



**A FRAMEWORK FOR CATEGORIZATION
OF INDUSTRIAL CONTROL SYSTEM
CYBER TRAINING ENVIRONMENTS**

THESIS

Evan G. Plumley, First Lieutenant, US Army
AFIT-ENG-MS-17-M-059

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Army, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-17-M-059

A FRAMEWORK FOR CATEGORIZATION OF INDUSTRIAL CONTROL
SYSTEM
CYBER TRAINING ENVIRONMENTS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Evan G. Plumley, B.S.C.S.
First Lieutenant, US Army

March 2017

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-17-M-059

A FRAMEWORK FOR CATEGORIZATION OF INDUSTRIAL CONTROL
SYSTEM

CYBER TRAINING ENVIRONMENTS

THESIS

Evan G. Plumley, B.S.C.S.
First Lieutenant, US Army

Committee Membership:

LTC Mason J. Rice, PhD
Chair

Lt Col. John M. Pecarina, PhD
Member

Mr. Stephen J. Dunlap
Member

Abstract

First responders and professionals in hazardous occupations undergo training and evaluations for the purpose of mitigating risk and damage. For example, helicopter pilots train with multiple categorized simulations that increase in complexity before flying a real aircraft. However in the industrial control cyber incident response domain, where incident response professionals help detect, respond and recover from cyber incidents, no official categorization of training environments exist. To address this gap, this thesis provides a categorization of industrial control training environments based on realism. Four levels of environments are proposed and mapped to Bloom's Taxonomy. This categorization will help organizations determine which training environment best aligns with their training needs and budgets.

Acknowledgements

I would like to thank my entire thesis committee and research team for their guidance and help through out this process.

I would like to specifically thank Stephen Dunlap for sharing his knowledge and time. This research would not have been possible without him.

I would also like to thank LTC Rice for serving as my advisor and guiding me through the thesis process.

Evan G. Plumley

I dedicate this thesis to my family. To my mother and father for their unconditional love and support. To my brothers for their guidance and leadership.

Table of Contents

	Page
Abstract	iv
Acknowledgements	v
List of Figures	ix
List of Tables	x
I. Introduction	1
1.1 Motivation	1
1.2 Contributions	2
II. Background	4
2.1 The Need for ICS Cyber Incident Response	4
2.2 Training Environments for ICS Incident Response	7
2.2.1 Government	8
2.2.2 Industry	10
2.2.3 Academia	10
2.3 Bloom's Taxonomy: A Cognitive Complexity Framework	11
2.4 Relating Bloom's Taxonomy to Existing Training Platforms	12
III. Methodology	16
3.1 A Framework for ICS Training Environment Development	16
3.1.1 Preparation	17
3.1.2 Detection and Analysis	19
3.1.3 Containment, Eradication and Recovery Phase	22
3.1.4 Post-Incident Activity	24
3.1.5 Training Administration for Environments	25
3.2 Categorization Levels of ICS Training Environments	27
3.2.1 Level 0 Training Environments	27
3.2.2 Level 1 Training Environments	27
3.2.3 Level 2 Training Environments	28
3.2.4 Level 3 Training Environments	29
3.2.5 Level 4 Training Environments	30
3.3 Mapping of Training Environment Levels to Bloom's Taxonomy	31
3.4 Construction of ICS Training Environments	33
3.4.1 Construction of Level 0 ICS Training Environments	34

	Page
3.4.2 Construction of Level 1 ICS Training Environments	34
3.4.3 Construction of Level 2 ICS Training Environments	36
3.4.4 Construction of Level 3 ICS Training Environments	37
3.4.5 Construction of Level 4 ICS Training Environments	47
IV. Conclusions	51
4.1 Conclusions of Research	51
4.1.1 Significance of Research	51
4.1.2 Recommendations for Future Research	52
Bibliography	53
Appendix A. Wastewater Environment Y-Box Communication and Interface Code	58
1.1 Wastewater Interface and Y-Box Communication Code	58
1.2 Y-Box Communication Engine	85
1.3 Wastewater Blower fan interface control	88
Appendix B. Prison Environment Y-Box Communication and Interface Code	90
2.1 Prison Interface and Y-Box Communication Code	90
2.2 Prison Guard Station Interface Code	111
2.3 Prison Indicator Image Display Code	117
2.4 Prison Mantrap Image Display Code	118
2.5 Prison Cell Door Image Display Code	123
Appendix C. Power Substation Environment Y-Box Communication and Interface Code	127
3.1 Power Substation Interface and Y-Box Communication Code	127

List of Figures

Figure	Page
1. AFIT automotive CAN bus emulation testbed.	29
2. Mounted stoplight control system used by AFIT for ICS security labs.	31
3. Level 3 wastewater treatment training environment.	41
4. RSLogix 5000 engineering workstation software for the wastewater environment.	41
5. Administrative interface for a real time view of attacking and defending effects on the wastewater environment.	42
6. Level 3 prison training environment.	44
7. HMI for the prison training environment.	45
8. Administrative monitoring view of the prison exercise environment.	45
9. Level 3 power substation training environment.	48
10. Power substation training environment programming software.	49
11. Power substation training environment monitoring HMI software.	50

List of Tables

Table	Page
1. Bloom's Taxonomy (revised) [15, 11].	12
2. Types of FSTDs [8].	14
3. Preparation: Skills and components.	19
4. Detection and Analysis: Skills and components.	21
5. Containment, eradication and recovery: Skills and components.	25
6. Post-Incident activity: Skills and components.	25
7. Training administration: skills and components.	26
8. Mapping ICS training environment levels to Bloom's Taxonomy.	34

A FRAMEWORK FOR CATEGORIZATION OF INDUSTRIAL CONTROL SYSTEM CYBER TRAINING ENVIRONMENTS

I. Introduction

1.1 Motivation

On the evening of April 17th, 2013 in the city of West, Texas, an act of arson at a fertilizer plant resulted in an explosion that killed 15 people; 10 were first responders fighting the fire. The deaths were the result of unprepared first responders who were not trained to handle a chemical fire and did not fully comprehend the explosive hazard of the materials within the plant [19, 10]. The Emergency Planning and Community Right to Know Act requires all companies to report any potential chemical hazards stored in their facilities. However, there are no legal requirements for the local first responders to train according to the reports.

The mechanical processes within an industrial environment are controlled by systems that generally lack cyber security and possess the ability to pose a physical threat to personnel and equipment. The first responders that are tasked to address cyber incidents within an industrial control environment must be properly trained and prepared to deal with various types of industrial cyber related situations. The training should include a holistic curriculum and a variety of training environments that enable responders to gradually increase their knowledge while ensuring necessary skills for handling incidents in a hazardous environment.

To avoid disasters like the Texas explosion, it is imperative to use training envi-

ronments that have the capability to provide training and assessment for the range of ICS incident response knowledge. The skills that a cyber ICS first responder needs must be identified to provide these training environments. Once the necessary skills are identified, the components of industrial control environments that necessitate those skills can be derived enabling the creation of training environments that ensure comprehensive ICS response capability assessment. There is currently no accepted method to categorize the existing cyber environments in terms of their capabilities and components.

This thesis proposes a framework for the identification and mapping of ICS cyber incident response skills to ICS training environment components. This thesis also proposes a categorization of training environments based on practicality and realism. This categorization helps organizations determine the best fit regarding cyber response training for personnel and budgets.

1.2 Contributions

This research proposes industrial control cyber response skills that a cyber first responder requires to effectively respond to a cyber incident in an industrial control environment and the training components that are necessary to assess the proposed skills. This research also proposes a categorization of tiered training environment that can be constructed to ensure a curricula that is inclusive of the presented skills and provides environment and exercise scenario examples for each category. The research presented makes the following contributions to the industrial control cyber response community.

1. Provides a survey of existing industrial control training environments.
2. Provides a framework which identifies the skills of an ICS incident responder and maps the skills to the components of an ICS training environment.

3. Categorizes training environments into five levels of increasing realism and relevance.
4. Leverages educational taxonomies for analysis and differentiation of training environments.
5. Introduces scenarios and components to match each level of the proposed categorization.

II. Background

2.1 The Need for ICS Cyber Incident Response

A lack of industrial control system (ICS) cyber defense has caused concern within the community. Contributing factors for the lack of defense include cost, diversity of the systems, and lack of demand from users who are reluctant to make any changes to their current operational system. The overdue awareness and scrutiny needed for ICS cyber security and incident response has only recently begun to propagate its way into the community. The long life cycle and high cost of industrial technologies results a reluctance for organizations to upgrade their system's security [27].

Due to the vulnerable nature of the operational technology, every organization should plan for proper cyber incident response resulting in a damage mitigating recovery and deep forensic analysis of the root cause of the incident. An improper response to a cyber incident in an industrial control environment can result in irrecoverable damages or ineffective corrective actions [27]. These proprietary and unique systems require experts in the specific field of operational technology (OT) cyber security for the purpose of incident response and forensics due to inherent discrepancies between the OT environment and traditional IT cyber security practices. Traditional defense software such as intrusion detection systems (IDS) and firewalls may not be deployable in an effective manner on ICS networks due to the proprietary nature of the protocols and the risk of such technology bringing down critical portions of the system or adjacent components. New control technology is becoming more security centric, however the technology was built with successful operation and durability as the priorities. A model commonly used to guide cyber security policies is the CIA triad which stands for confidentiality, integrity and availability. While IT systems may prioritize confidentiality and integrity over availability leading to system down-

times and security features that are inconvenient for system users, industrial control systems have prioritized availability to ensure that systems running critical processes don't experience downtime due to cumbersome security features. Older legacy systems are still widely deployed in critical infrastructure and industry with 70% of the existing technology being ten years or older in 2008 [9]. With the high cost of component replacement and the difficulties of taking the systems offline, there is a need for the capability to defend the existing infrastructure from real time cyber intrusions.

Increasing the cyber security stance of an industrial control facility is rarely trivial due to the proprietary nature of the hardware and software being used, however security must be increased to mitigate the possibility of cyber attacks causing facility damage or bodily harm to organization employees. Technology replacement and upgrades usually involve heavy participation from the vendor and integrator. To safely apply system changes, a deep analysis of the system is needed to understand the secondary effects of taking the system offline for any period of time. Equipment that has an integral part of the business operation may not be possible to replace in a seamless manner, leaving the organization with a low security legacy system. Software Patches for these systems requires extensive testing for compatibility before deployment which is impossible to do in many cases. Vulnerabilities may also be reported long before a patch is released due to a lack of product support from the vendor, delay for testing, or a limited demand. The devices that control the behavior of system sensors and actuators are the most valuable assets for an attacker and the difficulties of increasing the security in these devices make them the most vulnerable. [9]

Security regulations for industry and infrastructure are difficult to create and enforce due to the diversity of the systems, the inability of organizations to afford proper security and the possibility of regulations causing crippling damage to systems.

The National Institute of Standards and Technology (NIST) and the Department of Homeland Security (DHS) have both published multiple documents providing recommendations on improving industrial control system cyber security. The consensus of the documentation is to create policies for workers within an organization that discourages the opening of attack vectors onto the operational technology (OT) environment and the placement of firewalls or air gaps to separate the IT side of the organization from the OT side [24]. This strategy of isolation can easily fail due to improper deployment or security policy violations providing attackers direct access to the operational technology and the field devices that control the system endpoints. While organizational security regulations can be easily deployed and enforced on IT networks due to the prioritization of confidentiality and integrity, they contradict the emphasis of availability in the operational environment.

Most operations and management staff within ICS do not have the necessary skills to properly collect, analyze and examine the command and control traffic in their system and will have difficulty deciphering a cyber attack from a non cyber related malfunction [9]. They also most likely do not have the skills to properly isolate and eradicate a threat, therefore they will heavily rely on vendor or integrator support in times of malfunction or attack. Vendors hold proprietary knowledge about individual system components while integrators who have constructed the system as a whole will have the original configuration and logic files for the system as well as a cohesive understanding of the process. Support from a vendor should be discussed as part of an incident preparation plan to ensure that issues involving vendor contracts and warranties do not interrupt an immediate response to an attack. Vendor support may be necessary and the delay of response from a vendor can result in catastrophic damage or financial loss to the affected organization.

The system integrators who handle the logistics system construction and config-

urations have a crucial role in the matter of incident recovery and response. It is likely that the engineering and configuration files that were created at the time of a system's deployment are stored by the integrators for backup and reference purposes. The integrators bid for the contract to create the system that meets the organization's requirements for operation and they do so by contacting vendors to acquire support and components and then engineering the system itself. Proprietary knowledge of the system is held by the integrator who will have a deeper functional understanding than an everyday operator. To properly handle an incident, analysis of the whole system is necessary, however a lot of systems contain components from many different vendors making the integrators a valuable asset for operational knowledge.

The need for organizations to deploy a specific incident response team for ICS cyber is driven by high cyber risk and vulnerability in physically hazardous environments. DHS recommended practices for ICS incident response includes a plan for organizing an ICS security team internal to an organization [27]. This plan is not always possible due to lack of skilled staff in the area of cyber security or the lack of funding to higher security specialists. Dedicated emergency response teams need to be accessible to organizations for a guaranteed tailored response to ICS cyber attacks. Whether a response team is constructed internally, or contracted out to an external organization, a plan needs to be in place for an organization to deploy personnel in a swift and efficient manner.

2.2 Training Environments for ICS Incident Response

Contributing factors include cost, diversity of the systems, long life cycles and lack of demand from organizations that are reluctant to make changes to their operational system [27]. Most ICS organization personnel do not have the necessary skills to properly collect, analyze and examine the command and control traffic in their system

and will likely have difficulty deciphering a cyber attack from a non-cyber related malfunction [9]. Not every ICS organization can support a high level of training for their cyber response team, however, they may be able to support a lower level of training that balances organizational goals and budgets.

To properly train ICS cyber security professionals and response teams, training environments and capabilities that support training at different levels of expertise ranging from introductory simulations to full scale control systems need to exist. Effective full scale training should incorporate equipment from a number of vendors, show genuine cyber physical effects and include diverse configurations. The realism of a full scale testbed would allow trainees and response teams to experience the intricacies of a real system and provide insight into protocols, physical-safety-override systems and diverse configurations. Deficiencies in training environments and curricula have been shown to exist for ICS first responders in the training facilities and courses provided by government, industry and academia (e.g., see [3]).

2.2.1 Government.

Most training provided by the government for the ICS community is offered by the Department of Homeland Security (DHS). The focus, however, is on the effects of attacks and development of mitigation strategies as opposed to emergency incident response. The DHS ICS-CERT advanced training course provides a partial control system for demonstrating exploit effects and culminates in a red and blue team exercise where participants either attack or defend the control system [14]. The training environment does not represent a full scale system, however it does include realistic hardware and displays real physical effects. Participants for the advanced course are expected to have prior knowledge on both ICS and IT networks. The advanced course encourages discussion between IT and ICS professionals which enhances the devel-

opment of contextual knowledge for both communities. In a real incident response setting, the professionals from the IT and industrial control communities must productively communicate to avoid further damage and ensure a successful recovery. The exercises and training environment falls short in creating a fully complex system and leans heavily on traditional IT attack and defense rather than focusing on industrial control specific practices. Participants must travel to the facility in Idaho Falls, Idaho for the five day course. Idaho National Laboratory claims to have the capabilities to replicate any customer’s control system specifications to conduct simultaneous attacks on multiple systems or perform customized full-scale cyber attacks on an exact replica system [12].

Industrial control security research is conducted at Sandia National Laboratories. Sandia National Laboratories, which operates as a part of the U.S. Department of Energy’s National Nuclear Security Administration, is a leader in providing industrial control education and training for industry, government and academia [22]. They offer a SCADA Assessment Training Course which provides an educational survey of SCADA and ICS used within the infrastructure and industry. The primary purpose of the course is to provide methodologies and tools for assessing the security of industrial systems. The course is only offered at Sandia’s discretion to individuals with need-to-know and by invitation only [22].

The research laboratories also use an ICS testbed belonging to the DOE known as the National SCADA Test Bed where there are full scale realistic systems designated for research. The National SCADA Test Bed recreates real-world control systems for the purposes of research and education [26]. The realism and fidelity of the testbed emulates that of a real environment. However, this research facility is not currently being used to train SCADA/ICS cyber defenders or incident response teams [26].

2.2.2 Industry.

Industry provisioned training is similar to government training, offering to teach customers about ICS fundamentals, security auditing and assessment. The training environments often include hands-on laboratory experience that allows participants to become familiarized with the vendor's programmable logic controllers (PLCs) which may be networked in a way to simulate a real industrial environment. The vendors often travel to various locations and offer specific courses to individual organizations [3]. The classes are not tailored to train security experts and are more for the general education of anyone handling industrial control equipment. While these classes help in familiarization of components, they do not expose a potential cyber responder to the complexity of a complete system.

Training environments also exist as control environment simulation software. An example of control simulation software is the LogixPro-500 PLC Simulator which combines the RSLogix 500 engineering environment with the ProSim-II programmable process simulation that emulates a PLC [25]. While this simulation software is not security focused, it can assist in introducing beginner level defenders to the logic of industrial controllers.

2.2.3 Academia.

Academia also uses assorted testbeds for research and education. Mississippi State University has an ICS testbed that emulates a real world control system with physical processes for industrial control cyber security [16]. Other organizations in academia have constructed ICS environments with fully or partially simulated controllers and processes. Reaves et al. [20] created a testbed that fully simulates an ICS environment including the simulation of control systems and physical processes. Wertzberger et al.'s [28] environment combines real world control hardware with simulated physical

processes and network for a training environment that is a step beyond full simulation. While these environments are great for introductory training and conceptual education, they lack the complexity of a full scale system and emergency response procedures.

The SANS Institute, a cooperative research and education organization provides training for customers through online courses, classrooms or in a mentored settings around the world. Their effort to construct an ICS training environment resulted in the SANS CyberCity. The CyberCity is a scaled model of a small city that is controlled by computers, networks, control hardware and embedded devices emulating infrastructure including the power grid, water system, traffic system and HVAC systems [23]. This model city (controlled by ICS systems) is coupled with an on-site course in Austin, Texas that guides participants through various missions that provide exposure to components of industrial control systems. The course is an adequate overview of ICS components including human machine interfaces (HMIs), various industrial protocols and data historians. The model city training environment coupled with control hardware allows participants to view the physical effects of their cyber actions while providing context for realistic vendor supplied technology. The course succeeds in providing participants an overview in common security flaws and thwarting attacks against industrial systems.

2.3 Bloom’s Taxonomy: A Cognitive Complexity Framework

Educational frameworks have been created to provide insight on the cognitive value acquired from an educational activity (e.g., assigned projects or homework). An educational psychologist named Benjamin Bloom (1913-1999) sought to classify educational goals and objectives based upon cognitive complexity [11]. The result was Bloom’s Taxonomy which was revised in 2001 and is widely used today by teachers and

professors for structuring courses that encourage students to learn, apply knowledge and develop an array of thinking skills.

Bloom’s revised taxonomy consists of six major categories of educational goals which are listed in order from the least complex category to the most. The taxonomy illustrates the progression of cognitive complexity from basic understanding to the creation of original ideas and concepts. To develop original ideas within a field of study, you must first solidify a base of knowledge to gain holistic context which places emphasis and importance on all levels of Bloom’s Taxonomy. The framework provides a way to align educational tools to the level of skill and complexity that the tool is meant to invoke.

Table 1. Bloom’s Taxonomy (revised) [15, 11].

1. Remembering	Retrieving, recognizing and recalling relevant knowledge from long-term memory.
2. Understanding	Constructing meaning from oral, written and graphic messages through interpreting, exemplifying, classifying, summarizing, inferring, comparing and explaining.
3. Applying	Carrying out or using a procedure through executing or implementing.
4. Analyzing	Breaking material into constituent parts, determining how the parts relate to one another and their overall purpose through differentiating, organizing and attributing.
5. Evaluating	Making judgments based on criteria and standards through checking and critiquing.
6. Creating	Putting elements together to form a coherent or functional whole; reorganizing elements into a new pattern or structure through generating, planning or producing.

2.4 Relating Bloom’s Taxonomy to Existing Training Platforms

Bloom strongly suggests acquiring concrete knowledge before increasing the intricacy of the training environment presented to the student. In many fields of education, especially those with a high risk of incurring damage to property or injury, a form

of training simulation is often used to gradually introduce the trainee or student to more variables of complexity before attempting the authentic hazardous task.

Simulations to build a base of knowledge and comfort for trainees have been used in multiple hazardous and highly technical professions (e.g., military weapons and vehicle operation, aircraft piloting and astronautics). The U.S. Army uses multiple tank simulators to qualify their gunnery soldiers and drivers before operating a real tank [2, 1]. The Army also uses simulations for generic marksmanship training for its soldiers called the Engagement Skills Trainer. To take their training a step further, the Army also uses a training tool called the Virtual Convoy Operations Trainer where collective training can be practiced in a virtual environment and multiple soldiers can train together [4]. These simulations ensure soldier readiness before deployment into unpredictable and dangerous environments.

The U.S. Air Force trains pilots by assessing proficiency in simple simulations, gradually adding complexity into the simulations and finally conducting fully realistic live training. The Air Force boasts a wide range of simulation capabilities, from full mission simulators to desktop PC-based simulations [18]. The live training that the Air Force provides is comprised of real equipment, weaponry, environments and targets. Examples of live training are engaging aerial targets, deploying drones and firing munitions. Ensuring the competence of fighter pilots using realistic training scenarios before conducting real operations lowers the risk for damage (or death) in an inherently dangerous profession.

For private helicopter pilot training, different categories of training simulations, known as Flight Simulation Training Devices (FSTDs), have been defined with certification criteria. The European Aviation Safety Agency (EASA) is the certifying authority for the flight simulation training environments and has developed exact specifications that define each level of environment. The specifications define the ex-

act capabilities that each level of environment is required to provide to be certified, with specifications defining everything from the size of the cockpit to the sound of rain hitting the aircraft’s windshield in a virtualized environment [7]. The categories are: (i) Flight and Navigational Procedures Trainers (FNPT); (ii) Flight Training Devices (FTD); (iii) Full Flight Simulators (FFS); and (iv) Other Training Devices (OTD) covering any unregulated training tool or environment. Each of the categories serves a different purpose by introducing new variables and increasing realism (see Table 2). These simulation categories of increasing realism were created to ensure that pilots demonstrate mastery of the equipment and procedures in training to reduce risk during real flights.

Table 2. Types of FSTDs [8].

1. FNPT	A fixed-base generic system, primarily used for helicopter initial and refresher training, including basic and safety procedures, emergencies, navigation, instrument rating and multi-crew co-operation.
2. FTD	A fixed-based system, simulating a specific type of helicopter. In addition to the FNTP capabilities, the FTD is designed for rating pilots on specific helicopter types. This FSTD has a limited checking/testing capability due to the fact that it does not include a motion or vibration system.
3. FFS	A motion-base system, which provides, in a addition to the FTD, motion and vibration cues. It has the highest level of technical complexity and training capability and can be used for proficiency evaluation
4. OTD	Other training devices exist and can be defined as a training aid where a complete cockpit or flight deck is unnecessary. There are no regulations covering OTDs and can vary in form from a desktop computer to a helicopter dashboard. These devices are often used to drill pre-flight tasks or familiarize a pilot with a single cockpit instrument.

NASA houses a multitude of training simulators to familiarize its trainees with many environments and situations (e.g., launch, landing, payload and rendezvous activities) [17]. These simulations include both fixed-based and motion-based simula-

tors. Astronauts train for a combined total of 300 hours in these simulators to qualify for real operations. NASA training also extends outside of the virtual environment and attempts to recreate the environment of space with specialized aircraft, pools and other technologies. This training is necessary for the astronauts to gain comfort and confidence before operating in the hazardous and unpredictable environment of space.

Every training environment provided by these organizations are intended to gradually assimilate the trainees into a real, complex and diverse environment. Each environment serves a different purpose and is tailored toward the needs of the trainee. With every step forward in the training process, a new training platform is introduced which provides new concepts and builds the strong base of knowledge needed to operate in the unpredictable real environment.

III. Methodology

3.1 A Framework for ICS Training Environment Development

A framework for training environment development enables organizations to customize training components to fit their needs. A framework that enables ICS response organizations to map ICS incident response skills to training environment capabilities would ensure the training goals of the organizations are addressed. There is currently no accepted method to categorize the existing environments in terms of their capabilities and components.

The framework presented in this section identifies ICS first response skills and training environment components that facilitate those skills. The skills will be broken down into overarching phases of a cyber incident response that follow the National Institute of Standards and Technology (NIST) Incident Response Lifecycle [5]. The skills presented in each phase of the lifecycle are the result of the analysis and consolidation of multiple sources including the NIST National Cybersecurity Workforce Framework, DHS Recommended Practice publications, NIST guides for incident handling and ICS security and curricula for publicly available ICS security classes.

The NIST Incident Response Lifecycle is split into four iterative phases. The phases are (i) preparation, (ii) detection and analysis, (iii) containment, eradication and recovery, and (iv) post-incident activity. These general phases will be used for the categorization of the consolidated ICS incident response skills presented in this section. An additional section for components that enable effective training administration is also present in the framework.

3.1.1 Preparation.

The preparation phase of cyber incident response focuses on an organization's readiness to rapidly and effectively respond to a cyber attack. NIST describes the phase as establishing an incident response capability and taking preventive measures by ensuring the security of networks, systems and applications. This includes the gathering of hardware and software response tools and establishing lines of communication for the response team members [5].

When considering incident response preparation for an ICS network, the defense in depth strategy is not always possible for a response team who in most cases will only interact with a system once an incident has taken place. The time-sensitive responses for unfamiliar environments necessitates that the preparation phase be largely focused on the acquisition of generally applicable knowledge about ICS systems that can be used in a variety of environments.

Necessary Skills for Preparation.

- **Risk and Recovery Prioritization:** The ability to prioritize which components pose the biggest security threat for the system as a whole, determining which system components should be addressed.
- **Attack Vector Assessment:** The skill of understanding the attack path that an intruder might take when attempting to compromise the system. A responder must be able to understand how an attacker can gain access and the techniques used to manipulate and pivot through the network.
- **Communication with Asset Owner:** Communicating with the asset owner and employees is essential to gain an understanding of the operation of the system. This allows the responders to gain insight on the possible scope of

damage, network layouts and the limitations of the response effort. The skill enables the execution of all other skills for the preparation phase.

- **Competency with Control System Components:** A responder must understand the purpose of the components within a process and how the components (e.g., engineering software, control hardware and control interfaces) operate to gain insight on any irregularities, malfunctions or manipulations.

Components Necessary for Assessing Preparation Skills.

- **System Familiarization Components:** Examples of system familiarization components include descriptions of the devices that guide the risk and recovery prioritization of the response plan and network maps that assess a participant's ability to understand the role of operational technology within the industrial network.
- **Control Hardware:** Physical industrial control devices that facilitate the operation of a physical process. Examples include a PLC or a remote terminal unit (RTU).
- **Engineering Software:** The software used for programming and remote control of industrial control hardware. This software is often proprietary and provided by a vendor who develops control hardware.
- **Human Machine Interface (HMI) Software:** Software that allows the monitoring and control a physical process. This software typically lets a human interact with current tasks or analyze the operational status of a process.
- **Real Control Process:** A realistic process is one that is typically used in an industrial setting and provides a trainee with the opportunity to interact with

realistic physical process models. An example includes a wastewater cleansing process where large pieces of waste are extracted from wastewater using a large screening grate.

- **Varied Industrial Control Vendor Exposure:** The exposure of a trainee to multiple proprietary control technologies within a single training environment.

3.1.2 Detection and Analysis.

One of the most challenging parts of incident response is accurately detecting an attack and determining the scope of the problem [5]. This phase is complicated by the wide range of detection technologies that may be conflicting, inaccurate or incomplete. Asset owners may also have malfunctions that are not necessarily malicious. While an ICS response team is normally not the primary detector of the incident, a response team must be able to identify the potential signs of the problem and confirm through methods of detection and analysis that the problem is malicious. The team must have the capabilities to engage with all sources of incident indicators including intrusion detection systems (IDS), antivirus, security information and event management systems (SIEMs) and logging systems that are both network based and operating system based.

While traditional detection devices are valuable, an ICS specific team should be

Table 3. Preparation: Skills and components.

Skills	Components
Risk and Recovery Prioritization	System Familiarization Components
Attack Vector Assessment	Control Hardware
Communication with Asset Owner	Engineering Software
Competency with Control System Components	HMI Software
	Real Control Process
	Varied Industrial Control Vendor Exposure

able to apply their industrial control knowledge to detect any malicious effects that may be physically visible or viewable from control specific software. This adds a layer of complexity upon traditional IT intrusion detection. Spotting ambiguous ICS symptoms (e.g. slight configuration changes or altered control logic) can be challenging considering the many different types of proprietary technologies built into industrial systems. These skills and abilities rely heavily on the responders understanding of normal environment behavior.

Necessary Skills for Detection and Analysis.

- **Anomaly and Event Detection:** The ability to use software and hardware detection to pinpoint anomalies and events that impact the system operation.
- **Traffic Monitoring and Analysis:** The ability to monitor and filter traffic to track down malicious behavior in the network and the skill of analyzing and understanding the traffic that is being sent over the ICS network.
- **System Component Monitoring:** The ability to monitor control components and their logical execution to analyze functionality and detect abnormalities. This includes monitoring through software and physical means.
- **Log Analysis:** The forensic skill of analyzing logs on the network to trace an incident to its cause and track an attacker.

Assessing and honing the skills listed above requires a system that is capable of providing the assets that implement detection technologies, physical effects of malicious activity and realistic challenges of detection. The more realistic the traffic and detection systems are the more challenging the training can become. The size of the training system and amount of components can also expand the challenge of detecting malicious activity.

Training Components Necessary for Assessing Detection and Analysis Skills.

- **Physical Component Effects:** The physical operation of a control process (e.g., the actual pumping of water in a wastewater treatment plant environment).
- **System Logging:** The capability for the environment to log system interactions and industrial process data through the use of data historians or network logging tools. This enables a trainee to analyze and interact with industrial forensic data.
- **Realistic Industrial Network Traffic Generation:** The realism and variety of industrial network traffic in terms of real industrial protocols (e.g., Modbus, EtherNet IP or DeviceNet) providing a trainee with practical industrial protocol exposure for analysis and monitoring purposes.
- **Anomaly Detection Tools:** Software or hardware tools that allow the detection of anomalies on the industrial network through the analysis of network traffic or analysis of the physical process (e.g., Grassmarlin and Symantec Anomaly Detection for Industrial Control Systems).

Table 4. Detection and Analysis: Skills and components.

Skills	Components
Anomaly and Event Detection	Physical Component Effects
Traffic Monitoring and Analysis	System Logging
System Component Monitoring	Realistic Industrial Network Traffic Generation
Log Analysis	Anomaly Detection Tools

3.1.3 Containment, Eradication and Recovery Phase.

The phase of containment, eradication and recovery focuses of the responder's ability to choose and apply an appropriate strategy that includes quarantine, evidence handling, source identification, threat eradication and restoration [5].

Containment strategies include complete disconnection of the attacker or source of activity, sand boxing or network filtration. Any activity that mitigates the problem as a temporary solution to decrease the malicious activity and prevent further damage can be deemed containment. A strategy for containing the threat must take into consideration the possible consequences (e.g., internal damage, evidence preservation and solution duration) [5]. In an ICS specific environment, the strategy of disconnection can lead to catastrophic effects for the control process where components may depend upon each other for interoperability. The same problem can arise in an operational environment during an attempt to filter traffic. It is important for an ICS incident responder to understand system operations before making any isolation or containments decisions.

Evidence gathering and handling is used as a way to document the incident and for use in legal proceedings. Evidence should include any identifying information, information of personnel who handled and gathered the evidence, times and dates of occurrences contained in the evidence and evidence storage locations [5].

This phase should also focus on the identification of the attacking host, which is necessary to appropriately gather evidence, deploy containment strategies and eradicate the problem. Identification includes listing suspicious IP and MAC addresses, conducting research on the hosts and using continuous monitoring techniques on any suspicious host to confirm (or deny) the suspicions. It is crucial to identify all malicious hosts to properly eradicate the threat to the system.

If a control system component has been compromised and altered beyond repair,

a replacement may be necessary to ensure the integrity of the system. A software solution would be the identification and removal of any accounts that have been compromised or created in the incident or the removal of a malware implant in a system. To properly recover, a response team must be able to accurately assess the cause of the problem and apply fixes in the form of software patches, hardware replacement and ensure the removal of the access point. Once the threat has been completely eradicated and the system restored, a series of tests must be completed ensuring the return of the system to a normal state. Developing tests for different types of control systems can be challenging and requires the synthesis of system operation knowledge with the precise comprehension of the malicious effects.

Necessary Skills for Containment, Eradication and Recovery.

- **Evidence Gathering and Handling:** The skill of gathering and handling evidence in a way that does not compromise the investigation process.
- **Attacker Identification:** The ability to identify the source of the incident through forensic investigation.
- **Disconnection or Sandboxing of Attacker:** The ability to isolate the attack source from the network and ensure that no further damage can be done by the attacker.
- **Identify and Mitigate Exploited Vulnerabilities:** The ability to identify the vulnerabilities that were exploited in the attack and mitigate the security weaknesses.
- **Return of Systems to Operational State:** Rapidly returning the system to an operational state, mitigating any physical or financial losses and testing to ensure the system was restored properly.

To assess the ability of a responder to mitigate, document and eradicate an attack, a training environment must include elements that allow the trainee to conduct actions to stop an attack while also taking necessary actions to keep the system as functional as possible.

Training Components Necessary for Assessing Containment, Eradication and Recovery.

- **Physical Disconnect or Isolation Options:** The ability to physically disconnect portions of the system or the ability to isolate a portion of the system by some form of sandboxing.
- **Filtering Capabilities:** The ability to deploy filtering technology (e.g., firewall) in an effective manner within the system.
- **Real Malware or Attack Scripts:** Realistic attack scenarios or code that have genuine effects to the system allowing trainees to detect and defend against practical attacks.
- **Emergency Backup Operation Equipment:** The capability of deploying manual backup operations as a way to keep a critical process from completely failing. This allows a participant to prioritize system operation in necessary situations.
- **Acceptance Test Execution Capabilities:** The capability for a trainee to create and deploy acceptance tests for the system to ensure recovery.

3.1.4 Post-Incident Activity.

The post-incident activity phase is the synthesis of conclusions from gathered evidence.

Table 5. Containment, eradication and recovery: Skills and components.

Skills	Components
Evidence Gathering and Handling	Physical Disconnect or Isolation Options
Attacker Identification	Filtering Capabilities
Disconnection or Sandboxing of Attacker	Real Malware or Attack Scripts
Identify and Mitigate Exploited Vulnerabilities	Emergency Backup Operation Equipment
Return of Systems to Operational State	Acceptance Test Execution Capabilities

Necessary Skills for Post-Incident Activity.

- **Handling and Analysis of Malware:** The ability to understand the effect of malware and the proper way to handle it.
- **Incident Documentation:** The ability to synthesize conclusions from evidence and knowledge for the purpose of accurate documentation and response justification.

Training Components for Assessing Post-Incident Activity.

- **Documentation Standards:** A non-technology component to ensure the trainee understands the documentation process for their organization.
- **Malware Analysis Tools:** Analysis software that a trainee can use to dissect and analyze malware (e.g., IDA Pro, OllyDbg and WinDbg).

3.1.5 Training Administration for Environments.

The ability to ensure effective training and provide feedback for trainees is key for any training environment. While this is not part of the incident response lifecycle,

Table 6. Post-Incident activity: Skills and components.

Skills	Components
Handling and Analysis of Malware	Documentation Standards
Incident Documentation	Malware Analysis Tools

administration of training is paramount to the education of participants. The more detailed feedback that a trainer can provide, the more a trainee can learn from their experience in a training environment. To enable the trainer to effectively monitor activities in the training environment, certain range components are required that provide insight into the progression of the training activities. The following components are necessary for full monitoring capabilities within an environment.

Training Components Necessary for Training Administration.

- **Real Time View of Physical Signal Exchange:** The ability for an administrator to view the signal inputs and outputs of system endpoints (sensors and actuators). This is necessary to ensure the administrator is assessing an accurate representation of an environment even when the integrity of the system monitoring components within an environment has been comprised and are untrustworthy for assessment purposes [31].
- **Remote Administrative Monitoring:** The ability for an administering authority to assess trainees and control the exercise from a different physical location than the exercise environment.
- **Remote Participation:** The ability to administer exercises on the environment to participants located in physically separated locations.

Table 7. Training administration: skills and components.

Components
Real Time View of Physical Signal Exchange
Remote Administrative Monitoring
Remote Participation

3.2 Categorization Levels of ICS Training Environments

This section introduces levels of ICS training environments based on practical components and capabilities. While each level may have a varying range of capabilities, the primary delimiter of the levels is the practicality they provide in the context of a real industrial control system.

3.2.1 Level 0 Training Environments.

A Level 0 training environment is the presentation of information to a trainee through oral, written or graphical means. This type of environment is usually a classroom setting with lectures and presentation slides that provide a trainee with basic facts and explanation. This base of knowledge allows a trainee to understand the environment domain and provides meaning and context for the training. These training environments are easily coupled with higher level training environments in order to ensure a baseline of foundational knowledge before a trainee must apply their knowledge as skills.

3.2.2 Level 1 Training Environments.

A Level 1 training environment is completely software defined and operates within a program simulating an industrial controller or control system. This level of environment can provide simplified education and training for inexperienced trainees by defining controller operations and enabling familiarization with control process logic.

Existing examples of this environment include the LogixPro-500 PLC simulator and the Honeyd PLC interaction software [25, 29]. These environments provide no real physical interaction and are constructed of software defined capabilities. Basic interaction and programming can be implemented, however, there is no way to guarantee that these simulation programs will mimic the exact behavior of real control

hardware and software. For example, the Honeyd PLC simulation software supports 2000 TCP requests per second with 65,536 hosts compared to real programmable logic controllers which support significantly fewer connections [29]. These environments are also limited due to the inability to provide realistic defensive response interactions. Level 1 environments also do not allow for physical disconnect options that would be available to defenders in a real environment.

Fully simulated (Level 1) environments can provide many capabilities within high fidelity software but lacks real physical devices. Level 1 environments do not provide participant exposure to real control hardware, or vendor supplied engineering and HMI software. While a simulated environment can attempt to recreate engineering or HMI software, no actual communication can occur between vendor hardware and the software. Acquiring exposure to multiple industrial vendor hardware and software is not available in a strictly software defined environment.

3.2.3 Level 2 Training Environments.

A Level 2 training environment consists of an emulated control process that includes real physical effects but is not constructed of real vendor hardware and software. This level of environment can be constructed using embedded devices (e.g., Arduino, Raspberry Pi or BeagleBone) which can be programmed to control physical sensors and actuators using common programming languages (e.g., C and Python).

Level 2 training environments are used as training platforms and research testbeds for many organizations. Researchers at the Air Force Institute of Technology have created a Level 2 environment as a research testbed emulating a vehicle Controller Area Network (CAN) bus that is controlled by a BeagleBone Black (see Figure 1). The environment is used to test the effects of CAN bus attacks on vehicular control. The DHS ICS-CERT organization uses portable Level 2 training platforms to conduct

basic ICS classes and exercises. A company called Cybati has created Level 2 training kits called CybatiWorks which are Raspberry Pi control emulation devices that pair with low voltage sensors and actuators (such as stoplights represented as mounted LEDs) [6]. While Level 2 environments can show physical effects and emulate process control, the environments are still restricted to the code placed on the embedded devices and cannot be guaranteed to mirror the exact behavior of a real industrial controller.

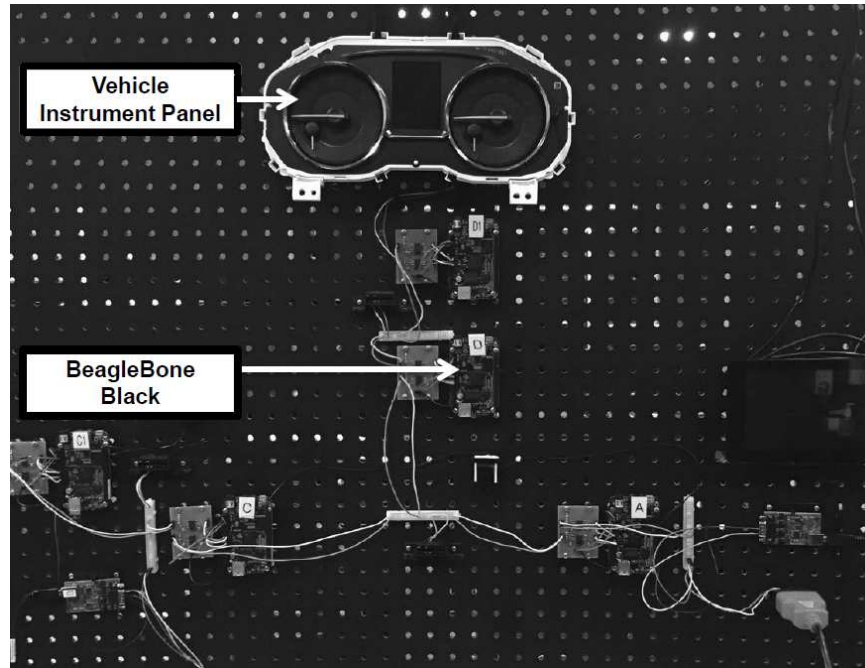


Figure 1. AFIT automotive CAN bus emulation testbed.

3.2.4 Level 3 Training Environments.

A Level 3 environment is constructed of real process control hardware and software controlling a partial industrial control system. Consider a wastewater treatment facility, an example of a Level 3 environment would be the hardware and software controlling the lift station portion of the wastewater treatment process. While these environments are not fully realistic, they do attempt to represent a scaled realism.

It allows for the familiarization of vendor equipment, industrial networks, realistic process logic and portability without the necessity of a full facility and costly maintenance.

Existing Level 3 environments are used for a variety of security related activities. Sandia SCADA Security Development Laboratory is used for testing and creating security practices, programs and protocols and would be considered a Level 3 environment [21]. Other examples include the Air Force Institute of Technology (AFIT) stoplight system controlled by Allen Bradley MicroLogix PLCs used to teach ICS defense classes (see Figure 2). The SANS Cybercity combines elements of a Level 2 range and a Level 3 to create a robust experience in a compact environment [23].

A Level 3 environment may be fully capable and have all exercise environment capabilities but it will still lack the realism of a full-scale system. It cannot supply an in-depth understanding of the physical cyber effects on a real world system where an incident or malfunction may have cascading effects do to the interconnection of the system processes [3].

3.2.5 Level 4 Training Environments.

A Level 4 training environment is a genuine industrial control system facility with functional processes. This would be an industrial control facility that is constructed exactly how it would be for its intended application and is used to conduct cyber exercises. Cyber exercises conducted in a real environment allows the trainee to have full immersion into the training scenarios where unpredictable cascading attack effects may require novel solutions.

Existing Level 4 training environments are located at the Atterbury-Muscatatuck Urban Training Center (MUTC) near Butlerville, Indiana. This training compound is owned and operated by the Indiana National Guard and is used for military and first

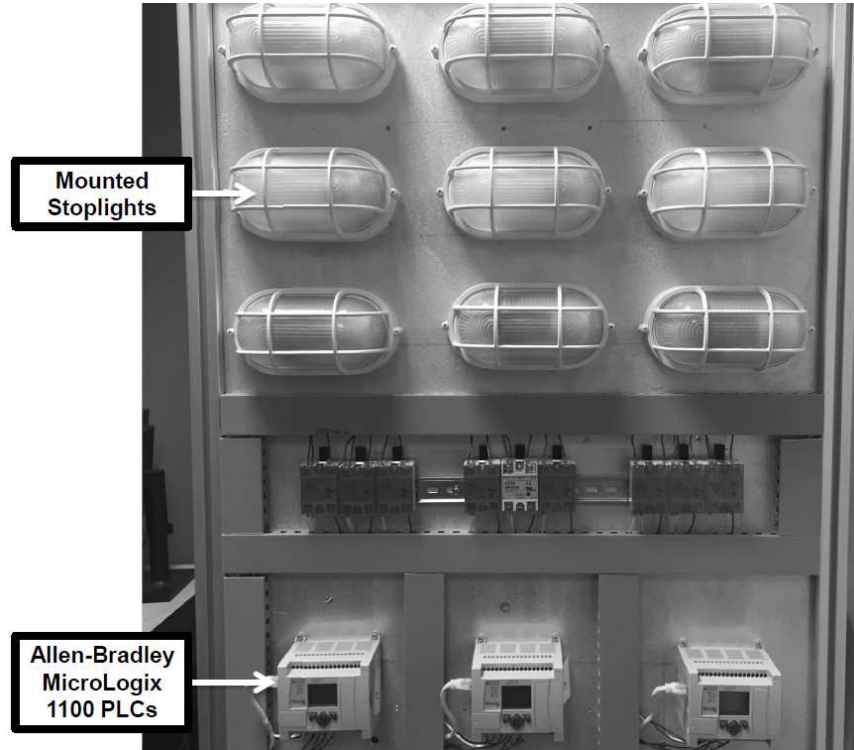


Figure 2. Mounted stoplight control system used by AFIT for ICS security labs.

response training and consists of many industrial facilities that make up many separate Level 4 environments. The facilities include a power plant, oil refinery, prison, hospital, subway station, power distribution and a wastewater treatment facility [13]. It should be noted that the cyber portion of the MUTC is somewhat lacking, however plans are being implemented to correct the deficiencies.

3.3 Mapping of Training Environment Levels to Bloom's Taxonomy

As the level of training environment increases so does the level of cognitive complexity and thinking that can be assessed in the environment. The complexity of training scenarios that can be presented in an ICS environment depends on the amount of realism which can provide real interaction and observations. Bloom's taxonomy can be mapped to the levels of ICS training environments in terms of training ICS defenders. The taxonomy is hierarchical in nature, therefore, an environment that

can administer exercises at higher levels of thinking can also assess all lesser levels in the taxonomy.

A Level 0 training environment is capable of presenting exercises and problems that address the first two cognitive levels of Bloom’s Taxonomy, specifically “Remembering” and “Understanding”. This level of training environment allows the trainee to construct meaning from the written, oral and graphical information being presented. This level alone does not allow the trainee to apply their knowledge through any kind of procedure or implementation.

A Level 1 fully simulated ICS training environment is capable of presenting exercises and problems that address the first two cognitive levels of Bloom’s Taxonomy. With a fully simulated training environment constrained to a program, you can test a trainee’s ability to recall and retrieve facts that have been programmed into the simulation, and the trainees can begin to interpret meaning from the lessons and make references and comparisons from the presented facts. Simulations that allow for simple PLC interaction can teach a trainee about traditional PLC behavior and basic concepts. However, there is no guarantee that the control behavior from a simulation will carry over into a real system due to the scripted limitations of software. This constrains Level 1 environments from assessing higher levels of Bloom’s Taxonomy. Another constraint for a Level 1 environment is the limited ability for an ICS defender to implement realistic security measures. Since a simulated environment is only capable of doing what it is programmed to do, a trainee cannot manipulate a network or change controller implementation to unanticipated configurations.

A Level 2 training environment, where emulated devices perform physical controller functions, provides a platform to assess the “Applying” and “Analyzing” levels of Bloom’s Taxonomy. With this training environment, a trainee can implement external defenses (e.g., firewalls and network isolation), dissect the environment, dif-

ferentiate components and attribute control to solve problems, however, a Level 2 environment cannot evaluate a trainee to Bloom’s “Evaluating” level because the criteria and standards that a trainee would need to base their judgments and critiques on would not be realistic enough.

A Level 3 environment is implemented with vendor supplied industrial hardware and software, but it is not a fully comprehensive environment. This level of environment is capable of administering exercises and training up to the “Evaluating” level of Bloom’s Taxonomy. This level of thinking is defined by making judgments and critiques based on criteria and standards. Evaluative thinking in an environment of this level can be done by comparing data and observations with standard operation criteria of the control component. This real data allows the participants of an exercise to perform realistic defensive evaluations on an ICS environment. A Level 3 environment struggles to assess trainees at the highest level of Bloom’s Taxonomy.

A Level 4 environment can assess and provide the highest level of thinking in Bloom’s Taxonomy. This level of thinking is defined by the ability to form a comprehensive view of a situation. In the context of ICS defense and incident response, this type of cognitive complexity cannot be achieved without a fully functional system. This environment can provide a trainee the opportunity to view and manipulate every possible element in an actual industrial environment. New and applicable solutions to defense problems can be applied to functional industrial systems. A Level 4 environment can help ICS defense trainees discover, correct and address real world problems with their creativity and see the ramifications (good or bad) of their actions (e.g., cascading effects).

3.4 Construction of ICS Training Environments

This section presents training environment examples for each of the five levels.

Table 8. Mapping ICS training environment levels to Bloom’s Taxonomy.

Bloom’s Taxonomy	Training Environment Levels				
	Level 0	Level 1	Level 2	Level 3	Level 4
1. Remembering	X	X	X	X	X
2. Understanding	X	X	X	X	X
3. Applying	-	-	X	X	X
4. Analyzing	-	-	X	X	X
5. Evaluating	-	-	-	X	X
6. Creating	-	-	-	-	X

3.4.1 Construction of Level 0 ICS Training Environments.

The construction of a Level 0 training environment consists of gathering and presenting information to a trainee. This environment contains no relevant hardware interaction and assessment in this environment can be presented as a written or oral exam. Interaction within this training environment would be note taking and posing questions to solidify topical understanding.

3.4.2 Construction of Level 1 ICS Training Environments.

The LogixPro-500 PLC simulator allows a trainee to provide or manipulate logic and displays the execution of the logic with simulated sensors and actuators. Using training scenarios in this simulated environment allows the assessment up to the “Understanding” level of Bloom’s Taxonomy by allowing the trainee to learn facts about control system operation and construct visual meaning and interpretation through the execution of the simulation. This training software environment is free to download [25].

Level 1 Scenario: Ladder Logic engineering.

- *Objective:* Given engineering specifications to control a garage door with open and closed sensors for an industrial control environment, provide logic that

allows the control hardware to execute the specifications.

- *Description:* Create Ladder Logic that allows the PLC to control a garage door with sensors that indicate when the door is opened or closed. The simulation should run the logic provided by the trainee and visualize the physical response with a garage door simulator.
- *Type:* Understanding the functionality and relationship between the control hardware and logic software.
- *Evaluation Criteria:* Correctly engineer the specified functionality within three hours.
- *Reference:* The provided engineering specifications for the garage door logic, the LogixPro-500 software help menu and the Rockwell Automation RSLogix user guide.

Below is a breakdown of the components that are represented in the LogixPro-500 simulation software training environment. The hardware and software needed is a single computer system with the LogixPro-500 simulation software installed.

Level 1 LogixPro-500 training environment components.

- **Control Hardware:** The control hardware is represented in this environment as a simulated PLC that executes the logic provided by a trainee.
- **Engineering Software:** The engineering software used in the simulation is a version of the Allen-Bradley RSLogix500 programming tool.
- **HMI Software:** The HMI software built into the simulation is an interactive visual representation of the sensors and actuators. This HMI displays how the sensors and actuators react to the logic from the simulated control hardware.

- **Physical Component Effects:** The physical effects of the system are represented on a computer screen.

3.4.3 Construction of Level 2 ICS Training Environments.

Raspberry Pi emulation devices can be programmed to emulate a network of stoplights using LEDs. Consider a scenario where a controller is attacked interfering with the timing of the lights. The trainee would have to determine the relationship between the components to interpret the cause of the incident. This scenario would assess the “Analyzing” level of Bloom’s Taxonomy. The cost of this environment with four stoplights is approximately \$200.

Level 2 Scenario: Stoplight Logic Manipulation.

- *Objective:* Given a network of emulated stoplights that are currently out of sync, return the lights to normal functionality and find the manipulated device.
- *Description:* Use network traffic monitoring and software analysis to understand which devices were impacted and find the source of the attack.
- *Type:* Understanding the functionality and relationship between the control hardware and logic software and applying response skills to mitigate the effects of the attack and return the system to normal functionality.
- *Evaluation Criteria:* Return the system to normal functionality within three hours and find the source of the attack within one hour.
- *Reference:* Network and system logs and the code on the Raspberry Pi devices.

The following is a breakdown of the components needed for this training scenario. The inherent hardware required is a Raspberry Pi development platform with the logic to control LED lights.

Level 2 Stoplight Network Training Environment Components.

- **Control Hardware:** The control hardware is represented in this environment by a network of Raspberry Pi emulation controllers.
- **HMI Software:** The HMI software built for this environment would have to be programmed to view and communicate with the Raspberry Pis.
- **Physical Component Effects:** The physical effects of the system are represented by LEDs.
- **System Logging:** Logging can be implemented into this network through the Raspberry Pi platforms and passive network monitoring software (e.g., Grassmarlin).
- **Physical Disconnect or Isolation Options:** The physical separation of controller devices allows for both network segmentation and physical network disconnection.
- **Filtering Capabilities:** Firewall filtering capabilities are built into the scenario to isolate the ICS network for any outside connections.

3.4.4 Construction of Level 3 ICS Training Environments.

Two examples of Level 3 environments are offered. The environments model a wastewater treatment plant and a prison facility. Components for each environment fit within a Pelican 1610 case (62.76 cm x 49.73 cm x 30.3 cm). The environments were created to be as cost effective as possible while including genuine control hardware devices. These scenarios assess thinking skills up to the “Evaluating” level of Bloom’s Taxonomy. Note that the descriptions of these environments are more detailed than

the other levels to show that high levels of interaction with genuine industrial control components can be achieved for a relatively low cost while maintaining high portability.

3.4.4.1 Wastewater Treatment Facility Environment.

The Level 3 wastewater treatment plant environment models the process of a wastewater aeration basin. An Allen-Bradley ControlLogix PLC controls the diffusion of oxygen into two chambers ensuring that the level of oxygen within the chambers meets the supplied specifications. If the oxygen levels become too high or low, an alarm will trigger which is displayed by two mounted red lights inside of the exercise environment. The oxygen fluctuates based on the speed of the blower fans and the dilation of oxygen valves. This closed control loop of oxygen level maintenance is controlled by the Allen-Bradley PLC and an Allen-Bradley PowerFlex 40 AC variable frequency drive (VFD). The PLC controls the valve dilation and computes the oxygen levels while the VFD dictates the fan speed based on the PLC's calculations. The equipment used to create this environment cost approximately \$16,500. Consider a scenario where a wastewater treatment plant experiences a cyber attack causing a PLC in the aeration basin to malfunction resulting in fluctuating oxygen levels.

Level 3 Example Scenario: Wastewater Treatment Aeration Basin Failure.

- *Objective:* Restore the aeration basin to full functionality and find the source and cause of the incident.
- *Description:* Using network traffic monitoring, HMI monitoring and visual monitoring the trainee must understand when the system fails and successfully recover the system by blocking access from the attacker and implement emergency

procedures to restore the failed control hardware to normal functionality.

- *Type:* Loss of system control and functionality.
- *Evaluation Criteria:* Return the system to normal functionality within three hours and find the source of the attack within one hour.
- *Reference:* ControlLogix PLC manual, PowerFlex 40 AC VFD manual, vulnerability reports of the control hardware and a map of the control network.

Level 3 Wastewater Treatment Facility Training Environment Components.

- **Control Hardware:** An Allen-Bradley ControlLogix PLC and an Allen-Bradley PowerFlex 40 AC VFD were used as the process control hardware for this environment (see Figure 3).
- **Engineering Software:** The engineering software used for programming and remotely controlling the Allen-Bradley PLC is RSLogix 5000 (see Figure 4).
- **Real Control Process:** The control process for the environment was modeled from a wastewater aeration basin and controls two zones of oxygen diffusion using a blower and two valves.
- **Physical Component Effects:** The speed of the blower fans are controlled by the VFD and the oxygen valves are controlled and effected by the PLC.
- **Realistic Industrial Network Traffic Generation:** The network traffic between the HMI workstation, engineering workstation, VFD and PLC is visible on the network and constitutes realistic ICS traffic using real industrial protocol communication including Ethernet/IP and Common Industrial Protocol (CIP).

- **Physical Disconnect or Isolation Options:** All machines on the network can be physically disconnected from their Ethernet ports and network isolation can be achieved through whitelist (and blacklist) capabilities of a Netgear ProSAFE network switch.
- **Filtering Capabilities:** Using an Ubiquiti EdgeRouterX router filtering can be done using simple firewall rules for any network connection in the environment.
- **Real Malware or Attack Scripts:** The PLC is vulnerable to a variety of attacks over the network making this model of PLC a realistic prospect of incident response.
- **Real Time View of Physical Signal Exchange:** Using Y-Box technology (for a detailed description of the Y-Box see [30]), an exercise administrator can view the operation of the process control endpoints and HMI to track an ongoing attack (see Figure 5).
- **Remote Administrative Monitoring:** The White Cell interface for the environment can be viewed using virtual network computing technology so that an exercise administrator can view the display of the status of the effected software and physical hardware.
- **Remote Participation:** Remote participation can be completed using remote access tools, however the capabilities of physical manipulation will be hindered.
- **Varied Industrial Control Vendor Exposure:** There is varied exposure to industrial components for trainees in this environment due to the use of a PLC component and a VFD component.

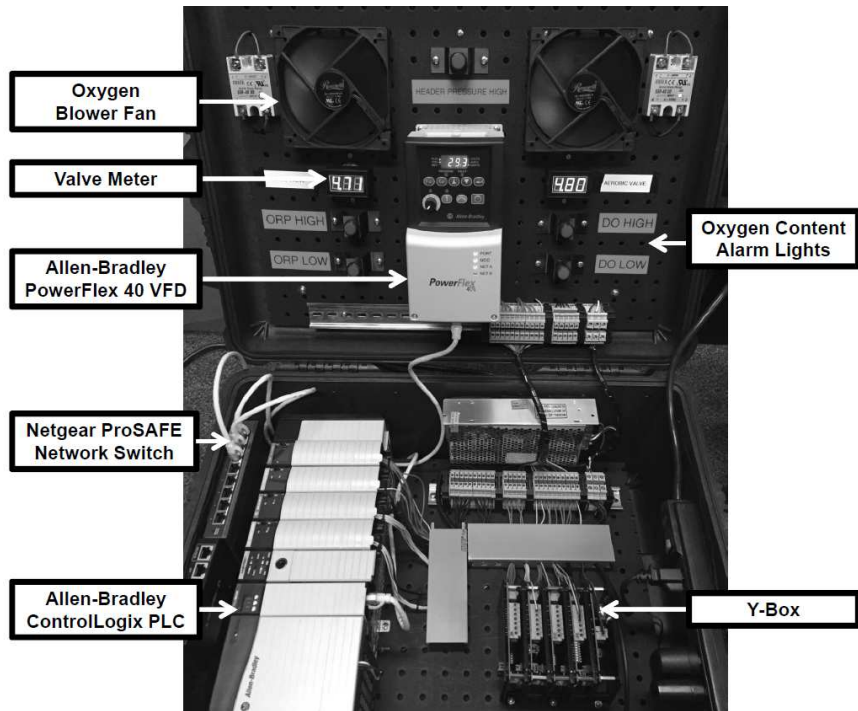


Figure 3. Level 3 wastewater treatment training environment.

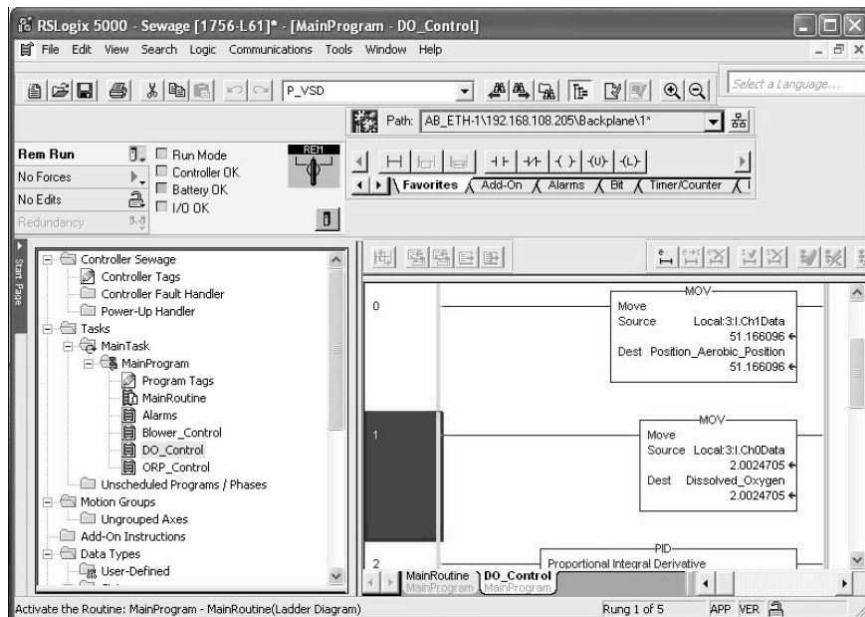


Figure 4. RSLogix 5000 engineering workstation software for the wastewater environment.

3.4.4.2 Prison Facility Environment.

The Level 3 prison facility training environment was modeled from a prison cell block containing three prison cells with door lock controls and a mantrap access

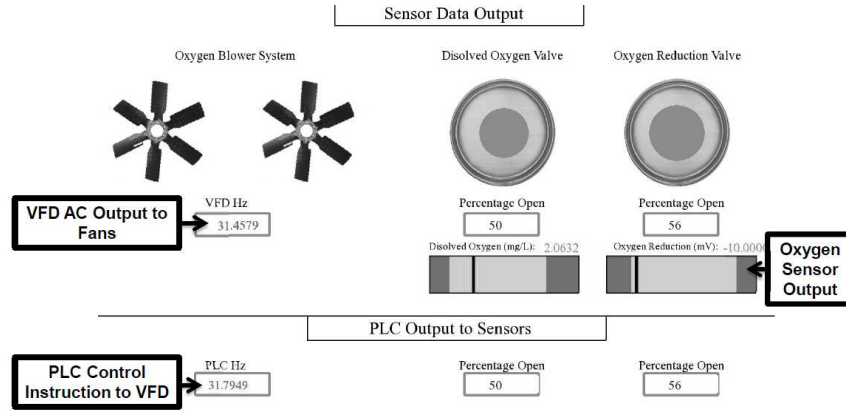


Figure 5. Administrative interface for a real time view of attacking and defending effects on the wastewater environment.

control system. An Omron PLC controls the function of the buttons, locks, security lights and alarm. This equipment cost an estimated \$1,400. Consider a scenario where a prison facility experiences a cyber attack that causes a PLC to malfunction opening prison door locks.

Level 3 Example Scenario: Prison Control System Failure.

- *Objective:* Restore the prison to full functionality and find the source and cause of the incident.
- *Description:* Using network traffic monitoring, HMI monitoring and visual monitoring the trainee must understand when the system fails and successfully recover the system by blocking access from the attacker and implement emergency procedures to restore the failed control hardware to normal functionality.
- *Type:* Loss of system control and functionality.
- *Evaluation Criteria:* Return the system to normal functionality within three hours and find the source of the attack within one hour.
- *Reference:* Omron CP1L PLC manual, vulnerability reports of the control hardware and a map of the control network.

Level 3 Prison Facility Environment Components.

- **Control Hardware:** An Omron CP1L PLC controls the function of the prison locks, buttons, lights and alarm (see Figure 6).
- **Engineering Software:** The engineering software component of the environment is the Omron CX-Programmer.
- **HMI Software:** The HMI to control the prison components (see Figure 7) was created using the open source Schneider Electric IGSS Free50 software.
- **Real Control Process:** The control process for the environment was modeled from the cell block of a prison located in the northwest United States.
- **Physical Component Effects:** The physical effects of the locks and lights can be operated through the HMI controls and by physically pressing the lock control buttons in the case.
- **Realistic Industrial Network Traffic Generation:** The network traffic between the HMI workstation, engineering workstation and PLC is visible on the network and constitutes realistic ICS traffic using real industrial protocol communication including Ethernet/IP, CIP and the proprietary Omron protocol.
- **Physical Disconnect or Isolation Options:** All machines on the network can be physically disconnected from their Ethernet ports.
- **Filtering Capabilities:** Using an Ubiquiti EdgeRouterX router filtering can be done using simple firewall rules for any network connection in the environment.
- **Real Malware or Attack Scripts:** The PLC is vulnerable to a variety of attacks over the network making this model of PLC a useful tool for training.

- **Real Time View of Physical Signal Exchange:** Using Y-Box technology, an exercise administrator can view the operation of the process control endpoints and the HMI to track an ongoing attack or to view real time effects of an attack (see Figure 8).
- **Remote Administrative Monitoring:** The White Cell interface for the environment can be viewed using VNC technology.
- **Remote Participation:** Remote participation can be completed using remote access tools, however the capabilities of physical manipulation would be hindered.

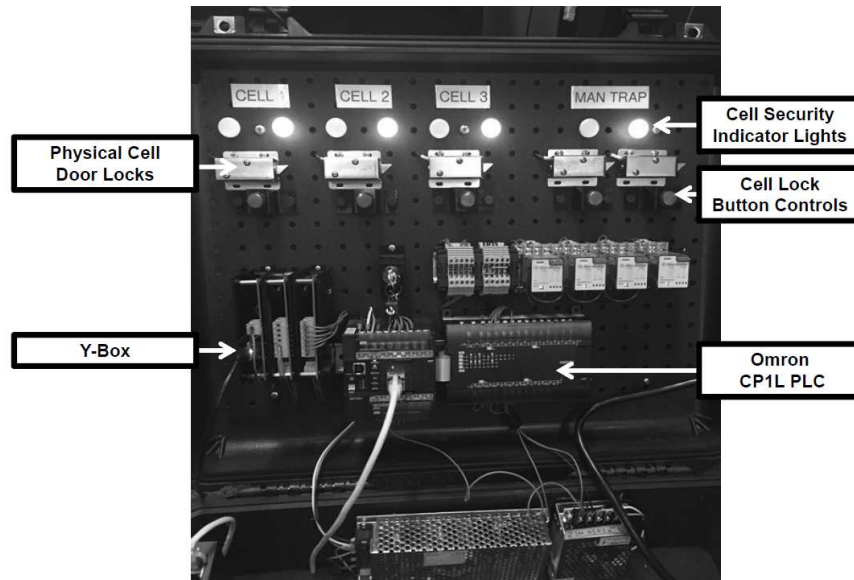


Figure 6. Level 3 prison training environment.

3.4.4.3 Power Substation Environment.

A power substation level 3 environment was constructed using a feeder protection relay and a power meter. These components have power that flows through them and cuts the power supply if the voltage or current becomes too high based on a designated limit. The power flows to three logically separated outlets where the

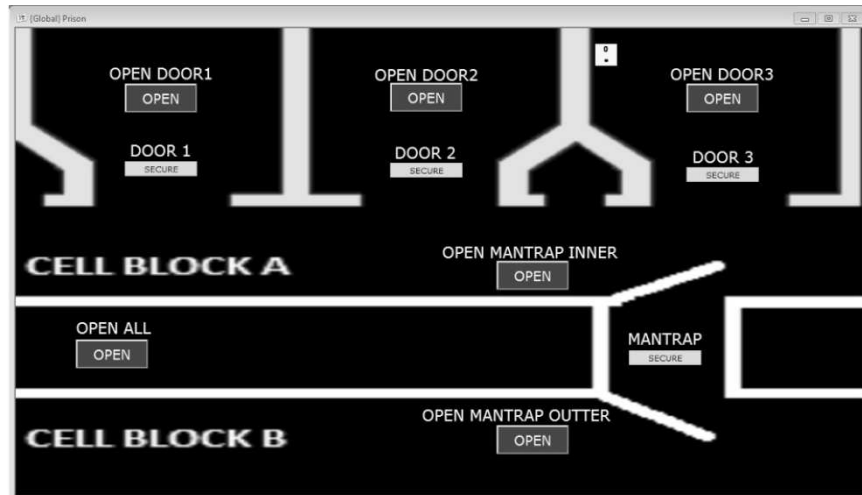


Figure 7. HMI for the prison training environment.

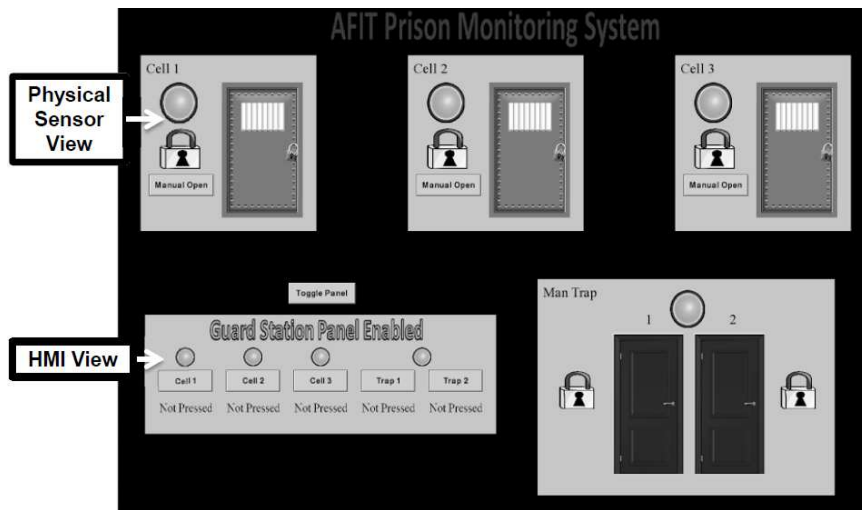


Figure 8. Administrative monitoring view of the prison exercise environment.

voltage on each outlet is displayed by voltage meters and an indicator light shows green for power or red if the power has been cut to the outlet. This environment can be used to showcase attack and defensive measures on power monitoring devices currently deployed as part of the national infrastructure. An example scenario that can be implemented into this environment would be the defense against and attack that alters the measurement of voltage and causes the relay to cut power supply to the outlets. This environment is valued at an estimated \$16,500 due primarily to the cost of the SEL power relay and Schnieder power meter.

Level 3 Example Scenario: Power Substation System Failure.

- *Objective:* Restore the power output to the outlets in the substation.
- *Description:* Using network traffic monitoring, HMI monitoring and visual monitoring the trainee must understand when the system fails and successfully recover the system by blocking access from the attacker and implement emergency procedures to restore the failed control hardware to normal functionality.
- *Type:* Loss of system control and functionality.
- *Evaluation Criteria:* Return the system to normal functionality within three hours and find the source of the attack within one hour.
- *Reference:* SEL-751A Feeder Protection Relay manual, Schneider Electric PowerLogic PM5300 Power and Energy Meter manual, vulnerability reports of the control hardware and a map of the control network.

Level 3 Power Substation Environment Components.

- **Control Hardware:** An SEL-751A Feeder Protection Relay and a Schneider Electric PowerLogic PM5300 Power and Energy Meter were used as the real vendor supplied control hardware for this training environment (Figure 9).
- **Engineering Software:** The Schneider Electric PowerLogic PM5300 uses ION Setup 3.0 as the configuration software. The Engineering software for the SEL-751A Feeder Protection Relay is the SEL-5030 AcSELeator QuickSet Software.
- **HMI Software:** The PowerLogic PM5300 uses a screen interface that is built into the front of the device. The SEL-751A Feeder Protection Relay uses the SEL-5030 AcSELeator QuickSet Software which has a built-in HMI view for the relay.

- **Physical Component Effects:** The power outlets in the environment are functional with power flowing through the feeder relay and the power meter to ensure the voltage and amperage stay within a specified range. If the range is surpassed, power will be cut to the outlets providing a real physical effect in the environment
- **Realistic Industrial Network Traffic Generation:** The network traffic between the HMI workstation, power relay and power meter provide realistic industrial network traffic with Modbus and DNP3 protocol capabilities.
- **Physical Disconnect or Isolation Options:** All machines on the network can be physically disconnected from their Ethernet ports and network isolation can be achieved through whitelist and blacklist capabilities of a Netgear ProSAFE network switch.
- **Filtering Capabilities:** Using an Ubiquiti EdgeRouterX router filtering can be done using simple firewall rules for any network connection in the environment.
- **Varied Industrial Control Vendor Exposure:** There is varied exposure to industrial components for trainees in this environment due to the use of a SEL Feeder Protection Relay component and a Schneider Electric Power Meter component.

3.4.5 Construction of Level 4 ICS Training Environments.

A Level 4 training environment corresponds to the “Creating” level of Bloom’s Taxonomy by presenting a trainee with a fully realistic scenario that enables the trainee to use all of their skills and knowledge to produce new solutions to unique and complex problems. Consider a power distribution plant, a scenario for a Level 4

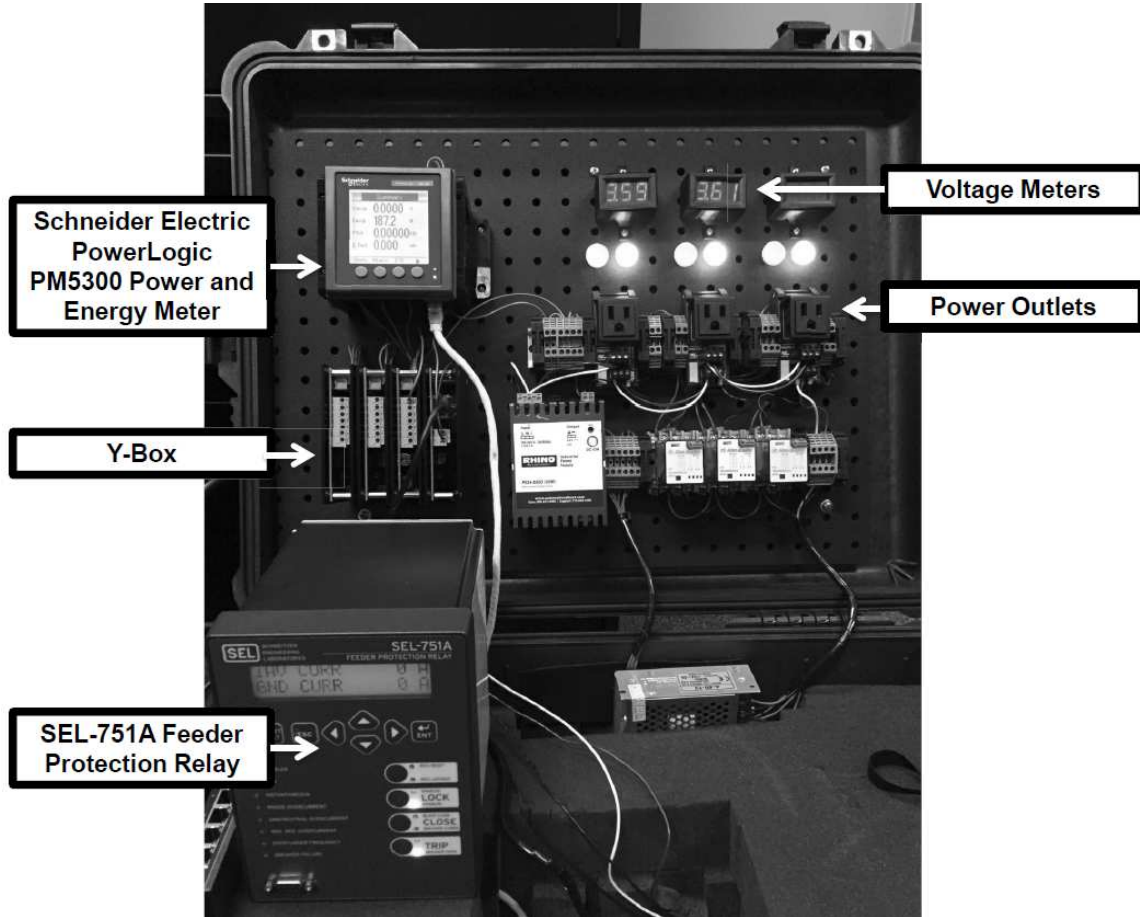


Figure 9. Level 3 power substation training environment.

environment would be the appearance of unusual traffic combined with unexplained fluctuations of power. Given the complexity of the environment with many components and connections, a trainee would have to be able to appropriately plan their response by prioritizing components on the network, narrow down the root cause of the malicious behavior and apply fixes to stop the incident and fully recover the system. With many signs of the incident resulting from a cascading effect within a real system, it can be difficult to determine the root cause of the incident and craft an appropriate response. With most industrial systems being uniquely constructed environments, the response of the trainee must be tailored to the environment. The components to construct a Level 4 power plant environment would be similar to that of an existing functional plant, however, additional engineering would be necessary to

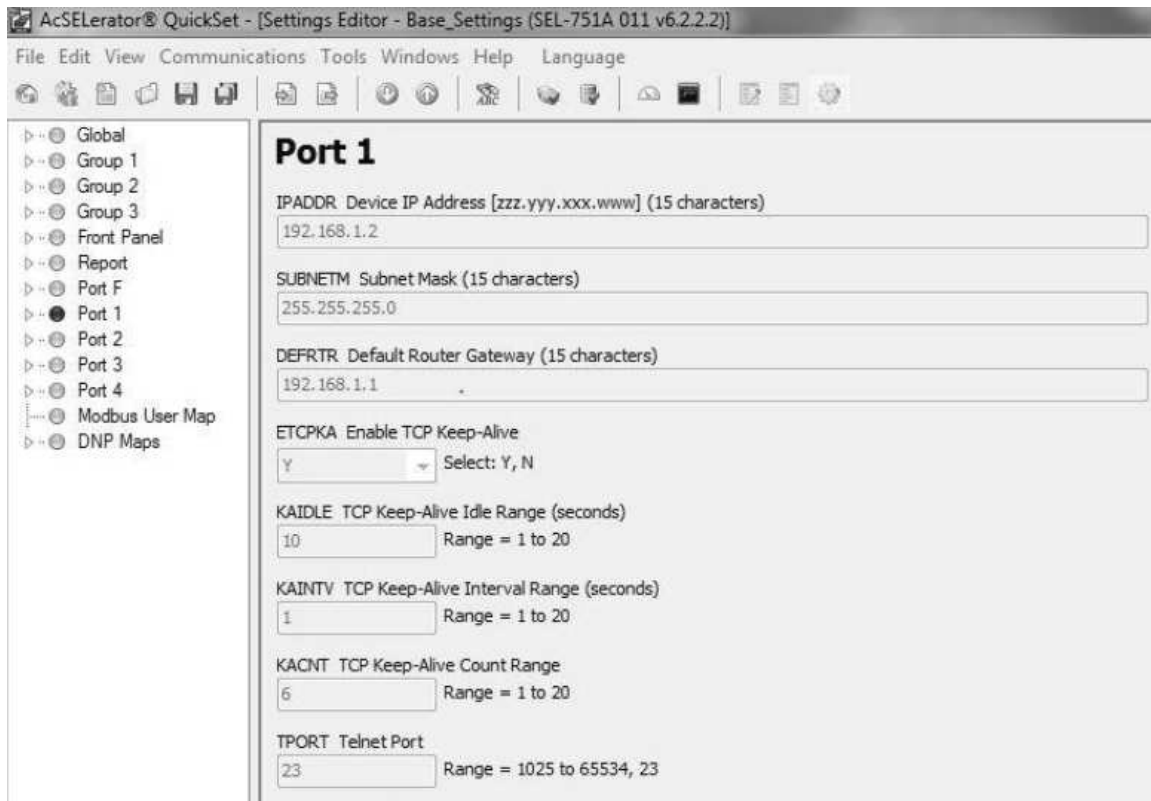


Figure 10. Power substation training environment programming software.

facilitate exercise control and monitoring. A Level 4 training environment may not be suitable to train beginners due to the risk of facility damage if an exercise does not go as intended. Instead, Level 1 and Level 2 environments can provide safe sandboxes for beginners to make mistakes and learn basic practices. Failsafe plans should also be considered when engineering a Level 4 environment for unpredictable situations during an exercise that could lead to facility damage. The cost of constructing a Level 4 power plant can vary widely from millions of dollars to over a billion. This Level 4 training environment is certainly not a mobile training platform, however, remote access may be possible.

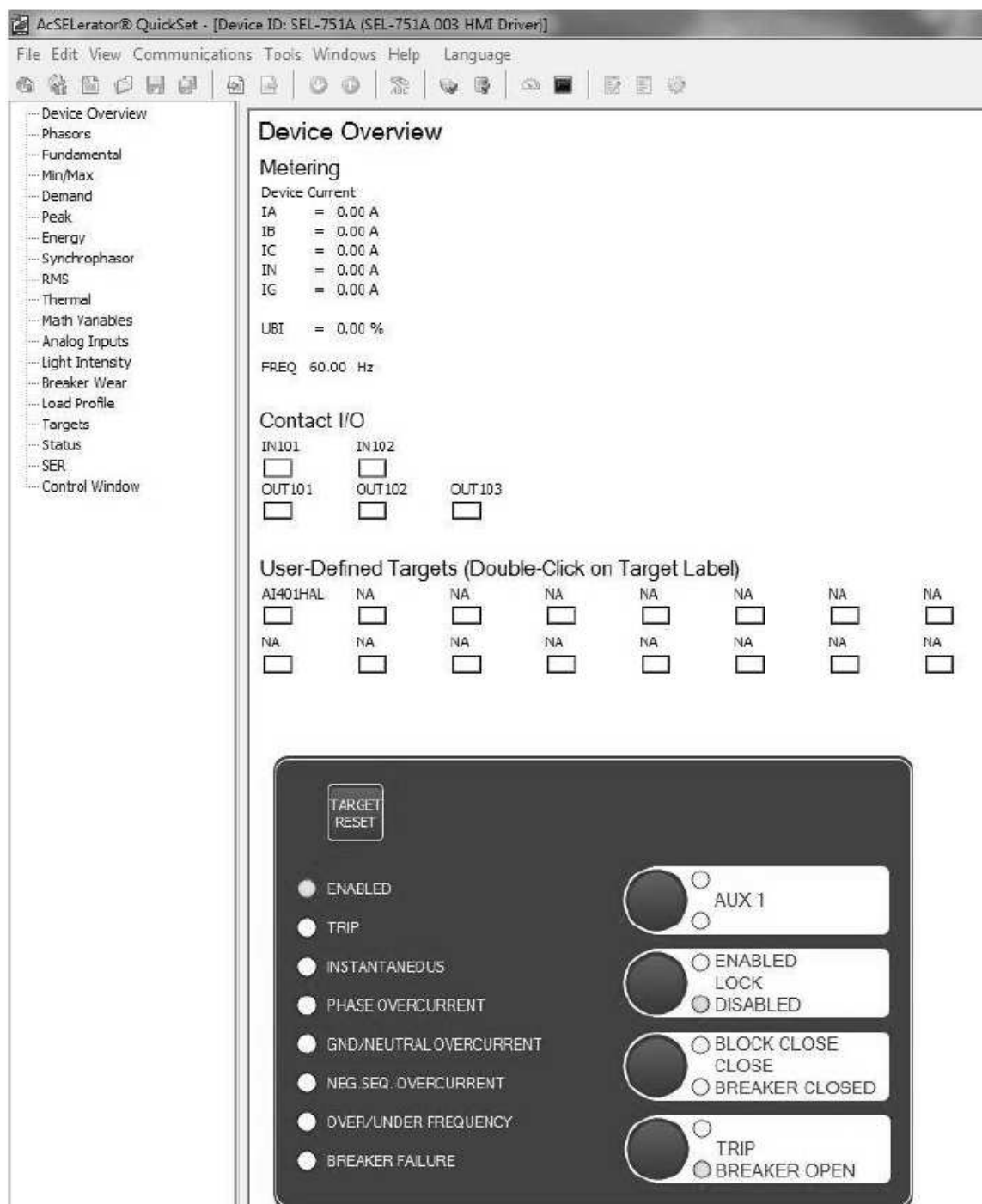


Figure 11. Power substation training environment monitoring HMI software.

IV. Conclusions

This chapter summarizes conclusions of the previous chapters.

4.1 Conclusions of Research

This thesis presents five categories of training environments that leverage Bloom's Taxonomy to address various levels of cognitive complexity for ICS cyber first responders. A categorization of training environments that provide complexity progression is necessary for ensuring the readiness and education of first responders. The proposed categories help determine which training environment best aligns with an organization's training goals. Examples of environments for each category level are presented with exercise scenario examples. Three Level 3 environments were constructed and deployed for training purposes to further solidify the proposed example concepts.

Level 1 environments may be appropriate for the average plant operator, while a Level 4 environment may be necessary for ICS cyber first responders that are called to handle a power plant incident. This will help prevent future incidences of improper response like the West, Texas fertilizer plant fire that could potentially involve a loss of life.

4.1.1 Significance of Research.

The successful construction and application of Level 3 environments confirms the hypothesis of attributing skill assessment capabilities to hardware and software components of training environments. The categorization of environments and component framework provide organizations the ability to understand levels of realistic fidelity within training environments and construct environments that target desired skill sets.

4.1.2 Recommendations for Future Research.

The recommendations are insights gained from the experience of this research.

4.1.2.1 Establishment of Environment Certification Authority.

This research presented in this thesis has served as a first step towards official regulation and certification of ICS training environments for incident responders. The creation of a governing authority that has the ability to certify environments as a designated level will allow organizations to train their incident responders to a certified level of proficiency. In the same way that organizations like the EASA certify flight simulators in order to guarantee a quality level of training for the pilots, research for the creation of a similar organization can provide a solidified categorization and provide the basis for trainee certification.

4.1.2.2 Categorization of ICS First Response Curriculum.

While this thesis focuses on the categorization of training environments, a strong curriculum is also necessary to ensure adequate training and education while leveraging the environments. A standardization of ICS cyber incident response training curriculum is the next step to establishing standardized and certified classes in this domain.

4.1.2.3 Establishment of Common Scenarios for Evaluation.

Exercise evaluation scenarios enable the assessment of trainees and standardization of assessment which guarantees a minimum level of proficiency demands that common elements of exercise scenarios be established in order to assess a baseline of knowledge and capabilities. Future research can establish a framework to create common scenarios and assessment practices in the ICS incident response domain.

Bibliography

1. C. Baltos, Soldiers Prepare on Tank Simulators for Saber Guardian Exercise, U.S. Army, Washington, DC (www.army.mil/article/171603/soldiers_prepare_on_tank_simulators_for_saber_guardian_exercise), 2016.
2. D. Beckstrom, Tank Gunnery Simulator: Getting Back to Basics, U.S. Army, Washington, DC (www.army.mil/article/143185/Tank_Gunnery_Simulator_Getting_Back_to_Basics), 2015.
3. J. Butts and M. Glover, How industrial control system security training is falling short, in *Critical Infrastructure Protection X*, M. Rice and S. Sheno (Eds.), Springer, Heidelberg, Germany, pp. 135–149, 2015.
4. H. Chang, Simulators Always Valuable in Military Training, U.S. Army, Washington, DC (www.army.mil/article/19599/simulators-always-valuable-in-military-training), 2009.
5. P. Cichonski, T. Millar, T. Grance and K. Scarfone, Computer Security Incident Handling Guide, NIST Special Publication 800-61 (Revision 2), National Institute of Standards and Technology, Gaithersburg, Maryland, 2011.
6. Cybati, CybatiWorks Homepage, Bloomington, Illinois (cybati.org/cybatiworks-one), 2015.
7. European Aviation Safety Agency, Certification Specifications for Helicopter Flight Simulation Training Devices, Annex to ED Decision 2012/011/R, Cologne, Germany, 2012.
8. European Helicopter Safety Team Component of ESSI, Teaching and Testing in Flight Simulation Training Devices (FSTD), For Helicopter Pilots, Instructors

- and Examiners (Training Leaflet), Cologne, Germany (essi.easa.europa.eu/ehest/wp-content/uploads/2015/12/206904_EASA_EHEST_HE_10.pdf), 2015.
9. M. Fabro and E. Cornelius, Recommended Practice: Creating Cyber Forensics Plans for Control Systems, DHS National Cyber Security Division, Control Systems Security Program, U.S. Department of Homeland Security, Washington, DC, 2008.
 10. M. Fernandez, Fire that left 15 dead at Texas fertilizer plant is ruled intentional, *New York Times*, May 11, 2016.
 11. M. Forehand, Bloom’s Taxonomy, From Emerging Perspectives on Learning, Teaching, and Technology, University of Georgia, Athens, Georgia (epltt.coe.uga.edu/index.php?title=Bloom%27s_Taxonomy), 2005.
 12. Idaho National Laboratory, INL Cyber Security Research: Defending the Network Against Hackers, Idaho Falls, Idaho (www.inl.gov/research/inl-cyber-security-research), 2014.
 13. Indiana National Guard, Atterbury/Muscatatuck MUTC Overview, Indianapolis, Indiana (www.atterburymuscatatuck.in.ng.mil/Ranges/MuscatatuckUrbanTrainingCenter/MUTCOverview.aspx), 2016.
 14. Industrial Control Systems – Cyber Emergency Response Team, Training Available Through ICS-CERT, Industrial Control System Cyber Emergency Response Team, Idaho Falls, Idaho (www.ics-cert.us-cert.gov/Training-Available-Through-ICS-CERT), 2016.
 15. D. Krathwohl, A revision of Bloom’s Taxonomy: An overview, *Theory into Practice*, vol. 41(4), pp. 212–218, 2002.

16. T. Morris, A. Srivastava, B. Reaves, W. Gao, K. Pavurapu and R. Reddi, A control system testbed to validate critical infrastructure protection concepts, *International Journal of Critical Infrastructure Protection*, vol. 4(2), pp. 88–103, 2011.
17. National Aeronautics and Space Administration, Johnson Space Center: Training for Space: Astronaut Training and Mission Preparation, Washington, DC (www.nasa.gov/centers/johnson/pdf/160410main_space_training_fact_sheet.pdf), 2006.
18. National Training and Simulation Association, Air Force Training 2015, Arlington, Virginia (www.trainingsystems.org/publications/AirForce.pdf), 2010.
19. M. Pell, R. McNeill and J. Roberts, Unprepared: Texas blast shows failure of emergency planning law analysis shows, *NBC News*, May 22, 2013.
20. B. Reaves and T. Morris, An open virtual testbed for industrial control system security research, *International Journal of Information Security*, vol. 11(4), pp. 215–229, 2012.
21. Sandia National Laboratories, National Supervisory Control and Data Acquisition (SCADA), Albuquerque, New Mexico (energy.sandia.gov/energy/ssrei/gridmod/cyber-security-for-electric-infrastructure/scada-systems), 2015.
22. Sandia National Laboratories, SCADA Training Courses, Albuquerque, New Mexico (energy.sandia.gov/energy/ssrei/gridmod/cyber-security-for-electric-infrastructure/scada-systems/education-and-training), 2015.

23. SANS Institute, SEC562: CyberCity Hands-on Kinetic Cyber Range Exercise, Bethesda, Maryland (www.sans.org/course/cybercity-hands-on-kinetic-cyber-range-exercise), 2016.
24. K. Stouffer, J. Falco and K. Scarfone, Guide to Industrial Control Systems (ICS) Security, NIST Special Publication 800-82, National Institute of Standards and Technology, Gaithersburg, Maryland, 2011.
25. The Learning Pit, LogixPro 500: From the World's Leading Supplier of Allen Bradley PLC Simulators, Whitby, Canada (www.thelearningpit.com/lp/logixpro.html).
26. U.S. Department of Energy, National SCADA Test Bed: Enhancing Control Systems Security in the Energy Sector, Washington, DC (www.energy.gov/sites/prod/files/oeprod/DocumentsandMedia/NSTB_Fact_Sheet_FINAL_09-16-09.pdf), 2009.
27. U.S. Department of Homeland Security, Recommended Practice: Developing an Industrial Control Systems Cybersecurity Incident Response Capability, Washington, DC, 2009.
28. N. Wertzberger, C. Glatter, W. Mahoney, R. Gandhi and K. Dick, Towards a low-cost SCADA test bed: An open-source platform for hardware-in-the-loop simulation, *Proceedings of the International Conference on Security and Management*, pp. 555–561, 2011.
29. M. Winn, M. Rice, S. Dunlap, J. Lopez and B. Mullins, Constructing cost-effective and targetable industrial control system honeypots for production networks, *International Journal of Critical Infrastructure Protection*, vol. 10, pp. 47–58, 2015.

30. J. Yoon, S. Dunlap, J. Butts, M. Rice and B. Ramsey, Evaluating the readiness of cyber first responders responsible for critical infrastructure protection, *International Journal of Critical Infrastructure Protection*, vol. 13, pp. 19–27, 2016.
31. J. Yoon, Framework for Evaluating the Readiness of Cyber First Responders for Industrial Control Systems, M.S. Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2016.

Appendix A. Wastewater Environment Y-Box Communication and Interface Code

This code was used to run the Y-Box interface of the aeration basin model of the wastewater treatment facility environment. This code simulates the aeration basin process and executes the experiment by automatically running trials with different experiment factors. The code is broken into three parts including the main interface code, Y-Box communication engine code, and code that controls the fan displays.

1.1 Wastewater Interface and Y-Box Communication Code

This code simulates the aeration basin process and presents a user interface displaying all data being provided by the Y-Box.

```
import pygame
from pygame.locals import *
import time
import _thread
import Ybox
import math

#
#####

# Author: Evan Plumley and Andrew Chaves
#
```

```

# This program is the communication engine for the Y-box and
# the GUI
# for monitoring and powering the physical wastewater system
#
#
#
#####

class WwtSim:
def __init__(self):
self._running = True
self.screen = None
self.size = self.width, self.height = 1450, 850
self.ybox = Ybox.Ybox()
self.erase_width = 140
self.erase_height = 35
self.vfd_spin = True

#####
# Chaves Initialization Code

#AI 0
#DI 1
#AO 2
#DO 3

```

#Fan 1 (3,0)

#Fan 0 (3,1)

#Low Left (3,3) ORP

#High Left (2,1) ORP

#Low Right (3,5)

#High Right (3,4)

#Top(3,6))

#Initialize Dissolved Oxygen and ORP and send to PLC

DO=2.0

ORP=-20

#Convert ORP (-50 to +50 scale) to something the ybox can use

(0-4095)

ORP=ORP+150

#Initial Valve Positions

Aerobic_Valve=50

Anaerobic_Valve=50

#Scale for sending DO (Converts 0-4 and 0-100) to ybox values

Scale_DO=math.ceil(4095/7)

Scale_ORP=math.ceil(4095/700)

Scale_Valves=math.ceil(4095/100)

#Scales and casts as an int DO, ORP, and Valves

Send_DO=**int**(DO*Scale_DO)

Send_ORP=**int**(ORP*Scale_ORP)

Send_Aerobic_Valve=Scale_Valves*Aerobic_Valve

Send_Anaerobic_Valve=Scale_Valves*Anaerobic_Valve

Fan_1=0

Fan_2=0

#Let it fly

self.ybox.sendAnWrite(2,0,Send_DO)

self.ybox.sendAnWrite(2,4,Send_ORP)

self.ybox.sendAnWrite(2,2,Send_Aerobic_Valve)

self.ybox.sendAnWrite(2,3,Send_Anaerobic_Valve)

self.ybox.sendWrite(3,0,Fan_1)

self.ybox.sendWrite(3,1,Fan_2)

self.ybox.sendWrite(3,2,0)

self.ybox.sendWrite(3,3,0)

self.ybox.sendWrite(3,4,0)

self.ybox.sendWrite(3,5,0)

self.ybox.sendWrite(3,6,0)

#What are we sending to the YBOX?

print('DO_Initial:', Send_DO)

print('\\nORP_Initial:', Send_ORP)

print('\\nSend_Aerobic_Valve_Initial:_', Send_Aerobic_Valve)

```
print( '\nSend_Anaerobic_Valve_Initial:_',
        Send_Anaerobic_Valve)
```

```
#Tell the infinite loop below what the intialized values are
... (Set them as the starting values)
```

```
self.Initial_DO=Send_DO
```

```
self.Initial_ORP=Send_ORP
```

```
Initial_Aerobic_Valve=Send_Aerobic_Valve
```

```
Initial_Anaerobic_Valve=Send_Anaerobic_Valve
```

```
#
```

```
#####
```

```
#Initialize screen
```

```
print ("Waste_Water_Monitor_Running")
```

```
pygame.init()
```

```
self.font = pygame.font.SysFont('Times', 25)
```

```
self.font2 = pygame.font.SysFont('Times', 25)
```

```
self.font3 = pygame.font.SysFont('Times', 35)
```

```
self.font4 = pygame.font.SysFont('Times', 20)
```

```
pygame.display.set_caption('Waste_Water_Treatment_Monitor')
```

```
self.screen = pygame.display.set_mode(self.size , pygame.
```

```
    HWSURFACE | pygame.DOUBLEBUF)
```

```
self.screen.fill((white))
```

```
self.screen.set_colorkey(white)
```

```

self.fan_width = 300
self.oxygen_content = 20

#add PLC output title
pygame.draw.rect(self.screen, black, (490, -5, 500, 55), 2)
self.screen.blit(self.font3.render('Sensor_Data_Output',
    True, (black)), (590, 5))

#divider line and input title
pygame.draw.line(self.screen, black, (0,630), (1450,630), 2)
pygame.draw.rect(self.screen, black, (435, 630, 600, 50), 2)
self.screen.blit(self.font3.render('PLC_Output_to_Sensors',
    True, (black)), (560, 635))

#add the fans
self.fan1cord = (25,150)
self.fan2cord = (325,150)

self.fan1 = pygame.image.load("wwtimages/fanBlade.jpg")
self.fan2 = pygame.image.load("wwtimages/fanBlade.jpg")

self.fan1.set_colorkey((0,0,0))
self.fan2.set_colorkey((0,0,0))

self.screen.blit(self.fan1, self.fan1cord)

```

```

self.screen.blit(self.fan2, self.fan2cord)

#add fan titles
self.screen.blit(self.font.render('Oxygen_Blower_System' ,
    True, (black)), (170, 95))

#add rectangles for fan speed display and their descriptions
self.screen.blit(self.font.render('VFD_Hz' , True, (black)),
    (230, 390))
pygame.draw.rect(self.screen , dark_grey , (210, 425, 150, 45),
    5)
self.screen.blit(self.font2.render('50' , True, (red)), (255,
    435))

#add the pipe valves
self.pipe1cord = (700,140)
self.pipe2cord = (1050,140)
self.pipe1 = pygame.image.load("wwtimages/pipe.png")
self.pipe2 = pygame.image.load("wwtimages/pipe.png")
self.screen.blit(self.pipe1, self.pipe1cord)
self.screen.blit(self.pipe2, self.pipe2cord)

```

```

#add valve titles

self.screen.blit(self.font.render('Disolved_Oxygen_Valve' ,
    True, (black)), (705, 95))

self.screen.blit(self.font.render('Oxygen_Reduction_Valve' ,
    True, (black)), (1050, 95))

#add rectangles and labels to display valve open percentage

self.screen.blit(self.font.render('Percentage_Open' , True, (
    black)), (740, 390))

self.screen.blit(self.font.render('Percentage_Open' , True, (
    black)), (1100, 390))

pygame.draw.rect(self.screen , dark_grey , (750, 425, 150, 45) ,
    5)

pygame.draw.rect(self.screen , dark_grey , (1110, 425, 150, 45)
    , 5)

#self.screen.blit(self.font2.render('50' , True, (dark_blue))
    , (800, 435))

#self.screen.blit(self.font2.render('100' , True, (dark_blue)
    ), (1160, 435))

#oxygen content ORP display

self.screen.blit(self.font4.render('Disolved_Oxygen_(mg/L): '
    , True, (black)), (680, 480))

pygame.draw.rect(self.screen , red , (680, 510, 50, 75), 0)
pygame.draw.rect(self.screen , green , (730, 510, 230, 75), 0)

```

```

pygame.draw.rect(self.screen , red , (960, 510, 20, 75), 0)
pygame.draw.rect(self.screen , black , (680, 510, 300, 75), 2)

self.screen.blit(self.font4.render('Oxygen_Reduction_(mV):' ,
    True, (black)), (1050, 480))

pygame.draw.rect(self.screen , red , (1035, 510, 50, 75), 0)
pygame.draw.rect(self.screen , green , (1085, 510, 230, 75), 0)
pygame.draw.rect(self.screen , red , (1315, 510, 20, 75), 0)
pygame.draw.rect(self.screen , black , (1035, 510, 300, 75), 2)

#
#####

# Add the logical output
#
#####

#add rectangles for fan speed display and their descriptions
self.screen.blit(self.font.render('PLC_Hz' , True, (black)),
    (230, 715))

```

```

pygame.draw.rect(self.screen , dark_grey , (210, 745, 150, 45) ,
5)
#self.screen.blit(self.font2.render('50' , True, (red)),
(115, 755))

#add rectangles and labels to display valve open percentage
self.screen.blit(self.font.render('Percentage_Open' , True, (
black)), (740, 715))
self.screen.blit(self.font.render('Percentage_Open' , True, (
black)), (1100, 715))
pygame.draw.rect(self.screen , dark_grey , (750, 745, 150, 45) ,
5)
pygame.draw.rect(self.screen , dark_grey , (1110, 745, 150, 45)
, 5)
#self.screen.blit(self.font2.render('50' , True, (dark_blue))
, (800, 755))
#self.screen.blit(self.font2.render('100' , True, (dark_blue)
), (1160, 755))

```

```

pygame.display.update()

try:
    _thread.start_new_thread(timedReads, (self,))
    _thread.start_new_thread(spinFans, (self,))
except Exception as e:
    print(e)
    self._running = True

#Handle all events
def on_event(self, event):
    if event.type == pygame.QUIT:
        self._running = False
    else: #Determine if button was clicked
        pygame.display.update()

def on_loop(self):
    pass

def on_render(self):
    pass

```

```

def on_cleanup(self):
pygame.quit()

#start program

def on_execute(self):
#if self.on_init() == False:
#self._running = False

while( self._running ):
for event in pygame.event.get():
self.on_event(event)
self.on_loop()
self.on_render()
self.on_cleanup()

def spinFans(self):

rotationDegree1 = 2 #rotation degrees for each fan
rotationDegree2 = 2
past = int(round(time.time() * 1000)) #getting starting
milisecond time to execute reads from the ybox

while True:
present = int(round(time.time() * 1000)) #getting present
time to comapre to past

```

```

#check to see if 100 milliseconds have passed

if present - past >= 40:
    past = present

#####

#rotate the fans

#####

rotationDegree1 = rotationDegree1 % 360 #rotation degree for
    fan 1
rotationDegree2 = rotationDegree2 % 360 #rotation degree for
    fan 2

##conduct the actual movements

if self.vfd_spin:
    newFan1 = rot_center(self.fan1, rotationDegree1)
    self.screen.blit(newFan1, self.fan1cord)
    newFan2 = rot_center(self.fan2, rotationDegree2)
    self.screen.blit(newFan2, self.fan2cord)
    rotationDegree1 = rotationDegree1 + 10 #move the dwgrees
    rotationDegree2 = rotationDegree2 + 10 #move the dwgrees
    pygame.display.update()

#method to monitor the PLC and chnage the display accordingly

```

```

def timedReads(self):
    white = (255, 255, 255)
    past = int(round(time.time() * 1000)) #getting starting
        milisecond time to execute reads from the ybox
    rotationDegree1 = 2 #rotation degrees for each fan
    rotationDegree2 = 2
    #waterSize1 = 1
    #waterSize2 = 100
    oxyBarPos = 0
    reduction = 0

    while True:
        present = int(round(time.time() * 1000)) #getting present
            time to comapre to past
        #check to see if 100 milliseconds have passed
        if present - past >= 100:
            past = present

        #Read the Setpoint for DO and ORP from PLC
        DO_Set_Point=(self.ybox.sendRead(0,0)*(1/(4095/7)))
        ORP_Set_Point=(self.ybox.sendRead(0,3)*(1/(4095/700)))

        #Read the output (%) of the PIDs controlling the VFD and
            Valves

```

```

Valve_Aerobic_Percent_Output=self.ybox.sendRead(0,4) #valve
percenatge
Valve_Anaerobic_Percent_Output=self.ybox.sendRead(0,5)
Frequency_Percent_Increase=self.ybox.sendRead(0,2)

#Scale valves and VFD speed values to 0-100, why is it 200?
zero_to_hun_scale=100/4095
Frequency_Scale=60/4095#100/4095

#Use the Scale
#####
#Display Frequecny sent to the VFP from the PLC
#####
Frequency_Percent_Increase_Scaled=(Frequency_Percent_Increase
    *Frequency_Scale) #0-60
print("PLC_Freq\n", Frequency_Percent_Increase)
pygame.draw.rect(self.screen, white, (215, 750, self.
    erase_width, self.erase_height), 0)
self.screen.blit(self.font2.render("%0.4f" %
    Frequency_Percent_Increase_Scaled, True, (red)), (235,
    755)) #displays 0-60 scale

#####
#Display Aerobic Valve updates
#####

```

```

Valve_Aerobic_Percent_Output_Scaled=int(
    Valve_Aerobic_Percent_Output*zero_to_hun_scale) # Aerobic
    valve open percentage
self.screen.blit(self.pipe1, self.pipe1cord)
pygame.draw.circle(self.screen, blue, (830,265),
    Valve_Aerobic_Percent_Output_Scaled)
pygame.draw.rect(self.screen, white, (753, 428, self.
    erase_width, self.erase_height), 0)
self.screen.blit(self.font2.render(str(
    Valve_Aerobic_Percent_Output_Scaled), True, (dark_blue)),
    (800, 435))
pygame.display.update()

```

```
#####
```

```
#Display Anerobic Valve updates
```

```
#####
```

```

Valve_Anaerobic_Percent_Output_Scaled=int(
    Valve_Anaerobic_Percent_Output*zero_to_hun_scale)# =
    Anerobic valve open percetage
self.screen.blit(self.pipe2, self.pipe2cord)
pygame.draw.circle(self.screen, blue, (1177,265),
    Valve_Anaerobic_Percent_Output_Scaled)
pygame.draw.rect(self.screen, white, (1115, 430, self.
    erase_width, self.erase_height), 0)
self.screen.blit(self.font2.render(str(
    Valve_Anaerobic_Percent_Output_Scaled), True, (dark_blue)

```

), (1160, 435))

#Print Valve positions and VFD speed

#print('Aerobic Valve Percent: ',

Valve_Aerobic_Percent_Output_Scaled)

#print('\nAnaerobic Valve Percent: ',

Valve_Anaerobic_Percent_Output_Scaled)

#print('\nVFD Speed Percent: ',

Frequency_Percent_Increase_Scaled)

*#Send the valve positions to the 3-10V meters and display to
the GUI*

#Logical Output to valves

#####

#Output to valves

Aerobic_Valve_Scaled_Display=Valve_Aerobic_Percent_Output

Anaerobic_Valve_Scaled_Display=Valve_Anaerobic_Percent_Output

self.ybox.sendAnWrite(2,2,Aerobic_Valve_Scaled_Display)

self.ybox.sendAnWrite(2,3,Anaerobic_Valve_Scaled_Display)

#display the outputs to the meter

Valve_Anaerobic_meter_display=int(

*Anaerobic_Valve_Scaled_Display*zero_to_hun_scale)# =*

Anerobic valve open percetage

```

Valve_Aerobic_meter_display=int( Aerobic_Valve_Scaled_Display*
    zero_to_hun_scale)# =Anerobic valve open percetage
pygame.draw.rect(self.screen , white , (760, 750, self.
    erase_width , self.erase_height), 0)
pygame.draw.rect(self.screen , white , (1120, 750, self.
    erase_width , self.erase_height), 0)
self.screen.blit(self.font2.render( str(
    Valve_Aerobic_meter_display), True, (dark_blue)), (800,
    755)) #aerobic
self.screen.blit(self.font2.render(str(
    Valve_Anaerobic_meter_display) , True, (dark_blue)),
    (1160, 755)) #anerobic

pygame.display.update()

#####Decrease DO AND ORP#####

#Subtract percentage of valves from 100 and use that to
    determine how much to lower DO and ORP
#The more anaerobic the water, the lower ORP and DO both are
DO_Subtract=(101-Valve_Aerobic_Percent_Output_Scaled)
ORP_Subtract=(101-Valve_Anaerobic_Percent_Output_Scaled)

#Subtract the values and divide by two to make the change
    less aggressive (helps the PIDs)
DO_Update=int(self.Initial_DO-(DO_Subtract/8))

```

```

ORP_Update=int(self.Initial_ORP-(ORP_Subtract/8))

#Update the previous oxygen value so next time through the
loop we are subtracting from the most recent correct value
self.Initial_DO=DO_Update
self.Initial_ORP=ORP_Update

#Let it fly
self.ybox.sendAnWrite(2,0,DO_Update)
self.ybox.sendAnWrite(2,4,ORP_Update)

#####INCREASE DO AND ORP#####

#Increase DO and ORP if valves are opening according to
equation
Slope_Up_DO=.15
Slope_Up_ORP=.15

#Multiply the valve percentage by a slope, add this value to
previous DO and ORP values
DO_Update=self.Initial_DO+(Slope_Up_DO*
    Valve_Aerobic_Percent_Output_Scaled)
ORP_Update=self.Initial_ORP+(Slope_Up_ORP*
    Valve_Anaerobic_Percent_Output_Scaled)

```

```

#Update the previous DO and ORP values so next time through
the loop we are adding to the correct value
self.Initial_DO=DO_Update
self.Initial_ORP=ORP_Update

#Cast as an integer for the YBOX's sake
DO_Update=int(DO_Update)
ORP_Update=int(ORP_Update)

#Show what is being sent
print( '\nDO_Update: ', DO_Update)
print( '\nORP_Update: ', ORP_Update)

#let DO and ORP fly
self.ybox.sendAnWrite(2,0,DO_Update)
self.ybox.sendAnWrite(2,4,ORP_Update)
pygame.display.update()

#####
# add and update oxygen level bar
#####
#oxyBarPos = oxyBarPos % 300
#oxyContent = oxyBarPos / 75 # this is between 0 and 4

```

```
#reduction = (oxyBarPos / 3) - 50 # reduction is between -50  
and 50
```

```
#dissolved oxygen
```

```
pygame.draw.rect(self.screen, white, (910, 480, 80, 30), 0)
```

```
if (DO_Update / (4095/7)) > 1 and (DO_Update / (4095/7)) <  
5.5:
```

```
self.screen.blit(self.font.render("%.4f" % (DO_Update /  
(4095/7)) , True, (dark_green)), (910, 480))
```

```
else:
```

```
self.screen.blit(self.font.render("%.4f" % (DO_Update /  
(4095/7)) , True, (red)), (910, 480))
```

```
oxyBarPos = (DO_Update / (4095/7)) * 42.85
```

```
pygame.draw.rect(self.screen, red, (680, 510, 40, 75), 0)
```

```
pygame.draw.rect(self.screen, green, (720, 510, 180, 75), 0)
```

```
pygame.draw.rect(self.screen, red, (915, 510, 45, 75), 0)
```

```
pygame.draw.rect(self.screen, black, (680, 510, 300, 75), 2)
```

```
if oxyBarPos >= 0 and oxyBarPos <= 300:
```

```
pygame.draw.line(self.screen, black, (oxyBarPos + 680, 510), (  
oxyBarPos + 680, 510+75), 6)
```

```
pygame.display.update()
```

```
#oxygen reduction
```

```
pygame.draw.rect(self.screen, white, (1270, 480, 120, 30), 0)
```

```
if ((ORP_Update / (4095/700)) - 150) > -30 and ((ORP_Update /  
(4095/700)) - 150) < 450:
```

```

self.screen.blit(self.font.render("%.4f" % ((ORP_Update /
(4095/700)) - 150) , True, (dark_green)), (1270, 480))
else:
self.screen.blit(self.font.render("%.4f" % ((ORP_Update /
(4095/700)) - 150) , True, (red)), (1270, 480))
orpOxyBarPos = (ORP_Update / (4095/700)) * .43
pygame.draw.rect(self.screen , red , (1035, 510, 50, 75), 0)
pygame.draw.rect(self.screen , green , (1085, 510, 210, 75), 0)
pygame.draw.rect(self.screen , red , (1295, 510, 40, 75), 0)
pygame.draw.rect(self.screen , black , (1035, 510, 300, 75), 2)
if orpOxyBarPos >= 0 and orpOxyBarPos <= 300:
pygame.draw.line(self.screen , black , (orpOxyBarPos +
1035,510), (orpOxyBarPos + 1035, 510+75), 6)
pygame.display.update()

```

```

#####GET TOTAL VALVE POSITION#####
#Get the total valve position (0-200): (Anaerobic (100) +
Aerobic (100) = 200 Total)
Total_Valve_Position=int(Valve_Aerobic_Percent_Output+
Valve_Anaerobic_Percent_Output)
Total_Valve_Scale=200/4095

#Print the Total Valve Position
print('\nTotal_valve:┐', Total_Valve_Position*
Total_Valve_Scale)

```

```

#Let the Total Valve position fly
self.ybox.sendAnWrite(2,5,Total_Valve_Position)

#####GET VFD TRUE SPEED #####
#Get the VFD's direct output from the Ybox for its speed NOTE
: the VFD must be running!!!!, not just on!!!
True_Frequency=self.ybox.sendRead(0,6)

print("True_Frequency_Raw", True_Frequency)

#Convert the signal to 0-60 hertz
True_Frequency_Scaled=True_Frequency*(60/4095)

if True_Frequency_Scaled < 5:
    self.vfd_spin = False
else:
    self.vfd_spin = True

#####
# Display true frequency to gui
#####
pygame.draw.rect(self.screen, white, (215, 430, self.
    erase_width, self.erase_height), 0)
self.screen.blit(self.font2.render("%4f" %
    True_Frequency_Scaled, True, (red)), (255, 435))

```

```

pygame.display.update()

#Print the true VFD frequency
print( '\nTrue_VFD_Frequency: ', True_Frequency_Scaled)

#Start fans if the VFD is runnning
if True_Frequency_Scaled > 5:
    self.ybox.sendWrite(3,0,1)
    self.ybox.sendWrite(3,1,1)
else:
    self.ybox.sendWrite(3,0,0)
    self.ybox.sendWrite(3,1,0)

#Print divider for next time through loop
print( '#####\n')

#####Obtain Alarms, check alarms, send to light if
necessary#####
DO_High=self.ybox.sendRead(1,0)
if DO_High==1:
    self.ybox.sendWrite(3,4,1)
else:
    self.ybox.sendWrite(3,4,0)

DO_Low=self.ybox.sendRead(1,1)

```

```

if DO_Low==1:
    self.ybox.sendWrite(3,5,1)
else:
    self.ybox.sendWrite(3,5,0)

ORP_High=self.ybox.sendRead(1,5)
if ORP_High==1:
    self.ybox.sendWrite(3,2,1)
else:
    self.ybox.sendWrite(3,2,0)

ORP_Low=self.ybox.sendRead(1,3)
if ORP_Low==1:
    self.ybox.sendWrite(3,3,1)
else:
    self.ybox.sendWrite(3,3,0)

Pressure_High=self.ybox.sendRead(1,4)
if Pressure_High==1:
    self.ybox.sendWrite(3,6,1)
else:
    self.ybox.sendWrite(3,6,0)

```

[illegible]

$\#rotationDegree2 = rotationDegree2 \% 360$ $\#rotation$ degree for
fan 2

```
# self.screen.blit(newFan2, self.fan2cord)
```

```
# pygame.display.update()
```

```
#rotationDegree1 = rotationDegree1 + 10 #move the dwgrees
#rotationDegree2 = rotationDegree2 + 10 #move the dwgrees
```

```
pygame.display.update()
```

```
def rot_center(image, angle):
    """rotate an image while keeping its center and size"""
    orig_rect = image.get_rect()
    rot_image = pygame.transform.rotate(image, angle)
    rot_rect = orig_rect.copy()
    rot_rect.center = rot_image.get_rect().center
    rot_image = rot_image.subsurface(rot_rect).copy()
    rot_image = rot_image.convert()
    return rot_image
```

```
if __name__ == "__main__" :
```

```
red = (225,0,0)
white = (255, 255, 255)
black = (0,0,0)
grey = (200, 200, 200)
dark_grey = (140, 140, 140)
blue = (130, 130, 255)
```

```

dark_blue = (0, 0, 150)
green = (0,240,0)
dark_green = (0,175,0)

```

```

wwtSim = WwtSim()
wwtSim.on_execute()

```

1.2 Y-Box Communication Engine

This code enables messages to be sent to and received from the Y-Box.

```

import time
import msvert
from pygame.locals import *
import serial
import _thread

```

```

#

```

```

#####

```

```

# Author: Evan Plumley

```

```

# Date: 6/20/2016

```

```

# version: 1.0

```

```

#

```

```

#The Ybox class

```

```

# dependencies: pygame, pyserial

```

```

#

```

```

#####

```

```

class Ybox(object):
def __init__(self):
    self.ser = serial.Serial(
port= 'COM4',
baudrate=115200,
parity=serial.PARITY_NONE,
stopbits=serial.STOPBITS_ONE,
bytesize=serial.EIGHTBITS,
timeout = 1
)
    time.sleep(1)
print ("Initialize Complete")

def readLine(self):
    line = self.ser.readline().decode()
    line = line.strip ()
return line

def sendRead(self, slot, channel):
    msg = "R" + str(slot) + "," + str(channel) + "\n"
    self.ser.write(msg.encode())

```

```

line = self.ser.readline().decode()
line = line.strip ()
#Give me an integer
split=line.split(',')
line=split[2]
line=int(line,16)
return line

def sendAnRead(self, slot, channel):
msg = "R" + str(slot) + "," + str(channel) + "\n"
self.ser.write(msg.encode())
line = self.ser.readline().decode()
line = line.strip ()
return line


def sendWrite(self, slot, channel, value):
msg = "W" + str(slot) + "," + str(channel) + "," + str(value)
    + "\n"
self.ser.write(msg.encode())
line = self.ser.readline().decode()
line = line.strip ()
return line


def sendAnWrite(self, slot, channel, value):
msg = "W" + str(slot) + "," + str(channel) + "," + str(value)
    + "\n"
self.ser.write(msg.encode())

```

```

line = self.ser.readline().decode()
line = line.strip ()
return line

```

```

def readAll(self , slot):
    slotstr = str(slot)
    msg = 'R1' + slotstr + 'A'
    self.ser.write(msg.encode())
    line = self.ser.readline().decode()
    line = line.strip ()
return line

```

1.3 Wastewater Blower fan interface control

This code enables the image processing and movement of the wastewater blower fan for the display interface of the environment.

```

import pygame
from pygame.locals import *

import pygamebutton

class BlowerFan:

    def __init__(self , sim=None, fan_num=0, x=0,y=0):
        if sim is not None:
            #load the fan onto the screen
            self.fan_num = fan_num

```

```
self.fan_image = pygame.image.load("wwtimages/fanBlade.jpg")  
sim.screen.blit(fan_image, (x,y))
```

```
def rotate(self):  
    pygame.transform.rotate(self.fan_image, 3)
```

Appendix B. Prison Environment Y-Box Communication and Interface Code

This code was used to run the Y-Box interface of the prison block model environment. This code reads and displays the status of the doors, locks and push buttons of the environment. The Y-Box communication engine shown in Appendix A is also used during the execution of this code.

2.1 Prison Interface and Y-Box Communication Code

This portion of the code controls the interface as a whole and updates the display while reading and writing to the Y-Box.

```
from pygame.locals import *
import guard_station
import prison_cell
import time
import _thread
import Ybox
import mantrap
import indicator

#
#####

# Authors: Evan Plumley and Jeff Guion
#
# This program runs the prison cell simulation displaying the
# value
```

of the PLC outputs and providing the PLC inputs

#

#

#

~~#####~~

class PrisonSim :

def __init__(self):

self._running = True

self.screen = None

self.size = self.width, self.height = 1400, 950

self.ybox = Ybox.Ybox()

self.buttonclick1 = False

self.buttonclick2 = False

self.buttonclick3 = False

self.buttonclick4 = False

self.buttonclick5 = False

self.buttontoggle1 = False

self.buttontoggle2 = False

self.buttontoggle3 = False

self.buttontoggle4 = False

self.buttontoggle5 = False

def on_init(self):

#Initialize screen

```

pygame.init()
self.font = pygame.font.SysFont('Times', 25)
pygame.display.set_caption('Ybox_Simulation')
self.screen = pygame.display.set_mode(self.size, pygame.
    HWSURFACE | pygame.DOUBLEBUF)
self.screen.fill((black))

x_padding = 42.5
y_padding = 95

cell_panel_width = 325
cell_panel_height = 300

#create the guard station panel with number of cells
self.guard_station_panel = guard_station.GuardStation(self,
    5, 50, 575)

#create each cell at given location
cell_one_panel = prison_cell.PrisonCell(self, 1, x_padding,
    y_padding)
cell_two_panel = prison_cell.PrisonCell(self, 2, 5*x_padding+
    cell_panel_width, y_padding)
cell_three_panel = prison_cell.PrisonCell(self, 3, 9*
    x_padding+2*cell_panel_width, y_padding)

```

```

#add all cells to list

self.cell_door_panels = [cell_one_panel , cell_two_panel ,
    cell_three_panel]

#add mantrap display

self.mantrap = mantrap.ManTrap(self , 3*x_padding+2*
    cell_panel_width , (2.2 * y_padding) + cell_panel_height)

#add the title of the program

main_title_dimensions = ((615, 70))
main_title_icon = pygame.image.load ("images/title.png")
main_title_icon2 = pygame.image.load ("images/title.png")
main_title_icons = [main_title_icon , main_title_icon2]
main_title = indicator.Indicator(self , 390, 10,
    main_title_icons , main_title_dimensions)

# set the initial states

self.ybox.sendWrite(0,9,1) #panel enabled

self.ybox.sendWrite(0,0,1) #light indicators green
self.ybox.sendWrite(0,2,1)
self.ybox.sendWrite(0,4,1)

self.ybox.sendWrite(0,10,1)
self.ybox.sendWrite(0,11,1)

```

```

pygame.display.update()

try:
    _thread.start_new_thread(timedReads, (self,))
except Exception as e:
    print(e)
    self._running = True

#Handle all events
def on_event(self, event):
    if event.type == pygame.QUIT:
        self._running = False
    else: #Determine if button was clicked
        for i, cell_btn in enumerate(self.guard_station_panel.
            cell_btns):
            if 'click' in cell_btn.handleEvent(event):
                try:
                    clickButton(self, i)
                except Exception as e:
                    print(e)

        for i, cell in enumerate(self.cell_door_panels):
            if 'click' in cell.key_btn.handleEvent(event):
                try:
                    openDoor(self, i)
                except Exception as e:
                    print(e)

```

```

if 'click' in self.guard_station_panel.disable_btn.
    handleEvent(event):
self.guard_station_panel.disable_clicked = True

pygame.display.update()

```

```

def on_loop(self):
pass
def on_render(self):
pass
def on_cleanup(self):
pygame.quit()

```

```

#start program
def on_execute(self):
if self.on_init() == False:
self._running = False

while( self._running ):
for event in pygame.event.get():
self.on_event(event)
self.on_loop()

```

```

self.on_render()
self.on_cleanup()

#pushes button for two seconds then releases it
def clickButton(self, i):
#door one button
    if i == 0 and self.buttonclick1 == False:
        self.buttonclick1 = True
        self.buttontoggle1 = True
    elif i == 0 and self.buttonclick1 == True:
        self.buttonclick1 = False
        self.buttontoggle1 = True

#door two button
    elif i == 1 and self.buttonclick2 == False:
        self.buttonclick2 = True
        self.buttontoggle2 = True
    elif i == 1 and self.buttonclick2 == True:
        self.buttonclick2 = False
        self.buttontoggle2 = True

#door three button
    elif i == 2 and self.buttonclick3 == False:
        self.buttonclick3 = True
        self.buttontoggle3 = True
    elif i == 2 and self.buttonclick3 == True:

```

```

self.buttonclick3 = False
self.buttontoggle3 = True

#door four button
elif i == 3 and self.buttonclick4 == False:
self.buttonclick4 = True
self.buttontoggle4 = True
elif i == 3 and self.buttonclick4 == True:
self.buttonclick4 = False
self.buttontoggle4 = True

#door five button
elif i == 4 and self.buttonclick5 == False:
self.buttonclick5 = True
self.buttontoggle5 = True
elif i == 4 and self.buttonclick5 == True:
self.buttonclick5 = False
self.buttontoggle5 = True

else:
print("Something_went_wrong_in_the_lock_reads")

#method to open the door via the manual key button

```

```

def openDoor(self , i):
    if self.cell_door_panels[i].doorClosed == True:
        self.cell_door_panels[i].cell_door.change_state()
        self.cell_door_panels[i].doorClosed = False

    elif self.cell_door_panels[i].doorClosed == False:
        self.cell_door_panels[i].cell_door.change_state()
        self.cell_door_panels[i].doorClosed = True

#method to monitor the PLC and chnage the display accordingly

def timedReads(self):
    past = int(round(time.time() * 1000)) #getting starting
        milisecond time to execute reads from the ybox
    while True:
        present = int(round(time.time() * 1000)) #getting present
            time to comapre to past
        #check to see if 100 milliseconds have passed
        if present - past >= 100:
            past = present
            #####
            #set button clikc values and write accordingly
            #####
            if self.buttonclick1 == True and self.buttontoggle1 == True:
                self.ybox.sendWrite(0,1,1)
                self.buttontoggle1 = False

```

```

elif self.buttonclick1 == False and self.buttontoggle1 ==
    True:
self.ybox.sendWrite(0,1,0)
self.buttontoggle1 = False

if self.buttonclick2 == True and self.buttontoggle2 == True:
self.ybox.sendWrite(0,3,1)
self.buttontoggle2 = False
elif self.buttonclick2 == False and self.buttontoggle2 ==
    True:
self.ybox.sendWrite(0,3,0)
self.buttontoggle2 = False

if self.buttonclick3 == True and self.buttontoggle3 == True:
self.ybox.sendWrite(0,5,1)
self.buttontoggle3 = False
elif self.buttonclick3 == False and self.buttontoggle3 ==
    True:
self.ybox.sendWrite(0,5,0)
self.buttontoggle3 = False

if self.buttonclick4 == True and self.buttontoggle4 == True:
self.ybox.sendWrite(0,7,1)
self.buttontoggle4 = False
elif self.buttonclick4 == False and self.buttontoggle4 ==
    True:

```

```

self.ybox.sendWrite(0,7,0)
self.buttontoggle4 = False

if self.buttonclick5 == True and self.buttontoggle5 == True:
self.ybox.sendWrite(0,8,1)
self.buttontoggle5 = False
elif self.buttonclick5 == False and self.buttontoggle5 ==
    True:
self.ybox.sendWrite(0,8,0)
self.buttontoggle5 = False

#####
#read the lock value for all doors from the PLC and then
    react
#####
for i in range(0, len(self.cell_door_panels)):
#acquiring the actual channel number
if i == 0:
readnum = 0
elif i == 1:
readnum = 3
elif i == 2:
readnum = 6
else:
print("Something_went_wrong_in_the_lock_reads")

```

```

resp = self.ybox.sendRead(1, readnum)
readnum = str(readnum)
if resp == ("r1," + readnum + ",1") and self.cell_door_panels
    [i].lockClosed == True: #checks for a lock state change
self.cell_door_panels[i].lock_indicator.change_state() #open
    lock
self.cell_door_panels[i].lockClosed = False #lock open state
    flag
#make sure I dont unecesariily change the state due to the
    manual key
if self.cell_door_panels[i].doorClosed == True:
self.cell_door_panels[i].cell_door.change_state()
self.cell_door_panels[i].doorClosed = False
pygame.display.update()

elif resp == ("r1,"+ readnum +",0") and self.cell_door_panels
    [i].lockClosed == False:
self.cell_door_panels[i].lock_indicator.change_state() #open
    lock
self.cell_door_panels[i].lockClosed = True #lock open state
    flag
#make sure I dont unecesariily change the state due to the
    manual key

```

```

if self.cell_door_panels[i].doorClosed == False:
    self.cell_door_panels[i].cell_door.change_state()
    self.cell_door_panels[i].doorClosed = True
    pygame.display.update()

#Checking for chnages for the indicator light
for i in range(0, len(self.cell_door_panels)):
    #mapping to secure lights for the plc
    if i == 0:
        writenum = 0
    elif i == 1:
        writenum = 2
    elif i == 2:
        writenum = 4

    if (self.cell_door_panels[i].doorClosed == False or self.
        cell_door_panels[i].lockClosed == False) and self.
        cell_door_panels[i].indicatorLight == True:
        self.cell_door_panels[i].cell_door_indicator.change_state()#
            change to red
        self.cell_door_panels[i].indicatorLight = False
        self.ybox.sendWrite(0,writenum,0) # turn the PLC light
        pygame.display.update()

    elif (self.cell_door_panels[i].doorClosed == True and self.
        cell_door_panels[i].lockClosed == True) and self.

```

```

        cell_door_panels[i].indicatorLight == False:
self.cell_door_panels[i].cell_door_indicator.change_state()#
    change to green
self.cell_door_panels[i].indicatorLight = True
self.ybox.sendWrite(0,writenum,1) # turn the PLC light
pygame.display.update()

#####
# read prison guard button statuses just for cell doors and
    mantrap
#####
for i in range(0, len(self.guard_station_panel.button_pushed)
    ):
#acquiring the actual channel number
if i == 0:
readnum = 1
writenum = 1
elif i == 1:
readnum = 4
writenum = 3
elif i == 2:
readnum = 7
writenum = 5
elif i == 3:
readnum = 10

```

```

writenum = 7
elif i == 4:
readnum = 12
writenum = 8
else:
print("Something_went_wrong_in_the_button_reads")

resp2 = self.ybox.sendRead(1, readnum)
readnum = str(readnum)

if resp2 == ("r1,"+ readnum +",1") and self.
    guard_station_panel.button_pushed[i] == False:
self.guard_station_panel.btn_statuses[i].change_state()
self.guard_station_panel.button_pushed[i] = True
self.ybox.sendWrite(0,writenum,1)
pygame.display.update()
elif resp2 == ("r1,"+ readnum + ",0") and self.
    guard_station_panel.button_pushed[i] == True:
self.guard_station_panel.btn_statuses[i].change_state()
self.guard_station_panel.button_pushed[i] = False
self.ybox.sendWrite(0,writenum,0)
pygame.display.update()

```

```

#####

```

```

#Read guard station light statuses

#####

for i in range(0, len(self.guard_station_panel.lights)):
    #acquiring the actual channel number
    if i == 0:
        readnum = 2
    elif i == 1:
        readnum = 5
    elif i == 2:
        readnum = 8
    elif i == 3:
        readnum = 13
    else:
        print("Something_went_wrong_in_the_button_reads")

resp3 = self.ybox.sendRead(1, readnum)
readnum = str(readnum)
if resp3 == ("r1,"+ readnum +",1") and self.
    guard_station_panel.light_green[i] == False:
        self.guard_station_panel.lights[i].change_state()
        self.guard_station_panel.light_green[i] = True
pygame.display.update()

```

```

elif resp3 == ("r1," + readnum + ",0") and self.
    guard_station_panel.light_green[i] == True:
self.guard_station_panel.lights[i].change_state()
self.guard_station_panel.light_green[i] = False
pygame.display.update()

#####

# Read the locks for the man trap and react appropriatley

#####

#lock reads for trap door one

#####

readnum = 9
writenum = 10
readnum_2 = 11
writenum_2 = 11
resp4 = self.ybox.sendRead(1, readnum)
resp5 = self.ybox.sendRead(1, readnum_2)
readnum = str(readnum)
readnum_2 = str(readnum_2)
#all reads done for both above

if resp4 == ("r1," + readnum + ",1") and self.mantrap.
    lock1Closed == True: #checks for a lock state change
self.mantrap.lock_indicator1.change_state() #open lock
self.mantrap.lock1Closed = False #lock open state flag

```

```

self.mantrap.trap_door1.change_state()
self.mantrap.door1Closed = False
self.ybox.sendWrite(0,writenum,0) # trap door in unsecure
if self.mantrap.indicatorLight == True:
    self.mantrap.indicatorLight = False
    self.mantrap.secure_indicator.change_state()
pygame.display.update()

elif resp4 == ("r1," + readnum + ",0") and self.mantrap.
    lock1Closed == False: #checks for a lock state change
    self.mantrap.lock_indicator1.change_state() #open lock
    self.mantrap.lock1Closed = True #lock open state flag
    self.mantrap.trap_door1.change_state()
    self.mantrap.door1Closed = True
    self.ybox.sendWrite(0,writenum,1) # trap door in unsecure
if resp5 == "r1,11,0" and self.mantrap.indicatorLight ==
    False:
        self.mantrap.indicatorLight = True
        self.mantrap.secure_indicator.change_state()
        pygame.display.update()

#lock reads for trap door 2
#####

```

```

if resp5 == ("r1," + readnum_2 + ",1") and self.mantrap.
    lock2Closed == True: #checks for a lock state change
    self.mantrap.lock_indicator2.change_state() #open lock
    self.mantrap.lock2Closed = False #lock open state flag
    self.mantrap.trap_door2.change_state()
    self.mantrap.door2Closed = False
    self.ybox.sendWrite(0,writenum_2,0) # trap door in unsecure
if self.mantrap.indicatorLight == True:
    self.mantrap.indicatorLight = False
    self.mantrap.secure_indicator.change_state()
    pygame.display.update()

elif resp5 == ("r1," + readnum_2 + ",0") and self.mantrap.
    lock2Closed == False: #checks for a lock state change
    self.mantrap.lock_indicator2.change_state() #open lock
    self.mantrap.lock2Closed = True #lock open state flag
    self.mantrap.trap_door2.change_state()
    self.mantrap.door2Closed = True
    self.ybox.sendWrite(0,writenum_2,1) # trap door secure
if resp4 == "r1,9,0" and self.mantrap.indicatorLight == False
    :
    self.mantrap.indicatorLight = True
    self.mantrap.secure_indicator.change_state()
    pygame.display.update()

```

*#read the key and execute only if the key holds the power to
do so*

```
resp6 = self.ybox.sendRead(1, 14)
if resp6 == ("r1,14,1") and self.guard_station_panel.
    panel_enabled == False and self.guard_station_panel.
    panel_keyControl == True:
self.ybox.sendWrite(0,9,1)
self.guard_station_panel.enablePanel()
pygame.display.update()
if resp6 == ("r1,14,0") and self.guard_station_panel.
    panel_enabled == True and self.guard_station_panel.
    panel_keyControl == True:
self.ybox.sendWrite(0,9,0)
self.guard_station_panel.disablePanel()
pygame.display.update()
```

*#read the simulation disable button and take control power
form the key*

```
if self.guard_station_panel.disable_clicked == True and self.
    guard_station_panel.panel_enabled == False:
self.guard_station_panel.panel_keyControl = False
self.guard_station_panel.disable_clicked = False
self.ybox.sendWrite(0,9,1)
self.guard_station_panel.enablePanel()
```

```

pygame.display.update()

if self.guard_station_panel.disable_clicked == True and self.
    guard_station_panel.panel_enabled == True:
self.guard_station_panel.panel_keyControl = False
self.guard_station_panel.disable_clicked = False
self.ybox.sendWrite(0,9,0)
self.guard_station_panel.disablePanel()
pygame.display.update()

#return control power to the key if the sim and key match
if self.guard_station_panel.panel_keyControl == False and
    resp6 == ("r1,14,1") and self.guard_station_panel.
    panel_enabled == True:
self.guard_station_panel.panel_keyControl = True

if self.guard_station_panel.panel_keyControl == False and
    resp6 == ("r1,14,0") and self.guard_station_panel.
    panel_enabled == False:
self.guard_station_panel.panel_keyControl = True

```

```

if __name__ == "__main__" :

    green = (200,0,0)
    white = (255, 255, 255)
    black = (0,0,0)
    grey = (200, 200, 200)
    dark_grey = (140, 140, 140)
    light_blue = (0, 0, 255)
    dark_blue = (0, 0, 150)

    prisonSim = PrisonSim()
    prisonSim.on_execute()

```

2.2 Prison Guard Station Interface Code

This portion of the code controls the guard station display and button interface for the prison environment.

```

from pygame.locals import *
import indicator
import pygamebutton

class GuardStation:

    def __init__(self, sim=None, num_cells=0, x=0, y=0):
        if sim:
            screen = sim.screen

```

```

font = sim.font
self.cell_btns = []
self.lights = []
self.btn_statuses = []
self.button_pushed = []
self.light_green = []
self.panel_enabled = True
self.panel_keyControl = True #enables the key o be overridden
self.disable_clicked = False
btn_height = 40;
btn_width = 100;
cell_padding_x = 25
cell_padding_y = 60
grey = (200, 200, 200)
dark_grey = (140, 140, 140)
white = (255, 255, 255)
black = (0,0,0)

title_height = 25
light_height = 35
light_width = 35
panel_width = btn_width * num_cells + (num_cells+1)*
    cell_padding_x
panel_height = btn_height + 2*cell_padding_y + title_height +
    light_height

```

```
guard_display_panel = pygame.draw.rect(screen, (grey), (x, y,
    panel_width, panel_height))
```

```
#flag for cell button toggle
for i in range(0, num_cells):
    self.button_pushed.append(False)
```

```
#flag for light to toggle
for i in range(0, num_cells - 1):
    self.light_green.append(True)
```

```
#Title
#screen.blit(font.render('Guard Station Panel', True, (black)
    ), (x+20, y+10))
```

```
#guard station image loading
green_icon = pygame.image.load("images/greenLightAlt.png")
red_icon = pygame.image.load("images/redLightAlt.png")
pressed_icon = pygame.image.load("images/pressed.png")
notPressed_icon = pygame.image.load("images/notpressed.png")
panelEnabled = pygame.image.load("images/panelenabled.png")
panelDisabled = pygame.image.load("images/panelDisabled.png")
```

```
cell_start_x = x + cell_padding_x
```

```

cell_start_y = y + cell_padding_y + title_height + (0.5 *
    light_height)
light_start_x = x + cell_padding_x + (btn_width * 0.5) - (
    light_width * 0.5)
light_start_y = y + cell_padding_y
status_start_x = x + cell_padding_x
status_start_y = y + cell_padding_y + title_height + (0.5 *
    light_height) + btn_height + (btn_height * 0.5)

#disable button to the right of the panel
self.disable_btn = pygbutton.PygButton((cell_start_x + (
    panel_width * 0.37) , y - cell_padding_y , 125, btn_height
) , "Toggle_Panel")
self.disable_btn.draw(screen)

#create enabled/disabled indicator
panel_icons = [panelEnabled , panelDisabled]
icon_dimensions = ((400,45))
self.enable_label = indicator.Indicator(sim , x + (0.18 *
    panel_width) , y + 5 , panel_icons , icon_dimensions , grey)

for i in range(0, num_cells):
    cell_num = i+1
    if cell_num == num_cells-1:
        button_text = "Trap_1"
    elif cell_num == num_cells:

```

```

button_text = "Trap_2"
else:
button_text = "Cell_%s" % cell_num

cell_btn = pygbutton.PygButton((cell_start_x , cell_start_y ,
    btn_width , btn_height), button_text)
cell_btn.draw(screen)

#Door light
if i == num_cells - 2:
light_icons = [green_icon , red_icon]
light_dimensions = ((35,35))
light = indicator.Indicator(sim, light_start_x + (0.63 *
    btn_width), light_start_y , light_icons , light_dimensions)
self.lights.append(light)
elif i == num_cells -1:
pass
else:
light_icons = [green_icon , red_icon]
light_dimensions = ((35,35))
light = indicator.Indicator(sim, light_start_x , light_start_y
    , light_icons , light_dimensions)
self.lights.append(light)

#button indicator
status_icons = [notPressed_icon , pressed_icon]

```

```

status_dimensions = ((100, 20))
status = indicator.Indicator(sim, status_start_x,
    status_start_y, status_icons, status_dimensions, grey)

#adding all objects to respective lists
self.cell_btns.append(cell_btn)
self.btn_statuses.append(status)

#adjusting placements for next iteration of icon and button
    creation
light_start_x += btn_width + cell_padding_x
cell_start_x += btn_width + cell_padding_x
status_start_x += btn_width + cell_padding_x

def disablePanel(self):
    if self.panel_enabled == True:
        self.panel_enabled = False
        self.enable_label.change_state()

def enablePanel(self):
    if self.panel_enabled == False:
        self.panel_enabled = True
        self.enable_label.change_state()

```

2.3 Prison Indicator Image Display Code

This portion of the code was created to enable images to be easily interchangeable such as a closed door changing to an open door image or a red light changing to green.

```
import pygame
from pygame.locals import *

class Indicator:

    def __init__(self, sim=None, x=0, y=0, icons = [], icon_size=
        None, background_color=None):
        if sim:
            self.state_num = 0
            self.x = x
            self.y = y
            self.icons = icons
            self.sim = sim
            self.icon_size = icon_size
            self.background_color = background_color
            start_state_icon = icons[0]

            if icon_size:
                start_state_icon = pygame.transform.scale(start_state_icon,
                    icon_size)
            sim.screen.blit(start_state_icon, (x, y))
```

```

def change_state(self, screen=None, state_num=None):
    if state_num is None:
        self.state_num = (self.state_num + 1) % 2
        state_icon = self.icons[self.state_num]
        if self.icon_size:
            state_icon = pygame.transform.scale(state_icon, self.
                icon_size)
        if screen is None:
            screen = self.sim.screen

        if self.background_color:
            shape = pygame.Surface(self.icon_size, flags=SRCALPHA)
            shape.fill(self.background_color)
            screen.blit(shape, (self.x, self.y))
            pygame.display.update()

        screen.blit(state_icon, (self.x, self.y))

```

2.4 Prison Mantrap Image Display Code

This portion of the code was created to show the state of the mantrap doors within the environment.

```

from pygame.locals import *

import pygamebutton

```

```

import indicator

class ManTrap:

    def __init__(self, sim=None, x=0, y=0, current_state=0):
        if sim is not None:
            cell_padding_x = 25
            cell_padding_y = 25
            title_height = 25
            indicators_panel_width = 100
            grey = (200, 200, 200)
            dark_grey = (140, 140, 140)
            white = (255, 255, 255)
            black = (0,0,0)
            screen = sim.screen
            font = sim.font
            #self.inLoop = False #if a thread is running on a door
            self.door1Closed = True
            self.lock1Closed = True
            self.door2Closed = True
            self.lock2Closed = True
            self.indicatorLight = True #True is green state

            panel_width = (3*cell_padding_x+200) * 2
            panel_height = 2*cell_padding_y+325+title_height

```

```
cell_door_start_x = x+ cell_padding_x+indicators_panel_width
    + 15
```

```
cell_door_start_y = y+(3 * cell_padding_y)+title_height
```

```
cell_control_panel = pygame.draw.rect(screen , (grey) , (x, y,
    panel_width , panel_height))
```

#Title and door labels

```
screen.blit(font.render('ManTrap', True, (black)), (x+10, y
    +10))
```

```
screen.blit(font.render('1', True, (black)), (
    cell_door_start_x + 60 , cell_door_start_y - 40))
```

```
screen.blit(font.render('2', True, (black)), (
    cell_door_start_x + 215 , cell_door_start_y - 40))
```

#Trap Door 1

```
trap_door_dimensions = ((130,260))
```

```
closed_door_icon = pygame.image.load ("images/closedTrap.png"
    )
```

```
open_door_icon= pygame.image.load ("images/openTrap.png")
```

```
cell_door_icons = [closed_door_icon , open_door_icon]
```

```
self.trap_door1 = indicator.Indicator(sim, cell_door_start_x ,
    cell_door_start_y , cell_door_icons , trap_door_dimensions)
```

#Trap Door 2

```

trap_door_dimensions = ((130,260))
closed_door_icon = pygame.image.load ("images/closedTrap.png"
)
open_door_icon= pygame.image.load ("images/openTrap.png")
cell_door_icons = [closed_door_icon , open_door_icon]
self.trap_door2 = indicator.Indicator(sim, cell_door_start_x
+ 150, cell_door_start_y , cell_door_icons ,
trap_door_dimensions)

```

#Control Panel icon loading

```

green_icon = pygame.image.load("images/greenLightAlt.png")
red_icon = pygame.image.load("images/redLightAlt.png")
lock_icon = pygame.image.load("images/locked.png")
unlock_icon = pygame.image.load("images/unlocked.png")

```

#Door light

```

indicator_start_x = cell_door_start_x + 105
indicator_start_y = cell_door_start_y - 80
indicator_icons = [green_icon , red_icon]
indicator_dimensions = ((65, 70))
self.secure_indicator = indicator.Indicator(sim,
indicator_start_x , indicator_start_y , indicator_icons ,
indicator_dimensions)

```

#lock unlock icon 1

```

lock_start_x = cell_door_start_x - 100
lock_start_y = cell_door_start_y + 70
lock_icons = [lock_icon , unlock_icon]
lock_dimensions = ((65,70))
self.lock_indicator1 = indicator.Indicator(sim, lock_start_x ,
    lock_start_y , lock_icons , lock_dimensions , grey)

```

```

#lock unlock icon 2

```

```

lock_start_x = cell_door_start_x + 310
lock_start_y = cell_door_start_y + 70
lock_icons = [lock_icon , unlock_icon]
lock_dimensions = ((65,70))
self.lock_indicator2 = indicator.Indicator(sim, lock_start_x ,
    lock_start_y , lock_icons , lock_dimensions , grey)

```

```

#door key 1

```

```

#key_start_x = cell_door_start_x - 110

```

```

#key_start_y = lock_start_y+85

```

```

#key_btn_width = 90

```

```

#key_btn_height = 40

```

```

#key_btn_text = "Unlock Key"

```

```

#self.key_btn1 = pygbutton.PygButton((key_start_x ,
    key_start_y , key_btn_width , key_btn_height), key_btn_text)
#self.key_btn1.draw(screen)

```

```

#door key 2
#key_start_x = cell_door_start_x + 300
#key_start_y = lock_start_y+85
#key_btn_width = 90
#key_btn_height = 40
#key_btn_text = "Unlock Key"

#self.key_btn2 = pygamebutton.PygButton((key_start_x ,
    key_start_y , key_btn_width , key_btn_height), key_btn_text)
#self.key_btn2.draw(screen)

```

2.5 Prison Cell Door Image Display Code

This portion of the code was created to show the state of the prison cell doors within the environment.

```

from pygame.locals import *

import pygamebutton
import indicator

class PrisonCell:

    def __init__(self , sim=None, cell_num=0, x=0, y=0,
        current_state=0):
        if sim is not None:
            cell_padding_x = 25

```

```

cell_padding_y = 25
title_height = 25
indicators_panel_width = 100
grey = (200, 200, 200)
dark_grey = (140, 140, 140)
white = (255, 255, 255)
black = (0,0,0)
screen = sim.screen
font = sim.font
self.inLoop = False #if a thread is running on a door
self.doorClosed = True
self.lockClosed = True
self.indicatorLight = True #True is green state


panel_width = 3*cell_padding_x+100+150
panel_height = 2*cell_padding_y+250+title_height


cell_door_start_x = x+2*cell_padding_x+indicators_panel_width
cell_door_start_y = y+cell_padding_y+title_height


cell_control_panel = pygame.draw.rect(screen, (grey), (x, y,
    panel_width, panel_height))


#Title

```

```

screen.blit(font.render('Cell_%s' % cell_num, True, (black)),
            (x+10, y+10))

```

#Cell Door

```

cell_door_dimensions = ((150,250))
closed_door_icon = pygame.image.load ("images/closedDoor.png"
)
open_door_icon= pygame.image.load ("images/openDoor.png")
cell_door_icons = [closed_door_icon, open_door_icon]
self.cell_door = indicator.Indicator(sim, cell_door_start_x,
    cell_door_start_y, cell_door_icons, cell_door_dimensions)

```

#Control Panel

```

green_icon = pygame.image.load("images/greenLightAlt.png")
red_icon = pygame.image.load("images/redLightAlt.png")
lock_icon = pygame.image.load("images/locked.png")
unlock_icon = pygame.image.load("images/unlocked.png")

```

#Door light

```

indicator_start_x = x+cell_padding_x + 10
indicator_start_y = cell_door_start_y
indicator_icons = [green_icon, red_icon]
indicator_dimensions = ((70,75))
self.cell_door_indicator = indicator.Indicator(sim,
    indicator_start_x, indicator_start_y, indicator_icons,
    indicator_dimensions)

```

```

#lock unlock icon

lock_start_x = indicator_start_x
lock_start_y = cell_door_start_y + 85
lock_icons = [lock_icon , unlock_icon]
lock_dimensions = ((75,75))

self.lock_indicator = indicator.Indicator(sim, lock_start_x ,
    lock_start_y , lock_icons , lock_dimensions , grey)


#door key

key_start_x = indicator_start_x - 20
key_start_y = lock_start_y+85
key_btn_width = 120
key_btn_height = 40
key_btn_text = "Manual_Open"


self.key_btn = pygbutton.PygButton((key_start_x , key_start_y ,
    key_btn_width , key_btn_height) , key_btn_text)
self.key_btn.draw(screen)

```

Appendix C. Power Substation Environment Y-Box Communication and Interface Code

This code was used to run the Y-Box interface of the Power Substation environment. The code enables the reading of power output to the outlets as well as the voltage as it travels through the meter. The Y-Box communication engine shown in Appendix A is also used in this software. The "Indicator" code shown in Appendix B is also leveraged in this code.

3.1 Power Substation Interface and Y-Box Communication Code

This code simulates and displays power output from power substation outlets.

```
import pygame
from pygame.locals import *
import time
import _thread
import Ybox
import math
import indicator
import pygbutton

#
#####

# Author: Evan Plumley
```

```

#
# This program is the communication engine for the Y-box and
#   the GUI
# for monitoring the power substation
#
#
#
#####

class PsSim:
def __init__(self):
    self._running = True
    self.screen = None
    self.size = self.width, self.height = 1350, 970
    self.ybox = Ybox.Ybox()
    self.prisonPower = True
    self.waterPower = True
    self.homePower = True
    self.meterAlarm1Trip = False
    self.meterAlarm2Trip = False
    self.line1amps = 31.1000
    self.line2amps = 31.1000
    self.clickhigh1 = False
    self.clickhigh2 = False
    self.clickmid1 = False

```

```

self.clickmid2 = False
self.clicklow1 = False
self.clicklow2 = False
self.clickzero1 = False
self.clickzero2 = False

startAmp = 400

#send original mA to the meters
self.ybox.sendWrite(0,2,startAmp) # write meter 1
self.ybox.sendWrite(0,3,1500)

self.ybox.sendWrite(0,1,startAmp) # 2
self.ybox.sendWrite(0,4,1500)


#line trips at 40 resets at 35
#nuetral trips at 90 and resets at 85


#Initialize screen
print ("here")
pygame.init()
self.font = pygame.font.SysFont('Times', 25)
self.font2 = pygame.font.SysFont('Times', 45)

```

```

self.font3 = pygame.font.SysFont( 'Times' , 35)
self.font4 = pygame.font.SysFont( 'Times' , 15)
pygame.display.set_caption( 'Power_Substation_Monitor' )
self.screen = pygame.display.set_mode( self.size , pygame.
    HWSURFACE | pygame.DOUBLEBUF)
self.screen.fill((white))
self.screen.set_colorkey(white)

#add the pygbuttons

btn_height = 35
btn1x = 275
btn2x = 450
btn3x = 625

self.screen.blit( self.font4.render( 'Line_1_trips_at_40_resets
    _at_35.' , True, (black)), (270, 900))
self.screen.blit( self.font4.render( 'Nuetral_(Line_2)_trips_at
    _90_and_resets_at_85' , True, (black)), (270, 920))

self.screen.blit( self.font4.render( 'Line_1_Amp_Control' ,
    True, (black)), (270, 650))
self.screen.blit( self.font4.render( 'Line_2_Amp_Control' ,
    True, (black)), (450, 650))

self.line1high_btn = pygamebutton.PygButton(( btn1x, 700, 125,
    btn_height), "High:_50")

```

```
self.line2high_btn = pygbutton.PygButton(( btn2x, 700, 125,  
    btn_height), "High: 50")
```

```
self.line1mid_btn = pygbutton.PygButton(( btn1x, 750, 125,  
    btn_height), "Mid: 31.1")
```

```
self.line2mid_btn = pygbutton.PygButton(( btn2x, 750, 125,  
    btn_height), "Mid: 31.1")
```

```
self.line1low_btn = pygbutton.PygButton(( btn1x, 800, 125,  
    btn_height), "Low: 8.1")
```

```
self.line2low_btn = pygbutton.PygButton(( btn2x, 800, 125,  
    btn_height), "Low: 8.1")
```

```
self.line1zero_btn = pygbutton.PygButton(( btn1x, 850, 125,  
    btn_height), "Zero")
```

```
self.line2zero_btn = pygbutton.PygButton(( btn2x, 850, 125,  
    btn_height), "Zero")
```

```
self.line1high_btn.draw(self.screen)
```

```
self.line2high_btn.draw(self.screen)
```

```
self.line1mid_btn.draw(self.screen)
```

```
self.line2mid_btn.draw(self.screen)
```

```
self.line1low_btn.draw(self.screen)
```

```
self.line2low_btn.draw(self.screen)
```

```
self.line1zero_btn.draw(self.screen)
```

```
self.line2zero_btn.draw(self.screen)
```

```
#add PLC output title
```

```
pygame.draw.rect(self.screen, black, (450, -5, 500, 55), 2)
```

```
self.screen.blit(self.font3.render('Power_Meter_and_Relay_  
Output', True, (black)), (480, 5))
```

```
#add the meter
```

```
self.meterpos = (300,150)
```

```
self.meter = pygame.image.load("psimages/meter.png")
```

```
self.screen.blit(self.meter, self.meterpos)
```

```
pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
```

```

#add the relay

self.relaypos = (800, 100)

self.relay = pygame.image.load("psimages/relay.jpg")

self.screen.blit(self.relay, self.relaypos)

pygame.draw.rect(self.screen, light_green, (916, 237, 280,
        65), 0)

#self.rice = pygame.image.load("psimages/rice.jpg")
#self.screen.blit(self.rice, (1030,239))


#add raiden and electricity

self.raidenpos = (35, 320)

self.raidenboltpos = (151, 330)

#self.raiden = pygame.image.load("psimages/pikachu.png")

self.raiden = pygame.image.load("psimages/raiden.png")

self.screen.blit(self.raiden, self.raidenpos)

self.raidenbolt = pygame.image.load("psimages/raidenpower.png
        ")

self.screen.blit(self.raidenbolt, self.raidenboltpos)

self.power1 = pygame.image.load("psimages/power.png")

self.power2 = pygame.image.load("psimages/power.png")

self.power3 = pygame.image.load("psimages/power.png")

self.power1pos = (720, 300)

self.power2pos = (720, 340)

self.power3pos = (720, 380)

self.screen.blit(self.power1, self.power1pos)

```

```

self.screen.blit(self.power2, self.power2pos)
self.screen.blit(self.power3, self.power3pos)

#add indicator lights for power at individual stations
green_icon = pygame.image.load("psimages/greenLightAlt.png")
red_icon = pygame.image.load("psimages/redLightAlt.png")
light_icons = [green_icon, red_icon]
light_dimensions = ((100,100))
self.light1 = indicator.Indicator(self, 850, 800, light_icons
    , light_dimensions)
self.light2 = indicator.Indicator(self, 1020, 800,
    light_icons, light_dimensions)
self.light3 = indicator.Indicator(self, 1190, 800,
    light_icons, light_dimensions)

self.screen.blit(self.font3.render("Prison", True, (black)),
    (860, 900))
self.screen.blit(self.font3.render("Water", True, (black)),
    (1030, 900))
self.screen.blit(self.font3.render("Home", True, (black)),
    (1200, 900))

lightning = pygame.image.load("psimages/lightning.png")
cover = pygame.image.load("psimages/whiteCover.png")
lightning_icons = [lightning, cover]

```

```

lightning_dimensions = ((40,100))

self.lightning1 = indicator.Indicator(self , 875, 700,
    lightning_icons , lightning_dimensions)
self.lightning2 = indicator.Indicator(self , 1045, 700,
    lightning_icons , lightning_dimensions)
self.lightning3 = indicator.Indicator(self , 1210, 700,
    lightning_icons , lightning_dimensions)

#####
# add the amps to the meter
#####

pygame.draw.rect(self.screen , white , (380, 226, 265, 255), 0)
self.screen.blit(self.font3.render("Amps" , True , (black)) ,
    (460, 235))
self.screen.blit(self.font3.render("1:" , True , (black)) ,
    (400, 300))
self.screen.blit(self.font3.render("2:" , True , (black)) ,
    (400, 340))

self.screen.blit(self.font3.render("%.4f" % self.line1amps ,
    True , (black)) , (460, 300))
self.screen.blit(self.font3.render("%.4f" % self.line2amps ,
    True , (black)) , (460, 340))

```

```
pygame.display.update()
```

```
try:
    _thread.start_new_thread(timedReads, (self,))
except Exception as e:
    print(e)
    self._running = True

#Handle all events
def on_event(self, event):
    if event.type == pygame.QUIT:
        self._running = False
    else: #Determine if button was clicked
    if 'click' in self.line1high_btn.handleEvent(event):
        self.clickhigh1 = True
    if 'click' in self.line2high_btn.handleEvent(event):
        self.clickhigh2 = True

    if 'click' in self.line1mid_btn.handleEvent(event):
        self.clickmid1 = True
```

```

if 'click' in self.line2mid_btn.handleEvent(event):
    self.clickmid2 = True

if 'click' in self.line1low_btn.handleEvent(event):
    self.clicklow1 = True
if 'click' in self.line2low_btn.handleEvent(event):
    self.clicklow2 = True

if 'click' in self.line1zero_btn.handleEvent(event):
    self.clickzero1 = True
if 'click' in self.line2zero_btn.handleEvent(event):
    self.clickzero2 = True

pygame.display.update()


def on_loop(self):
    pass
def on_render(self):
    pass
def on_cleanup(self):
    pygame.quit()


#start program

```

```

def on_execute(self):
#if self.on_init() == False:
#self._running = False

while( self._running ):
for event in pygame.event.get():
    self.on_event(event)
    self.on_loop()
    self.on_render()
    self.on_cleanup()


def timedReads(self):
    past = int(round(time.time() * 1000)) #getting starting
        milisecond time to execute reads from the ybox
    meter_current = 13.0001


while True:
    present = int(round(time.time() * 1000)) #getting present
        time to comapre to past
    #check to see if 100 milliseconds have passed

```

```

if present - past >= 40:
    past = present

#####
# read the SEL power state and display
#####
prisonState = self.ybox.sendRead(1,4)
waterState = self.ybox.sendRead(1,5)
homeState = self.ybox.sendRead(1,6)

#Prison change display
if self.prisonPower == True and prisonState == ("r1,4,1"):
    self.prisonPower = False
    self.light1.change_state()
    self.lightning1.change_state()
    self.ybox.sendWrite(2,0,1)

if self.prisonPower == False and prisonState == ("r1,4,0"):
    self.prisonPower = True
    self.light1.change_state()
    self.lightning1.change_state()
    self.ybox.sendWrite(2,0,0)

#Water change display

```

```

if self.waterPower == True and waterState == ("r1,5,1"):
    self.waterPower = False
    self.light2.change_state()
    self.lightning2.change_state()
    self.ybox.sendWrite(2,1,1)

```

```

if self.waterPower == False and waterState == ("r1,5,0"):
    self.waterPower = True
    self.light2.change_state()
    self.lightning2.change_state()
    self.ybox.sendWrite(2,1,0)

```

#Home change display

```

if self.homePower == True and homeState == ("r1,6,1"):
    print(homeState)
    self.homePower = False
    self.light3.change_state()
    self.lightning3.change_state()
    self.ybox.sendWrite(2,2,1)

```

```

if self.homePower == False and homeState == ("r1,6,0"):
    self.homePower = True
    self.light3.change_state()
    self.lightning3.change_state()
    self.ybox.sendWrite(2,2,0)

```

```

#####
# Read state of the meter alarm to forward to the relay
#####
meterAlarm1 = self.ybox.sendRead(1,1)
meterAlarm2 = self.ybox.sendRead(1,2)

if meterAlarm1 == ("r1,1,1"):# and self.meterAlarm1Trip ==
    False:
self.meterAlarm1Trip == True
self.ybox.sendWrite(2,4,1)

if meterAlarm1 == ("r1,1,0"):# and self.meterAlarm1Trip ==
    True:
self.meterAlarm1Trip == False
self.ybox.sendWrite(2,4,0)

if meterAlarm2 == ("r1,2,1"):# and self.meterAlarm2Trip ==
    False:
self.meterAlarm2Trip == True
self.ybox.sendWrite(2,5,1)

```

```

if meterAlarm2 == ("r1,2,0"): #and self.meterAlarm2Trip ==
    True:
    self.meterAlarm2Trip == False
    self.ybox.sendWrite(2,5,0)


pygame.display.update()


if self.clickhigh1 == True:
    self.clickhigh1 = False
    self.line1amps = 50.00
    self.ybox.sendWrite(0,2,700) # write meter 1
    self.ybox.sendWrite(0,3,4000)
    pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
    self.screen.blit(self.font3.render("Amps", True, (black)),
        (460, 235))
    self.screen.blit(self.font3.render("1:", True, (black)),
        (400, 300))
    self.screen.blit(self.font3.render("2:", True, (black)),
        (400, 340))


    self.screen.blit(self.font3.render("%.4f" % self.line1amps,
        True, (black)), (460, 300))

```

```

self.screen.blit(self.font3.render("%4f" % self.line2amps,
    True, (black)), (460, 340))
pygame.display.update()

if self.clickhigh2 == True:
    self.clickhigh2 = False
    self.line2amps = 50.00
    self.ybox.sendWrite(0,1,700) # write meter 1
    self.ybox.sendWrite(0,4,4000)
    pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
    self.screen.blit(self.font3.render("Amps", True, (black)),
        (460, 235))
    self.screen.blit(self.font3.render("1:", True, (black)),
        (400, 300))
    self.screen.blit(self.font3.render("2:", True, (black)),
        (400, 340))

self.screen.blit(self.font3.render("%4f" % self.line1amps,
    True, (black)), (460, 300))
self.screen.blit(self.font3.render("%4f" % self.line2amps,
    True, (black)), (460, 340))
pygame.display.update()

```

```

if self.clickmid1 == True:
    self.clickmid1 = False
    self.line1amps = 31.10
    self.ybox.sendWrite(0,2,400) # write meter 1
    self.ybox.sendWrite(0,3,3000)
    pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
    self.screen.blit(self.font3.render("Amps", True, (black)),
        (460, 235))
    self.screen.blit(self.font3.render("1:", True, (black)),
        (400, 300))
    self.screen.blit(self.font3.render("2:", True, (black)),
        (400, 340))

    self.screen.blit(self.font3.render("%.4f" % self.line1amps,
        True, (black)), (460, 300))
    self.screen.blit(self.font3.render("%.4f" % self.line2amps,
        True, (black)), (460, 340))
    pygame.display.update()

if self.clickmid2 == True:
    self.clickmid2 = False
    self.line2amps = 31.10
    self.ybox.sendWrite(0,1,400) # write meter 1

```

```

self.ybox.sendWrite(0,4,3000)
pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
self.screen.blit(self.font3.render("Amps", True, (black)),
    (460, 235))
self.screen.blit(self.font3.render("1:", True, (black)),
    (400, 300))
self.screen.blit(self.font3.render("2:", True, (black)),
    (400, 340))

self.screen.blit(self.font3.render("%.4f" % self.line1amps,
    True, (black)), (460, 300))
self.screen.blit(self.font3.render("%.4f" % self.line2amps,
    True, (black)), (460, 340))
pygame.display.update()

if self.clicklow1 == True:
    self.clicklow1 = False
    self.line1amps = 8.10
    self.ybox.sendWrite(0,2,100) # write meter 1
    self.ybox.sendWrite(0,3,1500)
    pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
    self.screen.blit(self.font3.render("Amps", True, (black)),
        (460, 235))

```

```

self.screen.blit(self.font3.render("1:", True, (black)),
    (400, 300))
self.screen.blit(self.font3.render("2:", True, (black)),
    (400, 340))

self.screen.blit(self.font3.render("%.4f" % self.line1amps,
    True, (black)), (460, 300))
self.screen.blit(self.font3.render("%.4f" % self.line2amps,
    True, (black)), (460, 340))
pygame.display.update()

if self.clicklow2 == True:
    self.clicklow2 = False
    self.line2amps = 8.10
    self.ybox.sendWrite(0,1,100) # write meter 1
    self.ybox.sendWrite(0,4,1500)
    pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
    self.screen.blit(self.font3.render("Amps", True, (black)),
        (460, 235))
    self.screen.blit(self.font3.render("1:", True, (black)),
        (400, 300))
    self.screen.blit(self.font3.render("2:", True, (black)),
        (400, 340))

self.screen.blit(self.font3.render("%.4f" % self.line1amps,
    True, (black)), (460, 300))

```

```

self.screen.blit(self.font3.render("%.4f" % self.line2amps,
    True, (black)), (460, 340))
pygame.display.update()

if self.clickzero1 == True:
    self.clickzero1 = False
    self.line1amps = 0
    self.ybox.sendWrite(0,2,0) # write meter 1
    self.ybox.sendWrite(0,3,0)
    pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
    self.screen.blit(self.font3.render("Amps", True, (black)),
        (460, 235))
    self.screen.blit(self.font3.render("1:", True, (black)),
        (400, 300))
    self.screen.blit(self.font3.render("2:", True, (black)),
        (400, 340))

    self.screen.blit(self.font3.render("%.4f" % self.line1amps,
        True, (black)), (460, 300))
    self.screen.blit(self.font3.render("%.4f" % self.line2amps,
        True, (black)), (460, 340))
    pygame.display.update()

if self.clickzero2 == True:
    self.clickzero2 = False

```

```

self.line2amps = 0
self.ybox.sendWrite(0,1,0) # write meter 1
self.ybox.sendWrite(0,4,0)
pygame.draw.rect(self.screen, white, (380, 226, 265, 255), 0)
self.screen.blit(self.font3.render("Amps", True, (black)),
    (460, 235))
self.screen.blit(self.font3.render("1:", True, (black)),
    (400, 300))
self.screen.blit(self.font3.render("2:", True, (black)),
    (400, 340))

self.screen.blit(self.font3.render("%.4f" % self.line1amps,
    True, (black)), (460, 300))
self.screen.blit(self.font3.render("%.4f" % self.line2amps,
    True, (black)), (460, 340))
pygame.display.update()

if __name__ == "__main__" :

red = (225,0,0)
white = (255, 255, 255)

```

```
black = (0,0,0)
grey = (200, 200, 200)
dark_grey = (140, 140, 140)
blue = (130, 130, 255)
dark_blue = (0, 0, 150)
green = (0,240,0)
dark_green = (0,175,0)
light_green = (30,210,30)
```

```
psSim = PsSim()
psSim.on_execute()
```

REPORT DOCUMENTATION PAGE					<i>Form Approved</i> OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 23-03-2017		2. REPORT TYPE Master's Thesis			3. DATES COVERED (From — To) Sept 2015 — Mar 2017	
4. TITLE AND SUBTITLE A Framework for Categorization of Industrial Control System Cyber Training Environments				5a. CONTRACT NUMBER		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Evan G. Plumley, First Lieutenant, USA				5d. PROJECT NUMBER 17G310		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765					8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-17-M-059	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Homeland Security ICS-CERT POC: Neil Hershfield, DHS ICS-CERT ATTN: NPPD/CSC/NCSD/US-CERT Mailstop: 0635 245 Murray Lane, SW, Bldg 410, Washington, DC 20528 Email: ics-cert@dhs.gov phone: 1-877-776-7585					10. SPONSOR/MONITOR'S ACRONYM(S) DHS ICS-CERT	
					11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.						
13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.						
14. ABSTRACT First responders and professionals in hazardous occupations undergo training and evaluations for the purpose of mitigating risk and damage. For example, helicopter pilots train with multiple categorized simulations that increase in complexity before flying a real aircraft. However in the industrial control cyber incident response domain, where incident response professionals help detect, respond and recover from cyber incidents, no official categorization of training environments exist. To address this gap, this thesis provides a categorization of industrial control training environments based on realism. Four levels of environments are proposed and mapped to Bloom's Taxonomy. This categorization will help organizations determine which training environment best aligns with their training needs and budgets.						
15. SUBJECT TERMS Industrial control systems, cyber incident response, cyber defense training, cyber training environments						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Barry E. Mullins, AFIT/ENG	
U	U	U	U	161	19b. TELEPHONE NUMBER (include area code) (937) 255-3636 x7979; barry.mullins@afit.edu	