-time monitoring approaches, there have been some research efforts to attribute an anomaly or adversary with data provenance. A recent work, PROV-CPS [31], collects provenance from resource-constrained embedded devices of the cyber-physical system. However, this research collects provenance from sensors only to identify anomalous measurements. Another notable work in this area is ProvThings [32] where a provenance collection framework is proposed for IoT apps and devices. ProvThings presents an automated instrumentation mechanism for IoT apps and device APIs. The collected provenance is then used to generate explanations for "why" a particular action occurred.

1.3 Thesis Statement

In this thesis, we address the problem of safety policy violations in a smart building environment. In doing so, a formal method needs to be developed to define high-level safety and security policies. Violation of these policies results in anomalous or conflicting situations among the devices. We propose a system to track the data flows to detect policy violation and actively enforce the safety and security policies of the system are not violated. The central thesis of this work is, therefore:

> *Data provenance can secure smart buildings by enforcing safety policies that prevent policy violations in smart buildings. Data provenance, the history of data, provides a rich source of information from the initiation of a system which helps in robust forensic analysis.*

While the anomalies for the smart building are vast, we focus specifically on the anomalies that arise from device and environmental feature conflicts. To that end, the thesis examines the following questions:

- *How do we define a set of sound and complete safety policies for large scale smart buildings, the violation of which lead to conflicting behavior in the domain?*. The automation of the system may lead the controller to command a device which is already performing a different action. As events are random, their occurrence cannot be managed most of the time. We formalize this type of policy violations in Chapter 3. An attacker can leverage these violationss or force some specific events to occur in order to take advantage of misconfigurations (i.e.rule conflicts) in the system.

- *How do we detect conflicts in smart buildings?* Having formalized the safety properties to detect conflicts, a novel approach is developed to track the data flow in smart buildings. The primary goal is to detect whether or not safety properties violations have occurred. This requires a way to track inputs and outputs, and a mechanism to model the evolution of the system from inputs to outputs. With these mechanisms in place, it becomes possible to ensure that the PLCs do not send commands to actuators that violate the safety and security policies of the system. To detect conflicts in smart buildings, we propose `PLC-PROV`, presented in Chapter 4.

- *How do we enforce security policies in smart buildings?* In order to ensure the safety and security of smart buildings, a methodology needs to be developed to enforce that no safety property violations are taking place. Data provenance is a robust approach to detect deviant values (i.e., sensor measurement and actuator action in smart buildings). This, combined with a conflict resolution method, can be effective for enforcing safety properties in smart buildings. To examine these, we propose `DEFEASIBLE-PROV` which is presented in Chapter 5.

1.4 Contribution

In answering the above questions, this thesis provides the following contributions:

- We propose a formal methods approach that considers the interactions among sensors, actuators, and controllers to detect safety policy violation. We prove the set of policies are sound and complete. The safety properties we formalized to consider conflicts as the preeminent threat to smart buildings. We show that the conflicts lead to additional actuation which eventually results in more energy consumption.

- We propose a mechanism to track the inputs and outputs of the system and compare them against the safety and security properties we have formalized. We employ *data provenance* for tracking the data flow to and from the PLCs and use that provenance to determine if a violation has occurred in smart buildings. Basically, provenance is the "history of data transformed by a system", and has been accepted as a novel approach to reason about the *context* in which an action is taken. Because PLCs are entirely event-driven, context is vitally important, and provenance is a natural fit for this sort of analysis as such.

- We propose an enforcement mechanism, `DEFEASIBLE-PROV`, to ensure conflicts in smart building operations are resolved with high efficacy. After analyzing the collected provenance graph, we develop methodologies to impose exceptions and superiority relationships among the smart building rules. Our proposed approach has the capability to resolve conflicts during run-time by continuous analysis of the information received from smart buildings.

CHAPTER 2: Background

This chapter summarizes the literature that discusses the basic infrastructure of smart buildings. Then we examine the literature that explores the Programmable Logic Controller(PLC) used for automating smart buildings. After describing the technologies used, we examine the vulnerabilities, threats, and attack surfaces there. Then, we discuss the current state of the art to detect, analyze, and mitigate the attacks and safety property violations in this domain. As one of our primary concerns is to deal with security policy violation/ conflicting situations in smart buildings, we study the methodologies and algorithms that include but are not limited to defeasible reasoning for mitigating conflicts. Finally, we conclude this chapter with some recent data provenance approaches that have been used in the IoT/ cyber-physical system domain.

2.1 Smart Building Infrastructure

A smart building consists of a number of sensors, actuators, and controllers connected with the wireless sensor network (WSN) or wireless sensor actuator network (WSAN). The devices (sensors, actuators, or controllers) include but are not limited to lights, thermostat, humidifier, speakers, security camera, video doorbell, door lock, window blinds, smoke detectors, carbon monoxide detectors, fire alarms, smartphones, and last but not least PLCs. Fortino et al., [6] analyzed all the requirements smart buildings based on WSANs with several parameters. The first one is the fast reconfiguration capability of the nodes (sensors, actuators). Some building management frameworks have capabilities to reconfigure nodes by sending packets while some do not have. It depends on the manufacturer of the nodes. In-node processing is an-

other important parameter of WSAN in buildings. Because energy saving is one of the major goals of smart buildings, in-node processing saves both bandwidth and energy by shifting from in-network processing (data fusion, shared variables, etc). It must have multi-hop support in order to cover the whole building. It should be capable of handling different types of sensor nodes with different features. A user or a controller can send a command to actuators (appliances, lights, radiators, etc) based on sensor values and rules. A building programming abstraction is proposed to capture the morphology of a building and to store the location of the devices (e.g, bathroom, lounge, seminar room, office) with their functionality (e.g., ambient temperature, lighting system, ventilation system). The building management system should have rules for operating the building in an energy saving and comfort providing manner. Also, smart buildings should have sufficient infrastructure to provide human-machine interaction to edit, delete, or add new rules to a controller or a set of controllers.

Devices like NEST Protect, Wemo Plugin, and Scout Alarm have gained popularity in materializing a home as a smart home. Some widely used smart home management systems are given in table 2.1 [1]. The second column is straightforward, indicating whether the standard is open source. The third column of the table indicates whether the standard or the system has dedicated controlling and management capabilities. This refers to the customization of the GUI for smart building applications. The fourth column in the table refers to Open Systems Interconnection (OSI) layers. The lower the layer, the closer it is to the physical medium of communication e.g., copper wire, optical fiber or air. On the other hand, the higher layer indicates the actual application and management intelligence. The last column indicates whether the standard supports functionality for the future smart grid.

However, for a smart building, in contrast with a smart home, the domain is much wider, and there are numerous devices and requirements to handle. PLCs are the most used and still considered the most reliable controllers to automate smart buildings

Table 2.1: Major Machine to Machine Communication standards, alliances and Building Management System (BMS) [1]

Name	Open Source	Offer BMS Services	OSI Layer Defined	Support Demand Response
Apple HomeKit	N	Y	(5)-(7)	N
Samsung SmartThing	N	Y	(5)-(7)	N
OpenHAB	Y	Y	(7)	N
Fairhair Alliance	N	Y	(5)-(7)	N
Thread	Y	N	(3)-(4)	N
Volttron	Y	N	(5)-(7)	Y
Weave	Y	N	(5)-(7)	N
AllJoyn	Y	N	(5)-(7)	N
Open Interconnect	Y	N	(5)-(7)	N
openADR	Y	N	(7)	Y
ZigBee	Y	N	(3)-(7)	N
Z-Wave	N	Y	(1)-(7)	N
Lora	N	N	(1)-(2)	N
SigFox	N	N	(1)-(2)	N
EnOcean	Y	N	(1)-(3)	N
IEEE 802.15	Y	N	(1)-(2)	N
IEEE 802.11	Y	N	(1)-(2)	N
6LoWPAN	Y	N	(3)	N

[16, 33]. Regardless of the current hype of smart home hubs, the need for PLC in managing a smart building cannot be removed overnight. According to [9], there is more than 10 billion US dollars in the PLC market worldwide.

Figure 2.1 shows a notional Industrial Control System, comprised of several components that together provide the ability to automate industrial processes. At the heart of this system is the PLC. The PLC takes input from the sensors and determines the appropriate commands for the actuators to adjust the environment. The logic for the PLCs are programmed with an engineering workstation that contains the IDE used by the programmer to develop the application logic for the ICS. Additionally, an ICS has one or more Human Machine Interfaces, or HMIs, that enable operators to view current and historical data from the ICS. The historical data is stored in the data historian and is often used for post-facto analysis of events.

The PLC applications are written in one of several programming languages includ-

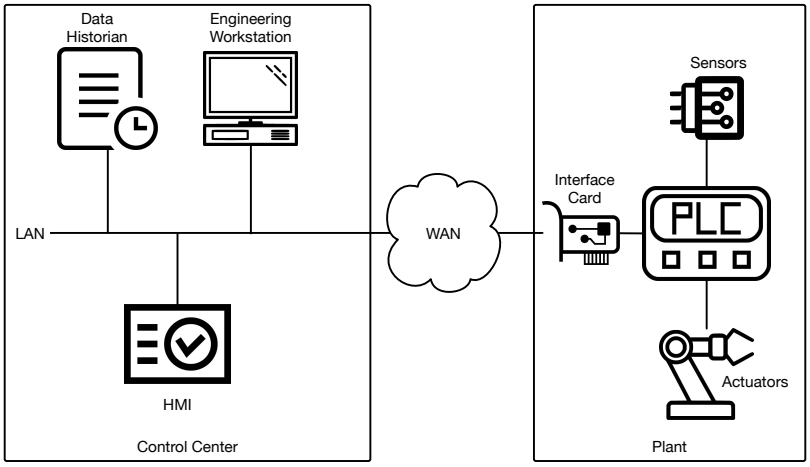

Figure 2.1: A simple PLC-based control system with the basic components of any industrial control system.

ing Instruction Lists (IL), Structured Text (ST), Functional Block Diagram (FBD), and Ladder Logic (LL). Regardless of the programming language used, the instructions control the features of the PLC including I/O control, communication, logical decisions, timing, counting, three mode proportional-integral-derivative (PID) control, arithmetic, and data and file processing. The inputs to the PLC come from a wide array of sensors such as temperature sensors, motion detectors, smoke detectors, water leak detectors, and surveillance cameras. The outputs of the PLC go to actuators that adjust the current environment. These actuators include thermostats, humidifiers, speakers, security cameras, video doorbells, door locks, and window blinds.

The topology of an ICS network can be broadly divided into two subnets. The first is the control network where the sensors and actuators interface with the PLC. In a more complex control system, there may also be a Master Terminal Unit, or MTU, that provides the control programs for the PLCs. This control network uses non-IP-based protocols such as Modbus [34]. The second network is the corporate network, where the historian, HMI, and engineering workstation are located. This is a traditional enterprise network, using standard IP protocols to communicate. In order

to link the corporate and control networks, interface cards are used to provide a bridge between IP-based protocols and the Modbus protocol. The PLC uses a Modbus/TCP protocol to send data to the historian. The PLC also provides an HTTP service for the historian to access and store historical data into a database.

For a distributed system like SCADA (Supervisory control and data acquisition) or DCS (Distributed Control System), a group of PLCs is assigned to different subsystems. This is mainly done to handle long distance communication among geographically dispersed assets (e.g., power grids, natural gas pipelines, water distribution, wastewater collection systems, railway transportation systems). The far-reaching nature of these systems necessitates numerous control systems responsible for controlling local operations but working in concert to ensure the global functioning of the system. While these systems are more complex, they rely on many of the same basic components of a smaller-scale ICS.

Even in localized ICS environments (e.g., smart buildings), it is common to rely on several PLCs that work in concert to provide a range of functions. Consider for example that there might be a PLC that controls the heating, ventilation, and cooling, or HVAC, system, one PLC that controls the elevators, one PLC that controls the door and window locks, and finally one PLC that monitors for hazard conditions (e.g., smoke, carbon monoxide, water and chemical leaks, etc.). While these systems can be implemented independently, there are often dependencies that need to be accounted for. Consider a case where the hazard monitoring PLC detects smoke in the building, it must send notifications to the elevator and door/window lock PLCs that this condition is present so that appropriate actions can be taken. Those actions might be to open the locks on the doors, and move the elevator to the ground floor and then lock out the use of the elevator. These dependencies ensure safety and efficiency in these automated systems. The followings are some examples where PLCs are used for automating smart building.

In Barz et al., [35], it has been shown how PLCs are used for operating a smart building. Devices are connected via a bus system, or star topology. With the help of PLCs, the automation for turning off the lights, lowering the blinds, increasing/decreasing room temperature, alarm on smoke and theft are possible. The authors discussed the Simatic S7-1200 System control which is operated with the help of a STEP7 program. This software provides interaction with the sensors and actuators through input addresses and commands the process through output addresses [36, 37]. The STEP7 Basic Software V12 has some functionalities that includes the configuration and parameterization of hardware, defining communication, programming, testing, developing documentation, and generating a display screen for the SIMATIC basic operating panels. The source program is written through a GUI. The program is run through the software which causes the code to be transferred from the software to the PLC. An operator can change the logic of the process by looking at sensory data and current states through the software. Hence, access authorization and authentication of the computer that has the STEP7 Basic Software is important.

Sysala and Neumann [38] proposed a smart family house control with the help of a commercially used PLC. However, they focused not only on the logical operation of the system, but also with the communication between the system and other devices like computers, tablets, and mobile phones. The project uses the "PLC Tecomat Foxtrot CP- 1006" by a Czech company named Teco. This PLC follows the international standards IEC 61131-3. The PLC provides the web server function to provide access to the functionality of PLC. Apart from controlling the heating, lighting, air condition system, controlled access, the PLC was capable of controlling infra-red (IR) devices like television. radio, satellite, and other multimedia devices.

Skeledzija et al., [39] proposed a modified version of a PLC (*littlePLC*) for automating a smart building. The main goal was to reduce energy consumption and carbon footprint significantly apart from ensuring better comfortability for the occupants of

the building. In this work, an ARM-based microcontroller unit was used instead of using commercially available PLCs. Although the *littlePLC* does not have industrial grade temperature operating range or other resistive techniques to an aggressive atmosphere, it provides a platform for easy implementation of control logic with a lower price. The smart building is divided into four subsystems; Central Processor Unit (embedded processor), Model Predictive Control algorithm (MPC), Programmable Logic Controller (littlePLC), and Wireless Sensor Network.

The most essential component of this system is the MPC that uses a dynamic model with the constraints of the system. MPC takes current measurement into account in order to forecast system behavior and to produce actions that lead to optimizing energy consumption. The CPU provides the resource to run the MPC. The sensors send the measurement to MPC through a wireless sensor network. Once MPC determines the optimal actions to be taken with these measurements, commands are sent to the actuators (HVAC system) via *littlePLC*.

2.2 Vulnerabilities

Generally, almost all PLCs are manufactured and made available commercially. Milinkovic et al., [2] listed the operating systems of some popular commercial PLCs. Those are given in Table 2.2. As can be seen, Microsoft Windows operating system is used in Siemens SIMATIC PLC. Hence, the vulnerabilities of Windows OS are also present in that PLC. Operating systems like OS-9 and VxWorks are less pervasive and therefore have fewer known vulnerabilities. However, VxWorks debug service (WDB Agent) has a known vulnerability over UDP port 17185 that allows complete access to the device, including the ability to manipulate memory, steal data, and hijack of the entire OS. Moreover, VxWorks has a weak password hashing implementation [40].

Although PLCs can be exposed to the same type of attacks as traditional IT equipment because of using the same operating systems, the security for the control system should not be the same. Milinkovic et al., [2] differentiated the focus of the

Table 2.2: Operating Systems of Some PLCs [2]

PLC	**Operating System**
Allen-Bradley PLC5	Microware OS-9
Allen-Bradley ControlLogix	VxWorks
Emerson DeltaV	VxWorks
Schneider Modicon Quantum	VxWorks
Yokogawa FA-M3	Linux
Wago 750	Linux
PLC reference platform	QNX Neutrino
Siemens SIMATIC WinAC RTX	Microsoft Windows

security goals of a PLC-based control system from the IT system in Figure 2.2. The assets and confidential data need to be more secured than anything else for general IT and therfore confidentiality becomes the most concern there. If the service is not available, it does not cause any damage or death. On the other hand, if the service is not available for a PLC-based system, the consequence may be catastrophic with causing death. We have mentioned some similar examples (e.g., train service interruption, sewage, and pipeline leakage) in chapter 1.

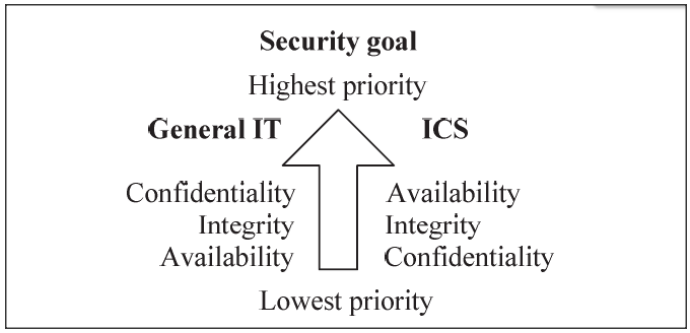

Figure 2.2: Security Goal Difference between IT and PLC based control system

Stouffer et al., [41] listed a number of vulnerabilities that are present in the Industrial Control System domain. They pointed out that the protocols (Distributed Network Protocol (DNP) 3.0, Profibus, and other protocols) used to communicate with PLC are freely available. Sometimes these protocols have very few security capabilities . Moreover, the use of clear text while communicating with the PLC make

it possible to eavesdrop on the communications. Inadequate authentication and access control to the workspace that has PLC application makes it more susceptible to attack.

A PLC-based system suffers from weak authentication and poor integrity checks because of the communication protocols used [17]. These vulnerabilities can easily be used to initiate man of the middle (MITM) attacks, denial of service (DoS) attacks, and memory corruption attacks (e.g., array, stack, and heap overflow, integer overflows, pointer corruption). An attacker can replay, modify, or spoof data as well as spoof devices because of the weak authentication system. Moreover, most PLC-based system use protocols that transmit clear text while communicating, thus making the system susceptible to eavesdropping. The reconnaissance for an attack on a smart building becomes straightforward in this case. In addition, not only do those protocols lack security capabilities, but also their documentation is available for free. A single network may have to handle both control and non-control type commands. The non-control traffic can initiate a DDoS attack to make some specific sensors and actuators unavailable.

2.3 Attacks and Threats

In addition to the real world attacks, there are some attacks proposed in some research papers to act proactively before they take place. Mclaughlin and Zonouz [42] proposed a new type of false data injection (FDI) attack where limited knowledge is enough for creating predictable malicious output in a large SCADA based controlled system. The authors claimed that prior works on FDI attack needed the attackers to have vast knowledge on the network topology and whole system for making an attack successful. Moreover, the attackers had no guarantee that their launched attack would have predictable consequences. Due to these restrictions, the authors present CaFDI (controller-aware false data injection) attack against any cyber-physical platform especially those which are controlled by PLC. The attack targets individual

monitoring and control substations where a limited knowledge about the substation configuration and control over a few sensors are enough. It is important to note that CaFDI attacks do not need any code upload to the PLC. The attack is carried out in two steps. The first step is to create an abstract formal finite state machine, Buchi Automaton, representing the controller code. CaFDI generally needs direct access to the PLC code to complete the Buchi Automaton. If not found, the state machine is completed through the observation of the target PLC's I/O behavior. The second step of CaFDI is to explore the generated automaton in order to find whether a malicious objective is satisfiable. Once a satisfiable state of a malicious objective is found, CaFDI calculates the malicious input values and sends them to PLC. In this way, the PLC generates malicious outputs as expected by the attack. In this paper, the authors discussed some remedies against such attacks. Undoubtedly, securing all sensors and channels between the sensors and PLC is the best way. However, due to the distributed nature, it is not always feasible. Physical tamper detection and redundant sensor deployments are some other ways to work against this attack. The best way is to verify the safety properties of PLC's code to make sure no malicious behavior will take place given any input vector.

McLaughlin and McDaniel [43] discussed a different type of attack based on the vulnerability that is possible with the uploading of malicious code to the PLC. This research proposed a proof-of-concept tool for generating PLC payloads based on device behavior in the target system. The authors argued that an adversary with no knowledge of the PLC's interface to the control system could not do much damage to that system. Moreover, in an attack like *Stuxnet* PLC version strings and device metadata are needed to verify the correct target. If the appropriate metadata or strings are not found in the PLC, it was ignored by the virus. In SABOT, it is not necessary to match the adversary control logic to the system control logic. Hence, the adversary need not know any version strings or vendor metadata a priori. SABOT

is capable of correctly identifying a target control logic out of some candidate control logic. Firstly, SABOT decompiles the PLC logic bytecode to an intermediate set of constraints on local, output, and timer variables. Then these constraints are translated into a process model using the NuSMV model checker. As the next step, SABOT attempts to find a variable to device mapping (VTDM). Whenever an adversary wishes to cause malicious activity, it does not know which device is referred by which memory. VTDM finds the mapping from the names in the adversary's specification/policy to the names of control logic model. The specifications are written using computational tree logic (CTL) formulas. The CTL formulas are given after the keywords of the logic. VDTM finds the mapping of these keywords to the control input and output memory address. It is important to note that while mapping from control input to output memory address, conflicts are possible. The authors attempted to solve that issue by introducing additional properties to the specification. In the last step, SABOT maps the names of adversary's generic payload to the control logic. The instantiated payload is recompiled into bytecode and later uploaded to PLC. For evaluating the accuracy, adaptability, performance, and scalability of SABOT, the experiments were conducted on container filling [44], motor control, traffic signal, pH Neutralization [45], and railway switching [46]. This research does not propose any counter measures against these attacks.

A similar type of attack on PLC is found in Garcia et al., [47]. They developed a rootkit, *HARVEY* for PLC that is capable of generating physics-aware stealthy attacks. The rootkit is capable of intercepting the PLCs' input and output values and generating semantically correct system states towards the control unit. However, the actual system states are unchanged. More precisely, the rootkit sits in the firmware layer and intercepts the input from sensor measurement. Then, it sends fake, but legitimate-looking sensor measurements to the HMI. The operator gets no clue on the real malicious sensor measurement. When legitimate commands come

from a non-compromised control unit, the rootkit simply discards it and issues actuation commands on its own. Actuations are performed in such a way that malicious behavior remains optimal. As an easy example, an attack on the pump system has been discussed in the paper. The goal of the malware is to increase the pressure in a pipe to damage it. If the rootkit changes the pressure arbitrarily, it would get caught because the sensor is sending an increase of pressure to the operator. This may even trigger an automatic safety mechanism. Hence, the malware must ensure that sensor readings, presented to operators are not suspicious. In doing so, *HARVEY* runs malicious code in parallel to the legitimate control logic. The outputs from both executions are calculated from the dynamic model as given in equation 2.1.

$$\begin{aligned} f(x,u) &= Ax + Bu + \omega \\ y &= Cx + \epsilon \end{aligned} \tag{2.1}$$

As mentioned earlier, *HARVEY* intercepts the output module write request and replaces them with malicious control output. The malicious output is calculated with manipulating minimum actuation in order to get maximum damage to the plant. *HARVEY*, at the same time, fabricates the sensor measurement as well. It calculates the sensor measurement as if the real control logic has been sent to the plant. At the same time, it runs malicious control commands to the plant and gets sensor measurements from that. However, it sends fake measurement to the operators only. The whole implementation was done by reverse engineering methods and only for PLCs manufactured by Allen Bradly which is the most common ICS suppliers in the United States. This research evaluated the effectiveness of *HARVEY* in a real-world power grid system.

Code injection in bytecode of a PLC firmware is proposed in [48]. The authors proposed a tool *PLCinject* in devising additional malicious code in the firmware. *PLCinject* works at any PLC if the bytecode is of MC7 format. The authors argued

that the lack of authentication in the PLCs can let the attackers gain access to those. An adversary with proper knowledge about the operational functionalities can download and upload code to it. The attacker is then capable of gaining access to other PLCs as well as other devices in a business network. That is, a network administrator has to be cautious when securing a business network not only from outside but also from inside. The PLCs can be used as a gateway by the attackers. With *PLCinject*, the adversary compromises the PLC in two steps. First, an SNMP (Simple Network Management Protocol) is implemented in order to get an overview of the network behind the PLCs. The second and last is step is to inject *SOCK* proxy to the PLC logic code in order to gain access of all PLCs via the compromised one. The authors noted that *SOCK* protocol is quite lightweight and easy to implement with respect to MC7 bytecode.

At this point, readers may wonder that patching the firmware of PLC can instantly solve the issue. The authors argued that firmware is not very often patched. One of the reasons behind this is that patching would interrupt the production process which has monetary impacts on the production. Another reason is that it may lead to a loss of production certificate or other kinds of quality assurance of the manufacturing company. The injection of malicious code was performed in Siemens S7-300 PLC that use Statement List (ST) as the native programming language. This source code generates MC7 bytecode in PLC. This research evaluated their attack by comparing the execution cycle times of three scenarios: 1) benign control program, 2) malicious control program with SNMP implemented, however without *SOCK* proxy loaded, 3) malicious code with *SOCK* proxy implemented. It takes more time on the second and third scenario than the first one. The authors noted that the upload of the code should not exceed 150 ms of time for a successful attack.

Li et al., [49] proposed a different type of attack that is possible in the controller's source program. The goal of the paper was to investigate a potential attack that

can trouble the basic operation of a controlled system (IoT, Industrial System). The authors argue that attacks like *Stuxnet* were totally out of everyone's knowledge. Therefore, research on potential attacks should bear equal importance with a recently occurred attack. In their paper, the authors defined the sequential logic as the physical process as the transitions between the different control commands executed by the actuators. Obviously, control commands are issued by the controller to the actuators. The false sequential logic attacks is the violation of the sequential order of the control command. There are lots of ways to interchange between any two control commands. However, the authors limited their work to two adjacent control commands. They provided a water pump system as an example of this attack. There are two pumps P1 and P2 from two different tanks T1 and T2, successively to a tank T3. It is designed that only one pump will run at a time. P1 will start first and P2 will start as soon as P1 finishes. In this case, a false sequential attack is visible when P2 starts before P1 finishes. There is a valve to drain liquid from T3. The authors defined six sequential logic attack with state-based modeling and showed the consequences of the attack through a MATLAB Simulink based environment. The attacker was capable of not draining liquid from tank T1, T2, and T3. Moreover, the attacker was capable of overflowing T3 before draining it properly. The authors pointed out that modification of sensor values can affect the control actuation through feedback. However, the question still remains how attackers are capable of getting access to a controller's code. Recent research [48] indicates an attacker can replicate a PLC's code through an insider or through preceding information gathering attack.

Last but not least, stealthy or covert attacks on a PLC-based system are discussed in Trcka et al., [50] where a smart building was considered as a case study. A stealthy attack is a common type of attack in a *cyber physical system* (CPS) like water and gas distribution centers, transportation network systems, etc. Networked control systems like smart buildings are a special form of cyber-physical system. Here, a network is

composed of sensors and actuators and those are controlled by a central controller. Security analysis or risk assessment is an important task to identify vulnerabilities and possible exploits in CPS. Sometimes, a predefined set of attack models are used to evaluate system vulnerabilities. Trcka et al., [50] proposed a model-based security analysis technique for the networked control system and chose a smart building environment as a case study. In a stealthy attack, an adversary drives the system into a bad state while staying undetected at the same time. For this work, the authors did not consider any specific detection scheme. Rather, they focused on a general method where an attack can be detected by the difference between the abnormal (attacked) and nominal measurement. That scheme is automated via formal verification. Typically, formal verification is an automatic technique that ensures whether or not the safety property is maintained throughout the operation. When any of the safety properties are violated, the authors used *simulink* to generate a counterexample. This counterexample determines the initial point of violation. The undetectability of an attack is defined with four rules. Firstly, the injected signal should be in an acceptable range. Secondly, the last state attack should follow the same relation to the current one. Thirdly, the measurement of a sensor with the injected signal must not cross the threshold. Lastly, the attack should not be any safe state of the system. In this work, safe states of a system have been defined based on the controlled system's dynamic. In this research, *Simulink Design Verifier* has been used as a formal verification tool. If no counterexample is generated during its operation, the controlled system can be declared safe. As a test case, a three-room home environment was chosen. There were two sensors and two actuators (thermostat) shared by all three rooms. The authors ran three experiments by making one sensor not secured, by making one communication channel from a controller to sensor compromised, and lastly, by devising one of the actuators to be compromised. In all the experiments, it was shown that the attacker was capable of compromising the system, yet remained

undetected with a variable attack length. However, the dynamic nature of the system was not discussed in this paper. The authors claimed that their approach is applicable to other monitoring and diagnostic systems as well. However, scalability will be an issue if this approach is applied to a large controlled system. The paper lacks this analysis and discussion. Moreover, in their test case, any actions taken on actuators were allowed that can creating conflicting situations. This can be avoidable through formal verification that ensures no known conflicts would occur.

2.4 Automation and Verification of PLC-based System

The automation and verification of PLC-based systems have drawn the attention of the research community. There have been several attempts to deploy policies that ensure safe behaviors of the PLC-based system. These can be divided into two categories: static and dynamic. Basically, static analysis does verification before the PLC program is released for operation, while the dynamic analysis contributes to checking for violations of security properties of a system at run-time.

2.4.1 Static Checking

Mclaughlin [26] introduced an enforcement mechanism for safety policies in a PLC-based system. When a PLC issues a command to an actuator, the current states of the system are checked and then decisions are made whether or not the command should be issued. The author named the enforcement mechanism C^2. It is placed in the control lines between a PLC and physical devices. The principal motivation behind this work is that most control systems do not have the architecture for storing state information of devices. C^2 mediates this limitation by acting as a middleware in a PLC-based system. It contains a set of declared conflicts among the devices of a system. The conflicts are checked when a device goes from one state to another depending on the command it receives from PLC. The author defined the states of the device as discrete and continuous. A single step motor can be considered a discrete

state device having three states namely, *forward, reverse, and off*. On the other hand, a continuous device (i.e., stepper motor) has a range bounded by an upper and a lower limit. When a device has issued a command, it has a minimum time allowed for the transition of states. During this time, the other devices that have conflicts with this device, are checked with their current state. When a conflict is found, the command is dealt with. In order to handle the situation, their framework devises TAP (Ternary Access Point) which allows C^2 to monitor as well as inject traffic between the control room and PLC. The control room has an operator that receives a notification when a conflicting situation arises. The PLC is issued a command thereafter. There are four ways to deny the command. The command can be totally *dropped*. Secondly, *approximation* can be done for continuous devices in order to determine the feasibility for truncating the command in some specific interval. Thirdly, *retry* can be done to replay the command. Lastly, *notification* can be sent to the PLC for further action. The authors developed a prototype that was tested in motor control, saw mill, pH neutralization, assembly line, and traffic light environment.

A similar type of work is found in McLaughlin et al.,[18] where a middle-ware ensures the safety of a PLC-based system sitting between the PLC and the devices. They named it TSV (Trusted Safety Verifier). The authors pointed out that a control system has a large trusted computing base (TCB) of commodity machines, firewalls, networks, and embedded systems. These large TCB introduces more vulnerabilities that can lead to more unsafe behaviors of the whole system. The patches for these vulnerabilities are not applied regularly. The operation of the plant should not be stopped due to these. Hence, this research tends to minimize the TCB to a verifier (TSV). TSV verifies the safety behavior of the code executed on PLC before it goes to actuation. The safety properties are written in temporal logic. In fact, TSV uses model checking to perform the verification. As model checking suffers from state explosion, TSV performs a symbolic execution to generate a mapping from path

predicates to symbolic outputs. Path predicates are basically a boolean expression that characterizes a set of input values that will cause a path to be traversed. This creates all possible executions of a single scan cycle. After this step, the model checking component of TSV is invoked. A set of temporal properties, combined with temporal qualifiers are inputted to TSV as the safety specification. These are verified across a state-based model of the PLC code. The authors named that model as *Temporal Execution Graph* (TEG). When a safety property is violated, TSV provides a counterexample to system operators. The code with no safety property violation will be given permission to execute from the PLC. The authors implemented TSV on a Raspberry Pi computer and put it in a way that it is capable of intercepting all controller-bound code. Traffic light, Assembly way, Sorter and Stacker of a conveyor belt, Rail interlocking, and PID (i.e., temperature controlling in a room) were taken as the testbed for this research. The performance, in terms of time requirements, were measured for all cases. The state space size due to model checking steps was shown for all the cases in order to validate the model's scalability. However, this research did not explain how symbolic execution can capture all input-output mapping. The more the possible number of inputs, the more chances there are for state-explosion in the model checking component of TSV.

As minimizing the trusted computing base(TCB) and state explosion in model checking were concerns in implementing TSV, the same research group presented a work [27], where TCB was minimized significantly. The author pointed out two of their previous works, TSV, and C^2 as static and dynamic enforcement of security policies in PLC logic, successively. Now, a new policy enforcement system is proposed as a small TCB. In order to understand the framework better, a brief explanation of how PLC works might be necessary. A PLC program consists of chains of Functions Blocks that are very similar to functions of traditional programming language. Each scan cycle of a PLC starts with the execution of an Organization Block (OB) which

invokes other FBs based on the sensor measurement found in the input memory region. A subset of PLC's function blocks is considered as the Guard Blocks (GBs) in this research. A set of run-time and upload-time checks are performed to ensure that only highly privileged users have access to GBs. The size of GBs is way less compared to the total number of function blocks in the PLCs. Hence, it contributes to minimizing the TCB for a system. Furthermore, GBs are the only code blocks for static analysis by TSV. This is how the state space can be reduced significantly. Now, this GB-based policy enforcement takes account of three critical components of any security policy the subjects, objects, and operations. Each function block will have a label as the subject. Username and password verification are done to check whether a personnel is highly privileged to create or change a function block of the program. The object label helps to identify who has created or changed the code and on what program blocks when the code is uploaded to the PLC. Lastly, there are operations that define what actions are performed on an object. This model is useful in preventing untrusted blocks from outputting memory modifications. A guard block does not write to any region by adversaries when the GB-based approach is applied. However, this research did not make clear how they distinguished an important functional block from the others. An attacker can gain access to a function block which is not considered as GB. The attacker can perform a malicious activity from that code region.

The work of McLaughlin et al., [18] was extended by Zonouz et al., [28] with an effort to find malicious PLC programs. The same *trusted safety verifier* TSV was used to verify across the state-based model *Temporal Execution Graph* (TEG) [18]. Here, a formal predicate for each safety property is negated and a state-based model, UR, was generated that satisfies the negated formal predicate. Now, a state-based Cartesian product model from TEG and UR is generated and named as P. Now, each state in P has two substates that refer to corresponding TEG and UR states. If a path is found in P from the initial state, the PLC program has an execution trace

in the model based on negated safety requirements. Safety policy violating counter-example with path condition and input vector are generated from their developed framework. However, the state explosion possibility for this case was not discussed as it creates three different state spaces (TEG, UR, and P).

As can be seen from the discussion in this section, verification of PLC programs has been taken with great importance in the research community. Hence, we focus on some recent works on the formalization or transformation of PLC (IEC 61131-3) programming languages. Markovic et al,. [24] classified them in the following three categories:

- XML-based transformation
- compiler-based formalization
- model-checking transformation

The XML-based transformation of a PLC program was first introduced in Younis et al., [51]. The main motivation for this work was to develop a technique for visualizing PLC program by reverse engineering. This work contributes to being an intermediate step for the formalization of a PLC program in order to perform verification and validation of its system properties. XML (eXtensible Markup Language) is a flexible meta-language for describing other languages. Analysis of any code needs a scanner (lexical analyzer) to generate a set of terminal symbols and a parser that checks the grammatical structure of the code. The lexical specification is already an invariant component of XML. Moreover, an XML-parser is capable of transferring an XML document to an abstract interpretation named Document Object Model (DOM) without using a grammar. In order for this to happen, the first step is to transfer a PLC program to a well-formed XML document. XSLT, the transformation language of XML performs this step. Then, this XML file is validated against the XML schema to conform the syntactical rules defined in the context of PLC programming language.

The next step is the identification of the instructions of the transformed XML document. This proves the semantics of the XML document is in accordance with the operation types of the PLC programming language. XSL, another transformation language of XML, is used here to visualize XML in a two column table in HTML. The two columns are instructions and instruction Id, found from the previous step. There has been some XML-based PLC program transformation. One of the most notables works is done by Marcos et al., [52]. They discussed three approaches to transform a PLC program using the XML-based approach. XML Schema Technology is one of them. An XML schema contains the lexical and syntactical constraints that define a new language. The PLCopen TC6 software uses XML schema that contains the elements and their relations of the PLC program. XML stylesheet transformation is another way that helps in exporting or importing PLC code. The derived data types or user-derived program organization units (POUs) are imported through this way. Lastly, the XML interface helps in the interoperability of different types of the programming language of the PLC. It helps in transferring a code, or a complete project from one development environment to another without loss of information. However, this research did not indicate how these types of XML technology can help in verification directly.

At this point of this work, it is important to note that Structured Text (ST) programming language, one of the most used languages for PLC, is our point of interest for automation and verification purpose. Readers may wonder why we have chosen ST over the other four languages of PLC. Firstly, it is a high-level programming language that is easily readable and writable due to its similarity with a once well-used programming language Pascal. Secondly, the European Organization for Nuclear Research (CERN) writes most of their PLC programs in ST. Lastly and most importantly, ST can be used as a pivot language to represent all five standard PLC language [53]. Darvas et al., [53] have shown that all other PLC programming

languages (Functional Block Diagram, Ladder Logic, Sequential Functional Chart, and Instruction List) can be mapped to Structured Text (ST) language, specifically Siemens ST' language. Therefore, the rest of the section explores some verification techniques on ST language verification and test case generation.

Compiler-based PLC program transformation for ST language was done by Rzonca et al., [54]. They created an ST language compiler which produces universal code to be executed in different machines. They named the tool as CPDev. They have described the scanner, parser, and code generator of the tool they have developed. This research can be used as a good starting point for verifying ST programs. Some other attempts to convert the ST program to other languages have been made by Sadolewski where the ST program was converted to Why[19] and ANSI C [20] program respectively. Then verification lemmas have been used to correct the program with `coq` prover. However, these approaches have the limitation on data type declarations and specification language. Therefore, researchers in CERN made an attempt to generate formal models automatically from ST code [22]. It has three main phases:

- Parsing ST source code with the help of Xtext, an open-source Eclipse-based framework.
- Transforming the parsed ST source code to an intermediate automata model. This is done with EMF (Eclipse Modeling Framework) in order to hide some difficulties from the developers
- Creating NuSMV formal models from the model found from the previous step.

Some real-life case study with this tool was conducted in [23]. The case studies depicted that model checking can be included for a PLC even though the automation engineers are not experts of the formal verification. This work contributed to reducing the intermediate model of [22] and counterexample analysis. Moreover, the tool was extended to be capable of creating formal models for other model checkers. The

tool has later been extended in [55] and named as `PLCverif` with an editor for PLC programs. It has full support for Structured Text (ST) language and partial for SFC (Sequential Function Chart). One can write a ST program directly in the editor of the PLC and then verify it across the safety requirements. An already existing program can also be imported into `PLCverif` for verifying. The users can define safety requirement policies in English sentences after following the guidelines to write those. This is helpful because all users do not need to be experts on CTL (Computational Tree Logic) or LTL (Linear Temporal Logic). Policy specifying directly with LTL or CTL was also given as a choice nevertheless.

One notable work in this regard was found in the work of Markovic [24] where the framework has three main components, `Control Template, Read Input Unit`, and `Function/ FB unit`. The basic idea of `control template` is adopted from [25] where Functional Block Diagram (FBD), another PLC language, was converted to UPPAAL environment. This is actually an automata representing the actions in the ST program. The second component was the `Read Inputs Unit` that contains the input templates for the input variables for the ST program. The last unit is the `Function Unit` that implements the behavior of the transformed functions of original ST program. With these three components, an ST program is converted to UPPAAL acceptable format. After UPPAAL executes this upon the temporal logic property, an execution path for testing is generated.

Last but not least, we want to mention the work of Biallas et al., [21] where they developed a tool to verify the safety properties of PLC that are written in the ST programming language. This tool takes an ST program as input. A user can specify the safety and liveness properties in LTL or CTL formulas. The names of the variables of these formulas must be the same as those in the ST program we want to verify. Once it is run to verify a property, the tool outputs whether the formula is satisfied for the ST program. If it is not satisfied, a counterexample is formed with the variables

that have caused this violation.

2.4.2 Dynamic Checking

Nicolson et al., [29] proposed a novel approach for verifying a formal specification of ICS components during run-time. They have used the Interval Temporal Logic(ITL)/ Tempura framework in doing so. It is capable of providing an early warning system in a PLC-based environment. ITL is flexible for both propositional and first-order reasoning with respect to a period of time, found in the description of a hardware and software system [56]. It offers a powerful proof technique for reasoning about properties that involve safety, liveness, and projected time. ITL has a matured executable subset named Tempura. It has been used extensively for hardware simulation and other areas where timing is important. In this research, Tempura was extended to include notation for PLC. In order to test Tempura, two attacks were launched against Siemens S7 PLC. The first exploit was detected before entering the next expected state in the duration of time expected. However, the same attack could be detected with regular signature-based IDS. The second attack was based on uploading a slightly modified malicious code. This is not detected by an existing system as the signature matches normal behavior. Tempura detected this attack as the state transition does not match the Tempura formula with time constraints. However, this work did not mention how they have specified safety and liveness properties with time constraints for Tempura. Later, the same research group extended this work in [30] where ITL/ Tempura definition of a Siemens S7-1200 PLC ladder logic was presented. They have shown the operators like `And, Or, Xor, In, Out, Set, Reset, Latches, Triggers, etc` are converted in ITL specification. Their developed monitoring methodology captures a snapshot of the current state (with values for markers, input, output, counters, and timers) of the PLC. Tempura was implemented to execute directly on the Arduino Yun that uses MIPS. This is connected with PLC. This is how they ensured that PLC does not need a high power computing device to

perform this task.

2.5 Conflicts in Smart Building and Solution Approaches

We have discussed in a previous section that conflicting actions in a PLC-based system can lead to an unsafe condition. In this section, we discuss what type of conflicting situations are possible in a smart building environment and what is the current state of the art for solving those. As of now, there is no prototype or formalism that is considered complete in terms of defining conflict in the smart building environment. Nevertheless, Sun et al., [15] made an effort in classifying conflicts in some finite categories. They classified the relation among all building management rules into 11 categories. Based on these relations, they classified all types of rule conflicts into five categories and claimed that any type of rule conflict can be classified into at least one of those categories. The first type of conflict, the authors talked about, is *shadow conflict*. When one rule for an actuator cannot be triggered due to the prior engagement of that actuator by another rule, this is called shadow conflict. The second type of conflict is the *execution conflict* where two contrary actions are directed for execution at the same actuator. Then, *environment mutual conflict* was mentioned where two contrary actions are made possible to take place by two different actuators (i.e., cooler and heater). The last two types of conflicts are *direct dependence conflict* and indirect dependence conflict, where two and more than two rules create a loop for the actuators, successively. Thus, it is likely to create an anomaly in the smart building environment. All the conflict classifications have been defined by formal rules. That is, the classification can be implemented in any formal verifying or model checking tool. More than 30,000 rules have been implemented to test their conflict scheme. For implementing the rules in a real building, the researches employed the sensors and actuators through wireless sensor and actuator network [57]. Though the rule conflict space is claimed to be complete, this work did not have any soundness and completeness proof. Moreover, the policies they have provided as

examples, are very simple. The detection and resolving of complex rules were left as future works in their research.

It is worth noting that one of the important reasons behind the emergence and growth of highly integrated and intelligent buildings is reducing energy consumption. But after looking at the real world implementation, the reduction in energy consumption has not been kept out of questions. The automation itself utilizes energy. The sensors are sending the status of the lights, temperature, humidity, pressure, doors, windows, and amount of smoke to the controlled system at regular intervals. In the work of [5], the data collection rate from the sensors has been reduced based on different time period of the day and environment status. The authors provided an example where the temperature of the building in winter can be set within a threshold. The sensors do not need to report about the temperature at a small interval. Setting the appropriate time interval for reporting data from the sensor can be an interesting research direction in conflict resolution. Based on the sensor measurement, an event occurs and that event triggers the actuation. The more events in a limited time frame, the more the chances for possible conflicts of the actions. However, limiting the event occurrence in order to avoid conflicts is not a smart idea. Overlapping events should be dealt with to provide the best operation of a smart building. Analysis of event logs can help to find the allowable time frame for event overlapping.

Kumar et al., [58] pointed out some reasons why automation should be considered with great importance for smart buildings. They have explained those with some real-world examples. The building occupants are not necessarily the programmers. Nor they do know the overall infrastructure of the building. The less the interaction between the occupants and the sensors/ actuators, the more comfortable and safe the system becomes for the occupants. However, it is obvious that a machine cannot deal with a conflicting situation as efficiently as a human can, especially when conflicting situations occur. The authors studied how a change of policy in one device

triggers others. Trigger-action programming and SMT based logic have been proposed by Nacci et al., [59] for detecting conflicts in the policies automatically. The authors formalized the rules in propositional formulas by prioritizing them and analyzed the formalization using the SMT solver Z3. Their proposed tool *BuildingRules* uses their previously developed web service for building management system, named *BuildingDepot* [60] for implementing the whole system in a building. However, one can easily argue about the basic assumption of this paper. The authors stated that conflict detections will be done automatically by their model. It has not been made clear what different types of conflict are possible for a smart building environment. Unless the conflict detection problem has been specified in a finite space, the deployment of an automation system is not possible. In addition to that, their solution approach is more or less dependent on trigger action based approach (*IF* something happens, *THEN* do something). This approach needs a stateful policy enforcement mechanism, especially in the smart building environment. As for example, we can say that both a cooler can turn on and off at the same time in terms of a conflicting situation. If the trigger-action based approach is implemented, the cooler will be turning on and off simultaneously and cause major damage. Finally, this research work lacks the scalability of the model in detecting and resolving conflicts as real-world implementation has not been shown there.

At this stage, we want to point out that there are some other states of the arts for developing defeasible logic programming. The framework, developed by Lam and Governatori [61] is considered as one of the robust approaches to handle the conflicting situation. In fact, their developed tool *SPINdle* to write defeasible logic, form theory database, and answer queries is well explained and user-friendly. Answering queries or concluding whether a query is true in the underlying theory, is the most important aspect of defeasible logic. Unlike classical logic programming, defeasible logic programming is capable of proving a theory from contradiction, though defea-

sibly. Any classical logic programming tool would have disproved that theory in the very first place. It can reason with incomplete and contradictory information. A defeasible theory is defined by Lam and Governatori [61] as a triple $(F, R, >)$ where facts and rules are denoted by F and R, respectively. The sign $>$ is used to assign superiority relation among the conflicting rules. The definition of facts is as same as it is for classical logical programming. However, *rules* are classified into *strict rules* and *defeasible rules*. Strict rules (represented by $\Longrightarrow$) bear the same functionality and representation as it is for *rules* in logic programming. The most interesting thing in the framework is the introduction of defeasible rules. Those are the rules that can be defeated by contrary evidence. The important contribution of this work is that they have introduced superiority relation among the rules. We will discuss this more in the domain of our smart home environment. Another component of their model is the *defeaters* (represented by $\sim>$). The defeater is used to prevent any conclusion from being drawn. As for example, one can put directly that if the bird is a penguin, it must not fly. Generally, any logic programming outputs whether a theory is provable or not. In addition to that, SPINdle outputs whether a theory is defeasibly provable or not. Decision making can change a lot if a policy maker finds a rule defeasibly provable.

At this point, we want to note that although *SPINdle* is being considered as a robust solution approach for some domains like smart building, stock market, battlefield modeling, it lacks a theory grounding mechanism. That is, no variable can be used as a predicate. For a smart home environment, this characteristic gave more advantages. We will talk about this later when we discuss the research proposed by Stavropoulos et al. [3]. Another shortcoming is the absence of a probabilistic feature. Here, superiority is assigned among the rules in the very first place. But this may change from time to time. A dynamic framework can be developed with probabilistic modeling in assigning superiority.

Recent research of Stavropoulos et al., [3] made a proper use of defeasible logic programming tool *SPINdle* that had been proposed by [61]. They modeled a multi-agent system with each agent having an energy saving policy for each room in an intelligent building. A middleware was proposed that connects the hardware layer at one end and the logic engine on the other end through a semantic web service. The policies can be edited by authorized personnel. The basic goal of the paper was to model the system so that energy saving is achieved. The authors confirmed that 4% of energy savings was achieved when it had been implemented in a Greek University. The numerical value of 4% may sound small, but, if all commercial or industrial buildings are brought under such a model, the cumulative effects of energy saving will contribute significantly to achieving energy saving. Later, they extended their model to have a defeasible logic engine in it for detecting and resolving rule conflicts. The overall architecture is shown in Figure 2.3. The sensors sense the components of ambient and there are indirect many to many relationships among the sensors and the actuators via the middleware. That means there is no direct feeding of data from sensors to actuators. The sensed data is sent to the database and the database updates the knowledge base according to its measurement. The conclusion of a rule can serve as a condition for a further rule and thereby *SPINdle* enables complex reasoning. It is to be noted that defeasible logic is capable of reaching an agreement after solving conflicts and inconsistencies among knowledge items. However, in the proposed work, the authors did not secure the communication from and to the middleware. It is also to be noted that the authors claimed the deployment of their model in a smart home environment is far richer (with ZigBee, Z-Wave, and RF) with a communication protocol. However, they did not state any of the security issues that are possible in those communication protocols. The aggregative vulnerability can lead to life-threatening states of the system. Furthermore, a malicious attacker can act as the man in the middle and rewrite policies. The automation system, proposed in this

paper, defines a series of actions. But it was not defined when to take the series of events. The authors left this work out of the scope of this paper. We know that SMT solver is a well-established way to get a satisfiable solution for a system with constraints and policies. We believe that an SMT solver, embedded with defeasible logic, can come forward with more fine-grained solutions. Because time plays an important role in the smart home environment, the policies are likely to get changed at a different time. *Simulink* with SMT solver *YICES* can be used to ensure safe action of the actuators at varying time periods.

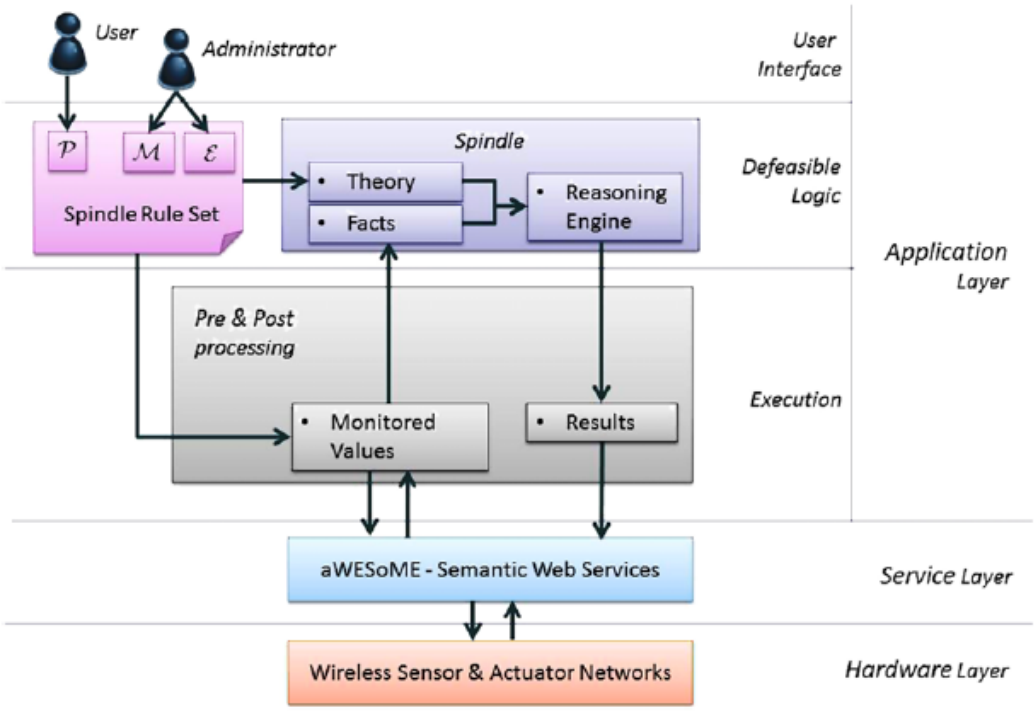

Figure 2.3: Defeasible Logic Agent Architecture for Smart Building [3]

The dealing of conflicting evidence can be better understood from the work of [62]. They chose the stock market as the domain to show how defeasible logic can help in solving the conflicting situation. In the stock market, the information keeps changing continuously. There are plenty of rules devised there to help the decision maker to decide when is a good time to buy a stock. Rather than thinking about a human in

the place of a decision maker, the authors proposed a multi-agent system that is able to monitor stock market, extract information and use defeasible logic programming (DeLP) to achieve the desired goal. By using defeasible logic programming, the multi-agent system can formulate arguments and counterarguments in deciding whether or not it will buy the stock. It is to be noted that DeLP is the extension of traditional logic programming and developed based on Prolog programs. DeLP has facts and strict rules the same as logic programming. In addition, it has presumption and defeasible rules. A presumption is similar to fact but it associates uncertainty on the evidence. When some rules are triggered based on presumptions, the rules are named as defeasible rules. In this paper [62], the authors gave an example where conflicting situations occurred while buying a stock. The multi-agent system was devised with different personality levels (e.g., very safe, safe, aggressive, risk-taker, brave) in taking a decision. The authors concluded that when the conclusions are derived from facts, the safe agents would buy the stock when it gets the conclusion to buy. If the multi-agent system is aggressive or risk taker, it would even buy the stock when the decision of buying the stock comes from presumptions. Though this work gives a basis in deciding with a different threshold of personalities, it does not give a clear idea which conflicting rule is winning over which and what is the basis for such a conclusion. The same research group later published a work [63] where a dialectical tree was proposed to show how an argument can defeat another to reach a conclusion.

It is noted by [63] a defeasible rule represents a weak connection between the head and body of a rule. Rather than going through any particular check, the knowledge base keeps changing with belief. This feature is the main contribution of defeasible logic. In fact, when contradictory goals are achieved, but defeasibly, DeLP provides argumentation formalism to validate the goal. Arguments are minimal and non-contradictory set of rules to reach a conclusion. The main point in their formalism is

that despite being the contradictory nature of DeLP, answer to a query must be supported by a non-contradictory set of rules. Moreover, they signified the importance of changing facts or knowledge base for which defeasible derivation takes place. They clearly mentioned that a conclusion from strict rules is more preferred than a conclusion from defeasible rules. Due to the nature of incomplete or changing knowledge, it is not easy to derive to a conclusion easily from strict rules. In order to explain which arguments defeat which arguments they proposed a dialectical tree. This type of tree is helpful for a better understanding of the conclusion that comes from a large pool of arguments and counter-arguments. The arguments and counter-arguments are also derived from sub-arguments when the rule base is large for a problem. Except for the root node, each parent node is the defeater of its child nodes. Two types of defeaters were introduced there: proper defeater and blocking defeater. An argument is a proper defeater of a counter-argument when that argument has higher priority over the counter argument or at least one sub-argument of the counter-argument. On the other hand, the concept of blocking defeater comes when both arguments and counter-arguments do not have a preference relation over each other. When the rule base is large, the authors showed that this type of situation is possible. It is important to observe that in this research, the priority relation of the rules is kept fixed. With the change of facts or knowledge-base, it is also possible the priority relationship among the rules can be changed. The priority change in rule base can be an interesting research direction.

Later, Martinez et al., [64] made an effort to make DeLP robust by full integration of presumption in reasoning. In that work, the authors defined presumption as a piece of information which is not fact or true always, but for the sake of reasoning, it will be considered as true tentatively. The main motivation of this research was that when evidence is gathered from multiple sources, it is not easy to scrutinize them as all are considered as facts. They explored a new idea where some evidence

is considered as presumptions, not facts. The proposed framework comprises of two model, *environmental model (EM)* and *analytical model (AM)*. An environmental model describes the preferences of the evidence in the knowledge bases. On the other hand, an analytical model is what we call queries in typical logic programming. It analyzes competing hypotheses for a scenario. Generally, the *analytical model* gives the framework the capability to reason out a conclusion of a given query. It is important to note that both EM and AM share the same variables and constants. In EM, the authors defined a formula that is built by predicates and boolean operators. Then, for each formula, the probability is assigned to each formula with an error tolerance. AM, on the other hand, helps the framework to warrant a conclusion based on the comparison given in EM. In this work, the probability to statements in EM is assigned either by a knowledge engineer or an automated system. However, it was not clear how an automated system can reason out for such probability assignment. The work of [65] can help in this regard. The main contribution of that will be discussed later in this dissertation. However, that work does not talk about the scalability issue as the search space for probability assignment would grow larger and larger with the number of evidence. The number of pieces of evidence increases the size of the knowledge base. Hence, the modeling of an automated system should consider the freshness of facts in the knowledge base.

2.6 Data Provenance in Security

Data provenance is defined as the "history of data transformed by a system" [66]. The provenance of a piece of data describes what the inputs and outputs of each process are, what processes were executed, and who had control of those processes during execution. Provenance is often represented as a directed acyclic graph (DAG) with nodes representing the data, processes, and controlling entities. The edges represent causal relationships between these nodes. For example, we provide a provenance graph in Figure 2.4 that represents the relationship and workflow among different

components of a smart building that are responsible for changing the status of a light. This is a simple example with three agents, two activities, and three entities. As can be seen from the figure, there are nodes of three different shapes. The agents are represented with pentagons, activities with rectangles, and entities with ovals. The workflow explains how a *Motion Sensor* changes the status of a light. An *Occupant* can also turn on/off the same light, which is explained in the graph as well. The *Motion Sensor* is associated with the activity *Motion Sensing* (represented by *wasAssociatedWith*). This activity generates *Motion Status* which is connected by the edge *wasGeneratedBy*. This *Motion Status* is used by another activity *Light Actuating*. This activity is associated with *Light* agent. Whenever *Light Actuating* activity finds motion from the *Motion Status* entity, it turns on the light and changes the *Light Status* activity. It is interesting to notice that the same *Light Actuating* activity is associated with another agent *Occupant* and uses the entity *Switch*. This *Switch* is operated by the occupant to change the *Light Status* entity.

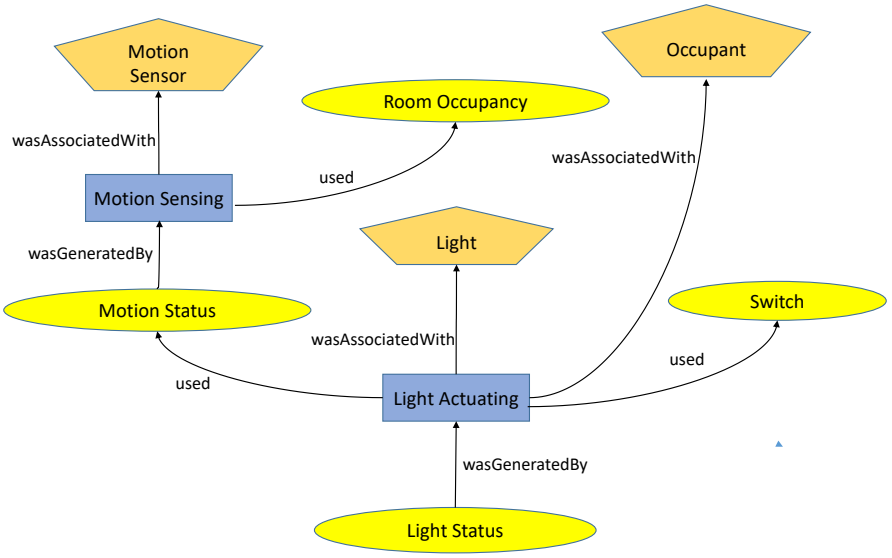

Figure 2.4: Notional Provenance Graph for a Light Actuation in a Smart Building

Provenance was first introduced in databases and computational sciences for tracing and debugging. However, more recently it has been proposed as a primitive for

building secure and resilient systems [67] that can "fight through" attacks. In order to provide such capabilities, novel collection, storage, and analysis mechanisms have been proposed to enable near-real-time analysis of provenance to support security and resilience decisions [68]. These mechanisms are being used to provide forensic analysis and intrusion detection capabilities [69].

As discussed in this chapter, conflicts in smart environment are dealt with deliberately by the research community. Defeasible reasoning is used for nearly two decades to resolve conflicts among information and policies. Though smart buildings have not been under cyber attacks so far, the PLCs used there were under some notable cyber attacks. As of now, static analysis approaches have been the most used methods to secure a PLC-based system. Recently, data provenance has been well-accepted to secure domains where the change of information needs to be tracked in runtime. It will be interesting to observe how provenance and defeasible reasoning can be combined to secure a PLC-based system, smart building.

CHAPTER 3: Detecting Conflicts in Smart Buildings

3.1 Introduction

3.1.1 Problem Statement

The distributed nature of a smart building leaves devices and communication channels exposed to attackers and many of these devices and protocols are resource-constrained. Moreover, thousands of policies in a building can lead to misconfiguration. An attacker is capable of triggering an event that leads to conflicting actions in the building and can achieve his goal by shortening devices'lifespan. Moreover, it is also possible for an attacker to leverage these conflicts and vulnerabilities to gain physical access to a smart building. Additionally, a series of attacks can be launched (cascading attack) [70].

Even with smart building technology in the early stages of development and deployment, the guarantees of maintaining the safe and secure operation of an environment in a smart building domain can attract more users. Even legacy, or dumb devices can be attached to the system and have effects on actions or environmental features.

Moreover, although the additional energy usage due to a single device's conflict may look negligible, the cumulation of excess energy usage of the whole system over a certain period can have a significant financial impact. Reduction of energy usage is one of the essential smart building technology contributions.

3.1.2 Contribution

In this chapter, we propose `IoTC`^{2}, a formal methods approach to ensure safety properties for the controllers and actuators in an IoT system. The main contributions of this research are:

- a formal approach to defining the safety properties of an IoT system
- a technique for detecting conflicts within the rulesets of the IoT system that violate the safety properties of the system
- an implementation of IoTC^2 that can be used in real-time to ensure the safety and energy efficiency of the IoT system

An IoT system provider, or the application developer, can leverage our framework while writing rules for an IoT system. We prove that the classification of safety properties, is sound and complete. Once the conflict creating actuators/controllers are distinguished by our framework, the application developer can modify operational rules to avoid conflicts as much as possible. If conflicts are unavoidable due to the uncertain nature of events, it is possible to manage them with the help of data provenance and defeasible reasoning.

Furthermore, IoT vendors can instrument the controller or application with our policies to monitor how many safety property violations have occurred and their impacts on environment and energy usage. This helps in classifying and differentiating anomalies in an IoT system. It should also be noted that actuation commands are issued from the controllers. Hence, multiple controllers that try to command the same actuator are in scope for this work. IoTC^2 contains all the rules for the IoT system so that when events trigger a rule or a set of rules, IoTC^2 makes sure the safety properties are maintained. When a violation of the safety properties occurs, it is due to conflicts in the rules defined for the system. The framework is also capable of detecting conflicts that are caused by misconfiguration of operational rules. The conflicts are detected even before the conflicting actions take place. IoTC^2 can identify the specific type of violation that has occurred. We have implemented an IoT environment simulation in Matlab's Simulink environment to understand the impact of conflicts in an IoT system.

3.2 Background

A basic IoT system is comprised of a number of sensors, actuators, and controllers sometimes combined into single devices. The sensors collect measurements of the current state of the environment or the system. There are operational rules stored in the controller. These rules are triggered based on events collected by the sensors. The rules define what actions should be taken by the various actuators connected to the controller. Simply put, an event triggers a rule and the rule triggers an action within the IoT system. The controller decides what action or set of actions to take, as defined by the rules installed in the controller. The controller issues commands to the actuator for performing the most appropriate action. The devices are connected through wired or wireless networks. While several components of an IoT system are interacting with one another as soon as an event occurs, it becomes challenging for the system itself to function without making conflicts. Conflicts, in this case, refer to the situations when more than one rule (triggered by an event) try to access the same actuator or affect the same environment feature at the same time. When the rules are written for automation, it is difficult to consider all potential conflicts. They occur when multiple events trigger at the same time. Conflicts are regarded as the most common, yet unsafe situations for a cyber-physical system [71, 15, 72, 73]

3.3 IoTC^2 Framework

The architecture of the IoTC^2 is given in figure 3.1. All rules/logic used for operation in each controller of the IoT system are the input for IoTC^2. Whenever there is a change in any rule, or a new rule needs to be added to the controller, the change needed is input to IoTC^2. The second type of input for our framework is the sensor measurement with timestamps. These two types of input act as the facts for Prolog logic. Three types of safety properties are defined in IoTC^2 namely controller safety policies, action trigger policies, and event handling policies. Regardless of the sen-

sor measurements and operational rules input to IoTC^2, it can find safety property violation in a complete and sound way (more about the proof is discussed in Section 3.4). To start, our framework receives copies of the traffic from the sensors to the controllers. As soon as IoTC^2 receives a sensor measurement, by using the rules/logic from the controller, IoTC^2 determines what actions the controller will emit for the actuators. Next, IoTC^2 determines the list of actuators, affected features, and issuing controllers. Using these lists, IoTC^2 determines whether or not these activities violate the safety properties (by creating conflicts) within the IoT system. IoTC^2 has the capability to output the number of conflicts and their type within the IoT system. In addition to conflict creating events, misconfiguration of rules within an IoT system can lead to a set of commands that can violate the safety properties of the system, which IoTC^2 is also capable of detecting.

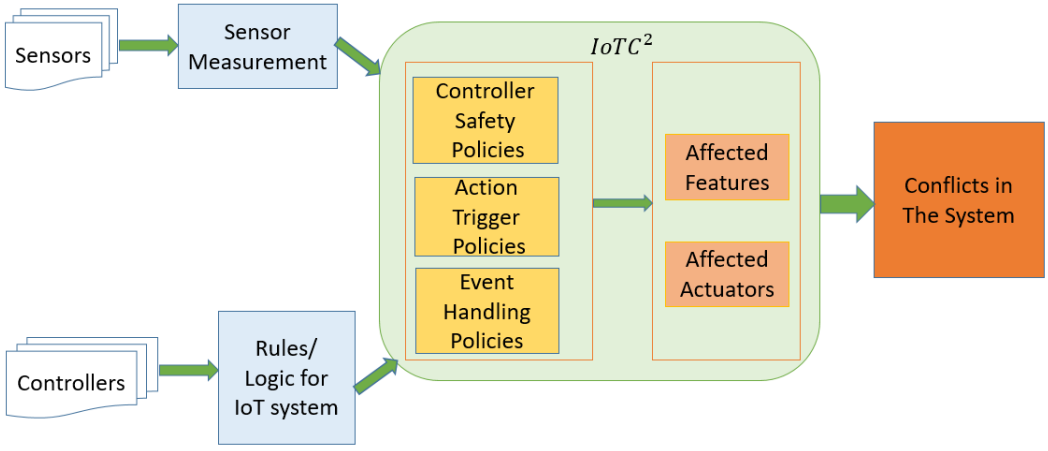

Figure 3.1: IoTC^2 Framework for Conflict Detection

While it is feasible to detect safety policy violations through formal methods, it is not easy to estimate the real world impact of conflicts with such techniques. The input to a system is important. Moreover, the cumulative effect on the environment features impact the energy usage. Hence, IoTC^2 needs to be implemented in a real world or a simulation test bed to characterize the real world impact of conflicts. When any of the safety properties are violated (controller safety policies, multiple action

trigger policies, multiple event handling policies), they affect the actuator states or environment features, which in turn cause different types of events and trigger the controller to issue conflicting commands to actuators. To calculate the impact of these commands,, the measurement from sensors and the actions from the actuators are necessary. The architecture for measuring impact of conflicts that are detected by the safety properties of IoTC^2 are given in Figure 3.2. The solid lines refer to the flow when no conflicts are assumed to occur at the system. However, in reality when conflicts occur it is captured by IoTC^2. Sensor measurement, actuator action, and controller's commands are monitored by IoTC^2. From these, energy usage overhead is calculated. The flow is represented by dotted lines. If safety property violations hold more frequently or for a long period of time, energy usage will differ notably from the expectation.

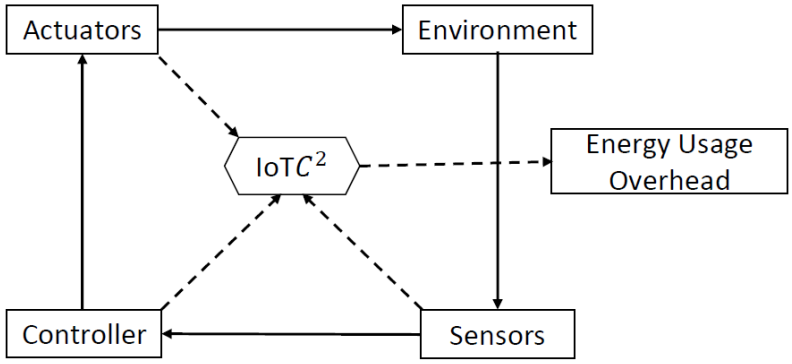

Figure 3.2: IoTC^2 Framework for Energy Usage Calculation due to Conflicts

3.4 Formal Method for Detecting Conflicts

In this section, we present our formal method for detecting safety property violations in IoT systems. In the following description, we rely on the notation defined in Table 3.1. We start by defining some general properties of the IoT system model, events, triggers, and actions.

Definition 1 *If there are features f_x and f_y such that changes in f_x affect f_y, then*

Table 3.1: Notation used

Notation	**Explanation**
e_i^t	Event i generated at time t
$a_{m,n}^{l,f}(t)$	Actuator m taking action n on feature f at location l
c^p	Controller p
x'	Object x' can be the same or different from object x
$\bar{x}$	Object $\bar{x}$ must be different from x

these features are dependent. It can be the case that feature f_x and feature f_y are not directly dependent, however feature f_x affects f_z and f_z affects f_y. In this case, f_y is indirectly dependent on f_x. Regardless, they are noted as:

$$f_x \stackrel{d}{=} f_y \tag{3.1}$$

Example: Temperature and humidity can be considered as two dependent features. Change in the temperature of a room leads to the change in humidity of that room [74]. Additionally, if a thermostat is shared between two rooms and corridors, change of temperature alone in one room can lead to temperature changes in the other room and corridor.

Definition 2 *If there are two events that are the same or similar by their characteristics and functionality and they occur within a bounded time frame, these events are called overlapping events. They are noted as:*

$$e_1 \stackrel{o}{=} e_2 \tag{3.2}$$

Whenever two events are not overlapping, they are considered disjoint events.

3.4.1 Controller Safety Policies

The controller is a crucial component of an IoT system that receives measurements from sensors and based on those measurements, it generates actuation commands for the appropriate actuators. We define the controller safety policies as follows:

- There are no two rules where two or more controllers can trigger the same actuator at the same time.

$$C_1 \colon \neg((e_i^t \Rightarrow a_{m \in c_p, n}^{l,f}) \wedge (e_{i'}^t \Rightarrow a_{m \in c_{\bar{p}}, n'}^{l,f'})) \tag{3.3}$$

  An IoT system has a number of rules for operating where the same actuator m is controlled by more than one controller c_p and $c_{\bar{p}}$. If the same actuator is accessed at the same time t, a conflict occurs. The actions (denoted by subscript n and n') on the actuators can be same or different, which does not change this policy. The affected features in this case are made different (superscript f and f') because the difference does not impact the safety policy. The impact of the potentially different features in creating conflicts is discussed in Section 4.4. The following are examples of conflict scenarios that can be captured by this rule.

  - A smoke detector and a water-leak detector can each trigger an alarm. We assume that the smoke detector and water-leak detectors are controlled by different controllers. But if the same alarm sounds at the same time, it will be hard to distinguish which event triggered the alarm. The policy formalized here restricts multiple controllers from triggering the same action at the same time.
  - Motion detected inside an elevator triggers the controller to prevent the elevator door from locking. On the other hand, an alarm in the building

will lock the door of the elevator so that no one can use it during the alarm. Here, the door of the elevator is the actuator while the action on it is operated by two controllers.

- There are no two rules where two controllers can trigger actions that affect the same, or dependent features at the same time.

$$C_2 \colon \neg((e_i^t \Rightarrow a^{l,f}_{m \in c_p, n}) \wedge (e_{i'}^t \Rightarrow a^{l,f'}_{m' \in c_{\bar{p}}, n'}) \\ \wedge (f = f' \vee f \overset{d}{=} f')) \tag{3.4}$$

There can be more than one rule that can trigger different actions (the subscript m and m' denote different actuators). However, these different actuators can impact the same, or dependent features. The same features are identified with '=' . On the other hand, the notion $\overset{d}{=}$ denotes the dependency among two features f and f'. An example of such a violation is:

  - A window opener and a thermostat are two actuators in a room which can be controlled by different controllers. However, their actuations can affect the same feature (temperature) of that room.

3.4.2 Multiple Action Trigger Policies

When an actuator is issued commands to perform multiple actions at the same time, conflicts can occur. In order to prevent conflicts, we have the following safety property:

- There are no two rules where two or more overlapping events (from any sensor) can trigger multiple actions on the same actuator.
  - Different action n'
  - Opposite action $\bar{n}$

– Dependent action $\hat{n}$

$$C_3 \colon \neg((e_i^t \Rightarrow a_{m,n}^{l,f}) \wedge (e_{i'}^t \Rightarrow (a_{m,\hat{n}}^{l,f'} \wedge a_{m,n^*}^{l,f'} \wedge a_{m,\bar{\bar{n}}}^{l,f'})) \\ \wedge (e_i^t \overset{o}{=} e_{i'}^t)) \tag{3.5}$$

In this formula, two events are differentiated by the subscript i and i'. The measurement of a sensor can trigger an event depending on whether that measurement is 'more than' or 'less than' or 'equal to' a set value. We define overlapping events as two or more events that occur within a specific time frame, or are related by their signature. The overlapping relation between two events e_i^t and $e_{i'}^t$ is denoted by $e_i^t \overset{o}{=} e_{i'}^t$. These overlapping events can trigger different actions (e.g. increase temperature or decrease temperature), opposite action (e.g. open the door and close the door), dependent action (beeping and flashing light on an alarm) and the same but overlapping action (increase temperature on a thermostat twice within 5 second). An action n on actuator m is the reference action and any action (n', $\bar{n}$, or $\hat{n}$) other than n is considered as the conflicting action on the same actuator m. An example of this safety policy violation is given below:

– Both room one and room two have temperature sensors but no thermostat. The corridor that joins both rooms has a thermostat, but no sensor. Temperature decreases in room one and temperature increases in room two can trigger the same thermostat. Hence, the thermostat can be triggered to increase the temperature and decrease the temperature at the same time.

- There are no two rules where overlapping events can trigger actions that affect

the same or dependent features.

$$C_4 \colon \neg((e_i^t \Rightarrow a_{m,n}^{l,f}) \wedge (e_{i'}^t \Rightarrow a_{m,\bar{n}}^{l,f'}) \\ \wedge (e_i^t \overset{o}{=} e_{i'}^t) \wedge ((f = f') \vee (f \overset{d}{=} f'))) \tag{3.6}$$

When an action is performed by an IoT device, it may affect one or more features (e.g. temperature, humidity, or luminance). There are some features the actuators affect directly while some features are impacted indirectly. As an example, humidity and temperature are considered as dependent features [75]. When the temperature goes up, it affects the humidity of a room if no moisture is added. This is because warm air can hold more water vapor than cool air. We differentiate two features by f and f'. There can be two rules that get activated at the same time by two overlapping events, resulting in two different actions, n and $\bar{n}$. These two actions affect features f and f' which are either the same or dependent. The following are examples of conflicts when this safety policy is violated:

- Room one and room two have a shared thermostat in room one. Hence, room one gets hotter than room two when the thermostat is turned on. Based on the temperature reading from room one, the thermostat is asked to turn off. However, the temperature measurement from room two will ask the controller to turn on the thermostat again. As mentioned earlier, temperature and humidity are dependent and therefore it is possible that humidity in two rooms will vary. If both rooms share a humidifier that is placed in room 2, the overlapping events can turn the humidifier on or off simultaneously. Apart from depending environment features being affected, more energy is needed for additional actuations.
- Luminance levels can be impacted by overlapping events. Window blind

and room lights are two different actuators that impact luminance.

- No two or more completely disjoint events can trigger multiple action on the same actuators

$$C_5\colon \neg((e_i^t \Rightarrow a_{m,n}^{l,f}) \wedge (e_{i'}^t \Rightarrow (a_{m,\hat{n}}^{l,f'} \vee a_{m,n^*}^{l,f'} \vee a_{m,\bar{\bar{n}}}^{l,f'})) \\ \wedge\neg(e_i^t \stackrel{o}{=} e_{i'}^t))) \tag{3.7}$$

In a large IoT system, it is not easy to distinguish overlapping events. Hence, we turn our attention to modeling the safety properties that are based on disjoint events. It is possible that these disjoint events are overlooked when devising the IoT operational rules, yet these rules can create conflicts within a single actuator. Examples for such conflicts are:

  - Management can impose a rule stating that when it is after 6 pm, the temperature need not be controlled. This means that the temperature of a smart building will follow the basic thermal model of the building. On the other hand, movement in a room will trigger the thermostat to increase/decrease the temperature for better occupant comfort.
  - Smoke detection and carbon monoxide detection can be two completely disjoint events that trigger multiple alarms to sound at the same time.

- No two completely disjoint events can trigger multiple actions that affect the same, or dependent features.

$$C_6\colon \neg((e_i^t \Rightarrow a_{m,n}^{l,f}) \wedge (e_{i'}^t \Rightarrow a_{m,\bar{n}}^{l,f'}) \wedge \neg(e_i^t \stackrel{o}{=} e_{i'}^t) \\ \wedge((f = f') \vee (f \stackrel{d}{=} f'))) \tag{3.8}$$

Two rules can be triggered by completely disjoint events at the same time. The actuator and its location are kept unchanged by notation m and l, respectively.

The actions are differentiated by n and n'. At the same location and with the same actuator, two features f and f' got affected. When these two features are dependent, $f \stackrel{d}{=} f'$ will return true.

- A window opening or closing and a thermostat turning on or off are two completely disjoint events that impact the temperature of the room
- Management can impose a rule that when it is after 6 pm, the thermostat and the humidifier should not be adjusted. On the other hand, movement in a room will trigger the thermostat to increase/decrease the temperature for better comfort. This has affect on humidity as these two features are dependent.

3.4.3 Multiple Event Handling Policies

A controller actuates an IoT device when an event occurs. There is relatively little control over how and when an event is generated. However, the way multiple events are handled, can be controlled. Therefore, we focus on formalizing event handling policies. The formalization is as follows:

- No single sensor with single objective can create more than one event within a specific time limit.

$$C_7 \colon \neg((e_i^t \Rightarrow a_{m,n}^{l,f}) \wedge (e_j^{\bar{t}} \Rightarrow a_{m,n}^{l,f})) \tag{3.9}$$

Here, two events i and i' are prohibited from same sensor j within a time limit t'. A sensor might send the same measurement to the controller more than once due to any physical or communication issue. This should be handled in a proper way so that same actions are not taken by the same actuator.

A sensor can send the same temperature measurement (e.g. 60F) twice to the controller within a 30 second interval. The controller would instruct the

thermostat to increase the temperature by 10F each time it receives the input from the sensor. Therefore, the temperature of the room is increased to 80F.

3.4.4 Completeness of IoT Safety Properties

Definition 3 *If an IoT system, comprised of sensors $s_{1...m}$, controllers $cntrl_{1...n}$, and actuators $a_{1...o}$, violates any safety properties $c_{1...p}$, a conflict $c^* \in Conflict$ has occurred (Here, $\in$ denotes* `belongs to`*).*

Completeness is the property of being able to prove all true things. That is, in order to declare our formal method complete, we have to make sure that existence of all conflicts can be found through IoTC^2. In order to analyze the safety policies formalized above in terms of controllers, triggered actions, and event handling, the policies were implemented using Prolog. If there exists a conflict $c^* \in Conflict$ in the IoT system operations, IoTC^2 finds it using the backward chaining. Prolog querey evaluation employs *Selective Linear Definite-clause with Negation as Failure* SLDNF [76]. However, the Depth First Search (DFS) strategy of Prolog makes it logically incomplete. With this strategy, the search begins from one node and traverses a single path to find the query answer, i.e. looking for conflicts. Whenever a conflict is found in the search space, Prolog does not traverse that branch to find another conflict even if another one exists. Our approach solves this limitation.

We followed the way proposed by [77] where the built-in dept-first search of Prolog was overruled. Rather, the implementation was based on iterative deepening of the query where each recursive call takes place at the top level of a conditional. In doing so, we have used tail-recursion. In each recursive call, the number of actions triggered, or the controllers associated with it, or the events that triggered the actions are stored in separate lists. In this way, the search space is being completed in lists. Then this list is traversed to find the conflicts in the system. Whenever, a resolution refutation is found, our Prolog implementation finds it and adds 0 (zero) to the accumulator

rather than stopping the search process on that node.

The iterative deepening strategy is potentially inefficient. One could argue that the size of lists will grow unbounded. However, we want to point out that the formulation for conflicts is dependent on the specific time the event has occurred. Say two actions a_1 and a_2 are triggered at the same actuator x at time t_1 and t_2. A conflict C occurs if and only if $t_1 = t_2$. These two actions cannot create conflicts on the same actuator if they are triggered at different times. If there are a total of m automation/operational rules in all the controllers of an IoT system, there can be at most n rules that are triggered at time t and it is obvious that $n << m$. Simply put, the search space for finding all conflicts in a given time is certainly limited.

3.4.5 Soundness of IoT Safety Properties

Definition 4 *The safety properties of* $\texttt{IoTC}^2$ *are sound, if for all sensors* $s_{1...m}$*, controllers* $cntrl_{1...n}$*, and actuators* $a_{1...o}$*, all possible operations in the system are a subset of the authorized operations allowed by* $\texttt{IoTC}^2$*.*

Soundness is the property of being able to identify the existence of something (conflict in our case) if that is proved by the system. In our case, if $\texttt{IoTC}^2$ finds a conflict it must exist in the system. If there exists a conflict in the IoT system, yet $\texttt{IoTC}^2$ cannot detect it, we call it unsound. As mentioned earlier, $\texttt{IoTC}^2$ is implemented in Prolog where it searches all safety property violations until it finds the ground truth. The ground truth are provided in the form of sensor measurement and operational rules. If there exists any resolution refutation, our implementation must find it because tail recursion is used there to avoid the built-in DFS of Prolog.

With the definition of soundness from 4, we can postulate that all safety properties, expressed in conjunctive normal form (CNF) make it logically sound as given in 3.10.

$$C_f : C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge C_6 \wedge C_7 \tag{3.10}$$

If the soundness and completeness conditions fail for $\texttt{IoTC}^2$, the negation of C_f in 3.10 will provide us an example of unsafe situation of the IoT system. More simply, a conflict has occurred.

3.5 Threat Modeling

An attacker can compromise an IoT system in different ways. As mentioned earlier, the IoT sensors have no or few security features in them. An attacker can target sensors and spoof it with falsified data. Additionally, an adversary can record sensor data and launch a replay attack to impact actuators and environmental features differently. However, our formal framework is not capable of detecting the availability of the devices, For example, a Denial of Service attack can take place whenever control and non-control traffic of IoT systems mix up in the network and some IoT communication packets drop. Our framework, in this case, is not capable of detecting such behavior of a network. It only detects attacks that are associated with actuations on the same devices/ environment features and the events that trigger those actuations. In this section, we focus on the threats that are associated with the policy violations that are discussed with respect to the seven safety policies we have formalized. An attacker succeeds in creating conflicts if he is capable of violating any of the safety properties. We define the situation as follows:

- An attacker (A) succeeds to compromise an IoT system if he violates c_i, where $i = 1, .., 7$. Formally,

$$A = \bigvee_{i=1..7} (\neg C_i) \tag{3.11}$$

The ways for compromisation is discussed as follows:

Reconnaissance of Physical World: An attacker can gather information by observing the physical components and environment around the IoT domain. The conflict between management rules and occupancy detection can be discussed in this context as explained under equation 3.7. An attacker can gather temperature sensor

data and figure out that after 6 pm the temperature of the rooms does not increase at all. This information entices an attacker to inject false information that motion has been detected in the building. This will force the controller to turn on the thermostat and control the temperature. In doing so, the attacker succeeds in causing additional energy usage in the smart building.

Reconnaissance of Smart Home Applications: Smart home users employ a number of applications for their home automation. Some applications (e.g., SmartThings mobile application) are self-published and do not go through an official review process. Most importantly, the source code is shared in a community forum [78]. An attacker can gather information from multiple application's source code and create conflicts from there. For example, consider a water leak and smoke detector who use the same alarm system. An attacker gathers this information after reviewing the corresponding applications. He can compromise the water leak detector to turn on the alarm. When the alarm keeps running for a long time, another application asks the sprinkler to turn on, having assumed that smoke is detected for such a long period of time. Water from the sprinkler can damage some important documents and electronics.

Random Compromise of Sensors: Due to resource limitations, sensors have few or almost no security features in them. An attacker can target some random sensors with the aim that the forged data will have chaining effects in creating conflicts. Moreover, because the same network is used for IoT and non-IoT traffic, it is also possible to affect the availability of some sensors and initiate replay attacks on the system. This would cause property damage or more energy usage in the long run.

3.6 Implementation

In order to evaluate the efficiency of our approach, we created an IoT environment using Matlab's Simulink. The basic purpose was to observe the interaction among the sensors, actuators, and controllers. Simulink is a robust and widely ac-

cepted simulation tool for power electronics, nuclear energy, manufacturing production, aerospace, transportation, supply chain management, etc. IoTC^2 was implemented in the Simulink testbed to monitor the testbed and detect conflicts. We designed a smart house with three rooms with corridors attaching each of the rooms. The thermal model of the house was adapted from [74]. The rooms have facilities for smoke detection, carbon monoxide detection, smart lights, smart window shutters and blinds, smart doors, etc. Operational rules for automating these IoT devices are implemented using appropriate Simulink blocks. The changes in the temperature and humidity of the smart house are designed using the same thermal model of the house to get reasonable heat transfer from the outside environment [74]. We ran each simulation five times and took the average to observe how accurately IoTC^2 can detect conflicts in the smart house successfully. We have used the built-in model verification blocks (e.g., Assertion, Check Dynamic Gap, Check Dynamic Range, etc) of Simulink to help our detection. IoTC^2 outputs the total count of each different type of detected conflict in a given simulation step regardless of how many events have occurred during that step. Not only are the detected conflicts shown, but also their effects on features or actuators. The Simulink model for the testbed is available in [79].

3.7 Evaluation

The evaluation section is divided in three parts.

- Can our approach measure the impacts of conflicts on environment features?
- Is there additional energy usage due to conflicts and can our approach estimate that?
- How many conflicts can our model can capture over varying simulation time?

For our evaluation, we created an IoT environment using Matlab's Simulink. The thermal model of the house was adapted from [74]. The rooms have facilities for

smoke detection, carbon monoxide detection, smart lights, smart window shutters and blinds, smart doors, etc. We ran each simulation several times to observe whether IoTC^2 can detect conflicts in the smart house successfully. We have used the built-in model verification blocks (e.g., Assertion, Check Dynamic Gap, Check Dynamic Range, etc) of Simulink to help our detection. IoTC^2 outputs the total count of each different type of detected conflict in a given simulation step. Not only are the detected conflicts shown, but also their effects on features or actuators.

3.7.1 Conflict Impact on Environment Feature

First, we evaluate the effects of conflicts on actuators or the environment feature. Our first experiment was conducted to observe whether two different actions on different actuators affect the same environmental feature. A smart window shutter is sometimes operated by the occupant with the help of an app from a tablet or smart phone. Similarly, the shutter is operated to keep open at different times of the day. On the other hand, the smart home is automated to turn the smart lights on whenever it finds movement in a room. In our setup, the probability of both the opening up the window shutter and turning on smart light was set as 0.10. The conflict occurs during a split second of time when both the window shutter and smart light turns on or off together given that someone is in the room. The luminance of the room then gets out of the range (> 450 or < 200) compared to a comfortable luminance range. The simulation was run for 500 units of time. Results of this experiment are shown in Figure 3.3. As can be seen from the figure, the luminance of the room exceeds the set bounds for a comfortable luminance level.

The second experiment measures the change in temperature of a room when a smart window shutter is opened or closed at the occupant's preference. As with the previous experiment, the window shutter can be opened from the occupant's smart phone or tablet. In addition to that, if the carbon monoxide of the house increases beyond a certain level, the smart home is instructed to open the windows immediately

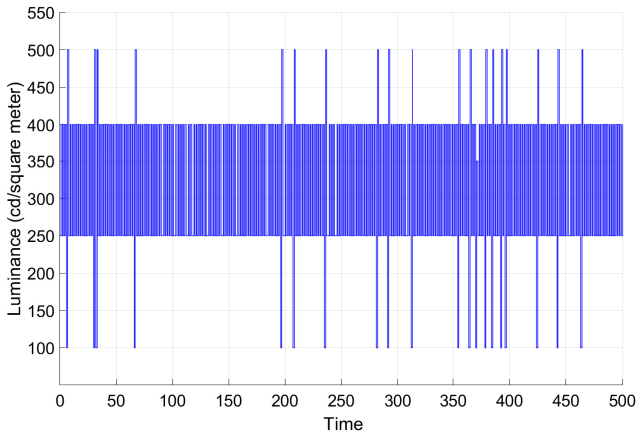

Figure 3.3: Luminance range of a room when smart window blinder and smart light are considered

for fresh air. This changes the temperature and humidity of the inside of the house. We designed this scenario in our Simulink environment and used the same thermal model of the house to get reasonable heat transfer from the outside environment [74]. The results of this experiment are shown in Figure 3.4. The blue line indicates the expected temperature reading. However, the red line in this figure indicates how much the temperature reading deviates due to conflicts between the window shutter opening and the thermostat operating rules. This can result in more actuation of the thermostat which will be discussed later in Figure 3.10.

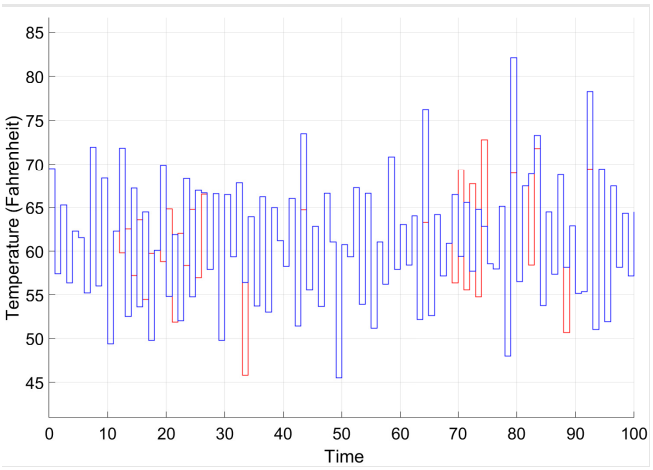

Figure 3.4: Effect on Temperature when thermostat and window shutter works at the same time

The third experiment captures the impact on temperature of a shared corridor

between two rooms. The two rooms are set to have different temperature based on the occupants' preferences. Let us assume the corridor has the thermostat, but no temperature sensor. On the other hand, both rooms have individual sensors. We also assume that room one has the window open which affects the normal temperature of the room by lowering the ambient temperature of room one. On the other hand, room two has no influence from the outside environment, i.e., the window in room two is closed. However, due to two different sensor measurements, the thermostat is instructed to turn on and off more frequently than expected. The temperature, as shown by the red line in Figure 3.5, is the temperature of the corridor, calculated by Simulink. The blue line is the temperature reading during different times from room two. Our intuition says this blue line is supposed to be the temperature reading of the corridor, but it is impacted by the temperature from the other room.

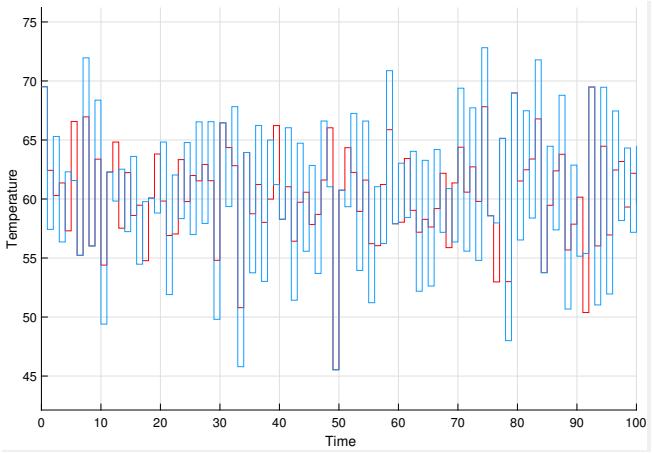

Figure 3.5: Effect on temperature of the corridor when it is connected by two rooms of different temperature

The fourth experiment characterizes how changes in one environment feature can influence the another environment. Here, we consider the fact that the temperature in a smart home is dependent on thermal radiation and humidity as it is mentioned in [75]. The humidity changes with the temperature of the room. The temperature changes with thermostat and humidifier. Also, the window shutter opening changes the temperature and humidity of the room. The scenario described above is tested

and shown in Figure 3.6. The expected humidity reading is shown by the blue line, while the real humidity reading is shown by the red line. Events like the window shutters opening and the humidifier running have caused the temperature of the room to fluctuate. This deviation is overlooked, yet creates discomfort for the occupants and more energy usage due to additional actuation.

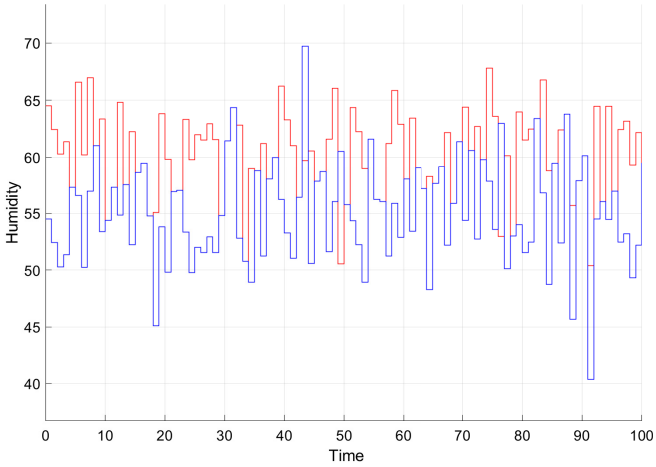

Figure 3.6: Effect on Humidity when thermostat and window shutter combinedly changes the temperature and humidity

3.7.2 Conflict Impact on Energy Usage

To this point, we have observed how the environment features change due to conflicts. Now, we conduct the second type of experiments to observe the difference of energy usage due to conflicts. For this experiment, we choose a test case where temperature in a room is affected by two factors, the state of a window, and outside temperature variation. It should be noted that other factors like humidity, lighting condition, and occupancy level were ignored for this experiment. We keep those parameters constant in order to find how window shutter and outside temperature affect the energy usage by varying the values. The observation is shown in Figure 3.7. For better understanding, our simulation test bed measures the energy usage as shown by blue line. Later, we run the experiment with IoTC^2 employed in the test bed. As can be seen from the figure, even with only 7% chance of a window being opened, and 5F temperature variation outside the smart home, our system captures considerable

overhead energy usage. Apparently, both values are reasonably low to mimic a real world scenario. We increased both the probability of the window being opened and temperature variation slightly more for the next two runs of our experiments and found the energy usage is getting higher.

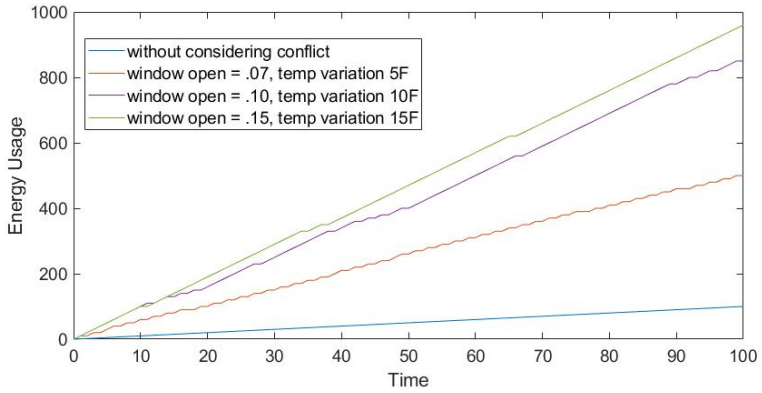

Figure 3.7: IoTC^2 Calculates Energy Usage Overhead due to Conflicts

The above experiment was conducted only on the actuation on the thermostat of a room. The next experiment is conducted to observe total energy usage of a smart home. The line shown in Figure 3.8, is the difference between observed and expected energy usage over simulated time. The observed energy usage is calculated with IoTC^2 monitoring each actuation. On the other hand, the expected energy usage assumes that energy usage due to conflicts is negligible. Unfortunately, the figure 3.8 clearly shows that conflicts have affects on energy usage. Due to the conflicts, there are additional count of actuations to devices. Some actuations continue for longer time in addition to their frequencies.

3.7.3 Conflict Count with IoTC^2

To this point, we have conducted our experiments to characterize how different types of conflicts can impact the actuators or the environment features. Now, we move our focus on counting the number of conflicts in a given time by varying different parameters. First, we consider the case where the same alarm (actuator) is triggered when smoke is detected or a water leak is detected. The rules for triggering an alarm

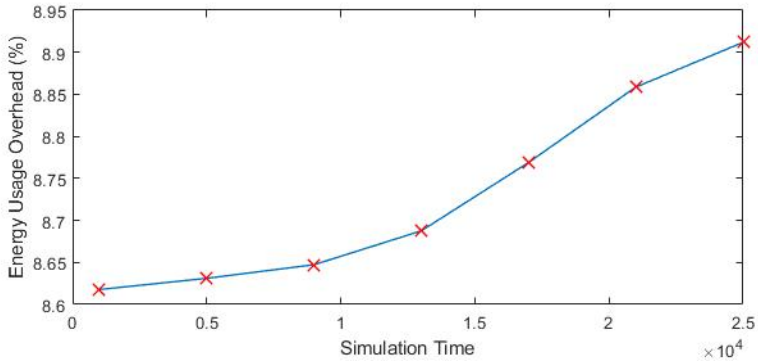

Figure 3.8: IoTC^2 Calculates Energy Usage Overhead Over Time

in those cases are installed in two different controllers. We consider the probability of smoke detection = 5% and water leak detection = 7% in each time unit. The simulation was run for 2000 time units which is a large number compared to the experiments conducted to get the impacted environment features. The reason for using such a large number is to get more data points about conflicts. It is shown in Figure 3.9 that the same alarm is triggered by different events at the same time with the increase of simulation time. Next, we increase both smoke detection and water leak detection to 10% and find the increase in total conflict count. This type of conflict should be considered as an attacker can compromise only the water system, yet convince the occupants of the smart home that the alarm is due to a fire. That eventually can lead the occupants/target of a smart home to evacuate.

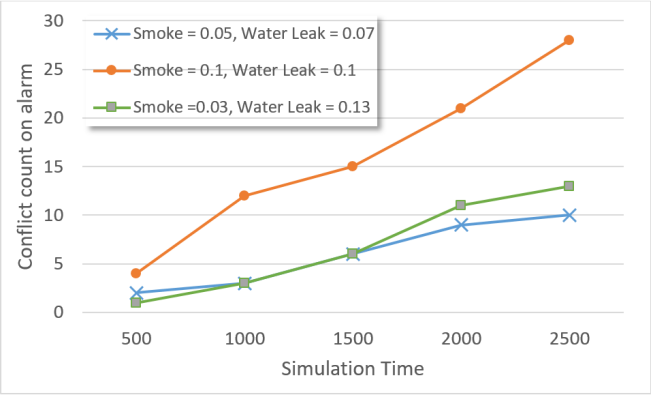

Figure 3.9: Conflict count when the same alarm is triggered by multiple events

In the next experiment, we count the number of conflicts when the window shutters of the smart home are open and the thermostat turns on at the same time. The impact

of such conflicts is shown in Figure 3.4. As can be seen from Figure 3.10, the number of conflicts increases when the window shutter is opened more frequently.

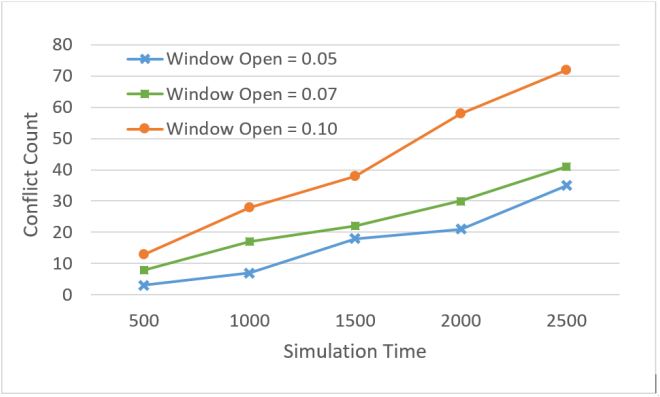

Figure 3.10: Frequency of thermostat being actuated more than usual due to the window being opened

Next, we considered humidity as a dependent feature of temperature. The thermal model of the house is kept the same. The humidity changes based on the temperature which triggers the humidifier. The effect of conflicts in such cases are discussed in Figure 3.6. First, we run the simulation as if the humidity is not affected by temperature. Next, we re-run the simulation with humidity being dependent on temperature. We count the number of times the humidifier is turned on due to conflicts. As can be seen in Figure 3.11 when there is more temperature variation outside the smart home, the humidifier inside the house is turned on more often.

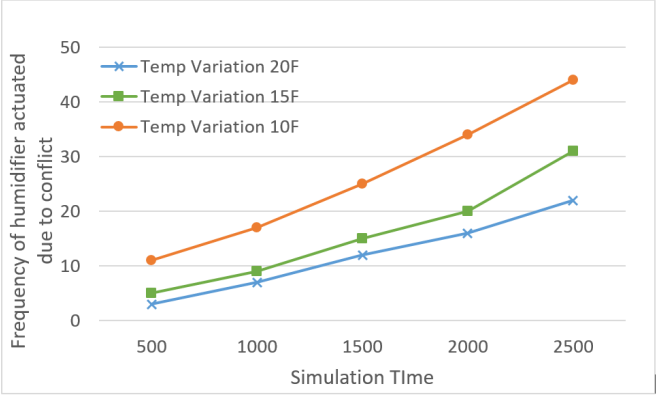

Figure 3.11: Additional actuation count on the humidifier due to temperature difference in two adjacent rooms

We have observed in Figure 3.3 how conflicting actions can affect the same feature.

We have seen that the luminance of the room can exceed a comfortable range due to conflicting actions between the smart lights and the smart window blinds. We next count how many times this conflict happens over a specific period of time. We set the probability of the window blinds being opened to 2%. This is a very low number which indicates that only 2 out of 100 unit times a window blinder is opened. We then increased this probability while keeping the probability of turning the light on constant (at 10%) which is reasonably low because if there is an occupant and the window blinds are closed, then the smart lights will turn on immediately. However, the purpose of this experiment is to show that the luminance can get out of range even though there is a low probability of conflicts. Figure 3.12 shows that the luminance of the room is out of range more frequently when there is a greater chance of the window blinds being opened, given that the smart light is turned on at the same moment. Similarly, some incidents were observed where a room is too dark when both the window blinds and the smart lights turn off.

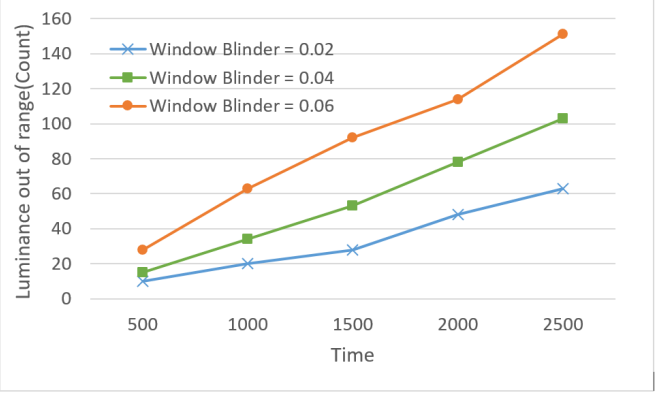

Figure 3.12: Total count of the luminance range exceeding the comfortable range due to conflicts

The next experiment measures the number of conflicts between management rules and operational rules for an IoT system. For a smart building, the management rules may state that the thermostat is not active after 6pm. However, if there are lots of people in a room with their mobile devices or computers on, the temperature of the room will increase. The regular operational rules will then tend to actuate

the thermostat to cool the room. Here, we run the simulation for a room in two different ways simultaneously while keeping all parameters same. In the first run, we consider that the occupancy of a room has an effect on temperature and the thermostat is actuated by following basic operational rules. The second run assumes that room occupants and electronic devices do not have a measurable impact on the temperature of the room and that the management rules dictate the operation of the thermostat. Our goal is to observe whether conflicting events cause more actuation of the thermostat. As expected, more actuations of the thermostat are needed when conflicts occur. The building management authorities either overlook this additional actuation, or ignore the comfort of the occupants. The additional number of actuations is shown on Y-axis of Figure 3.13. This number increases when the simulation is executed for a longer time.

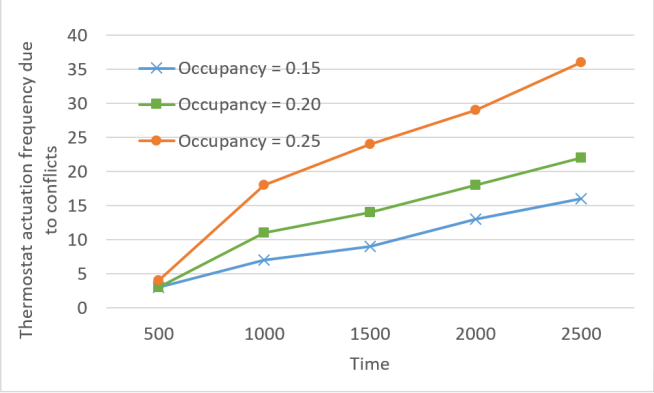

Figure 3.13: Additional count of the thermostat being actuated due to conflicts between management rules and operational rules

The last experiment conducted, is very similar to the previous one. Here, the count of humidifier actuations is measured due to conflicts between operational rules and management rules. We counted the number of additional actuations needed for the humidifier in the smart home. When there are more occupants, the air quality degrades and the temperature of the room changes as well. The humidifier turns on to make the environment of the room more comfortable. However, the management rule stipulates not turning on the humidifier after a certain time of the day (say 6 pm).

There can be conflicts between management rules and regular operational rules based on the occupancy of a room. Similar to the previous experiment, we ran simultaneous experiments with different assumptions. During the first run, we assume that room occupancy has negligible impact on the humidity and that the smart building will be operated based on the management rules. For the other run, we make the opposite assumption, namely that the occupants in the room will have a measurable impact on the humidity of the room. The results are shown in Figure 3.14.

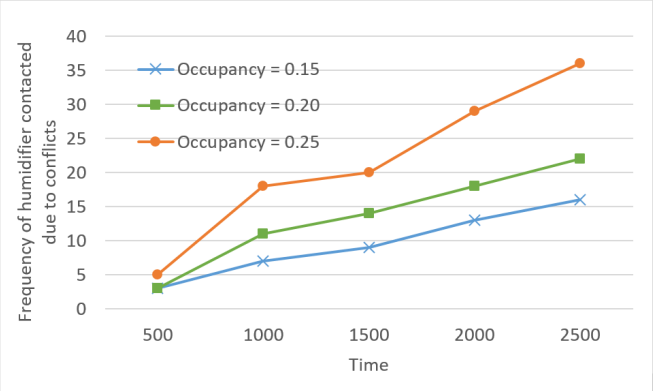

Figure 3.14: Additional frequency of humidifier being actuated due to conflicts between management rules and random occupancy

3.8 Related Work

Most of the research efforts in IoT has been on management, efficiency, interoperability, and deployment of these systems in the real world. Recently, confidentiality, access control, privacy, and trust issues of IoT technology have been discussed in [80, 81, 82]. In IoTSAT [83], a formal framework was proposed for security analysis based on device configurations, network topologies, user policies, and IoT-specific attack surface. Recently, the work by [84] proposes a verification framework with satisfiable module theory (SMT) for a smart environment with respect to event-condition-action (ECA). However, this research did not propose either safety properties or address conflicts for the smart environment. The work by [15] made an effort to specify the relation among all building management rules and classified all type of rule conflicts into five categories. Our approach is quite different. Rather than

classifying conflicts, we specified safety properties for the components of IoT and the violation of those properties leads to conflicts. CityGuard [73] proposed an approach to intercept actions for a smart service to detect and resolve conflicts for smart cities. The authors argued that safety requirements vary under different contexts with different granularity and therefore the safety requirements do remain the same for different domains. We claim that our formal methods approach covers a complete set of basic safety properties to detect conflicts, thereby applicable to any cyber-physical domains (e.g., smart home, smart building, smart city, smart transportation).

Some preliminary work in the areas of formal modeling and verification for IoT driven domains has been done [85, 86]. The closest to this work in terms of detecting conflicts are Depsys [71], and HomeOS [87]. Depsys specified and detected conflicts after collecting the functionalities of 35 smart apps used in smart home. It detects the conflicts after they have occurred and in order to address the conflicts priorities have been set to the apps so that no two apps can access the same actuator. HomeOS is a large-scale application running on a centralized server that enables devices to talk to one another. That is, it acts as an event handler which aids in resolving device conflicts. Our approach, on the other hand, detects conflicts as soon as an event is generated which may immediately cause an action or a set of actions that result in conflicts. We have left the automated resolution of conflicts as future work.

3.9 Discussion

The conflicts that are possible in IoT systems are often overlooked both in the design phase and during operations. IoT is an automated system and hence the accumulated effects of conflicts on an environment feature or actuator can have more effects than initially anticipated. The safety and security of IoT systems is largely dependent of its conflict-less behavior. Hence, the safety properties we formalized in $\texttt{IoTC}^2$ consider conflicts as the preeminent threat to the safety and security of IoT system. Furthermore, our model has shown how conflicts can lead to additional

actuations which in turn resulted in more energy consumption. In addition to the contributions mentioned above, we believe our proposed framework will have significant impacts when employed in the *policy monitor* block of *ProvThings* [32]. However, the enforcement of the proposed safety policies for a system is an open challenge. The implementation of an inlined reference monitor (IRM) [88] for enforcing the safety policies of our framework is another interesting research direction. IoT systems are emerging more and more in our daily life, and mechanisms are needed to ensure that these systems are safe, secure, and energy efficient if IoT systems are to be widely deployed and accepted.

CHAPTER 4: Detecting Safety and Security Faults in PLC Systems with Data Provenance

4.1 Introduction

4.1.1 Problem Statement

One of the biggest problems with smart building networks is that the security of these control networks is limited at best. The communication protocols used in these control networks lack authentication and integrity checking for messages [17]. These weaknesses make it possible to initiate many commonly-known attacks such as man-in-the-middle attacks, denial of service attacks, memory corruption attacks, replay attacks, and spoofing attacks. While enterprise networks can rely on a wide-range of security mechanisms including IPsec, transport layer security (TLS), and virtual private networks (VPNs) to secure their communications, such mechanisms are difficult to deploy on these control networks, leaving them vulnerable to network-based attackers. Furthermore, many of the commonly applied mitigations fail to cover PLC-based systems [89].

What is needed are mechanisms that can monitor the inputs and outputs of the ICS and ensure that critical safety properties are not violated. This requires an understanding of the desired safety properties, a way to track inputs and outputs, and a mechanism to model the evolution of the system from inputs to outputs. With these mechanisms in place, it becomes possible to ensure that the PLCs do not send commands to actuators (i.e., the devices that interact with the physical environment) that violate the safety and security policies of the system.

4.1.2 Contribution

In this chapter, we propose PLC-PROV, a mechanism to track the inputs and outputs of the system and compare them against the specified safety and security properties. PLC-PROV relies on tracking *data provenance* for the PLCs and using that provenance to determine if a violation has occurred. Provenance, in short, is the "history of data transformed by a system", and has been proposed as a building block for systems that require the ability to reason about the *context* in which an action is taken. Since PLCs are entirely event-driven, context is vitally important, and as such provenance is a natural fit for this sort of analysis.

4.2 Background

4.2.1 Data Provenance

Data provenance is defined as the "history of data transformed by a system" [66]. The provenance of a piece of data describes what the inputs and outputs of each process are, what processes were executed, and who had control of those processes during execution. Provenance is often represented as a directed acyclic graph (DAG) with nodes representing the data, processes, and controlling entities. The edges represent causal relationships between these nodes.

Provenance was first introduced in databases and computational sciences for tracing and debugging. However, more recently it has been proposed as a primitive for building secure and resilient systems [67] that can "fight through" attacks. In order to provide such capabilities, novel collection, storage, and analysis mechanisms have been proposed to enable near-real-time analysis of provenance to support security and resilience decisions [68]. These mechanisms are being used to provide forensic analysis and intrusion detection capabilities [69].

4.3 Design

Due to the distributed nature of PLC systems an attacker can trigger an event that leads to conflicting actions for the same object or feature of the plant/environment. Let us consider a smart building as an example where PLC is used [16, 33, 35, 38, 39]. An attacker can compromise a carbon monoxide detector and trigger a false alarm by indicating an unsafe carbon monoxide level. This sends a command to the windows to open allowing fresh air into the building and trigger an audible alarm. Occupants will evacuate the building and a thief can use the open windows to enter the building. Another example is creating multiple events that trigger a thermostat to increase and decrease the temperature of a room at the same time. Sending two different commands in the thermostat at the same time continuously can damage it, by artificially shortening the device's lifespan. In this way, the attacker not only damages an asset but also may drive the occupants of the room to leave due to fluctuations in the comfort level of the room. In addition to the attacks described above, an attacker can create a series of attacks or a cascading attack [70]. Moreover, misconfiguration in the smart building operation is possible as there are numerous rules or policies for taking actions by the controllers after events have occurred.

4.3.1 PROV Modeling

Our approach to addressing the requirements in `PLC-PROV` is to identify the common concepts present in smart building systems as shown in Figure 4.1 and define a unified provenance model for smart building based on the W3C PROV-DM [90].

Figure 4.1 is a notional example of the type of graph that is generated. Let us consider a conference room in a smart building which has two temperature sensors and one thermostat as the actuator. Therefore, we have three agents *Temperature Sensor 1*, *Temperature Sensor 2*, and *Thermostat* in the system. Two *Sensing* activities are associated with two separate sensors and one *Actuating* activity is associated with

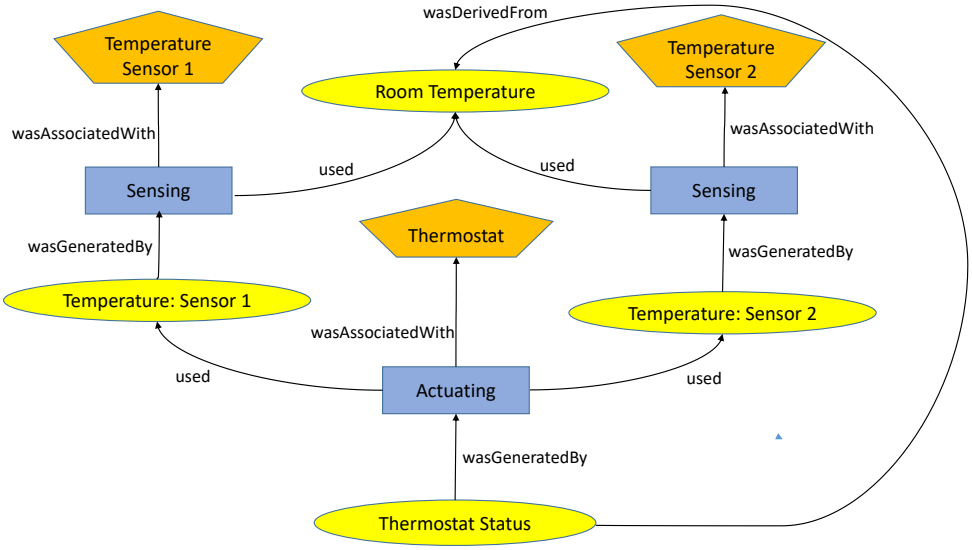

Figure 4.1: Notional provenance model of a PLC-based system for the change in temperature of a room

Thermostat agent. The *Sensing* uses the room *temperature entity* for performing their activities. Similarly, *Actuating* uses two separate entities of temperature to change the *Thermostat* status. The sensors send the measurements to the PLC. Based on the rules programmed into the PLC, it issues commands to the *Thermostat* actuator for necessary action. Because more than two entities are used for *Actuating* activity, it can create conflict on changing the thermostat status. (Please refer to 4.4 for a detailed conflict detection scenario).

As much as the example in Figure 4.1 explains how provenance captures the flow of a smart building system, we need a generic provenance model to capture whatever is happening with the automation of smart building systems. With such models, we are able to utilize provenance data that are collected from `Codesys` *Traces*. A unified model with causal relation enables the same terminology for provenance to be used on any PLC-based system. The general model is shown in Table 4.1. We map each concept to the PROV model to capture the overall behavior of the system. For example, a motion detector (sensor agent) senses the environment condition (environment condition entity) and sends that to a PLC by performing the sensing activity. The

Table 4.1: Model for Representing Provenance of a PLC System (Smart Building)

Concept	**Description**	**Prov Model**	**Example**
Sensor	A device to collect specific environment condition	Agent	motion detector, temperature sensor
Actuator	A device that changes environment condition	Agent	light, thermostat
User	A person that changes environment condition by actuating a switch	Agent	Occupants of a building
Management	A person/ group of persons who imposes automation and emergency rules on devices	Agent	operator
Action	A command issued by the controller to change the state of a device	Activity	Turning Light on/off
Sensing	Activity to capture the environment condition	Activity	Motion detection
Environment Condition	Measurement of sensor about the environment condition	Entity	Temperature, motion state
Actuator State	Current state of an actuator	Entity	Light state (on or off)

PLC sends an actuation command which is performed by turning on/off (actuating activity) a light (actuator agent). That is, the turning on/off uses the environment condition entity to produce a new light state (actuator status entity). Eventually, this action affects the ambient condition (environment condition entity) again.

In order to understand the graph in a better way, let us go through the notations used there. An agent for a device d is expressed as $d\_agent{:}x$, where x is the data sample index. For the same data index x, the notation for an activity for the device d is $d\_activity{:}x$. Entity has a slightly different notation as $d{:}x$ for the same device. The value y for an entity is given as $ex{:}value\ y$.

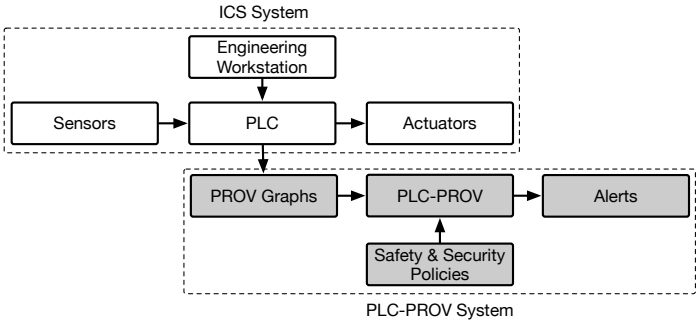

Figure 4.2: PLC-PROV architecure

4.3.2 PLC-PROV Architecture

A formal methods approach for detecting conflicts in IoT systems is presented in [91]. A PLC-based system (e.g., smart home, power supply, water supply, wastewater management, and traffic control system) works on the same sensor-actuator-controller functionalities. Therefore, we adopted the safety policies defined in [91] as the basic policies to analyze using the collected provenance graphs. PLC-PROV will check whether there is any violation of the defined safety and security properties. If found, it is reported as an anomaly in the system, which also provides the administrator with the detailed traces that are needed to understand the impact of the anomaly and aid in root-cause analysis. The basic architecture is shown in Figure 4.2.

To start, our framework traces the variables designated for the sensors and actuators that are connected to a PLC. The controller issues commands to the actuators based on these sensor measurements. These variables map sensor inputs and actuator actions, enabling interaction between the various components. A PLC contains the core rules/logic, written in any of the five programming languages of IEC-611131 standard, for controlling the plant/environment.

A PLC has to be operated with software that provides interaction with the sensors and actuators. In our research, we use CODESYS[1] which is a development system for PLC applications. Variable values are collected by CODESYS with timestamps into traces of system execution.

[1] https://www.codesys.com/

These traces are the input for our developed provenance management tool where we have used an open source provenance management library, `prov` [92]. This library minimizes integration complexity for the application developers. It is also capable of integrating provenance the multiple abstraction levels, a feature that enables reasoning about provenance both at the sensor reading level (micro) and at the environment/plant level (macro). As `prov` is targeted for a microservice-based system, it fits well for our case where we collect traces from disparate components of the system, similar to microservices. Moreover, `prov` has incorporated the use of `NetworkX` [93] which provides more flexibility for researchers and developers to traverse through a complex network of provenance data. With the help of `prov`, a provenance graph is generated, showing the evolution of the system from sensor readings through the PLC and finally to the actuators. The steps for provenance recording and graph generation are given in Algorithm 1. Initially, our provenance graph is empty (line 2). It is generated while parsing the $Traces$, collected from `Codesys`. The $Traces$ contain the labeling that reveals active agents in each time-slice (line 3). The automation rules are input so as to find the relation mapping Map of sensors and actuators for each command (line 4-6). These steps generate activities (i.e., sensing, and actuating) and put them into Map. With the help $Traces$, the entities (e.g., sensor measurement, actuator action) are recorded for each device in a given timestamp. Map has the relationship stored among different components of a system (i.e., wasAssociatedWith, wasGeneratedBy, used, wasDerivedFrom). $Traces$ and Map together contribute in generating the provenance graph. Finally, data in $Traces$ are stored in the provenance graph P (line 7- line 15).

The provenance graphs collected by `PLC-PROV` enable an administrator to answer the following questions:

- Has an actuator been actuated more than once at the same time?
- If an actuator receives multiple commands at the same time, are these same or

Algorithm 1 Provenance Recording and Graph Generation

0: **procedure** RECORDPROVENANCE($Traces, Rules$)
0: $\emptyset \leftarrow$ ProvenanceGraph P
0: $agent(1, .., i) \leftarrow$ *labels of Traces*
0: **for** each rule r in Rules **do**
0: $Map(1, .., m) \leftarrow sensor(r) \cup actuator(r)$
0: **end for**
0: **for** each timestamp t in Traces **do**
0: $entity(1, .., j) \leftarrow Traces(t) \cup Map$ {sensors'}
0: $entity(1, .., k) \leftarrow Traces(t) \cup Map$ {actuators'}
0: $activity(1, .., l) \leftarrow Traces(t) \cup Map$
0: $p(t).wasAssociatedWith(agent(i), activity(l))$
0: $p(t).used(activity(l), entity(l))$
0: $p(t).wasGeneratedBy(entity(m), activity(l))$
0: $p(t).wasDerivedF(entity(m), entity(m-1))$
0: $P \leftarrow P \cup (p(t))$
0: **end for**
0: **return** P
0: **end procedure**=0

different?

- What are the reasons behind conflicting actions?
- Which are the sensors influencing conflicting commands?
- Has any sensor measurement gone beyond normal range? If yes, how many times did that happen and how long did it last?
- Are there more than two actions affecting the same environment feature?

In order to detect conflicts in a PLC-based system, we propose Algorithm 2. The goal of this algorithm is to count the conflicts as well as determine the affected actuators, affected environment features, and the sensors behind these conflicts. We initialize these parameters from line 2- line4. According to our provenance graph generation, the status of a feature or actuator state is derived from the its prior state. That is, when we find a feature (an entity in the graph) is changed, we traverse through the provenance graph to find which activity has generated this entity. Then,

we search more in the graph to see whether more than one entities were used by this activity. If we see that is the case, our next step is to find the value of the entities. If we see that there are more than one entities with value 1, we can conclude that the actuation was conducted by two entities which were generated by two different agents. Our algorithm reports a conflict and appends the affected feature and actuator in a list, $featureList$ $deviceList$, successively.

Algorithm 2 Conflict Detection with Data Provenance

```
0: procedure DETECTION(ProvenanceP)
0:   ∅ ← deviceList
0:   ∅ ← featureList
0:   0 ← conflictCount
0:   for each actuation activity a in graph P do
0:     entities en_k ← used by(a)
0:     if Σ_{k=1}^{K} en_k > 1 then
0:       value(n) ← entityGeneratedBy a
0:       value(m) ← entityWasDerivedTo n
0:       if value(n) ≠ value(m) then
0:         get agent ag(1,...,i) wasAssociatedWith a
0:         List entity e(1,...,j) wasUsedBy a
0:         conflictCount ← conflictCount + 1
0:         deviceList ← deviceList ∪ ag(1, ..., i)
0:         featureList ← featureList ∪ e(1, ..., j)
0:       else
0:         get agent ag(1,...,i) wasAssociatedWith a
0:         conflictCount ← conflictCount + 1
0:         deviceList ← deviceList ∪ ag(1, ..., i)
0:       end if
0:     end if
0:   end for
0:   return conflictCount, deviceList, featureList
0: end procedure=0
```

4.4 Evaluation

In order to evaluate how our system works, we designed a testbed for a smart building system. A smart building is an exclusive example of a PLC-based controlled system. Our evaluation is based on a number of safety property violations. Even-

tually, we define these violations as conflicts. At first, we describe the conflict type and their description as per [94]. We map safety property violations of a smart building to the conflicts. `PLC-PROV` captures these conflicts which we represent with two provenance graphs of two different safety property violations. Lastly, we demonstrate how efficiently `PLC-PROV` captures these conflicts with respect to collected samples and conflict count. The testbed design is described as below:

4.4.1 Testbed

Our testbed provides a flexible platform to connect sensors and actuators to our PLCs. We have used Wago [2] controllers with several input, output, and end modules. The input and output modules support only 24V power through the Wago 787-612 power unit. Because we have used some devices that operate on 5V and some on 250V, we used a relay switches and optocouplers to complete the system. These devices enable our testbed to support devices at multiple voltage levels. To include a variety of sensors and actuators, we have used a Raspberry PI 3 (RPI3), and an Arduino Uno. These devices act as remote terminal units (RTU) to collect sensor measurement for the Wago PLC. Arduino has the capability of including a good number of digital and analog sensors in the most cost-effective way compared to Wago digital and analog modules. Apart from connecting a good number of digital sensors, RPI3 can host a server to send all the sensor measurements, collected from both Arduino and RPI3, to the Wago PLC. The Arduino was connected to the RPI3 through a serial port. A webserver in RPI3 sends collected sensor measurements to the Wago PLC using the `WagoLibHTTP` library. In this way, the Wago PLC is still our primary controller to execute the rules and therefore, does not change the impact of the work. The testbed is shown in Figure 4.3. The devices that are used in this testbed are given in Table 4.2:

[2]https://www.wago.com/us/building-technology

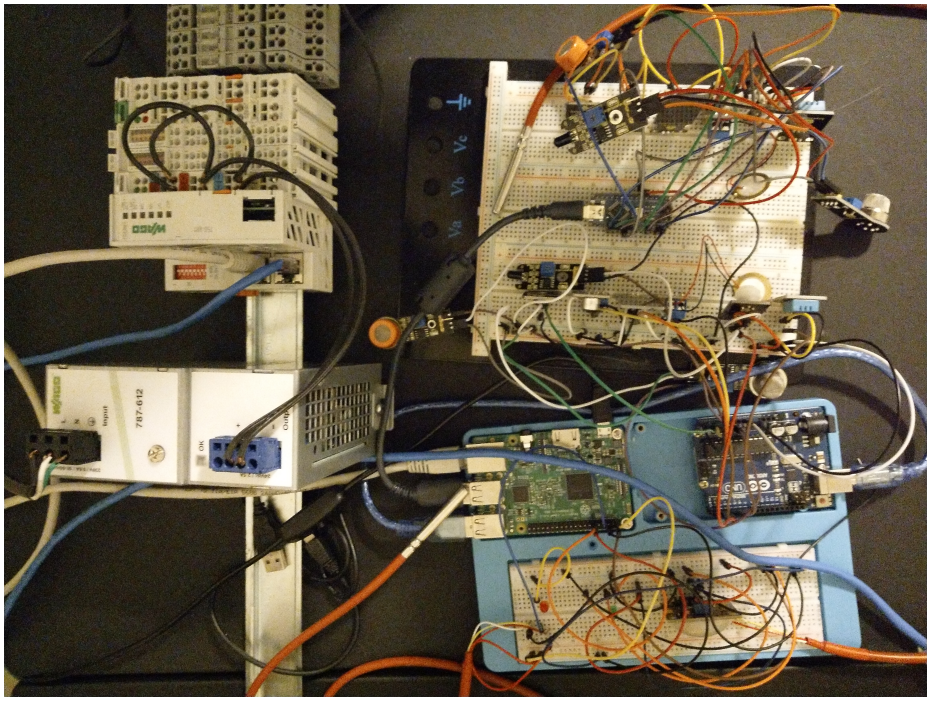

Figure 4.3: Smartbuilding testbed

4.4.2 Safety Property Violations

We identify some potential safety property violations which are derived according to our testbed as designed in 4.4.1. These are the properties that are checked for violation over some variable period of time. Whenever these violations occur, they create conflicts. These property violations are proposed in such a way that they refer to conflicts only as proposed in [94]. That is, all the safety property violations are categorized among those seven conflict creating scenarios. The conflict types and corresponding descriptions are given in 4.3.

For our testbed, we have defined 27 safety policies as described in Table 4.4. Violation of any leads to a conflict. The traces we collect from our testbed comprise of all the variables that store the measurement and states of sensors and actuators. If we collect data of only 1 sample size, at the most extreme case, it is possible that all 27 safety policies are violated. That is, there are 27 conflicts in the system for that collected data. It is also possible that no conflicts are found in that collected sample.

Table 4.2: Testbed for PLC-PROV Evaluation

Manufacturer	**Device**
Wago	750-881 controller, PFC 200 controller
Wago	750-1405 â 16 channel digital input module
Wago	750-461 - 2 channel analog input module
Wago	750- 1504- 16 channel digital output module
Wago	750-600 end module
Wago	750-612 power unit
Wago	optocouplers 859-795
Wago	optocouplers 859-796
Wago	optocouplers 859-702
Wago	optocouplers 750-461
Adafruit	Raspberry Pi 3
Adafruit	Arduino Uno
Parallax, Keyestudio	PIR motion sensor
T-pro	DS18b20, PT100 temperature sensor
Keyestudio	LM35 temperature sensor
Keyestudio, Jekewin	DHT11 temperature humidity sensor
Keyestudio	Ks0349 active/ passive buzzer module
Keyestudio, HiLetgo	TEMT6000 ambient light sensor
Keyestudio	Ks0349 steam sensor
Keyestudio	Ks0349 water sensor
Keyestudio	Ks0349 sound sensor
Keyestudio	Ks0349 vibration sensor
Keyestudio	push button module
Adafruit, Keyestudio	MQ2 gas sensor
Adafruit, Keyestudio	MQ3 alcohol sensor
Adafruit, Keyestudio	stepper motor, servo motor
Gowoops, Keyestudio	HC-SR04 liquid measure sensor
eBoot	led lights
Elonco	breadboard

Table 4.5 refers to the relations between safety property violations in smart building systems and corresponding conflict types that are described in [94]. When a safety property is violated, it falls under one or more conflict categories. For example, when policy $p1$ is violated, it falls under conflict type c_1, and c_3. When the flame and water sensors from a different controller trigger the same alarm, it is classified as c_1. On the other hand, when they are part of the same controller, they still create a conflict on the same alarm and this is categorized as c_3.

Detection of safety property violation $p1$ is shown in Figure 4.4. Notice that the graph is based on two samples only. Sensor *water1* and *flame1* can trigger the same

Table 4.3: Conflict Description

Conflict Type	**Description**
c_1	No two or more controllers can trigger the same actuator at the same time
c_2	No two or more controllers can trigger actions that affect the same, or dependent features at the same time
c_3	No two or more overlapping events can trigger multiple actions on the same actuator
c_4	No two overlapping events can trigger actions that affect the same or dependent features
c_5	No two or more completely disjoint events can trigger multiple action on the same actuators
c_6	No two or more completely disjoint events can trigger multiple actions that affect the same, or dependent features
c_7	No single sensor with single objective can create more than one event within a specific time limit

alarm1 at the same time. We represent this scenario in the provenance graph.

For the safety property $p1$, there are three agents, *water1_ agent*, *flame1_ agent* and *alarm1_ agent*. These agents are associated with three activities, *flame1_ sensing*, *water1_ sensing*, and *alarm1_ actuating*, respectively. The two sensing activities sense the environment and generate entities *flame1* and *water1*. These two entities are used by *alarm1_ actuating* to generate the status of the alarm as *alarm1* entity. When *flame1* and *water1* have the same value 1, it is referred as a conflict to the device *alarm1*. That is, *alarm1* is actuated at the same time. In this example, there was no conflicts for first set of data. This is shown with a dashed green line. However, for the second sample data, `PLC-PROV` detects conflict which are represented by red solid box. As can be seen, the same activity was influenced by two activities (both entities have value 1) at the same time. A similar example of whether $p6$ is violated in our collected sample, is illustrated below.

As can be seen in Figure 4.5, both motion detection *motion1* and a light switch *switchLight1* are capable of actuating the same light *light1* at the same time. For the first sample, the conflict did not take place which is shown with a dashed green

Table 4.4: Violation of Safety Properties that Lead to Conflicts

	Safety Violations
$p1$	Flame sensors and water detectors sensor actuate the same alarm
$p2$	Gas sensors and flame sensors actuate the same alarm
$p3$	Flame sensors have different rules (operational and management) to access the same elevator door
$p4$	Gas sensor and motion detection access the same elevator door
$p5$	Gas sensor and management rule access the same window shutter
$p6$	Motion detector and switch light access the same light
$p7$	Two or more motion detectors access the same light continuously
$p8$	Flame and gas sensor actuate the same window shutter differently
$p9$	Temperature sensor and flame sensor actuate a sprinklers differently in case of fire
$p10$	Temperature sensors with variety of measurements triggers a sprinkler differently
$p11$	Shared temperature sensors access a single thermostat differently
$p12$	Multiple humidity sensors access a common fan/humidifier
$p13$	Temperature sensor and humidity sensor regulate the same thermostat/fan
$p14$	motion detectors in corridors access the same light in opposite ways
$p15$	Management rule and occupantâs preference affect the same thermostat
$p16$	Management rule and occupantâs preference affect the same window shutter
$p17$	Management rule and occupantâs preference affect the same lights
$p18$	Management rule of Window blind and occupantâs preference on a roomâs blinder affect light sensitivity
$p19$	Emergency rule and human presence conflict on a light
$p20$	Emergency rule and human presence conflict on the same door
$p21$	Emergency rule (shut down thermostat) and human presence (regular rule on thermostat) conflict on thermostat operations
$p22$	Management rule and motion detection affect temperature in a room and corridor
$p23$	Management rule and emergency rule with different logic actuate the main entrance
$p24$	The same corridor light is turned off (management rule) and turned on (emergency rule) with different logic to access the corridor light
$p25$	Management rule (turning on) and emergency rule (turning off) with different logic actuate the same thermostat
$p26$	Management rule and emergency rule with different logic (emergency triggers camera to turn on, management rule asks camera to turn off after office hour) actuate the same camera
$p27$	An elevator is kept operational (management rule) and shut down (emergency rule)

Table 4.5: Relationship between IoTC^2 and Safety Property Violations in the testbed

	C_1	C_2	C_3	C_4	C_5	C_6	C_7
$p1$	✓		✓				
$p2$	✓		✓				
$p3$	✓				✓		
$p4$	✓		✓				
$p5$		✓			✓		
$p6$		✓			✓	✓	
$p7$		✓	✓	✓			
$p8$		✓		✓			
$p9$	✓		✓				
$p10$	✓		✓				
$p11$		✓	✓	✓			✓
$p12$		✓	✓	✓			✓
$p13$		✓	✓	✓			✓
$p14$		✓		✓			
$p15$		✓			✓	✓	✓
$p16$	✓	✓			✓	✓	
$p17$	✓	✓			✓	✓	
$p18$	✓	✓			✓	✓	
$p19$			✓		✓	✓	
$p20$	✓				✓		
$p21$	✓		✓				
$p22$	✓	✓			✓	✓	✓
$p23$	✓				✓		
$p24$	✓				✓	✓	
$p25$	✓				✓	✓	✓
$p26$	✓		✓				
$p27$	✓				✓	✓	✓

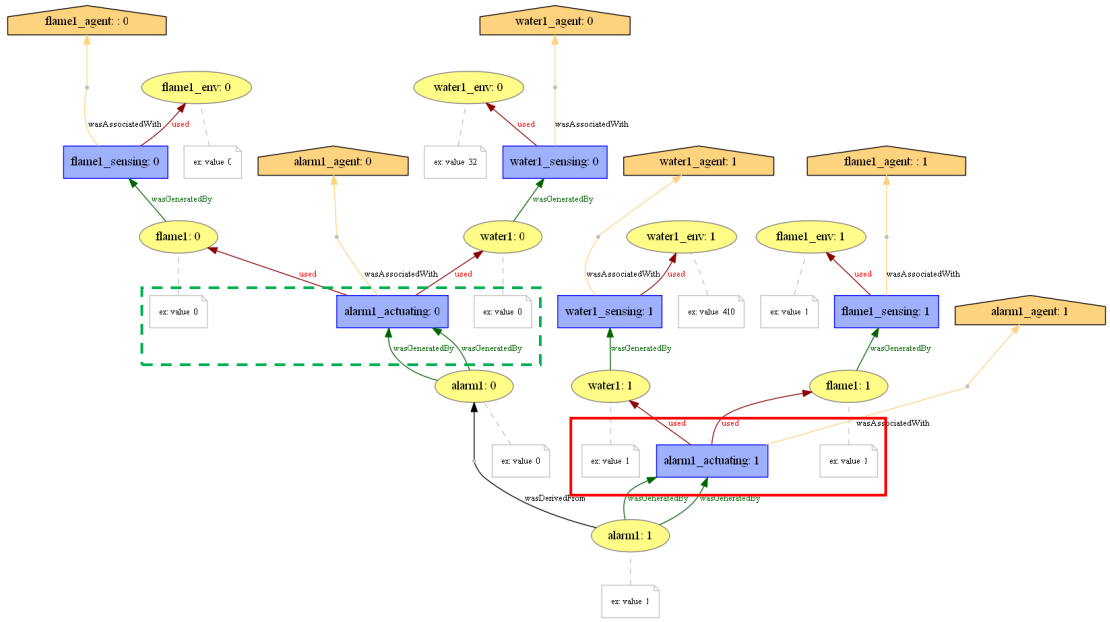

Figure 4.4: detects $p1$ violation

box. However, for the second sample, the light is switched at the same time of motion detection. Both entities have the same value 1 (as marked with red solid box). That is, the lighting activity is triggered at the same time by two different agents. Therefore, it creates conflicts on *light1*. PLC-PROV reports it as conflict.

The conflict detection algorithm of PLC-PROV was evaluated for Figure 4.4 and 4.5 on a dataset of size 2. With our testbed, we have collected up to 8000 sample of data in order to observe which sensors are causing the most conflicts and what actuators are affected the most in our testbed. The evaluation result is given in Table 4.6. Because an event is not predictable, the occurrence of conflicts on a device cannot be mapped linearly. The same goes with the conflict creating sensors. We cannot depict their frequency with the size of data. However, note that with PLC-PROV, we can find the conflict creating sensors affected actuators with frequency of occurrences. In this way, we know which devices need the most attention by the smart building authority in order to resolve conflicts. The resolution of conflict is left as our future work.

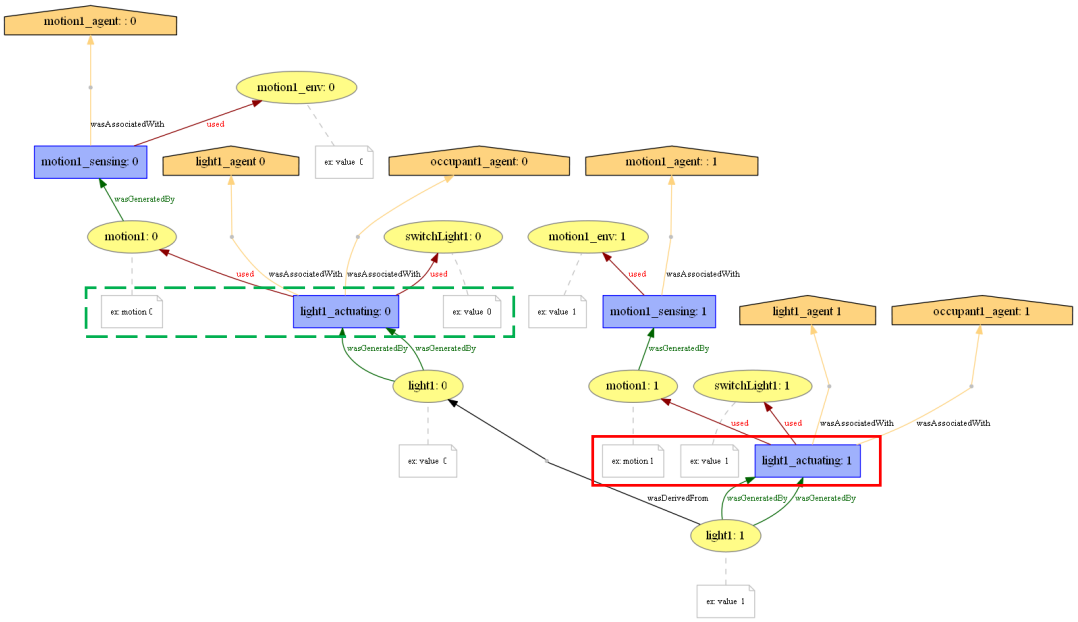

Figure 4.5: PLC-PROV detects $p6$ violation

4.4.3 PLC-PROV Execution Time

In this section, we summarize our finding on the execution time of PLC-PROV. We have a total 27 policies for our testbed. We have collected sensor and actuator measurements for 1000 time-stamps. For experimental purpose, we ran our system for 5 policies only in the first experiment. Then, we incremented the number of policies by 5 for each subsequent experiment. As can be seen from Figure 4.6, it takes more time for the machine when there are more policies. This is an expected behavior because more policies includes more sensor and actuator states, and thereby adds more nodes and edges in the graph. Therefore, it takes more time to traverse a bigger graph. However, it should be noted that each policy does not add the same number of nodes in the graph. For example, $p8$ does not add any new nodes for flame and gas as the corresponding nodes for agents, entities, and activities were already included with $p1$ and $p2$. However, necessary edges are added to the graph for representing the actuation to take place on the window shutter.

For our next experiment setup, we observe the execution time of our system with different size of collected samples from Codesys traces. As can be seen from Table

Table 4.6: Some Selected Sensors (causing conflicts) and Actuators (affected by conflicts in our testbed)

	Data Size							
Devices	1000	2000	3000	4000	5000	6000	7000	8000
temperature sensor 1	86	67	125	148	202	295	298	280
motion sensor 1	61	149	221	233	349	240	471	462
flame sensor 1	21	46	103	85	122	124	152	162
gas sensor 1	20	43	88	79	114	133	134	143
humidity sensor 1	37	13	43	67	61	89	81	71
thermostat 1	48	51	63	87	118	131	215	204
light 1	32	44	74	69	139	140	111	214
window blinder 1	55	21	63	100	110	81	85	154
sprinkler 1	4	7	18	24	24	27	13	31
door 1	18	43	77	89	80	118	143	135

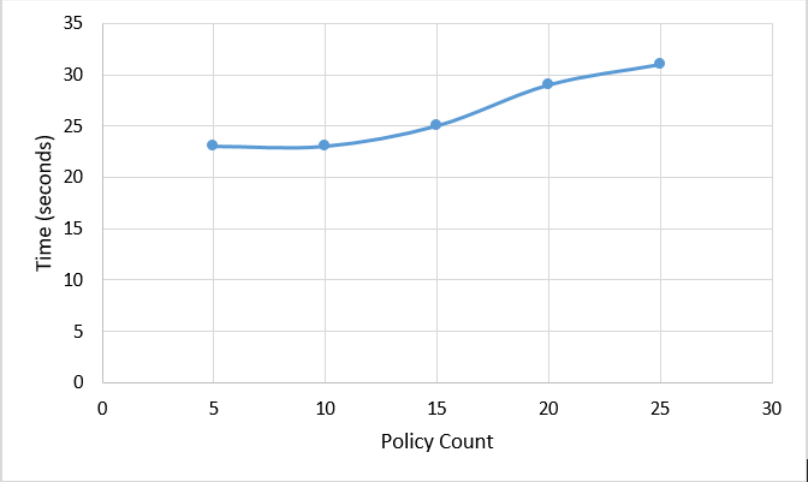

Figure 4.6: PLC-PROV Execution time with increase in policy count

4.2, we have variety of devices for our evaluation. We run the testbed to collect samples and then evaluate whether any of the safety properties, given in Table 4.4, are violated. At first, our collected sample size was smaller, and hence we get a small number of conflicts. When we increase the size of the sample, as expected more conflicts are detected. When the sample size is increased, there are more nodes in the provenance graph to be traversed PLC-PROV to detect conflicts. The observation is shown in Figure 4.7 with two y-axis, left for time and right for conflict count.

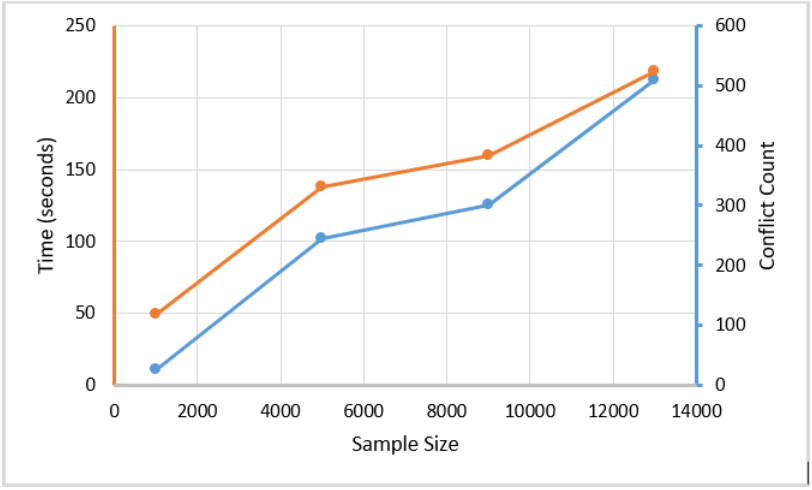

Figure 4.7: PLC-PROV Execution Time with Increase of Collected Sample and Conflict

4.5 Related Work

The closest work to PLC-PROV is PROV-CPS [31] where provenance was collected from resource-constrained embedded devices of the cyber-physical system. However, this research collects provenance from sensors only to identify anomalous measurements. On the contrary, apart from collecting provenance from the sensor measurement, our work covers the actions of the actuators and the dependencies among the PLCs in finding malicious activities. Our approach is complete in expressing the causality and dependencies among the data objects through the provenance graph. Another notable work in this area is ProvThings [32] where a provenance collection framework is proposed for IoT apps and devices. ProvThings presents an automated instrumentation mechanism for IoT apps and device APIs. The collected provenance is then used to generate explanations for "why" a particular action occurred. Our work captures provenance data for all sensors and actuators in order to detect safety and security policy violations.

There have been several other attempts to deploy security policies with static verification, dynamic verification, and the hybrid of these two approaches. Static verification (model checking) is proposed in TSV [18] where a middleware ensures the safety

of a PLC-based system sitting between PLC and the devices. TSV verifies the safety behavior of the code executed on PLC before commands reach the actuators. The safety properties are written in temporal logic which is verified using model checking. While this work verifies the system's behavior, there are some other works that started the verification from PLCs' source program [19, 20]. Later, others proposed mechanism to automatically generate formal models from PLC programs [21, 22, 23, 24, 25].

While static analysis performs the verification before the PLC program is released for operation (i.e. compile-time), dynamic analysis ensures that policies are not violated at run-time. C^2 [26] introduced an enforcement mechanism for safety policies in PLC-based system. When a PLC issues a command to an actuator, the current states of the system are checked and then decisions are made whether or not the command should be issued through their enforcement mechanism, C^2. In this work, concerns about the size of the trusted computing base (TCB) and state explosion in the model checking were expressed. The reduction of the size of the TCB size was addressed considerably in [27]. This work merges the static and dynamic analysis of TSV and C^2. The works by McLaughlin, et. al. focus specifically on safety properties. This was subsequently extended in [28] with an effort to find malicious PLC programs. Another approach to dynamic analysis is proposed in [29] using Interval Temporal Logic (ITL) and the Tempura framework, which aims to provide early alerts in PLC-based systems.

Later, this work was extended in [30] where an ITL/Tempura definition of a Siemens S7-1200 PLC ladder logic was presented. Their developed monitoring methodology captures a snapshot of the current state (with values for markers, input, output, counters, and timers) of the PLC. Tempura was implemented to execute on an Arduino Uno connected to the PLC, ensuring that the PLC does not need a powerful computing node to perform the computations.

While static analysis has proven promising, the number of possible inputs and out-

puts for a PLC system can lead to a state explosion. Furthermore, dynamic analysis suffers from a coverage problem, where only executed code paths are verified. Symbolic execution can minimize the state space, but cannot guarantee complete verification of outputs (actuation command) from input sets (sensor measurement). For these reasons, what is needed is a mechanism that can provide high-level safety and security policy descriptions that can be enforced at run-time where the appropriate context can be considered.

4.6 Discussion

This chapter focuses on the integration of data provenance in PLC controlled systems in order to detect safety policy violations there. We have modeled data provenance that considers user input (through switches), actuators' state (through the controller), and sensors' measurement (to the controller). Therefore, we claim that our model is complete in expressing the causality and dependencies among the data objects in a PLC-controlled system. We evaluated our model with a smart building testbed. With our developed tool, we can know the most vulnerable sensors to create conflicts. The actuators that are affected by a good number of conflicts can get major attention too. The execution time for our developed tool is very nominal compared to the large size of data it handles. It turns out that data provenance has great potential applicability in PLC controlled systems where the change of sensor measurement and actuator actions take place very frequently. Despite being used in critical infrastructures, PLCs have little or almost no security. The integration of `PLC-PROV` is capable of enforcing adequate safety and security policies.

CHAPTER 5: Conflict Resolution in Smart Buildings

5.1 Introduction

5.1.1 Problem Statement

Due to the limited computational and memory capacities of the end devices of a smart building, it is not possible to have all the devices programmable. Hence, an end device itself does not know whether it is impacted by or contributing to conflicts. A smart building connects a myriad of devices. The more the number of devices, the more exposed the system becomes for misconfiguration in terms of conflicts. Furthermore, some policies need to be enforced due to emergency or management rules which can conflict the regular operations. Conflict verification is designed to prevent a conflict from occurring. However, events can occur any time or an attacker can force specific events to happen. Hence, apart from checking whether a recently triggered event causes a conflict, the resolution of conflicts is equally, if not more important.

Defeasible reasoning is a robust approach to resolving conflicts apart from detecting them in some areas (e.g., stock market, cyber attribution) where the information received are incomplete or contradictory. While classical logic requires the complete state of all information before beginning the reasoning process, defeasible logic, on the other hand, provides the competence that new information can abandon the previously established conclusions and adopt new ones. In doing so, apart from detecting conflicts, defeasible logic resolves conflicts by enforcing a relationship among the rules that create conflicts. It also devises an approach to prevent a conclusion from being drawn. It should be noted that a formal methods approach for detecting conflicts in IoT system is presented in $IoTC^2$ [91]. In our research, we replaced the

conflict detection component of defeasible reasoning with $IoTC^2$, because it is more fine-grained and comprehensive for PLC-based systems.

5.1.2 Contribution

In this chapter, we propose `DEFEASIBLE-PROV`, a mechanism to resolve the conflicts as detected by $IoTC^2$. In doing so, we use `PLC-PROV` [95] to track the data (as inputs and outputs) of the smart building system. `DEFEASIBLE-PROV` relies on tracking *data provenance* to compare the information against the specified safety and security properties as defined in $IoTC^2$. If a policy violation is found, it provides the administrator with the detailed traces that are needed to understand the impact of the violation (i.e., conflicts). Next, it traces the rules that are triggered by these conflict creating sensors. These rules are given less priority for execution to resolve conflicts in the system. In order to enforce the superiority relation among the rules, we develop a compiler for PLC source programs (IEC-61131). Our tool is capable of enforcing defeasible reasoning in PLC programs for conflict resolution in the system.

The contributions of the chapter are as follows:

- Develop an approach to generate an abstract syntax tree (AST) with necessary information about event triggering and actuation assignment location of a PLC program (IEC-61131 Structured Text)
- Implement a methodology that identifies the rules/logic that are responsible for conflicts in smart building systems
- Implement a systematical approach to devise defeasible reasoning for generating a secure PLC program

5.2 Background

5.2.1 Defeasible Logic Programming

Defeasible logic deals with conflicts among the information that are used as knowledge items for reasoning purposes. As a simple example, a piece of new information arrives in smart building systems for which two rules get triggered, and the conclusion of these rules negate each other.

Unlike classical logic programming, defeasible logic programming is capable of proving a theory from contradiction and incompletion, though defeasibly. Any classical logic programming tool would have disproved that theory in the very first place. A defeasible theory is defined by Lam and Governatori [61] as a triple $(F, R, >)$ where facts and rules are denoted by F and R, respectively. The sign $>$ is used to assign superiority relation among the conflicting rules. The definition of facts is the same as it is for classical logical programming. However, *rules* are classified into two separate criteria: *strict rules* and *defeasible rules*. Strict rules (represented by $->$) bear the same functionality and representation as it is for *rules* in logic programming. For example, we can consider a *strict rule* like the following:

$$bird(X) -> fly(X)$$

That is, when we have any input as a bird, the rule ensures it flies. What if this is not true always? However, we want to note that penguins are considered as bird too. When the input is penguin in our knowledge base, it is represented as:

$$bird(penguin)$$

This piece of knowledge makes our previous rule a wrong one. Therefore, we make this rule as a *defeasible rule* (represented by $\Longrightarrow$). These *defeasible rules* are the rules that can be defeated by contrary evidence (e.g., penguin). With defeasible rule,

we relax our previous rule where we impose all birds can fly. Rather, we would like to denote that if the creature is a bird there it may fly. The rule can be written by defeasible logic as follows:

$$bird(X) => fly(X)$$

This gives us the flexibility that despite being a bird, a penguin cannot fly.

One may wonder how can we impose a penguin to not fly. Defeasible reasoning framework introduces *defeaters* (or exceptions) (represented by $\sim>$) for this purpose. A defeater is used to prevent any conclusion from being drawn. A new rule is added as such:

$$bird(penguin) \sim> \neg fly(X)$$

It is interpreted as: if the bird is a penguin, it must not fly. In other words, *defeaters* can be seen as exceptions in the knowledge base.

Another important aspect of defeasible reasoning is devising superiority relations among the automation rules. Generally, any logic programming provides an output on whether a rule in the form of theory is provable or not in the system. When a rule is not strictly provable, it may be defeasibly provable if it is conflicted by another rule. Let us adopt an example from [3], which was used in a smart environment context. For a room X, two rules $r1$, and $r2$ exist. $r1$ turns on the cooler when it finds the temperature is high. On the other hand, $r2$ turns off the same cooler when it does not find anybody in the room. These two rules can trigger opposite actuations to the same cooler at the same time. In order to solve this, $r2$ is given more priority so that $r1$ will not trigger in the case when both rules are triggered at the same time.

$$r1 : tempHigh(X) \implies switchOnCooler(X) \tag{5.1}$$

$$r2 : \neg motion(X) \implies switchOffCooler(X) \tag{5.2}$$

$$r2 \geq r1 \tag{5.3}$$

5.3 DEFEASIBLE-PROV Design

Due to the distributed nature of smart buildings, it is possible that a number of sensors try to actuate one actuator. For example, there can be more than one temperature sensor to regulate the temperature of a hallway of a smart building. Consider a rule which triggers a thermostat to turn on when the temperature is below 60F. In the same system, there is another rule which asks the same thermostat to turn off when the temperature is above 80F. An attacker can leverage this type of miscofiguration to create conflict and make the thermostat dysfunctional. Moreover, if the room has two temperature sensors, but one thermostat, an attacker can compromise the sensors and send two different values, 81F and 59F, to the controller. In this way, both of the rules, as mentioned above, will be triggered to decrease and increase temperature at the same time. Based on the conflict impacted devices, rules should be prioritized, and exceptions should be made for the conflict creating sensors.

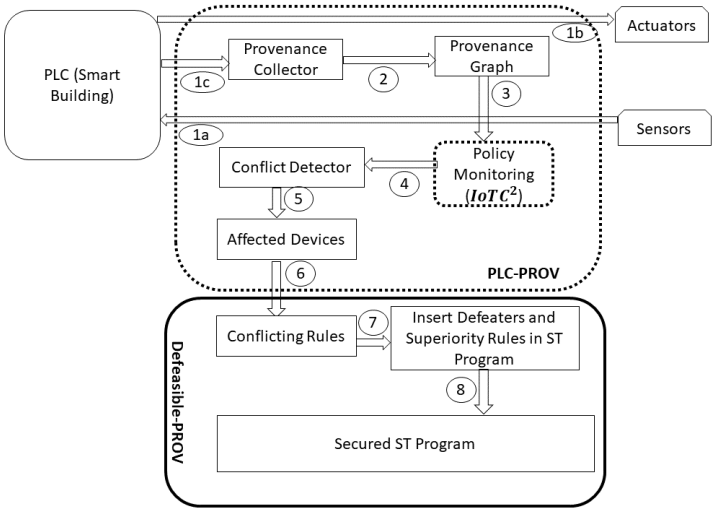

Figure 5.1: Architecture of DEFEASIBLE-PROV

The architecture of DEFEASIBLE-PROV with the flow is given in Figure 5.1.A basic smart building system is comprised of a number of sensors, actuators, and PLCs. The sensors collect measurements of the current state of the environment or the system

(1a). The operational rules are programmed in the PLCs. These rules are triggered based on events collected by the sensors. The rules define what actions should be taken by the actuators that are connected to the PLCs. Simply put, an event triggers a rule, and the rule triggers an action in the system. The controller decides what action or set of actions to take, as defined by the rules installed in the controller. The controller issues commands to the actuator for performing the most appropriate action (1b). This flow is given in the upper portion of Figure 5.1. As can be seen from the figure, the above-mentioned flow is captured by a framework, `PLC-PROV` [96], as the first step to track inputs and outputs and to model the evolution of the system from inputs to outputs. We use CODESYS, a development system for most PLC applications, which emits traces for sensor measurement and actuator states of the system (1c). `PLC-PROV` traces the variables designated for sensors and actuators with timestamps during a system execution, which is eventually the provenance collection scheme. The provenance recorder compiles collected provenance from CODESYS traces and converts them into the smart building provenance model as described in [95]. From there, the provenance graph is generated, depicting the evolution of the system from sensor readings through the PLC and controller to the actuators (2). In the next step, this provenance graph is traversed to determine if a safety property violation, as described in $IoTC^2$, has occurred (3). When found, they are regarded as conflicts (4). The *conflict detector* module of `PLC-PROV` outputs the sensors which create conflicts and the actuators who are impacted by those conflicts (5). The next step is where `DEFEASIBLE-PROV` starts. The conflict impacted devices are searched in a smart building ruleset (6). These rules become subject to the defeasible reasoning framework (7). Finally, a secured PLC program is generated. More details on the (6),(7), and (8) steps are given in Algorithm 3.

As can be seen from Algorithm 3, a list of conflict creating sensors and conflict impacted actuators that are outputted from `PLC-PROV`, the rules of a smart build-

ing are traced. Therefore, the algorithm takes a list of sensors, a list of actuators, and the PLC source program for a smart building. In searching for rules of a PLC program, an abstract syntax tree (AST) is generated using IEC-61131 grammar (line 2). `DEFEASIBLE-PROV` traverses the AST to find where those actuators are assigned values. If there are emergency rules, `DEFEASIBLE-PROV` does not pick up that rule (line 5). Other rules are blocked (line 6). The conflict creating actuators are identified in this step. It should be noted that all rules but the emergency rules are marked as defeasible rules. The emergency rules are noted as the strict rules. Simply put, emergency rules cannot be defeated by any other rules in our defeasible logic framework. When some sensors create a large number of conflicts, our framework ensures that no rule triggers from those sensors (line 8,9). This, in other words, is the implementation of *defeaters* of defeasible logic programming. We make exceptions for these sensors by not triggering any action out of them. `DEFEASIBLE-PROV` traces the rules in AST to search the associated with conflict creating/ impacted devices. Now, `DEFEASIBLE-PROV` imposes a *superiority relation* in the PLC source program to prohibit the conflict creating rules to trigger (line 10, 11). Finally, the instrumented PLC source program is regenerated from AST (line 15). The output of this algorithm is a secured ST program, which can be seen as the last block in Figure 5.1 .

As an example, there are two rules r_1 and r_2 which are triggered by sensor measurement m_1 and m_2, respectively. Let us assume that both rules try to issue different commands to device d at the same time. At this point, it should be noted that defeasible reasoning detects conflicts first before resolving them (e.g., `SPINdle` [61]). In this research, we have adopted the conflict resolution component of the defeasible reasoning framework only. For conflict detection, we have used $IoTC^2$ for formally design conflict creating safety property violations and later `PLC-PROV` to detect those on runtime. `PLC-PROV` provides us with a list of devices that are the most impacted by conflicts. By using the list, our framework traces the rules that are associated

with the conflict creating sensors and impacted actuators. As we revisit the example again, `PLC-PROV` identifies m_1 as anomalous. Therefore, `DEFEASIBLE-PROV` enforces that r_1 would have less priority than r_2 in terms of execution. In this way, the conflict gets resolved.

10

Algorithm 3 Conflict Resolution with Defeasible Reasoning

```
0: procedure RESOLUTION(SensorS, ActuatorA, ProgP)
0:   AST ← P
0:   for each actuator a in A do
0:     find rule r from AST where a is assigned
0:     if r is an emergency rule then
0:       block all other rules where a is assigned
0:     else
0:       arrange S in descending order
0:       pick the n sensors s(1,..,n)
0:       find actuation a in AST triggered by s_n
0:       get rule r where a is assigned
0:       block rule r
0:     end if
0:   end for
0:   P ← AST
0:   return P
0: end procedure=0
```

5.4 Evaluation

In order to evaluate how our system works, we built a testbed for a smart building system. Details of the testbed are given in 4.4.1. Conflicts in the system are defined based on the safety property violations as described in [94]. We identify some potential safety property violations which are derived according to our testbed. Whenever these violations occur, they are referred as conflicts. These property violations are proposed in such a way that they refer to conflicts only as proposed in [94]. That is, all the safety property violations are categorized among those seven conflict creating scenarios. Details about safety property violations are given in 4.4.2.

5.4.1 DEFEASIBLE-PROV Implementation

DEFEASIBLE-PROV takes a PLC source program that is programmed in Structured Text (ST) programming language as an input. ST is the only high level programming language for PLC [97]. Moreover, ST programing language is preferable for research purposes. CERN (European Organization for Nuclear Research) has been conducting exclusive research on instrumenting ST programs of PLC [23]. However, we also want to note that our tool has the capability of parsing other PLC programming languages. DEFEASIBLE-PROV adopted the grammar file and tokenizer from an open source project, iec2xml [98]. We developed a new parser on top of this project to parse a ST program into an Abstract Syntax Tree (AST). Finding all assignments of a device with their line numbers in the source program was made possible with our implementation.

5.4.2 DEFEASIBLE-PROV Efficiency

The efficacy of DEFEASIBLE-PROV is determined by the number of conflicts it can eliminate from a smart building system. One can aim for 100% resolution of conflicts. However, in a real-world setting, it is hard to achieve because we do not have control over events that trigger actions in the system. Blocking all the conflict creating rules at once can resolve almost all conflicts in the system. However, it hinders the basic automation rules of a system. For example, a motion detector and a switch can turn the same light at the same time. Blocking one of these associated operational rules can resolve conflict on the light. However, both rules are important for smart building operations. Rather, the focus of this research is resolving conflicts on the light if it is impacted the most. If we observe that to be true, DEFEASIBLE-PROV finds the rules that are associated with changing light status and then make priority in command execution.

As mentioned before in the design section, DEFEASIBLE-PROV receives information

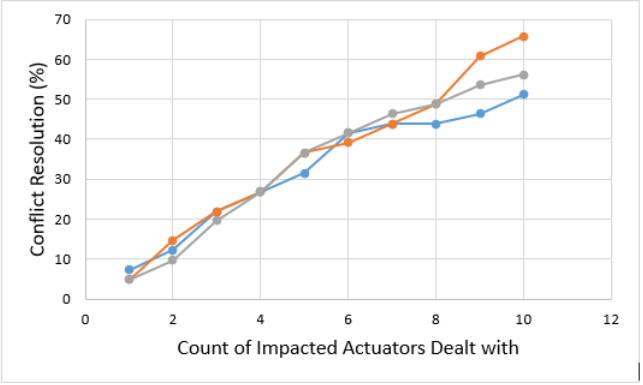

Figure 5.2: DEFEASIBLE-PROV Efficiency with Increase in Dealt Actuators

about most conflict creating/impacted devices from *PLC-PROV*. From there, we vary the device count, which is used by DEFEASIBLE-PROV to resolve conflicts. In the first experiment, we increase the count of handled actuators to observe the efficacy of our tool. There can be more than one smart building rule associated with the most conflict impacted actuator. In such a case, only one rule is allowed to operate. DEFEASIBLE-PROV blocks other rules as per devising the superiority relationship of defeasible reasoning. We conducted this experiment with 3 sets of data where each set has a different combination of impacted actuators. Therefore, conflict resolution efficacy is not the same for the same count of dealt actuators. However, it is important to notice from Figure 5.2 that when our tool handles more actuators, more conflicts are resolved.

In our next experiment, we consider only conflict-creating sensors to resolve conflicts. That is, we implement the idea of a defeater of defeasible reasoning. Whenever a sensor creates the most conflicts, we make an exception that any rules which are triggered by this sensor will be blocked. We increase the count of the handled sensors to observe how many conflicts DEFEASIBLE-PROV can resolve. We conducted this experiment with 3 sets of data with a variety of conflict-creating sensors received from *PLC-PROV*. As can be seen from Figure 5.3, the more sensors that are considered, the more conflicts that are resolved.

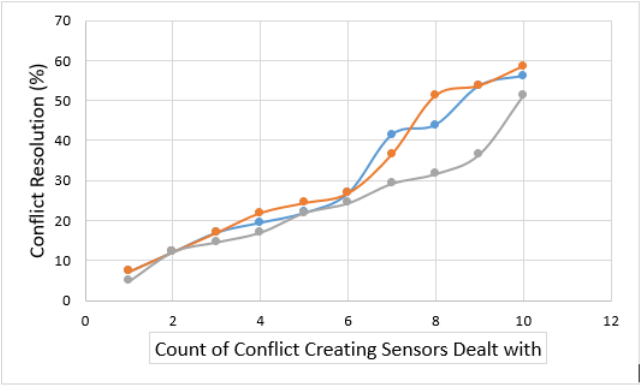

Figure 5.3: DEFEASIBLE-PROV Efficiency with Increase in Dealt Sensors

The last experiment on DEFEASIBLE-PROV efficacy is conducted to observe the impact of dealing with the combination of sensors and actuators to resolve conflicts. That is, both defeaters and superiority relationships are implemented to observe the efficacy of our tool. Like the previous two experiments, we have used 3 sets of data with a variety of conflict creating/impacted devices. As we see from Figure 5.4, a slightly better conflict resolution is achieved compared to the previous two experiments when the same number of devices are dealt with. More conflicts could be resolved if the most conflict creating sensors were not correlated with the most conflict impacted actuators. For example, a motion detector creates conflict in a light. Therefore, when both the motion detector and light are used for resolving conflicts, only a limited number of conflicts are resolved. However, some sensors have one to many mapping. For example, a motion detector sensor can actuate a light and a door. In this case, more conflict resolution is possible if DEFEASIBLE-PROV considers that motion detector.

It is very important to observe how scalable DEFEASIBLE-PROV is. The number of sensors and actuators can vary from building to building. It is possible that a high number of smart building devices contribute to conflict creation. Therefore, it is important to observe how time efficient DEFEASIBLE-PROV is in the conflict resolution process. In doing so, we vary the number of devices that DEFEASIBLE-PROV handles

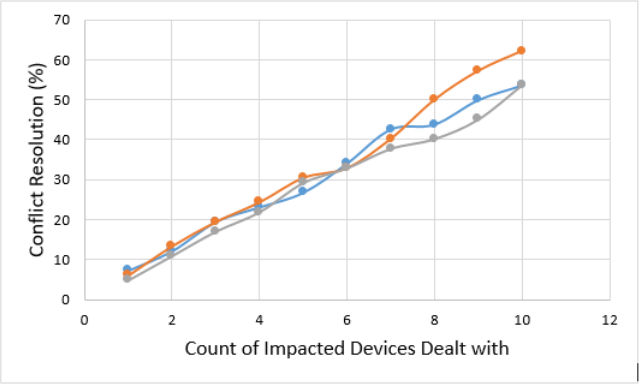

Figure 5.4: DEFEASIBLE-PROV Efficiency with Increase in Dealt Devices

for resolving conflicts. At first, our experiment is conducted to handle the actuators only and shown in Figure 5.5. That is, DEFEASIBLE-PROV resolves conflicts that are associated with the most conflict impacted actuators only. At first, we take only one actuator into consideration to see how much time DEFEASIBLE-PROV needs to execute. Then, we increase the number of actuators to see the impact of execution time. As can be seen from the figure, there is a very marginal increase in time as we increase the actuator count. Also, notice that DEFEASIBLE-PROV needs very little time (1 second) to execute.

The next experiment was conducted in a way so that DEFEASIBLE-PROV resolves conflicts based on the conflict creating sensors only. That is, we have not examined the impact of actuators in resolving conflicts in this experimental setup. We increase the sensor count to observe the behavior of execution time. It is interesting to notice that the DEFEASIBLE-PROV takes almost the same time with the sensors as it takes with the actuators. The execution time does not increase much, with an increase in the number of sensors (shown in Figure 5.5).

The last experiment is conducted when both sensors and actuators are considered by DEFEASIBLE-PROV to resolve conflicts. Like the previous two experiments, we increase the count by 1 in each run to observe how timing differs for DEFEASIBLE-PROV execution. As can be seen from Figure 5.5, DEFEASIBLE-PROV takes a little bit more

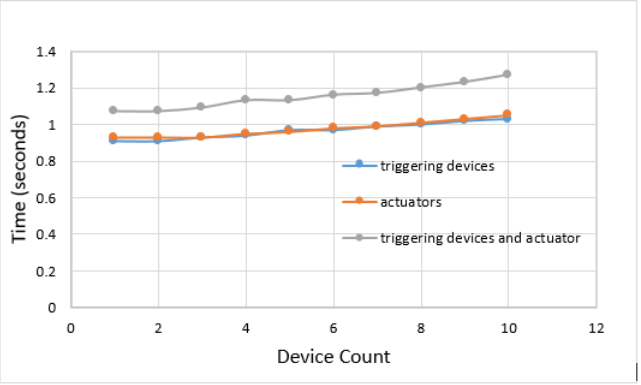

Figure 5.5: `DEFEASIBLE-PROV` Execution time

time (0.25 seconds) than the previous two experiments for each experimental run. This is reasonable because `DEFEASIBLE-PROV` has to explore the abstract syntax tree of a PLC program for sensors and actuators separately, which adds more time complexity in the execution. However, the increment pattern for all three experiments is more or less the same.

5.5 Related Work

Conflicts are common in a smart environment where several components are exchanging information to make an operation. One important way of dealing with conflicts in automation is by using planning techniques. Rules were organized hierarchically for execution [99]. The learning phase for the knowledge base accompanied by planning was proposed in [100] for resolving conflicts in the ambient intelligence domain. This research work was improved with a scheduling algorithm where priority values were assigned among conflicting rules [101].

Defeasible reasoning has been successfully applied to several fields (e.g., smart cities, smart building, stock market) where interoperability and data sharing are fundamental for their services [102, 103, 104]. For example, in smart cities, when data is shared by multiple stakeholders, their policies conflict on a regular basis. Defeasible logic is used to build a trust model among all parties in an effort to resolve these conflicts [105]. Another research attempt applies the concept of agent negotiations for

deciding which of the conflicting rules to trigger in a smart environment context [106]. Defeasible logic has been successfully used for UAVs navigation system's simulation, for solving conflictual rules [107]. Business processes also use defeasible logic or some variation of it to impose exception in the received knowledge and to propose regulation of policies [62, 63, 108, 109, 110].

There have been several research efforts to deal with the uncertainty, ambiguity, and conflicts in a smart building/smart home environment. One of the significant approaches was made by [111], where logical rules were distributed among agents. Agents were distributed smart environment rules in such a way that they don't create conflicts. However, when merged globally, defeasible reasoning is applied to avoid conflicts. The preferences among the rules are assigned based on an agent's requirement at a certain moment. There are some other research where defeasible reasoning was used for smart building/ home environment [99, 100, 101]. Some proposed agent-based systems [3] with a specific defeasible logic reasoner `SPINdle` [61]. All these works focus on achieving a certain goal (i.e., energy saving) while imposing a superiority relationship among the rules. Our approach is different in this way. We tend to devise the defeasible logic framework based on the root cause behind the conflicts.

5.6 Discussion

This research focuses on the integration of defeasible reasoning in PLC controlled systems in order to resolve safety policy violations there. The proposed framework uses data provenance to investigate the root cause behind conflicts. Data provenance suits the smart buildings domain very well in expressing the causality and dependencies among the data objects to detect conflict there. To the best of our knowledge, this is the first-ever approach where conflict detecting or impacted devices are considered for devising defeasible reasoning in appropriate places of a PLC source program. We evaluated our model with a real-world smart building testbed. We observe the promising efficacy of our tool to resolve conflicts. The execution time of the tool is

very nominal as well. It turns out that defeasible reasoning combined with data provenance has high potential applicability in PLC controlled systems where the change of sensor measurement and actuator actions take place very frequently. Despite being used in critical infrastructures, PLCs have little or almost no security. The integration of `DEFEASIBLE-PROV` is capable of enforcing adequate safety and security policies.

CHAPTER 6: Conclusions and Future Work

Smart building technology is growing to achieve energy efficiency, sustainability, and maximizing worker productivity. However, the vulnerabilities associated with PLC can impact smart buildings the same or even worse way as in other domains (e.g., train signaling system, pipeline system, telecommunication system). Despite being used widely in many critical infrastructures (e.g., smart building management, power generation, water and wastewater management, traffic control systems, oil and natural gas, chemical, pharmaceutical, pulp and paper, food and beverage, automotive, and aerospace), PLCs use protocols which make these control systems vulnerable to many common attacks, including man-in-the-middle attacks, denial of service attacks, and memory corruption attacks (e.g., array, stack, and heap overflows, integer overflows, and pointer corruption). With these attacks, it is possible to create conflicts on certain devices of smart buildings, thereby disrupting functionality.

To this point, very few research efforts have been made to detect conflicting situations using formal methods in smart building systems. Moreover, conflicts can lead to additional actuation of devices, which eventually can increase the energy usage of a smart building beyond a desired threshold. We provide a formal method approach, IoT Conflict Checker ($\texttt{IoTC}^2$), to ensure the safety of device behavior concerning conflicts. While other works have their safety policies [112, 73, 113, 78] similar to the ones shown in Table 4.4, our formal method approach provides a general framework that fits for any IoT/smart building safety property violations that lead to conflicts. Any of our proposed policy violations result in conflicts. We defined the safety policies for controller, actions, and triggering events and implemented those with Prolog to prove the logical completeness and soundness. In addition to that, we have implemented

the detection policies in Matlab Simulink Environment with its built-in Model Verification blocks. We created a smart-home environment in Simulink and showed how the conflicts affect actions and corresponding features. We have also experimented with our method's scalability, efficiency, and accuracy in a simulated environment.

Later, we integrated data provenance in smart building systems to detect safety policy violations on runtime. Our provenance modeling considers sensing, actuating, user input, controller's command, actuators' state, and sensors' measurement to collect provenance and generate a graph from there. We claim that our model is complete in expressing the causality and dependencies among the data objects in a smart building system. We evaluated our model with a testbed that mimics a real-world smart building system. With our developed tool, we know the most vulnerable sensors to create conflicts. Moreover, most conflict impacted actuators can be identified for further attention. Our tool is capable of processing a large size of data within a satisfactory time range.

Our research concludes that data provenance has great potential applicability in smart buildings. The integration of `PLC-PROV` is capable of enforcing adequate safety and security policies in a smart building system. `PROV-CPS` [31] is the closest to our work, where sensor measurements are used to identify malicious data. Conflicts can occur without even inserting malicious data. Therefore, we modeled provenance with more causalities, actuation status, and change in environment features to capture provenance in smart building systems.

Apart from detecting conflicts, our research also aims to resolve them. We propose another framework, `DEFEASIBLE-PROV`, which uses data provenance to resolve conflicts. The root cause behind conflicts, found from `PLC-PROV`, are well accommodated for defeasible reasoning framework to resolve conflicts. To the best of our knowledge, ours is the first-ever approach to consider impacted devices in a defeasible reasoning framework. While other works focus on theoretical or analytical frame-

works with defeasible reasoning [114, 100, 101, 72], our work is more on the implementation paradigm, which is instrumenting a PLC program for smart building. That is, we can adopt their algorithms and instrument them in smart buildings with `DEFEASIBLE-PROV`. The execution time the tool is very nominal as well. It turns out that defeasible reasoning combined with data provenance is a promising approach towards a secure smart building system. Our approach shows great efficacy in resolving conflicts.

6.0.1 Challenges

Because formal method approaches suffer from state explosion problems, it will be interesting to observe if our model is challenged with a bigger smart building environment. The scalability of our method is important to make its feasibility in the real world. Moreover, our testbed for evaluating `PLC-PROV` and `DEFEASIBLE-PROV` consists of the PLCs that could be instrumented by ourselves. For a real smart building, this flexibility may not be present. Therefore, the accessibility of PLCs in a smart building can be an intriguing challenge. Another implicit challenge is the execution time needed for `PLC-PROV` to traverse the provenance graph. As the graph size gets larger with time and the increase of devices, the computation needs to be distributed in real-world implication.

6.0.2 Future Work

As the execution time of `PLC-PROV` is a concern, we plan to implement parallel computing approaches for provenance graphs [115]. The linear dependencies among the subgraphs, especially when dynamic integration of devices are possible, will be an interesting research direction. It should be noted that `PLC-PROV` provides us with the list of conflict impacted devices. It will be compelling to use this data for designing automated temporal logic for safety and security purposes in smart buildings. We also tend to observe how our tool behaves when dynamic changes in safety policies

are possible in the system. Another vital research direction can be predicting what devices would be conflicted in a certain period using deep learning techniques.

The enhancement of `DEFEASIBLE-PROV` with a multi-agent system will be a compelling future research direction. Each agent should be responsible for enforcing security policies for each subdomain of the provenance graph. The agents should be communicating with each other to achieve a larger goal (e.g., resilience, energy efficiency) of the system. Moreover, we have developed a parser for PLC programs. The development of a program analysis tool from `DEFEASIBLE-PROV` can contribute to the safety and security of smart buildings significantly.

REFERENCES

[1] G. Lilis, G. Conus, N. Asadi, and M. Kayal, "Towards the next generation of intelligent building: An assessment study of current automation and future iot based systems with a proposal for transitional design," *Sustainable cities and society*, vol. 28, pp. 473–481, 2017.

[2] S. A. Milinković and L. R. Lazić, "Industrial plc security issues," in *Telecommunications Forum (TELFOR), 2012 20th*, pp. 1536–1539, IEEE, 2012.

[3] T. G. Stavropoulos, E. Kontopoulos, N. Bassiliades, J. Argyriou, A. Bikakis, D. Vrakas, and I. Vlahavas, "Rule-based approaches for energy savings in an ambient intelligence environment," *Pervasive and Mobile Computing*, vol. 19, pp. 1–23, 2015.

[4] D. Clements-Croome *et al.*, "Master planning for sustainable liveable cities," in *6th International Conference on Green and Efficient Building and New Technologies and Products Expo*, vol. 29, 2009.

[5] S. Folea, D. Bordencea, C. Hotea, and H. Valean, "Smart home automation system using wi-fi low power devices," in *Automation Quality and Testing Robotics (AQTR), 2012 IEEE International Conference on*, pp. 569–574, IEEE, 2012.

[6] G. Fortino, A. Guerrieri, G. M. O'Hare, and A. Ruzzelli, "A flexible building management framework based on wireless sensor and actuator networks," *Journal of Network and computer applications*, vol. 35, no. 6, pp. 1934–1952, 2012.

[7] D. Clements-Croome, "Sustainable intelligent buildings for people: a review," *Intelligent Buildings International*, vol. 3, no. 2, pp. 67–86, 2011.

[8] A. Fensel, S. Tomic, V. Kumar, M. Stefanovic, S. V. Aleshin, and D. O. Novikov, "Sesame-s: Semantic smart home system for energy efficiency," *Informatik-Spektrum*, vol. 36, no. 1, pp. 46–57, 2013.

[9] TechNavio, "Global Industrial Control Systems (ICS) security market 2014-2018," 2014.

[10] C. Interactive, "Teen hacker faces federal charges: Worcester regional airport caused computer crash that disabled massachusetts airport," 1998.

[11] T. Smith, "Hacker jailed for revenge sewage attacks," 2001.

[12] C. News, "Virus disrupts train signals," 2003.

[13] M. Abrams and J. Weiss, *Bellingham, Washington, control system cyber security case study.* publisher not identified, 2008.

[14] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, p. 6, 2011.

[15] Y. Sun, X. Wang, H. Luo, and X. Li, "Conflict detection scheme based on formal rule model for smart building systems," *Human-Machine Systems, IEEE Transactions on*, vol. 45, no. 2, pp. 215–227, 2015.

[16] W. Kastner, G. Neugschwandtner, S. Soucek, and H. M. Newman, "Communication systems for building automation and control," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1178–1203, 2005.

[17] J. L. Rrushi, *SCADA Protocol Vulnerabilities*, pp. 150–176. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[18] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel, "A trusted safety verifier for process controller code.," in *NDSS*, vol. 14, 2014.

[19] J. Sadolewski, "Automated conversion of st control programs to why for verification purposes," in *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pp. 849–854, IEEE, 2011.

[20] J. Sadolewski, "Conversion of st control programs to ansi c for verification purposes.," *e-Informatica*, vol. 5, no. 1, pp. 65–76, 2011.

[21] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade. plc: A verification platform for programmable logic controllers," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 338–341, ACM, 2012.

[22] D. Darvas, E. Blanco, and B. Fernández Adiego, "Transforming plc programs into formal models for verification purposes," tech. rep., 2013.

[23] B. F. Adiego, D. Darvas, J.-C. Tournier, E. B. Vinuela, and V. M. G. Suárez, "Bringing automated model checking to plc program developmentâa cern case studyâ," *IFAC Proceedings Volumes*, vol. 47, no. 2, pp. 394–399, 2014.

[24] F. Markovic, "Automated test generation for structured text language using uppaal model checker," 2015.

[25] E. P. Enoiu, A. Čaušević, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson, "Automated test generation using model checking: an industrial evaluation," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 335–353, 2016.

[26] S. McLaughlin, "Cps: Stateful policy enforcement for control system device usage," in *Proceedings of the 29th Annual Computer Security Applications Conference*, pp. 109–118, ACM, 2013.

[27] S. McLaughlin, "Blocking unsafe behaviors in control systems through static and dynamic policy enforcement," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2015.

[28] S. Zonouz, J. Rrushi, and S. McLaughlin, "Detecting industrial control malware using automated plc code analytics," *IEEE Security & Privacy*, vol. 12, no. 6, pp. 40–47, 2014.

[29] A. Nicholson, H. Janicke, and A. Cau, "Position paper: Safety and security monitoring in ics/scada systems.," in *ICS-CSR*, pp. 61–66, BCS, 2014.

[30] H. Janicke, A. Nicholson, S. Webber, and A. Cau, "Runtime-monitoring for industrial control systems," *Electronics*, vol. 4, no. 4, pp. 995–1017, 2015.

[31] E. Nwafor, *Trace-Based Data Provenance For Cyber-Physical Systems*. PhD thesis, Howard University, 2018.

[32] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter, "Fear and logging in the internet of things," in *ISOC NDSS*, 2018.

[33] W. Granzer, F. Praus, and W. Kastner, "Security in building automation systems," *IEEE Transactions on Industrial Electronics*, vol. 57, no. 11, pp. 3622–3630, 2010.

[34] Modbus Organization, "Modbus Application Protocol Specification V1.1b3." `http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf`.

[35] C. Barz, S. Deaconu, T. Latinovic, A. Berdie, A. Pop-Vadean, and M. Horgos, "Plcs used in smart home control," in *IOP Conference Series: Materials Science and Engineering*, vol. 106, p. 012036, IOP Publishing, 2016.

[36] G. A. Dunning, *Introduction to programmable logic controllers*. Cengage Learning, 2005.

[37] D. Popescu, "Automate programabile. construcÈie, funcÈionare, programare Èi aplicaÈii," *Matrix, Bucharest, Romania*, 2011.

[38] T. Sysala, M. Pospíchal, and P. Neumann, "Monitoring and control system for a smart family house controlled via programmable controller," in *Carpathian Control Conference (ICCC), 2016 17th International*, pp. 706–710, IEEE, 2016.

[39] N. Skeledzija, J. Cesic, E. Koco, V. Bachler, H. N. Vucemilo, and H. Dzapo, "Smart home automation system for energy efficient housing," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on*, pp. 166–171, IEEE, 2014.

[40] U.-C. V. N. 362332, "Wind river systems vxworks debug service enabled by default," 2012.

[41] K. Stouffer, J. Falco, and K. Scarfone, "Guide to industrial control systems (ics) security," *NIST special publication*, vol. 800, no. 82, pp. 16–16, 2011.

[42] S. McLaughlin and S. Zonouz, "Controller-aware false data injection against programmable logic controllers," in *Smart Grid Communications (SmartGridComm), 2014 IEEE International Conference on*, pp. 848–853, IEEE, 2014.

[43] S. McLaughlin and P. McDaniel, "Sabot: Specification-based payload generation for programmable logic controllers," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 439–449, ACM, 2012.

[44] K. T. Erickson and J. L. Hedrick, *Plant-wide process control*, vol. 4. John Wiley & Sons, 1999.

[45] A. Falcione and B. H. Krogh, "Design recovery for relay ladder logic," *IEEE Control Systems*, vol. 13, no. 2, pp. 90–98, 1993.

[46] N. G. Ferreira and P. S. M. Silva, "Automatic verification of safety rules for a subway control software," *Electronic Notes in Theoretical Computer Science*, vol. 130, pp. 323–343, 2005.

[47] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics attacking plcs with physical model aware rootkit," in *Proceedings of the Network & Distributed System Security Symposium, San Diego, CA, USA*, pp. 26–28, 2017.

[48] J. Klick, S. Lau, D. Marzin, J.-O. Malchow, and V. Roth, "Internet-facing plcs as a network backdoor," in *Communications and Network Security (CNS), 2015 IEEE Conference on*, pp. 524–532, IEEE, 2015.

[49] W. Li, L. Xie, Z. Deng, and Z. Wang, "False sequential logic attack on scada system and its physical impact analysis," *Computers & Security*, vol. 58, pp. 149–159, 2016.

[50] N. Trcka, M. Moulin, S. Bopardikar, and A. Speranzon, "A formal verification approach to revealing stealth attacks on networked control systems," in *Proceedings of the 3rd international conference on High confidence networked systems*, pp. 67–76, ACM, 2014.

[51] M. B. Younis and G. Frey, "Visualization of plc programs using xml," in *American Control Conference, 2004. Proceedings of the 2004*, vol. 4, pp. 3082–3087, IEEE, 2004.

[52] M. Marcos, E. Estevez, F. Perez, and E. Van Der Wal, "Xml exchange of control programs," *IEEE Industrial Electronics Magazine*, vol. 3, no. 4, 2009.

[53] D. Darvas, I. Majzik, and E. B. Viñuela, "Generic representation of plc programming languages for formal verification," in *Proc. of the 23rd PhD Mini-Symposium*, pp. 6–9.

[54] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Programming controllers in structured text language of iec 61131-3 standard," *Journal of Applied Computer Science*, vol. 16, no. 1, pp. 49–67, 2008.

[55] D. Darvas, E. Blanco Vinuela, and B. Fernández Adiego, "Plcverif: A tool to verify plc programs based on model checking techniques," 2015.

[56] S. Zhou, H. Zedan, and A. Cau, "Run-time analysis of time-critical systems," *Journal of Systems Architecture*, vol. 51, no. 5, pp. 331–345, 2005.

[57] H. Salarian, K.-W. Chin, and F. Naghdy, "Coordination in wireless sensor–actuator networks: A survey," *Journal of Parallel and Distributed Computing*, vol. 72, no. 7, pp. 856–867, 2012.

[58] V. Kumar, A. Fensel, and P. Fröhlich, "Context based adaptation of semantic rules in smart buildings," in *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, p. 719, ACM, 2013.

[59] A. A. Nacci, B. Balaji, P. Spoletini, R. Gupta, D. Sciuto, and Y. Agarwal, "Buildingrules: a trigger-action based system to manage complex commercial buildings," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, pp. 381–384, ACM, 2015.

[60] Y. Agarwal, R. Gupta, D. Komaki, and T. Weng, "Buildingdepot: an extensible and distributed architecture for building data storage, access and sharing," in *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pp. 64–71, ACM, 2012.

[61] H.-P. Lam and G. Governatori, "The making of spindle," in *Rule Interchange and Applications*, pp. 315–322, Springer, 2009.

[62] A. J. Garcıa, D. Gollapally, P. Tarau, and G. R. Simari, "Deliberative stock market agents using jinni and defeasible logic programming," in *Proceedings of esawâ00 engineering societies in the agentsâ world, workshop of ecai 2000*, 2000.

[63] A. J. García and G. R. Simari, "Defeasible logic programming: An argumentative approach," *Theory and practice of logic programming*, vol. 4, no. 1+ 2, pp. 95–138, 2004.

[64] V. Martinez *et al.*, "On the use of presumptions in structured defeasible reasoning," 2012.

[65] S. Parsonsa, K. Atkinsonb, K. Haighc, K. Levittd, P. M. J. Rowed, M. P. Singhf, and E. Sklara, "Argument schemes for reasoning about trust," *Computational Models of Argument: Proceedings of COMMA 2012*, vol. 245, p. 430, 2012.

[66] World Wide Web Consortium and others, "PROV-Overview: an overview of the PROV family of documents." `https://www.w3.org/TR/prov-overview/`, 2013.

[67] T. Moyer, K. Chadha, R. Cunningham, N. Schear, W. Smith, A. Bates, K. Butler, F. Capobianco, T. Jaeger, and P. Cable, "Leveraging data provenance to enhance cyber resilience," in *Cybersecurity Development (SecDev), IEEE*, pp. 107–114, IEEE, 2016.

[68] X. Han, T. Pasquier, and M. Seltzer, "Provenance-based intrusion detection: Opportunities and challenges," in *10th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2018)*, (London), USENIX Association, 2018.

[69] Y. Xie, D. Feng, Z. Tan, and J. Zhou, "Unifying intrusion detection and forensic analysis via provenance awareness," *Future Generation Computer Systems*, vol. 61, pp. 26–36, 2016.

[70] S. COBB, "10 things to know about the october 21 iot ddos attacks," 2016.

[71] S. Munir and J. A. Stankovic, "Depsys: Dependency aware integration of cyber-physical systems for smart homes," in *Cyber-Physical Systems (ICCPS), 2014 ACM/IEEE International Conference on*, pp. 127–138, IEEE, 2014.

[72] M. Ma, J. A. Stankovic, and L. Feng, "Cityresolver: A decision support system for conflict resolution in smart cities," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 55–64, IEEE, 2018.

[73] M. Ma, S. M. Preum, and J. A. Stankovic, "Cityguard: A watchdog for safety-aware conflict detection in smart cities," in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, pp. 259–270, ACM, 2017.

[74] M. Simulink, "Thermal model of a house." `https://www.mathworks.com/help/simulink/examples/thermal-model-of-a-house.html`. Accessed: 2018-07-27.

[75] R. J. De Dear and G. S. Brager, "Thermal comfort in naturally ventilated buildings: revisions to ashrae standard 55," *Energy and buildings*, vol. 34, no. 6, pp. 549–561, 2002.

[76] M. Triska, "Theorem proving with prolog." `https://www.metalevel.at/prolog/theoremproving`. Accessed: 2018-07-27.

[77] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "Swi-prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[78] Z. B. Celik, P. McDaniel, and G. Tan, "Soteria: Automated iot safety and security analysis," *arXiv preprint arXiv:1805.08876*, 2018.

[79] A. A. Farooq, "Iot testbed for iotc2," Dec. 2018.

[80] Y. Fathy, P. Barnaghi, and R. Tafazolli, "Distributed spatial indexing for the internet of things data management," in *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pp. 1246–1251, IEEE, 2017.

[81] F. Alsubaei, A. Abuhussein, and S. Shiva, "Quantifying security and privacy in internet of things solutions," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–6, IEEE, 2018.

[82] H. Kinkelin, V. Hauner, H. Niedermayer, and G. Carle, "Trustworthy configuration management for networked devices using distributed ledgers," *arXiv preprint arXiv:1804.04798*, 2018.

[83] M. Mohsin, Z. Anwar, G. Husari, E. Al-Shaer, and M. A. Rahman, "Iotsat: A formal framework for security analysis of the internet of things (iot)," in *Communications and Network Security (CNS), 2016 IEEE Conference on*, pp. 180–188, IEEE, 2016.

[84] C. Vannucchi, M. Diamanti, G. Mazzante, D. Cacciagrano, R. Culmone, N. Gorogiannis, L. Mostarda, and F. Raimondi, "Symbolic verification of event–condition–action rules in intelligent environments," *Journal of Reliable Intelligent Environments*, vol. 3, no. 2, pp. 117–130, 2017.

[85] F. Corno and M. Sanaullah, "Design-time formal verification for smart environments: an exploratory perspective," *Journal of Ambient Intelligence and Humanized Computing*, vol. 5, no. 4, pp. 581–599, 2014.

[86] A. Coronato and G. De Pietro, "Formal design of ambient intelligence applications," *Computer*, vol. 43, no. 12, pp. 60–68, 2010.

[87] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and P. Bahl, "An operating system for the home," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 25–25, USENIX Association, 2012.

[88] F. B. Schneider, G. Morrisett, and R. Harper, "A language-based approach to security," in *Informatics*, pp. 86–101, Springer, 2001.

[89] J. Pincus and B. Baker, "Mitigations for low-level coding vulnerabilities: Incomparability and limitations."

[90] P. Groth and L. Moreau, "Prov-overview. an overview of the prov family of documents," 2013.

[91] A. A. Farooq, E. Al-Shaer, T. Moyer, and K. Kant, "Iotc2: A formal method approach for detecting conflicts in large scale iot systems," in *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, IEEE, 2019.

[92] Trung Dong Huynh, "A library for W3C Provenance Data Model supporting PROV-JSON, PROV-XML and PROV-O (RDF)." `https://pypi.org/project/prov/`.

[93] A. Hagberg, D. Schult, P. Swart, D. Conway, L. Séguin-Charbonneau, C. Ellison, B. Edwards, and J. Torrents, "Networkx. high productivity software for complex networks," *Webová strá nka https://networkx. lanl. gov/wiki*, 2013.

[94] A. Al Farooq, E. Al-Shaer, T. Moyer, and K. Kant, "Iotc 2: A formal method approach for detecting conflicts in large scale iot systems," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pp. 442–447, IEEE, 2019.

[95] A. Al Farooq, J. Marquard, K. George, and T. Moyer, "Detecting safety and security faults in plc systems with data provenance," in *2019 IEEE International Symposium on Technologies for Homeland Security (HST)*, pp. 1–6, 2019.

[96] A. Al Farooq, J. Marquard, K. George, and T. Moyer, "Detecting safety and security faults in plc systems with data provenance," in *2019 IEEE International Symposium on Technologies for Homeland Security (HST)*, pp. 1–6, IEEE, 2019.

[97] R. Ramanathan, "The iec 61131-3 programming languages features for industrial control systems," in *2014 World Automation Congress (WAC)*, pp. 598–603, IEEE, 2014.

[98] V. Birk, "pypeg â a peg parser-interpreter in python," 2014.

[99] F. Amigoni, N. Gatti, C. Pinciroli, and M. Roveri, "What planner for ambient intelligence applications?," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 35, no. 1, pp. 7–21, 2004.

[100] M. Cristani, E. Karafili, and C. Tomazzoli, "Energy saving by ambient intelligence techniques," in *2014 17th International Conference on Network-Based Information Systems*, pp. 157–164, IEEE, 2014.

[101] M. Cristani, E. Karafili, and C. Tomazzoli, "Improving energy saving techniques by ambient intelligence scheduling," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pp. 324–331, IEEE, 2015.

[102] G. Antoniou, M. J. Maher, and D. Billington, "Defeasible logic versus logic programming without negation as failure," *The Journal of Logic Programming*, vol. 42, no. 1, pp. 47–57, 2000.

[103] A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni, "An abstract, argumentation-theoretic approach to default reasoning," *Artificial intelligence*, vol. 93, no. 1-2, pp. 63–101, 1997.

[104] H. Prakken, *Logical tools for modelling legal argument: a study of defeasible reasoning in law*, vol. 32. Springer Science & Business Media, 2013.

[105] E. Daga, A. Gangemi, and E. Motta, "Reasoning with data flows and policy propagation rules," *Semantic Web*, vol. 9, no. 2, pp. 163–183, 2018.

[106] E. Sierra, A. Hossian, D. Rodríguez, M. García-Martínez, P. Britos, and R. García-Martínez, "Intelligent systems applied to optimize buildingâs environments performance," in *IFIP International Conference on Artificial Intelligence in Theory and Practice*, pp. 237–244, Springer, 2008.

[107] H.-P. Lam and G. Governatori, "Towards a model of uavs navigation in urban canyon through defeasible logic," *Journal of Logic and Computation*, vol. 23, no. 2, pp. 373–395, 2013.

[108] G. Governatori, F. Olivieri, A. Rotolo, S. Scannapieco, and M. Cristani, "Picking up the best goal," in *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pp. 99–113, Springer, 2013.

[109] G. Governatori, F. Olivieri, S. Scannapieco, A. Rotolo, and M. Cristani, "The rationale behind the concept of goal," *Theory and Practice of Logic Programming*, vol. 16, no. 3, pp. 296–324, 2016.

[110] F. Olivieri, G. Governatori, S. Scannapieco, and M. Cristani, "Compliant business process design by declarative specifications," in *International Conference on Principles and Practice of Multi-Agent Systems*, pp. 213–228, Springer, 2013.

[111] A. Bikakis, G. Antoniou, and P. Hasapis, "Strategies for contextual reasoning with conflicts in ambient intelligence," *Knowledge and Information Systems*, vol. 27, no. 1, pp. 45–84, 2011.

[112] V. Nagendra, A. Bhattacharya, V. Yegneswaran, A. Rahmati, and S. Das, "An intent-based automation framework for securing dynamic consumer iot infrastructures," in *Proceedings of The Web Conference 2020*, pp. 1625–1636, 2020.

[113] Z. B. Celik, G. Tan, and P. D. McDaniel, "Iotguard: Dynamic enforcement of security and safety policy in commodity iot.," in *NDSS*, 2019.

[114] T. G. Stavropoulos, E. Kontopoulos, N. Bassiliades, J. Argyriou, A. Bikakis, D. Vrakas, and I. Vlahavas, "Rule-based approaches for energy savings in an ambient intelligence environment," *Pervasive and Mobile Computing*, vol. 19, pp. 1–23, 2015.

[115] A. Al Farooq, "Probabilistic modeling of erroneous human response to in-vehicle route guidance systems: A domain decomposition-based algorithm," 2013.