

Improving Usable Security and System Safety

by

Christopher W. Perr

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 2, 2014

Keywords: Usability, Security, Systems Engineering, Fault Tree Analysis, Architecture,
Static Code Analysis, Software Engineering, SCADA

Copyright 2014 by Christopher W. Perr

Approved by

Jerry Davis, Chair, Associate Professor of Industrial and Systems Engineering
John A. Hamilton, Jr., Co-Chair, Alumni Professor of Computer Science
Richard Seseck, Assistant Professor of Industrial and Systems Engineering
David Umphress, Associate Professor of Computer Science

Abstract

Usability in information technology systems plays of vital role in conducting operations safely and securely. The methods for judging usability vary greatly, which makes assessing any aspect of usability difficult. In cases of either safety or security, the inability to identify usability shortcomings can be costly. This has been shown to be especially true for the Supervisory Control and Data Acquisition (SCADA) systems responsible for controlling national infrastructure.

The literature reveals a gap in analysis methods for the identification of unsafe system states. The manner of entering these states is typically referred to as a misconfiguration, and may be accidental or intentional. Code analysis methods are used to create abstract representations of source code and binaries. The generated representations are used to indicate a level of correctness in regards to coding practices. However, there is no method currently being used for ladder logic written for SCADA systems which indicates correctness beyond identifying code which will fail to run correctly.

In addressing this shortcoming, a static code analysis method was developed for the generation of abstract representations of possible system states, namely Fault Trees. Inputs into ladder logic are treated as initiating events, and the pathways through a system are mapped with the possible end states emphasized for further analysis. In this way inputs to a system can be looked at in regards to the possible states which the operator can place the system in. If the undesirable states are identified, engineering methods can be applied to mitigate or remove the undesired state. This may improve both security and usability in a system.

To test the effectiveness of the static code analysis method created, a usability experiment was conducted. A model SCADA pipeline was created based on case studies and

pipeline accident reports from the National Transportation and Safety Board. Twenty five test subjects operated the model pipeline in mock critical operating conditions. The model was first programmed using a simple ladder logic program for control. This program underwent the code analysis method studied, and was reconfigured to correct for the possibly unsafe system states discovered. Users were asked to reconfigure pipeline flow using both control programs to drive the model. Test subjects were then asked to ‘attack’ the model with both the simple program and the program that had undergone code analysis. This was done to more deliberately test the strength of the method, and to explore the relationship between usability and security.

From this study it was shown that usability improvements, in relation to the model tested, could be made by identifying unsafe system states by using the code analysis method proposed. The number of accidental user misconfigurations resulting in an alarm condition and intentional user misconfigurations resulting in an alarm condition was significantly reduced. It is believed that the method described shows promise as a means for conducting code analysis for the improvement of both usability and security in SCADA systems.

Acknowledgments

I have a lot of people to thank for their help.

- Special thanks to my beautiful bride, Jacquelyn Perr. She was the one to tell me to go for it, and has been with me for my successes and my failures.
- Thank you to both sets of parents. Wayne and Ruth Ann, thank you for being the best parents I could ask for. Gary and Robin, thank you for supporting Jackie and I along the way.
- Thank you to my sister, Jeannette, for telling me what needed to be done and that I could do it. Thank you to Alex and Danny for the much needed breaks.
- Thank you to my lab mates for being an endless source of proofreaders, sounding boards, and programming help. Thank you Alex, Ben, Chris, Devin, Greg, Jim, Pat and Sara.
- Thank you to Dr. John A. Hamilton Jr. for bringing me to Auburn, funding me, and mentoring me along the way. I would not have gotten to Auburn without him.
- Thank you to Dr. Jerry Davis. I would not be finishing at Auburn without him.
- Thank you to Dr. David Umphress. Dr. Umphress always left his office door open to me. A privilege I took advantage of. I would not have been able to stay Auburn without his help.
- Thank you to Dr. Richard Sesek. Dr. Sesek was a constant supporter, and I truly appreciate all the time he gave me.
- Thank you to Kyle Jennings for all the help with my statistics questions.

- Thank you to Dr. Casey Cegielski for taking the time to serve as outside reader.
- Thanks to Indusoft for sharing an academic license of Indusoft WebStudio with me.
- Thanks to Isograph for sharing an academic license of Reliability+ software with me.

Table of Contents

Abstract	ii
Acknowledgments	iv
List of Figures	ix
List of Tables	xii
Acronyms, Terminology and Abbreviations	xiii
1 Introduction	1
1.1 Research Objectives	3
1.2 Research and Dissertation Organization	3
2 A Review of the Literature on Security, Safety, and Usability in SCADA Systems	5
2.1 Introduction	5
2.2 Introduction to SCADA	7
2.2.1 SCADA Concerns and Current Best Practices	10
2.3 Code and Failure Analysis	17
2.4 Limitations of the Existing Research	20
2.4.1 Incorporating Code Analysis and Safety Engineering Tools for Improving Usable Security in SCADA	21
3 Developing a Code Behavior Analysis Method to Identify Possible Misconfiguration Errors in SCADA	22
3.1 Abstract	22
3.2 Introduction	22
3.3 Methods	23
3.3.1 Objective	23
3.4 Development	24

3.4.1	Method Development	27
3.5	Results	38
3.6	Discussion	38
3.7	Conclusion	39
4	Developing Experiment	40
4.1	Abstract	40
4.2	Introduction	41
4.3	Methods	42
4.3.1	Objective and Hypothesis	42
4.3.2	Experimental Design	43
4.3.3	Subjects	52
4.3.4	Experimental Apparatus	52
4.3.5	Protocol	54
4.4	Results	59
4.4.1	Time to Completion Difference for Users	59
4.4.2	Errors Made by Users	60
4.4.3	Successful Attacks by Attackers	61
4.5	Discussion	62
4.6	Conclusion	64
5	Conclusion	65
5.1	Introduction	65
5.2	Summary of Findings	65
5.3	Limitations of Research	66
5.4	Recommendations for Future Research	68
	Bibliography	69
6	Appendices	75
6.1	IRB	75

6.1.1	IRB Approved Consent Form and Recruitment Documents	75
6.1.2	IRB Research Protocol Review Form and Submitted Documents . . .	79
6.2	Indusoft Educational Agreement	94
6.3	Test Data	100
6.4	Fault Tree Generation	101
6.4.1	Sinks and Sources Permutations	101
6.5	Test Hardware	104
6.6	Screenshots	107
6.7	Source Code	112
6.7.1	VisualBasic Engine for InduSoft Web Studio 1	115
6.7.2	Test Reconfiguration Visual Basic Script for InduSoft Web Studio . .	163

List of Figures

1.1	Venn Diagram for Safety, Security, and Usability	3
2.1	Example SCADA Architecture	7
2.2	Basic PLC Operation	9
2.3	Example Ladder Logic Program	9
2.4	Example Verilog Program	10
3.1	Example PLC Operation	24
3.2	RSLogix Micro Screen Shot	26
3.3	Example Door Interlock	27
3.4	Interlock Ladder Logic Program	28
3.5	Interlock Verilog Pseudocode	28
3.6	Interlock Program File	29
3.7	Interlock Fault Tree Diagram	30
3.8	Interlock State Diagram	31
3.9	Improved Interlock Ladder Logic Program	31
3.10	Improved Interlock Program File	32

3.11	Improved Interlock Fault Tree Diagram	33
3.12	Improved Interlock State Diagram	34
3.13	Override Interlock Ladder Logic Program	35
3.14	Override on Improved Interlock Program File	35
3.15	Override Interlock Fault Tree Diagram	36
3.16	Simplified Override Interlock Fault Tree Diagram	37
3.17	Override Interlock State Diagram	37
4.1	Pipeline Usable Security Simulator	46
4.2	Unimproved Pipeline Ladder Logic	49
4.3	Pipeline 1 Fault Tree Example	51
4.4	Pump 1 Control Rung	52
4.5	IOgraph Mouse Tracking Overlay	54
4.6	Main Menu	55
4.7	Tutorial Screen	56
4.8	Experiment Screen	58
6.1	Raspberry Pi with PiFace	104
6.2	MicroLogix 1000	105
6.3	Test Computer	106

6.4 RSLogix Running Simple Test PLC Code 107

6.5 Large Version Pipeline Usable Security Simulator 108

6.6 IOgraph Mouse Tracking Overlay 1 109

6.7 IOgraph Mouse Tracking Overlay 2 110

6.8 IOgraph Mouse Tracking Overlay 3 111

6.9 Improved Ladder Logic for Pipeline Model Part 1 112

6.10 Improved Ladder Logic for Pipeline Model Part 2 113

List of Tables

2.1	Comparison of Security Focus in Information Technology Systems and ICSs . . .	12
3.1	Cut Set for Unimproved Interlock Fault Tree	30
3.2	Cut Set for Override Interlock Fault Tree	37
4.1	Cut Set for Pipeline 1	51
4.2	Descriptive Statistics for Time to Completion on Each Test in Seconds	60
4.3	Descriptive Statistics for Time to Completion on Each Test in Seconds: No Outlier	60
4.4	Descriptive Statistics for the Errors in Each Test Based on Test Subject	61
4.5	Successful Attacks Made Per Test	62
6.1	Number of Errors for Each Test Organized by Subject	100

Acronyms, Terminology and Abbreviations

Black Hat Hacker : a hacker who ‘violates computer security for little reason beyond maliciousness or for personal gain’ [Moore 2010].

CERT CC: Cyber Emergency Response Team Coordination Center

CIP : Critical Infrastructure Protection

Code Smell : A symptom in the source code of a program that possibly indicated a deeper problem.

COTS : Commercial Off-the-Shelf Components

DHS : Department of Homeland Security

DPCD : Digital Cyber Protection Control Device

ETA : Event Tree Analysis

FMEA : Failure Modes Effects Analysis

FTA : Fault Tree Analysis

GAO : General Accounting Office

Grey Hat Hacker : a hacker whose activities fall somewhere between white and black hat hackers in a variety of practices [Moore 2010].

GUI : Graphical User Interface

HDL : Hardware Description Language

HMI : Human Machine Interface

I/O : Input and output

ICS : Industrial Control System

ICS-CERT : Industrial Control System Computer Emergency Response Team

NCSD : National Cyber Security Division

NERC : North American Electric Reliability Corporation

Pigging : the practice of using devices known as pigs to perform various maintenance operations on a pipeline.

PLC : Programmable Logic Controller

SCADA : Supervisory Control and Data Acquisition

UML : Unified Modeling Language

White Hat Hacker : a ethical computer hacker, or a ‘computer security expert’, who specializes in penetration testing [Moore 2010].

Chapter 1

Introduction

“The most destructive scenarios involve cyber actors launching several attacks on our critical infrastructure at one time . . . attackers could also seek to disable or degrade critical military systems and communication networks. The collective result of these kinds of attacks could be a cyber Pearl Harbor. ” - Leon Panetta
[Panetta 2012]

In March of 2007, Idaho National Laboratory conducted a test for the Department of Homeland Security (DHS). This test simulated what could happen if an attacker was able to take control of a power generation turbine. The purpose was to investigate system vulnerabilities on Supervisory Control and Data Acquisition (SCADA) systems within utility companies. The test is now commonly referred to as the Aurora Generator Test.

In the Aurora Generator Test, researchers were able to gain access to the SCADA system controlling a power generator turbine. With access to the controls, researchers were able to open a circuit breaker, wait for the generator to slip out of synchronism with the load, and close the breaker [Zeller 2011]. This caused the power generation turbine to enter an out-of-phase condition, resulting in violent physical and magnetic stresses that shut down the generator and caused severe damage.

In 2005, two years before the Aurora Generator Test took place, a researcher named Charles Mozina at Beckwith Electric was looking at the same problem from a different perspective. Mozina described a problem where a circuit breaker could be accidentally opened and closed at exactly the wrong moment. He called the mistake a multi-phase generator fault [Mozina 2005]. Mozina’s paper described several other undesirable situations where operators are able to accidentally close breakers and cause damage to generators. The

multi-phase generator fault Mozina described is commonly referred to as a ‘million-dollar mistake’ [Weiss 2014]. After the Aurora Generator Test, Joe Weiss, a managing partner at Applied Control Solutions, explained the basic tenets behind the attack.

“Aurora exploits basic physics. It is a basic tenet of power engineering that you do not close a breaker with the grid out of phase. Creating an out-of-phase condition and starting equipment (intentionally or unintentionally - doesn’t have to be malicious) will create significant torque on the equipment that can cause substantial hardware damage.” [Weiss 2014]

The Aurora Generator Test exposes how a safety and usability issue could be exploited by an attacker, at which point the safety and usability issue becomes a security vulnerability. In this case, the solution to both problems was to insert a Digital Cyber Protection Control Device (DPCD) between the substation and the load. The DPCD detects an out of phase condition event and isolates the substation before the demand of the grid can be applied to the equipment [Swearingen et al. 2014]. The NERC CIP-002, a standard provided by the North American Electric Reliability Corporation (NERC) for Critical Infrastructure Protection (CIP), requires this protection for all generators deemed part of the nation’s critical infrastructure [NERC 2008]. If this solution, which is now required because of the Aurora Generator test, had been required previously, the security vulnerability may have been avoided and the usability and safety issues resolved.

The tightly-coupled nature which exists between usability, safety, and security in SCADA systems is a factor that this research seeks to explore. This tightly-coupled nature, as illustrated in the Venn diagram in Figure 1.1, suggests that if a vulnerability can be detected and mitigated (or repaired), then in some cases all three problems can be at least partially solved (**X**). It does not matter if the flaw is discovered due to a safety inspection, a usability walkthrough, or a security audit.

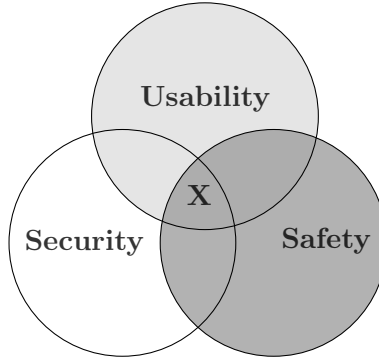


Figure 1.1: Venn Diagram for Safety, Security, and Usability

1.1 Research Objectives

The literature, as reported in Chapter 2, reveals a gap in the incorporation of behavioral static code analysis in identifying misconfiguration errors and unsafe system states. While computer security researchers are exploring code analysis as a means to provide program behavior, program behavior and static code analysis is typically ignored in the development of Industrial Control Systems (ICS). This is especially true in regards to user error prevention. The identification of unsafe system states is critical in preventing possible user misconfiguration errors and security vulnerabilities. ICSs currently struggle with reliance on commercial off-shelf-components (COTS) and increasing complexity. This is a problem for which code analysis is being developed [Brumley et al. 2011]. Thus, the objective of this research is to bridge the gap between code analysis and safety engineering. By using static code analysis as a means for providing accurate program behavior as an input to system safety engineering tools, unsafe system states may be better identified and subsequently improve system security and reduce misconfiguration errors.

1.2 Research and Dissertation Organization

The chapters of this dissertation are organized according to the ACM publication format as well as the Auburn University dissertation guide. The dissertation is comprised of five

chapters. Chapter One is a traditional introduction, and Chapter Five is a traditional conclusion. Chapter Two is a comprehensive review of the literature on Security, Safety, and Usability in SCADA Systems. Chapter Three is the explanation and development of a method for exploring the application of static code analysis to identify usability issues which contribute to insecure and unsafe system states. Chapter Four reports on the application of this method to a SCADA system modeled on historical NTSB scenarios, and the usability experiment conducted to test the static code analysis method developed.

Chapter 2

A Review of the Literature on Security, Safety, and Usability in SCADA Systems

“The Internet and all the systems we build today are getting more complex at a rate that is faster than we are capable of matching. So while security in reality is actually improving [...] the target is constantly shifting and as complexity grows, we are losing ground.” -Bruce Schneier [Chan 2012]

2.1 Introduction

On 8 November 2011, a possible cyber attack was reported at a water treatment plant in Springfield, Illinois. It was difficult to tell what happened. There was a remote access in the networking logs from a foreign IP address. A pump had been continually cycled on and off until it had failed. When this information was leaked to the media, the Department of Homeland Security responded with the following statement:

“At this time there is no credible corroborated data that indicates a risk to critical infrastructure entities or a threat to public safety.” [Finkle 2012].

‘pr0f’, a grey hat hacker, was incensed at the lack of attention being given to this problem. To prove a point, pr0f made a series of posts to pastebin (<http://pastebin.com/Wx90LLum>) and twitter (https://twitter.com/pr0f_srs). While all of the posts highlighted vulnerabilities in public Industrial Control Systems (ICSs), one contained a screenshot showing access to the Human Machine Interface (HMI) for the South Houston, Virginia Water Treatment Plant. pr0f had shown how easy it was to gain access to these systems. [CNET 2014]

“As for how I did it, it’s usually a combination of poor configuration of services, bad password choice, and no restrictions on who can access the interfaces.”

[CNET 2014]

On 23 November 2011, the Industrial Control System Computer Emergency Response Team (ICS-CERT) released a one page report which contained the following statement regarding the initial incident in Illinois.

“After detailed analysis of all available data, ICS-CERT and the FBI found no evidence of a cyber intrusion into the SCADA system of the Curran-Gardner Public Water District in Springfield, Illinois.” [CERT 2014]

prof, in a very public manner, had shown that public ICSs are vulnerable. As an individual prof had discovered, exploited, and released vulnerabilities to multiple public SCADA systems. He had also done this while publicly admitting that he does not work with SCADA systems or in computer security, and is not funded to do this work. This raised the unsettling questions as to why the United States had not yet experienced a major recorded cyber attack on the SCADA systems in charge of critical national infrastructure.

Retired Army General Keith Alexander gave one possible explanation in 2012 when serving as the head of the National Security Agency.

“They’re practicing.” [Bloomberg 2014]

James Lewis, a cybersecurity fellow at the Center for Strategic and International Studies, provided the following commentary on possible protections to our national infrastructure.

“You can engage intelligence operations to try to judge their [possible attackers] capabilities and intent, or fall upon your knees and beg critical infrastructure to make themselves a harder target, and we’re doing both.” [Bloomberg 2014]

The goal of this dissertation is to develop a method which addresses the second path which James Lewis offered. How can critical infrastructure be made a harder target? First, a basic understanding of and research regarding SCADA systems and security is required.

2.2 Introduction to SCADA

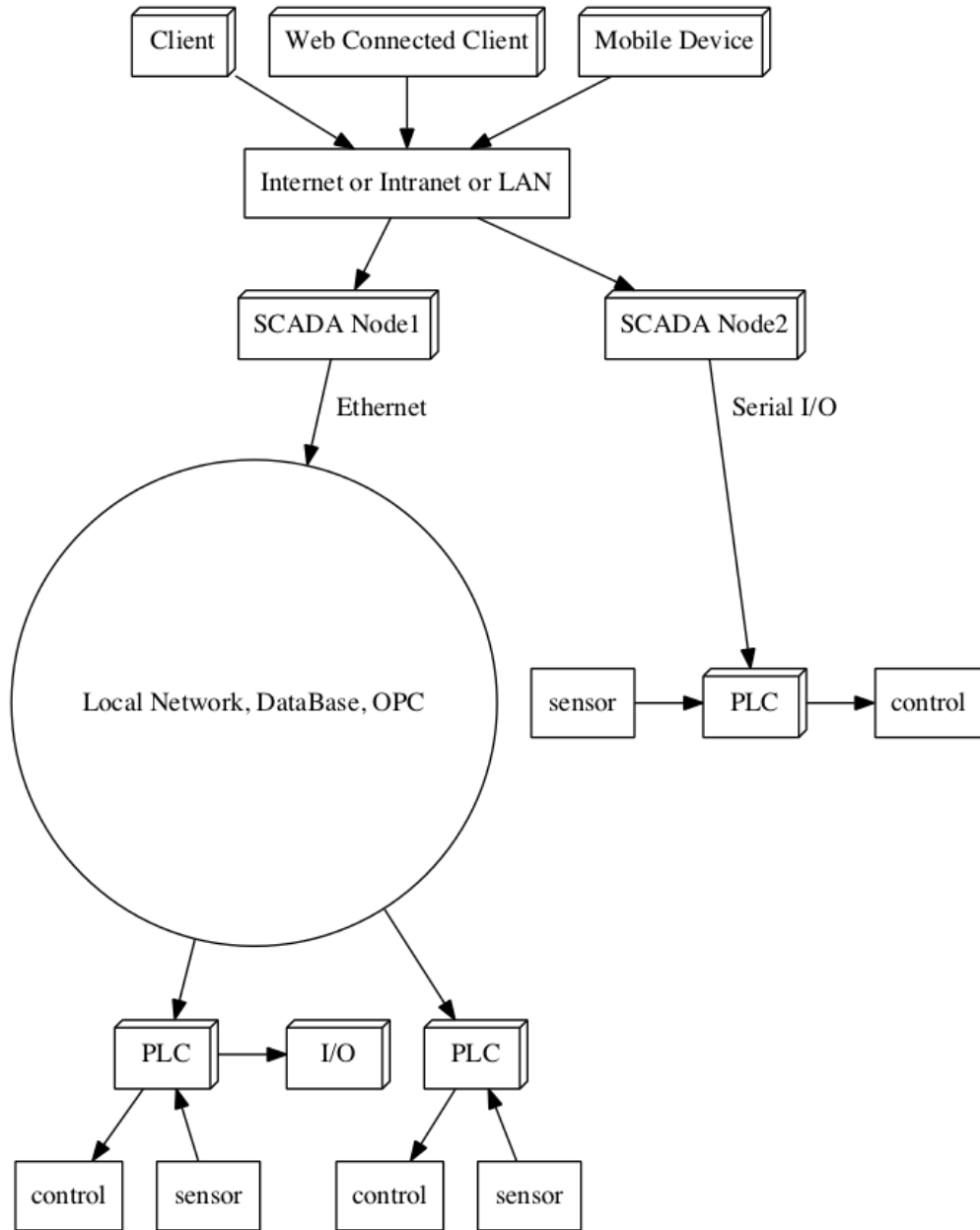


Figure 2.1: Example SCADA Architecture

As this dissertation is concerned with identifying usable security issues in Supervisory Control and Data Acquisition Systems (SCADA) systems, it is important to first develop the vocabulary and a basic understanding of SCADA. SCADA is used to monitor and control

large scale processes. SCADA systems consist of highly specialized computer systems that control critical infrastructure.

Figure 2.1 provides an example architecture for a SCADA system. The top of the figure is the client level, which can include devices ranging from laptops to cell phones. The client is a software product running on the device which typically consists of a Human Machine Interface (HMI) and the means for connecting to the SCADA Node. The connection to the SCADA Node may be via Internet, Intranet, Serial Connection or via the Local Area Network (LAN). [Broadwin 2014]

A SCADA Node communicates in real-time with the automation equipment and control systems via Serial, Ethernet or proprietary communications (commercial drivers). The SCADA Node re-transmits data between the automation hardware and the clients. In this manner, the SCADA Node acts as a SCADA ‘server’. The SCADA Node also communicates with a database, or Object Linking and Embedding for Process Control (OPC) server.[Broadwin 2014]

Daneels provides the following definition of a SCADA system which helps to address the lowest level component of the SCADA system, the Programmable Logic Controller (PLC):

“SCADA stands for Supervisory Control And Data Acquisition. As the name indicates, it is not a full control system, but rather focuses on the supervisory level. As such, it is purely a software package that is positioned on top of hardware to which it is interfaced, in general via Programmable Logic Controllers (PLCs)" [Daneels and Salter 1999].

A PLC is a specialized microcontroller that uses programmable memory to store instructions and to implement functions such as logic, sequencing, timing, counting and arithmetic to control machines and processes [Bolton 2009]. It is the microcontroller which interfaces with the sensors and controls for the task being performed. Figure 2.2 is a graphical representation of how a PLC operates.

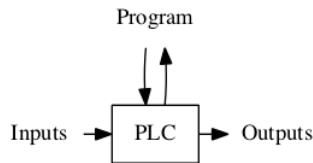


Figure 2.2: Basic PLC Operation

The program on a PLC is typically written using either Ladder Logic or a Hardware Description Language (HDL), and the program is transferred to the PLC using either software provided by the manufacturer or a specialized handheld programming tool.

Ladder logic is primarily used by PLC manufacturers in the United States. Ladder logic was originally a written method to document the design and construction of relay racks, which are nineteen inch wide racks originally used to hold railroad signaling equipment or telecommunication relays. Since then, ladder logic has become a programming language which represents a program using a graphical diagram similar to circuit diagrams. The name, ladder logic, comes from the observation that these programs resemble the vertical rails and horizontal rungs of a ladder. Figure 2.3 is an example ladder logic program which turns a control on or off via a switch. Ladder logic is well suited to control scenarios where binary values are concerned, and where interlocking and sequencing is the primary control problem. Due to the nature of how ladder logic programs are executed, race conditions are possible. [Erickson 1996] A race condition is a hazard caused when the sequence or timing of an action is critical to the output, but the sequencing or timing may not be controllable. Typically, PLCs are programmed and operate under strict constraints to avoid race conditions.

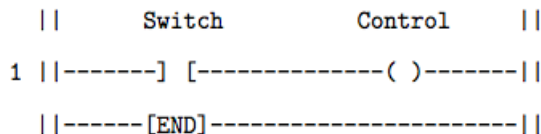


Figure 2.3: Example Ladder Logic Program

In contrast, Hardware Description Languages (HDLs) are specialized computer languages used to program the structure, design and operation of electronic circuits, as well as digital logic circuits. HDLs look more like a typical programming language as HDLs are a textual description consisting of expressions, statements and control structures. One important difference between most programming languages and HDLs is that HDLs explicitly include the notion of time. Verilog, a common HDL standardized as IEEE 1364, is the example HDL used in this dissertation. Figure 2.4 is a translation of Figure 2.3 into Verilog.

```
always@(*) Control = Switch
```

Figure 2.4: Example Verilog Program

With the common vocabulary and basic information established regarding SCADA systems, it is now possible to discuss the problems facing SCADA systems and the research being conducted in regards to SCADA security and usability.

2.2.1 SCADA Concerns and Current Best Practices

SCADA Security Concerns

Early communication in SCADA systems began using serial networks such as the RS-232 communication standard, which is still used today. These standards only support minimal functionality with little or no attention to security [NRC 2002]. Messages are sent clear text and are accepted without authentication [NRC 2002]. Meanwhile, SCADA systems have been connected to newer communication mediums such as Ethernet, wireless, shared lines, and the public Internet, meaning that SCADA systems are no longer isolated [NRC 2002]. In addition, SCADA systems are now largely dependent on Commercial Off-the-Shelf Components (COTS) which expose SCADA systems to known threats to those components [DHS 2009]. Further, the high cost and fragility of SCADA systems makes it expensive and unlikely that the systems in use, and communication standards, will be updated [DHS 2009].

In response to the threat caused from the lack of authentication and reliance on COTS [Falco et al. 2002] [Fernandez and Fernandez 2005] [Hildick-Smith 2005] the President’s Critical Infrastructure Protection Board, the United Kingdom National Infrastructure Security Coordination Centre, the Chemical Industry Data Exchange, and the General Accounting Office (GAO) have all developed their own security recommendations [Ralston et al. 2007]. Given the high cost of updating and the previously isolated nature of SCADA systems, industry was slow to respond to pressures to change [Ralston et al. 2007].

Table 2.1, adapted from a 2009 report from the Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) at DHS, helps to briefly layout the major security focuses in SCADA [DHS 2009].

Security Topic	Information Technology (IT)	Industrial Control Systems (ICS)
Antivirus and Mobile Code	Very common; easily deployed and updated	Can be very difficult due to impact on ICS; legacy systems cannot be fixed easily
Patch Management	Easily defined; enterprise wide remote and automated	Very long runway to successful patch install; OEM specific; may impact performance
Technology Support Lifetime (Outsourcing)	2-3 years; multiple vendors ubiquitous support	10 - 20 years; same vendor
Cyber security Testing and Audit (Methods)	Uses modern methods	Testing has to be tuned to the system; modern methods inappropriate for ICS; fragile equipment breaks
Change Management	Regular and scheduled; aligned with minimum use periods	Strategic scheduling; non-trivial process due to impact
Asset Classification	Common practice and done annually; results drive cyber security expenditure	Only performed when obligated; critical asset protection associated with budget costs
Incident Response and Forensics	Easily developed and deployed; some regulatory requirements; embedded in technology	Uncommon beyond system resumption activities; no forensics beyond event recreation
Physical and Environmental Security	Poor (office systems) to excellent (critical operations systems)	Excellent (operations centers, guards, gates and guns)
Secure Systems Development	Integral part of development process	Usually not an integral part of systems development
Security Compliance	Limited regulatory oversight	Specific regulatory guidance (some sectors)

Table 2.1: Comparison of Security Focus in Information Technology Systems and ICSs

Currently the recommended defense for these issues is adapting a Defense in Depth Strategy [DHS 2009] [CERT/CC 2013], which may prove unsustainable and inadequate to the problem [Small 2011].

SCADA Usability and Safety Concerns

Security is a growing concern in SCADA systems. At the same time, safety programs and accident prevention maintain a level of extreme importance. SCADA systems are responsible for critical national infrastructure where there can be almost no interruption in service. When industrial scale systems fail, the cost can be severe. The release of a cloud of methyl isocyanate in Bhopal, India is one example of operator error that led to over 2,000 fatalities [Mank 1992]. Safety programs and practices represent a unique area of application in that the methods proposed have been tried and tested for much longer than modern IT security programs [Leveson 2011] [Mil Std 882 2002].

‘Accidents happen’ is a frequently used mantra, and in many cases the explanation is recorded as ‘operator error’. There is a tendency to believe that errors cannot be avoided, and are just part of having people in the system. The following passage comes from Nancy Leveson’s book, *Engineering a Safer World*, and discusses the time period when the engineering community first began to question if operator error could be prevented and accidents avoided. This opened the door to looking at ways to make systems safer by including human factors considerations [Leveson 2011].

“The tendency to blame the operator is not simply a nineteenth century problem because it persists today. After World War II, the Air Force had serious problems with aircraft accidents; for example, from 1952 to 1966, 7,715 aircraft were lost and 8,547 people killed. Most of these accidents were blamed on pilots. Some aerospace engineers in the 1950’s did not believe the cause was so simple

and argued that safety must be designed and built into aircraft just like performance, stability, and structural integrity are built in. Although a few seminars were conducted and papers written about this approach, the Air Force did not take it seriously until they began to develop intercontinental ballistic missiles: there were no pilots to blame for the frequent and devastating explosions of these liquid-propellant missiles." [Leveson 2011]

In addressing operator error in both safety and security situations researchers look to understand what makes a system usable. Alma Whitten stated what she saw as a shortcoming in addressing this concern in her 2004 dissertation.

"...there was little research directly investigating usability for security." [Whitten 2004]

During this period of time, multiple researchers worked to address the problem of usable security, mainly focusing on adjusting the user interface to be more intuitive and to follow better design practices [Whitten 2004] [Zurko and Simon 1996] [Garfinkel 2005] [Wixon and Wilson 1997]. Whitten references a paper by Anderson on *Why Cryptosystems Fail*. Much of the early work in usability and computer security, including Whitten's work, was focused on cryptography and usability [Whitten and Tygar 1999], seemingly because of the strong relationship with cryptography and security. Recent research in the industrial control realm has applied the same design improvement thinking to address safety concerns, and has demonstrated some success [Ikuma 2013].

The problem, seemingly, is that usability engineering does not necessarily prevent users from making mistakes in either security or safety instances. The mistakes are still allowed. The hope is simply that users will make fewer mistakes because they better understand their actions and the consequences [Nielsen 1994]. The concept of foreseeable misuse addresses this overall shortcoming in regards to safety engineering as a tool to address safety, usability,

and security. Usually, foreseeable misuse is referred to in the sense of ‘reasonably’ foreseeable misuse which is a legal term.

Foreseeable That which is credibly reasonable to expect, not merely what might conceivably occur [Black and Garner 1999].

Misuse Erroneous, improper, or unorthodox [Simpson et al. 1989].

Foreseeable Misuse Unorthodox usage which is objectively reasonable to expect.

Foreseeable misuse helps to highlight the problem with commonly practiced usability testing for error prevention as it stands. The tenets of usability testing are generic, and do not reliably detect that which is ‘objectively reasonable to expect’ [Dix 2004]. Whitten makes this same point in her dissertation when she argues for determining possible user actions before a usability test, ‘well-ahead-of-time’, instead of the current model which practices ‘just-in-time’ user guidance. [Collins et al. 1997] [Whitten 2004]

Greer makes the point that security improvements could hinder proper system function, and notes that improperly implemented security is a typical reason for security failures as employees will typically circumvent security in such situations [Geer 2006]. This is an issue already raised by researchers in more information technology based approaches to usability and security [Sasse 2003] [Adams and Sasse 1999] [Adams and Sasse 2001]. Safety researchers seemingly have a better methodology on how to handle this problem. First, a ‘hierarchy of controls’ is applied to safety problems when found via an engineering method. This method is not present in computer security. The safety hierarchy of controls is:

1. Elimination : Removing the hazard.
2. Substitution : Replacing a hazard.
3. Engineered Controls : Isolation from a hazard.
4. Administrative Controls : Change how people work. (example: training)

5. Personal Protective Equipment : Provide backup protection from a hazard.

[Mannan and Lees 2005]

Identifying issues in a complex system can be difficult and, as such, requires more complex engineering approaches. One of those methods is Fault Tree Analysis (FTA). DeLong provides us with a basic breakdown of FTA [DeLong 1970]. Clemens additionally provides well organized materials which include the strengths and limitations of FTA [Clemens 2002]. Butler, while working for NASA, created a Fault Tree Compiler. The Fault Tree Compiler was a program designed to predict top event probability for a given fault tree [Martensen and Butler 1987]. Today, multiple commercial and open source tools exist which compute predicted events from a fault tree [Isograph 2014] [Auvation 2014] [ItemSoftware 2014]. In 1992, Leveson wrote a paper on how to automate safety verification of programs written in the Ada programming language, the main conclusion being that this technique worked well in analyzing systems with a strong cyberphysical interaction. The success of this method was also hypothesized to have been tied to the engineers viewing their programs in a different manner [Leveson et al. 1991]. Winter tested a similar method on a automated control system device with similar results [Winter 1995]. Both authors stated that the practice worked best as a additional tool to existing code inspection methods and should not be used as the sole method for possible fault identification.

There are several sets of tools in the same family as FTA relevant to this discussion. Event Tree Analysis (ETA) differs from FTA in that ETA is a forward, bottom up, logical modeling technique for both success and failure that explores responses through a single initiating event and lays a path for assessing probabilities of the outcomes and overall system analysis [Clemens and Simmons 1998]. Failure modes and effects analysis (FMEA) is a review of as many components, assemblies, and subsystems as possible to identify failure modes, their causes and their effects [Clemens and Simmons 1998]. There have been previous attempts to apply FMEA to software with success in identifying failure points, but this research did not consider usability or security [Reifer 1979].

Previously, the point was made that accidents and attacks can follow similar paths. In capitalizing on this feature, Schneier developed the concept of an attack tree. An attack tree is a way of thinking about and describing the security of systems, and can be extended to the development of an automatic database which describes the security of a system. This can help in making decisions about how to improve security, or the effects of a new attack on security. In this way, attack trees can be used by both attackers and defenders. [Schneier 1999]

Attack trees have previously been applied to SCADA systems at an administrative level, demonstrating that they can be a useful tool for modeling threats and vulnerabilities in a wide variety of systems not limited to Internet or IT systems. [Byres et al. 2004] [Edge 2007]

Though the research suggests that FTA and other system safety risk management tools have been applied to software, few have actually attempted to integrate at the software level. None have attempted to integrate with executables. The most success in integrating system safety tools with software dependent systems has come from looking at systems with strong cyber-physical attributes, and in using the system safety tools as an auditing method. [Leveson et al. 1991] [Byres et al. 2004]

2.3 Code and Failure Analysis

In looking at current trends in computer security, Static and Dynamic Code Analysis has recently shown promise in identifying flaws and program errors which can create possible safety and security issues. The testing program, *Fortify* by Hewlett Packard, is a commercial software product used to identify threats via Static Code Analysis.

Other academic and open source attempts at the same level of vulnerability detection have been attempted. In one case, the human factors risks associated with programming have been identified. Murphy created a open source programming graphical user interface (GUI) which applied analysis techniques and used a novel interface to identify ‘code smell’ while programming [Murphy-Hill and Black 2010].

Code smell was first discussed by Fowler, who defined code smell as a symptom in the source code of a program that possibly indicates a deeper problem. Code smells are usually not bugs. They are not technically incorrect and do not prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future.[Fowler 1999]

PEP8 is an example of a code standard for the Python programming language. In part, PEP8 was designed as a code standard to improve maintainability in Python code. The means to improving maintainability was to make the code easier to understand and more readable. An example of doing this was by implementing a standard tab and space rule, and integrating the standard into popular Python editors. This included automating layout tasks by editing rulesets for VIM, the popular Linux terminal text editor. PyLint is an automated tool for the detection of code smells based on the PEP8 standard [Logilab 2013].

Static Code Analysis tools exist for most of the common programming languages. CERT developed the secure coding standard for Carnegie Mellon, and has standards for C, C++, Java and Perl [CERT/CC 2013]. Numerous tools test against the CERT Secure coding standard, including Fortify, Rose, and the Secure Coding Analysis Laboratory (SCALe). Murphy dives deeper into the types of behavior in general that need to be considered [Murphy-Hill and Black 2012] .

Static Code Analysis suffers from shortcomings which are addressed by another type of analysis, Dynamic Code Analysis [McCabe 2013]. Dynamic Code Analysis is the practice of analyzing software as it is run on a virtual processor, and has the ability to monitor code as it executes. Dynamic Code Analysis makes it possible to look at program behaviors after the code has been compiled. This type of analysis also allows for detection of compiler compromises. It is possible that a compiler could be written which introduces vulnerabilities and malware into a program. Static Code Analysis would be unable to detect this type of malicious behavior, but it may be detected using Dynamic Code Analysis.

Where Static Code Analysis is based on ‘smell’, Dynamic Code Analysis is based on ‘taints’. Taints, like smells, were developed to combat the automatic generation of exploits. Without the ability to automate detection of malware by creation of signatures or other means, it would be difficult to ‘keep up’ with the attackers. Newsome discusses the need for automatic taint analysis, and discusses in depth the application of taint analysis and the issues with granularity, false positives, and the need for dynamic detection. Dynamic detection operates by identifying un-trusted input sources, monitoring the propagation of the untrusted input through a program, and detects when tainted data can be used in a dangerous manner. Newsome summarizes the types of attacks that can be discovered. [Newsome and Song 2005]

“This approach allows us to detect overwrite attacks, attacks that cause a sensitive value (such as return addresses, function pointers, format strings, etc.) to be overwritten with the attacker’s data. Most commonly occurring exploits fall into this class of attacks" [Newsome and Song 2005].

Newsome also discusses the benefits of their approach.

“Our technique is based on the observation that in order for an attacker to change the execution of a program illegitimately, he must cause a value that is normally derived from a trusted source to instead be derived from his own input" [Newsome and Song 2005].

Tools such as BAP [Brumley et al. 2011] and BitBlaze [Song et al. 2008] make use of an intermediary compilation to create a model of symbolic execution using the Simple Intermediate Language (SimpIL). In this way, data can be tracked between *sources* and *sinks*, or the points where data originates in a program and where it eventually goes. The main steps of analysis are checking the taint introduction, taint propagation, and taint status. The taint status is the value mainly used to determine the behavior of a program. The issues

which then are important to this factor are the variance between the time of detection and the time of the attack, and the possible recourses to detected ‘bad behavior’. Schwarz makes it apparent that this method could be used for more than just security scenarios. Dynamic taint analysis could possibly be used to detect and prevent human caused safety and security misconfigurations at runtime.

Some tools and methods exist for code monitoring of ladder logic. Zifferer has a graphical tool for monitoring SCADA ladder logic operation [Zifferer 1994]. Moon attempted the automatic verification of sequential control systems using temporal logic, which like most other SCADA code auditing techniques, is focused on detecting race conditions [Moon et al. 1992] [Moon 1994]. Zoubek looked at automatic verification of ladder logic programs, mostly for certifying correct logical operation [Zoubek et al. 2003]. Petri nets have shown promise in the ability to identify errors in both timing and logic in more complex SCADA systems, and could serve as a better method for code analysis on SCADA Systems [Uzam and Jones 1998].

2.4 Limitations of the Existing Research

One primary limitation has been identified in the review of the existing literature. The current state of the art in usable security for SCADA systems is highly fractured. Usability research in SCADA is focused on the user interface [Ikuma 2013]. Security in SCADA systems is focused on defense in depth [Small 2011] [DHS 2009]. The only research in SCADA which considers the operational programs at the lowest level is typically concerned with race conditions. In code analysis, there is almost no consideration of safety or usability. This leaves a opportunity for research to be conducted in the application of safety engineering methods to code analysis for the improvement of usable security in SCADA systems.

2.4.1 Incorporating Code Analysis and Safety Engineering Tools for Improving Usable Security in SCADA

The introduction of this dissertation looked at the Aurora Generator Test. Aurora was a demonstration of utilizing a known safety vulnerability, after defeating defense in depth protections, to cause physical damage to a power generation system. If the safety issue would have been solved when it was discovered, the researchers conducting the test would have been unable to attack the generator by opening and closing a breaker with the generator out-of-phase.

Code analysis may have identified this unsafe system state [Uzam and Jones 1998], and programmatically disallowed activation from an input if provided the correct safety information. Safety engineering may be incorporated into this method to give engineers a different view of their work [Leveson et al. 1991], and to provide the means to ‘engineer out’ possibly unsafe future states. However, there has been no attempt to include code analysis and safety engineering methods for the improvement of usable security in SCADA systems. This research aims to develop a method for applying code analysis and safety engineering methods to SCADA systems, and to conduct a usability test to identify the strengths and weaknesses of this method.

Chapter 3

Developing a Code Behavior Analysis Method to Identify Possible Misconfiguration Errors in SCADA

3.1 Abstract

Misconfigurations, depending on system design, can place a system into a possibly unsafe state. It does not matter if the action is committed intentionally or accidentally. To help identify the actions which can lead to misconfiguration errors in SCADA systems, a static code analysis method was developed. This method creates abstract representations from the rungs of ladder logic program used to control a Programmable Logic Controller. The abstract representation used is based on Fault Tree Analysis, and hopefully makes it possible to provide an indicator of correctness in regards to usability, safety, and security.

3.2 Introduction

Programmers employ static code analysis to check programs for errors without executing them. Static code analysis provides an initial indicator of correctness. By constructing a model of a program, the programmer has additional means to review what was created. The abstract representation can be used to identify patterns which are indicative of possible errors [Louridas 2006]. In many cases, programs may be syntactically correct: the program will compile and run, but may still allow for behavioral errors. Analyzing code, specifically code reviews, is one preferential method to detect and eliminate these types of errors. However, code review is not always possible. It can be costly and difficult to both train programmers and to bring them together. [Bardas 2010]

In stark contrast, usability is measured only after a prototype has been created. Only then can a usability test be conducted. There is no quick test to check for an indicator of "correctness" in usability. Usability testing is typically considered separate from the main engineering task, and as Whitten points out, there has been little work in investigating the relationship between usability and security [Whitten 2004].

Given the limitations of current usability testing and security research, the first contribution to research by this dissertation is the development of a static code analysis method for SCADA systems which incorporates safety engineering tools, such as FTA, to identify possibly unsafe system states. In the same way that static code analysis is being used to identify unsafe code patterns, or smells, this method tries to identify unsafe states resulting from both accidental or malicious system misconfigurations. In identifying these unsafe states, it is possible to simultaneously remove a user's ability to make costly mistakes, and to eliminate possible attack vectors. Such a method could help to provide engineers with an initial indication of correctness regarding usable security.

3.3 Methods

3.3.1 Objective

The objective of this study was to develop a method for evaluating code written at the lowest level of a SCADA system, the PLC, in order to generate indicators and patterns for evaluating usability and security. This section of the dissertation is focused on the engineering capabilities and realities of working with commonly used hardware and programs found in SCADA systems. Experimentation at this point could not yet be conducted until a method and an initial test bed had been created. As such, this study focuses on the test bed creation and the development and refinement of the method used for the static analysis.

3.4 Development

PLC operation, as shown in Figure 3.1, is relatively basic. The PLC is the main controller for a physical process. The PLC takes input, operates on that information programmatically, and manipulates outputs. Other controllers may exist in a complex system, but the lowest level logic being operated on in a SCADA system exists in the PLC. In this way, when provided with the inputs and outputs, it is possible to understand the system state and the logic being considered solely by understanding what is taking place within the PLC, or PLCs, provided that the input and output information is correct.

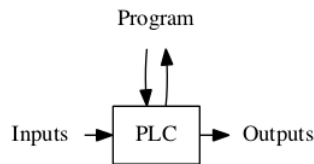


Figure 3.1: Example PLC Operation

Initially, development for this study was conducted using a Raspberry Pi with a PiFace Digital Input Output controller. This hardware is shown in Figure 6.1. The main strength behind this hardware and software test bed was that the PiFace allowed for PLC-like operation, and is programmable in Python. Python is unique in that a coding standard has already been developed [Guido van Rossum 2013] and tools such as Pylint exist to conduct static code analysis. Pylint already allows for the creation of Unified Modeling Language (UML) type representations of Python programs by using tools such as PyReverse [Logilab 2013].

While the Raspberry Pi was initially promising for this research, the lack of conformity to what was in common use in SCADA systems was immediately notable. PLCs are designed to be highly reliable, and as such their programming deviates too widely for existing code analysis methods to apply. Typically, PLCs are programmed using either ladder logic or HDL. A different test device needed to be identified.

There is no shortage of PLCs in use in modern SCADA systems. The Allen-Bradley MicroLogix 1000 was selected because of the availability to program both a physical hardware device and a virtual device. A photograph of the Allen-Bradley Micrologix 1000 is shown in Figure 6.2. There are many guides available for programming Allen-Bradley PLCs. Programs for the MicroLogix 1000 and other Allen-Bradley PLCs can be written using RSLogix Micro. The programs are transferred to the PLC using RSLinx Classic. Additionally, the PLC behavior can be emulated on a Windows machine by using RSLogix Emulate 500. This combination of tools provided all of the hardware and software needed.

The manner in which the program is written, stored, and run on the PLC is important. Allen-Bradley provides a separate guide for this information. Allen-Bradley PLCs, and most other PLCs, function on an operating cycle which follows this order: [Allen-Bradley 1997]

1. Input scan
2. Program scan
3. Output scan
4. Service communications
5. Overhead

The operating cycle helps to ensure reliable behavior, and attempts to avoid race conditions. Much of the study in conducting code analysis on a PLC has been to detect race conditions [Zoubek et al. 2003]. The PLC processor provides control via the program written to the device in the form of a Processor File. This file is broken down into a set of Program Files and Data Files to be more manageable. The Program Files contain an ordered set of files which contain the controller information (file 0, file 1), the main ladder program (file 2), the interrupt subroutines (file 3), and any other subroutine programs (files 4-15). The file of interest for the purposes of static code analysis is the main ladder program (file 2). [Allen-Bradley 1997]

The Data Files contain status information associated with external input and output (I/O). This includes all output and input states (file 0 and file 1), the status of the control operation (file 2), internal relay logic storage (file 3), counter and accumulator information (file 5 thru file 7), and the timer (file 4). [Allen-Bradley 1997]

The PLC main program was written using a ladder logic editor. The ladder logic editor uses a graphical programming language based on electrical relay diagrams. In this case, RSLogix Micro and LD Micro were used to create the programs. Figure 3.2 provides a screenshot of RSLogix Micro running a basic control program while connected to the hardware MicroLogix 1000.

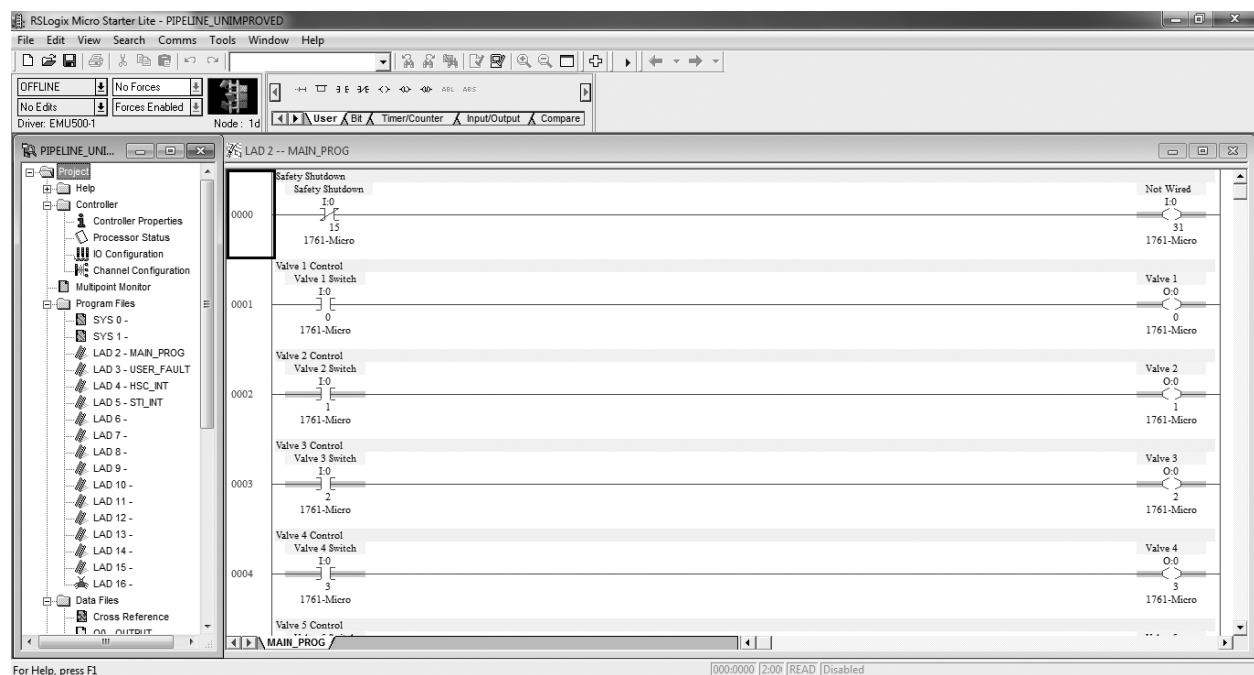


Figure 3.2: RSLogix Micro Screen Shot

In order to demonstrate the method used to conduct code analysis on a PLC program, an example program was used. A door interlock is an easily understood engineering problem with interesting safety considerations. A interlock is commonly used to prevent undesired states in a system. A household example is the door on a microwave. When the door is open the circuit is broken and the microwave will not operate. This prevents the undesired state of the microwave running with the door open.

Another common application of an interlock at the scale which we are considering would be an airlock. In an airlock, an inner door and an outer door are used in order to prevent exposure from one area to another. The inner door and outer door are connected by an intermediate space which can be pressurized or decontaminated as needed. Figure 3.3 is a graphical representation of a door interlock.

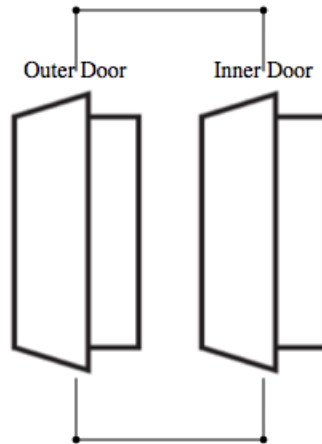


Figure 3.3: Example Door Interlock

3.4.1 Method Development

Unimproved Program Analysis

The ladder logic in Figure 3.4 was written to control both the inner and outer door independent of any safety measures, and was the most basic way to control the doors. In this program there is no interlock between the doors. When the inner door switch is pressed, the inner door opens. When the outer door switch is pressed, the outer door opens. The switch and control represent the state of the door. In reality, a sensor may be required to correctly show the state of the door as either open, closed, or possibly in an intermediate state.

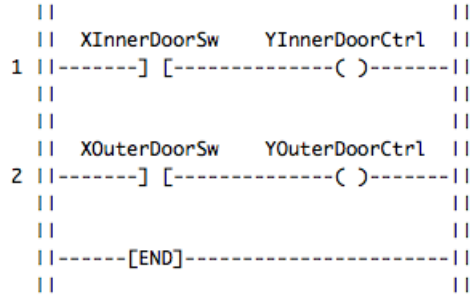


Figure 3.4: Interlock Ladder Logic Program

Figure 3.5 shows the translation of the ladder logic program to Verilog. This demonstrates a key task if this method is to work with European PLCs which are programmed with HDL instead of ladder logic. Additional tools can be used with HDL as well. GTKWave, as an example, simulates an HDL program’s execution as a testing measure. The method of translation will not work in all cases as HDL is a more complex language and it may not be possible or justifiable to translate all of the functionality of a HDL into a ladder logic program.

```

always@(*) YInnerDoorCtrl = XInnerDoorSw
always@(*) YOuterDoorCtrl = XOuterDoorSw

```

Figure 3.5: Interlock Verilog Pseudocode

The next step in the method was to create a meaningful representation from the ladder logic code for analysis. FTA can be created by treating each of the initiating contacts on the individual rungs of the ladder logic program as an initiating event for each branch of the fault tree. For example, the contacts labeled ‘XInnerDoorSw.’ and ‘XOuterDoorSw.’ become the initiating events for the branches of our first fault tree. Additional contacts on the rung can be joined to the corresponding branch on the fault tree by using the appropriate logical gate for inclusion. The branch of the fault tree will end with the rung, in our case at the ‘coils’ labelled ‘YInnerDoorCtrl’ and ‘YOuterDoorCtrl’. The top-event and possibly

intermediary events may not be able to be generated from the ladder logic and will have to be created from further analysis.

The main ladder logic program file will vary depending on the editor used. Figure 3.6 is a file created by LD Micro. The program file typically begins with information on the type of editor used and the processor for which the program is intended. This information was removed as it is not needed, and does not provide behavioral information. The I/O list provides the list of input and output values. In this case, XInnerDoorSw and XOuterDoorSw are the door switches, and YInnerDoorCtrl and YOuterDoorCtrl are the door controls. The program is sub-divided into rungs. If additional logic were included on the rungs, it would have been placed in parallel sections of code nested under each rung. This is shown in later figures.

```
IO LIST
  XInnerDoorSw at 0
  XOuterDoorSw at 0
  YInnerDoorCtrl at 0
  YOuterDoorCtrl at 0
END

PROGRAM
RUNG
  CONTACTS XInnerDoorSw 0
  COIL YInnerDoorCtrl 0 0 0
END
RUNG
  CONTACTS XOuterDoorSw 0
  COIL YOuterDoorCtrl 0 0 0
END
```

Figure 3.6: Interlock Program File

With this file, it is possible to write a script which parses the ladder logic into a \LaTeX document. This creates the beginning of the fault tree. In the example case, a top event which describes the eventual physical world event is needed (Both Doors Open). Similar to test driven development, the initial event should be a undesirable event. If no undesirable event can be detected, then there is no necessary action to be taken and the representation will serve as a indicator of correctness. First, the appropriate logic needs to be drawn from the ladder logic program.

In this example, the rungs are joined by an ‘and’ logic gate as they both need to be activated in order to move down the branch. The observation is made that the air lock integrity would be violated in this case. Figure 3.7 was created using L^AT_EX, and was also modeled using OpenFTA.

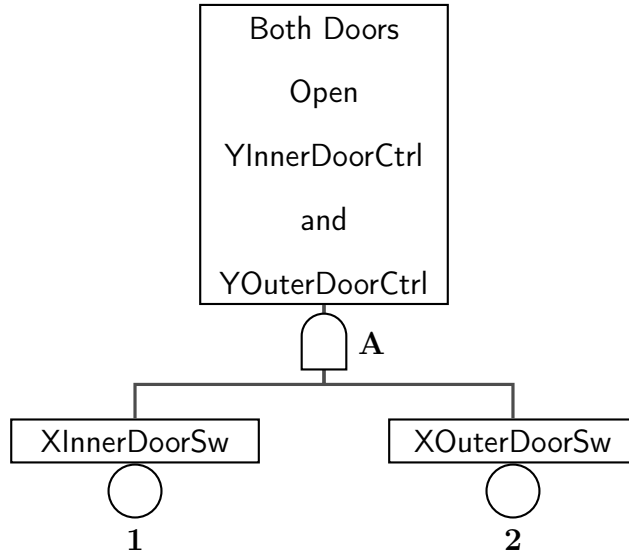


Figure 3.7: Interlock Fault Tree Diagram

For this example a cut set was created automatically using OpenFTA. A cut set is the set of events that will assure that the system will not function [Moriarty 1990]. In our case the cut set is the set of component failures that will cause system failure. The cut set is given in a single row in Table 3.1 indicating that preventing either initiating event can prevent violating the integrity of the airlock. The cut set and minimal cut set in this example are the same, and were automatically generated by OpenFTA.

Table 3.1: Cut Set for Unimproved Interlock Fault Tree

1	2
---	---

To further illustrate this point, a state diagram in Figure 3.8 shows the transition to possible end states based on initiating events. In this case, since the doors are being

opened and closed by an operator, the state in ‘Both Doors Open’ is the possibly dangerous misconfiguration state caused by opening both doors without shutting a door. With this possible misconfiguration in mind, the next step is to attempt to apply the hierarchy of controls to ‘engineer out’ the misconfiguration.

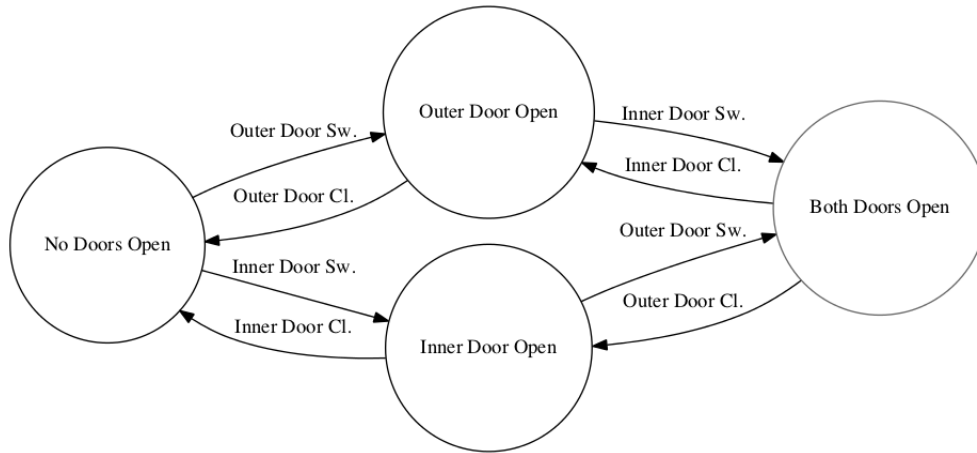


Figure 3.8: Interlock State Diagram

Improved Program Analysis

To improve the program, and to avoid the conceivable accidental state of an operator simultaneously opening both doors at once, a negated contact is included in the ladder logic program. A negated contact is indicated by a slash in the contact symbol. Figure 3.9 shows the change in logic.

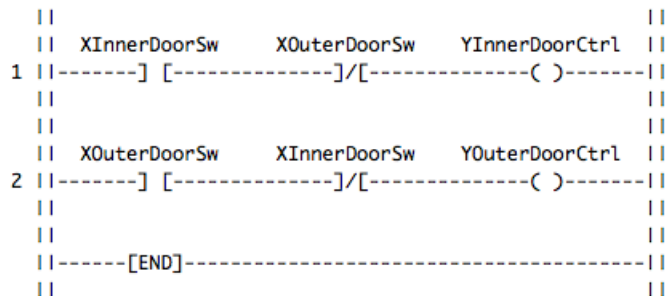


Figure 3.9: Improved Interlock Ladder Logic Program

Figure 3.10 shows the improved program file in LD Micro. In this case, the additional negated contact is placed inline for each individual rung. The logical statement is made that the corresponding door control coil cannot be operated if the alternate door switch is closed. In this case, the parallel statement causes the fault tree to be expanded to include ‘NOT’ logic as an initiating event. Figure 3.11 shows the resulting fault tree demonstrating that if one door is opened the other door cannot be opened. There are multiple ways to demonstrate this logic, and some rule sets may make automation possible but will result in overly complex fault trees without additional analysis. Simultaneously, this may make the resulting fault tree cumbersome as the model increases in complexity.

```
IO LIST
  XInnerDoorSw at 0
  XOuterDoorSw at 0
  YInnerDoorCtrl at 0
  YOuterDoorCtrl at 0
END

PROGRAM
RUNG
  CONTACTS XInnerDoorSw 0
  CONTACTS XOuterDoorSw 1
  COIL YInnerDoorCtrl 0 0 0
END
RUNG
  CONTACTS XOuterDoorSw 0
  CONTACTS XInnerDoorSw 1
  COIL YOuterDoorCtrl 0 0 0
END
```

Figure 3.10: Improved Interlock Program File

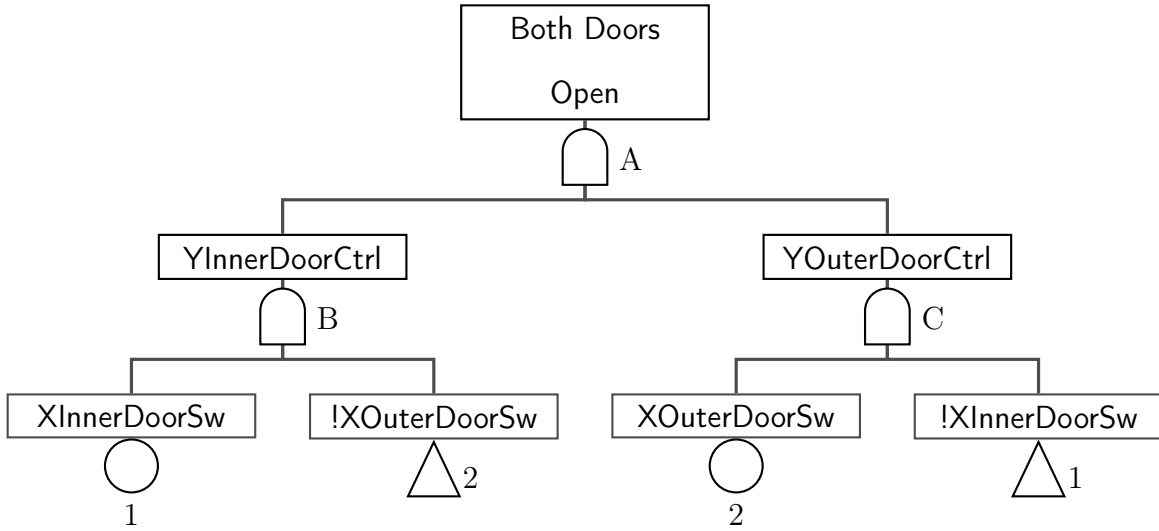


Figure 3.11: Improved Interlock Fault Tree Diagram

Figure 3.12 is the state diagram resulting from the changes made. This state diagram shows that the state for ‘Both Doors Open’ is no longer possible when controlled from the PLC. Sabotage and physical damage are possible ways to open both doors, but are not considered as they are not means which can be judged via code analysis. The inability to open both doors might be a potentially unsafe engineering state depending on the specification required for the doors. There may be a time when both doors need to be opened simultaneously for installing large equipment, cleaning, or evacuation. The next demonstration shows how risk and mitigation steps might be included in the method in order to handle scenarios where completely removing a state is less ideal.

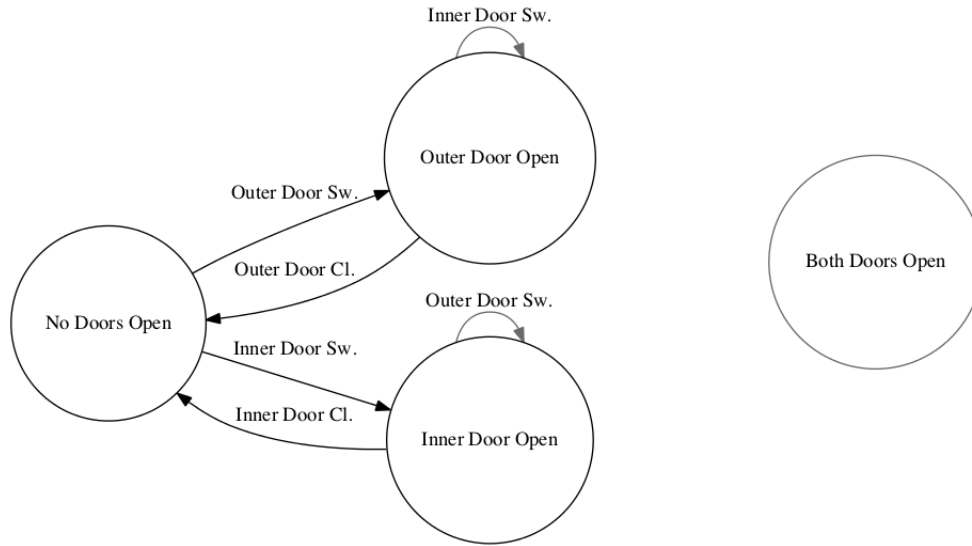


Figure 3.12: Improved Interlock State Diagram

Improved Program Analysis with Design Trade-Offs

The options for mitigating possibly unsafe states are numerous and depend largely on the design decisions and trade-offs made by the stakeholders in charge of the project. For this implementation, it is speculated that the state for both doors needs to be allowed but should prevent accidental activation. Additionally, the override switch could be maintained strictly in the physical realm where security is more assured. Other measures may include adding a delay to the action, or providing a warning when the override is selected and both doors are opened.

Figure 3.13 shows the inclusion of the override contact to the ladder logic program as well as the addition of an alarm for when the override is activated. Figure 3.14 is the updated program file generated by LDMicro.

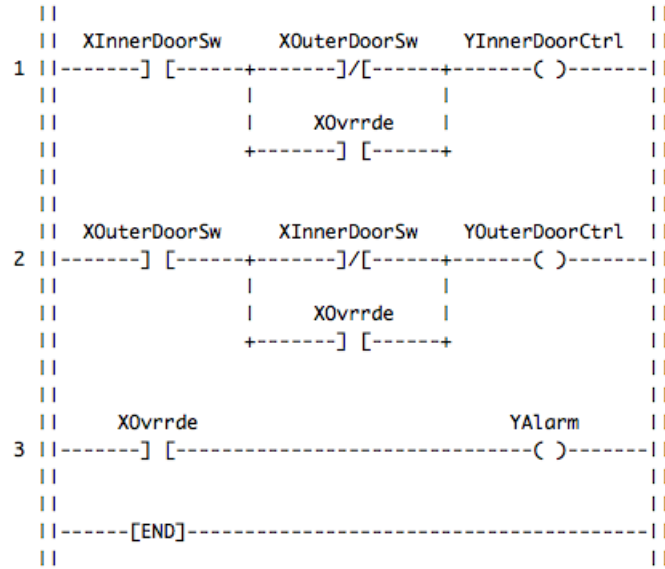


Figure 3.13: Override Interlock Ladder Logic Program

```

IO LIST
  X1 at 0
  X2 at 0
  XOvrnde at 0
  Y1 at 0
  Y2 at 0
  Yalarm at 0
END

PROGRAM
RUNG
  CONTACTS X1 0
  PARALLEL
    CONTACTS X2 1
    CONTACTS XOvrnde 0
  END
  COIL Y1 0 0 0
END
RUNG
  CONTACTS X2 0
  PARALLEL
    CONTACTS X1 1
    CONTACTS XOvrnde 0
  END
  COIL Y2 0 0 0
END
RUNG
  CONTACTS XOvrnde 0
  COIL Yalarm 0 0 0
END

```

Figure 3.14: Override on Improved Interlock Program File

The fault tree in Figure 3.15 was created using the method described, and was generated using \LaTeX . It was necessary to manually include the final end state and the override intermediate state. The left and middle branches were created by directly referencing the ladder logic for Rung 1 and Rung 2. The alarm rung was not modeled as it does not lead to a potentially undesirable system state. This makes the fault tree more complex than necessary, but is still technically correct.

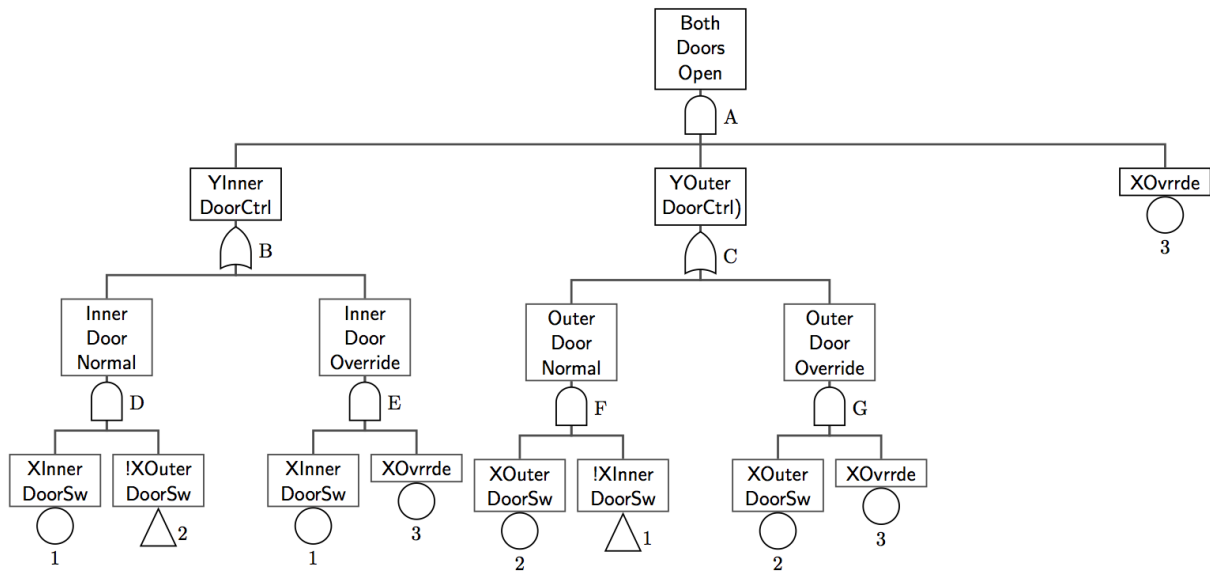


Figure 3.15: Override Interlock Fault Tree Diagram

With the inclusion of a new intermediate state and additional logic, a new cut set can be generated. The fault tree was created and validated using OpenFTA. OpenFTA was also used to generate the cut set and minimal cut set. The cut set in Table 3.2 now shows that all three contacts need to be activated in order to transition into a possible unsafe state where both doors can be opened simultaneously. An equivalent fault tree can be created using alternate logic. In both Figure 3.15 and Table 3.2, the alarm branch was excluded despite the alarm being added on a separate rung showing that it is activated with the override contact.

Table 3.2: Cut Set for Override Interlock Fault Tree

1	2	3
---	---	---

A simplified FTA can be created which is helpful in understanding the change in the system from the addition of the override.

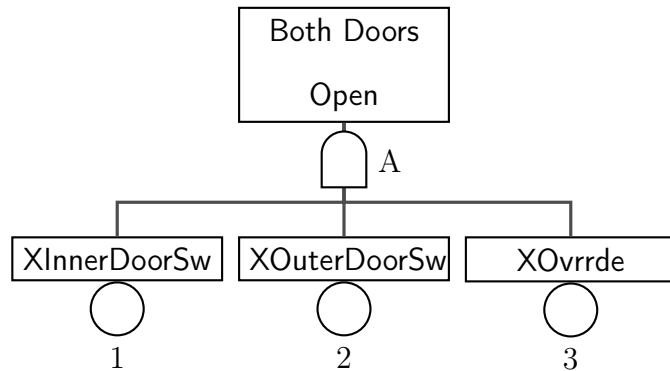


Figure 3.16: Simplified Override Interlock Fault Tree Diagram

Figure 3.17 is a final state diagram created to show that it is now possible to open both doors of the airlock, but necessitates the activation of an override switch.

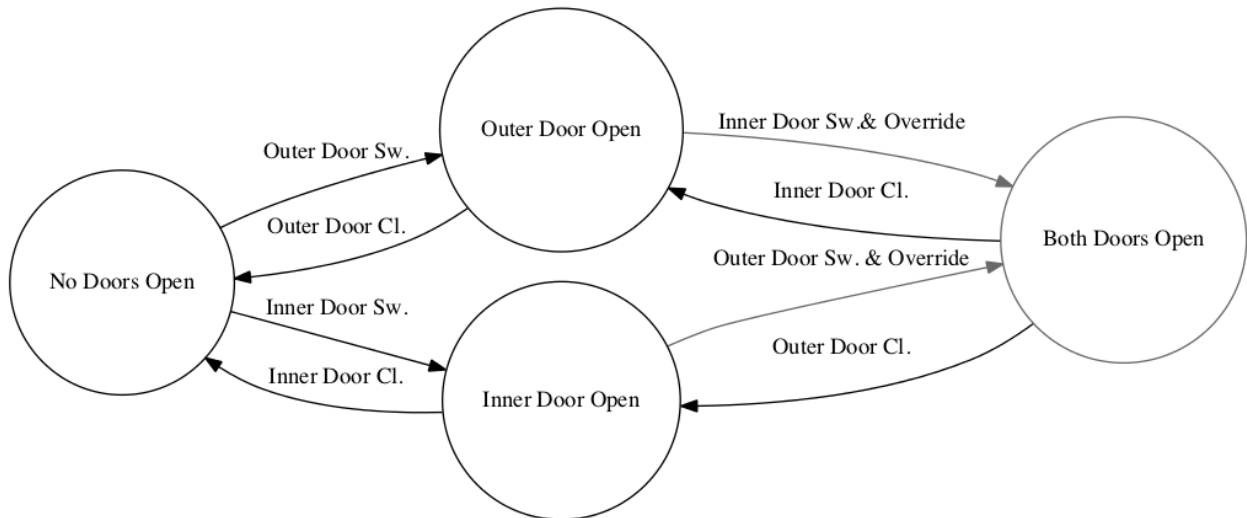


Figure 3.17: Override Interlock State Diagram

3.5 Results

The result of this study is the development of a static code analysis method for generating fault trees from the main ladder logic program file run on PLCs. The ladder logic written has been tested on both PLC hardware and PLC virtualization tools. This method attempts to create a meaningful representation of the program used to control the lowest level component of a SCADA system. The hope is that this representation can be used to provide an indication of the level of correctness towards usable security.

3.6 Discussion

The static code analysis method described works with the common functions available to both ladder logic and HDL which are easily translatable to the components and logic used in FTA. However, the vocabulary used in FTA can be limiting, while the vocabulary available to HDL and ladder logic are very expressive. An example of this problem came in that few of the FTA editors tested support the ‘NOT’ logical symbol which is commonly used in ladder logic. As such, compromises have had to be used occasionally.

With the complication in languages comes a problem with the simplicity of the method described. As shown in Figure 3.15 versus Figure 3.16, the fault trees generated using this method can be more complex than necessary, and may need to be simplified. This leads to an additional issue with complexity.

The generation of fault trees using this method encounters formatting issues as the fault trees generated begin to include more nodes. As the fault trees grow in complexity readability decreases. A vector based graphical solution might be needed for a more complex system in order to view the fault trees from varying levels of abstraction. This brings us to a possible problem with one of the focuses of this study.

One of the focuses of this method is on security. Defense in depth, while previously criticized as possibly proving unsustainable and inadequate to the problem, is still necessary

even if this method is successful in identifying and removing possible misconfigurations. Recent SCADA attack research has shown the ability to overwrite PLC logic which would subvert the efforts made in applying this static code analysis method [Queiroz et al. 2009].

Finally, an additional limitation of this method is the reliance on a significant level of manual interference and system knowledge to audit the fault trees generated. It is difficult to measure the cost and benefit of applying this method. This is due in large part to the wide variety of tasks for which PLCs are used. There has been a great deal of success in the language used in ladder logic programming in this study, and to mitigate the possibility of encountering a program where this method will fall short, the method has been tried on a variety of ladder logic programs running on hardware and multiple virtualization tools.

3.7 Conclusion

The method described in this chapter is a starting point for conducting static code analysis to identify misconfiguration errors related to usable security. The method has been demonstrated to work on sample programs from a variety of hardware and software tools. The next chapter discusses efforts made to study and test the effectiveness of this method by conducting a usability experiment.

Chapter 4

Developing Experiment

4.1 Abstract

Misconfigurations can be costly mistakes that place a system into an unsafe state. In order to address these issues from a usability, safety and security standpoint with regards to SCADA systems, a static code analysis method has been developed which generates Fault Tree Analysis (FTA) artifacts from ladder logic programs running on a Programmable Logic Controller (PLC). The static code analysis method generates a fault tree, which is used to identify possibly unsafe system states, and the user actions that may lead to that unsafe state. The hope is to provide an indicator of program correctness. In order to evaluate the strengths and weaknesses of the proposed method, a model pipeline SCADA system was developed. The static code analysis method was applied to this model, and controls were implemented to mitigate or remove the unsafe states identified. Twenty five subjects operated the model pipeline under mock critical operating conditions using ladder logic that had undergone code analysis and ladder logic that had not. Test subjects were also asked to ‘attack’ the model pipeline in order to cause as many intentional misconfiguration errors as possible. In this experiment, it was shown that for this model there was a significant decrease in the number of misconfiguration errors that test subjects made, both unintentionally and intentionally.

4.2 Introduction

On 10 June 1999, an operator was overseeing a sixteen inch gasoline pipeline in Bellingham, Washington when both the primary and secondary SCADA computers began to generate errors. The system became unresponsive. Pressure in the pipeline began to build in a section of the pipeline presumably damaged by an excavator. A full restart of the system was conducted to regain control. At that time, the operator was notified of a gasoline smell. It took 1 hour and 15 minutes from the time the pipeline started leaking until the isolation valves were finally closed. The resulting fire killed three people and injured eight others.[NTSB 2002a]

Seventeen years earlier, in 1982, a similar situation occurred at the Trans-Siberian natural gas pipeline. An estimated 3-kiloton explosion was the result of suspected software bugs in code responsible for controlling the pipeline. The main difference between the Bellingham, Washington incident and the Trans-Siberian pipeline accident was intent.

Allegedly, the bugs were maliciously left in the code. Thomas Reed, a former Secretary of the Air Force, wrote the following about the attack. [Reed 2007]

“ ...the pipeline software that was to run the pumps, turbines, and valves was programmed to go haywire, after a decent interval, to reset pump speeds and valve settings to produce pressures far beyond those acceptable to the pipeline joints and welds. The result was the most monumental non-nuclear explosion and fire ever seen from space" [Reed 2007].

The reason these two cases are relevant is that they both deal with usability, security, and software for SCADA systems. In both cases, the system was allowed to enter an unsafe state. Both cases might have been prevented by programs operating at the highly reliable PLC level. Code analysis may have been capable of detecting the unsafe system states and preventing the commands and subsequent actions which directed the system to enter those states.

The purpose of this study was to identify a means for conducting static code analysis relevant to usability, safety and security in SCADA systems. By providing engineers with an indicator as to the level of code correctness, corresponding usability errors may be detected during development and accidents prevented.

4.3 Methods

4.3.1 Objective and Hypothesis

The objective of this experiment was to test the ability of a static code analysis method to aid in improving usability in SCADA systems. Usability, in this case, was treated as a key method for enhancing both safety and security. The code analysis method created an abstract representation of PLC ladder logic as branches for the creation of a fault tree. The fault tree was then used to identify possibly unsafe system states. The unsafe systems states, when viewed from a usability standpoint, are indications of possible user misconfigurations which need to be addressed by a hierarchy of controls. In these cases, code reconfiguration is needed to either engineer out the possibility of the unsafe state entirely, or the programmer or engineer will need to provide a warning or mitigation step before allowing the system to enter the unsafe state. Typically, these unsafe states can be difficult to detect as code and systems increase in complexity.

In order to test the method as it applied to usability, an experiment was created. The hypotheses of the experiment were:

Hypothesis 1 : The model pipeline SCADA system on which code analysis and reconfigurations have been conducted will have fewer errors caused from accidental misconfigurations in comparison to a model pipeline SCADA system on which code analysis and reconfiguration has not been conducted.

$$H_0 : \mu_{\text{test subject accidental alarms without analysis}} = \mu_{\text{test subject accidental alarms with analysis}} \quad (4.1)$$

$$H_1 : \mu_{\text{test subject accidental alarms without analysis}} > \mu_{\text{test subject accidental alarms with analysis}}$$

The experiment also asked test subjects to ‘attack’ the model pipeline SCADA system created. The second hypothesis of the experiment was:

Hypothesis 2 : The model pipeline SCADA system on which code analysis and reconfiguration have been conducted will have fewer errors caused from intentional misconfigurations in comparison to a model pipeline SCADA system on which code analysis and reconfiguration has not been conducted.

$$H_0 : \mu_{\text{test subject intentional alarms without analysis}} = \mu_{\text{test subject intentional alarms with analysis}} \quad (4.2)$$

$$H_1 : \mu_{\text{test subject intentional alarms without analysis}} > \mu_{\text{test subject intentional alarms with analysis}}$$

Additionally, the experiment also tested if there would be a difference in the amount of time for test subjects to complete the assigned tasks between the model pipeline SCADA system on which code analysis and reconfiguration had been conducted and the model pipeline SCADA system on which code analysis and reconfiguration had not been conducted :

Hypothesis 3 : The difference in the amount of time that test subjects take to complete a task on a model pipeline SCADA system on which code analysis and reconfiguration has been conducted will be less in comparison to the amount of time it will take a test subject to complete identical tasks on an otherwise identical model SCADA system on which code analysis has not been conducted.

$$H_0 : \mu_{\text{time to task completion without analysis}} = \mu_{\text{time to task completion with analysis}} \quad (4.3)$$

$$H_1 : \mu_{\text{time to task completion without analysis}} > \mu_{\text{time to task completion with analysis}}$$

4.3.2 Experimental Design

In order to test these hypotheses, a model pipeline SCADA system was created and the code analysis method applied. To realistically model a pipeline SCADA system, case studies conducted by the National Transportation Safety Board (NTSB) were referenced. The overall summary of the case study is made available from the NTSB [NTSB 2002a].

The findings of the case study are provided verbatim from the conclusions section of the NTSB report, and are the result of thirteen case studies conducted, twelve site visits, and sixty nine individual interviews.

1. Most hazardous liquid pipeline operators use SCADA systems to monitor and control their pipelines.
2. Operators reported that SCADA systems enhance both the safety and efficiency of their pipelines.
3. Implementations of graphical standards developed for pipeline operations will increase the likelihood that leaks will be detected quickly and that resulting damage from the leaks will be minimized.
4. An effective alarm, review/audit system will increase the likelihood of controllers appropriately responding to alarms associated with pipeline leaks.
5. Requiring controllers to train for leak detection tasks using simulators or non-computerized simulations will improve the probability of controllers finding and mitigating pipeline leaks.
6. Because the report form used by the Office of Pipeline Safety for companies to report liquid pipeline accidents (PHMSA F 7000-1) does not require operators to provide information about fatigue, such as controller work schedules, it is not possible to empirically determine the contribution of fatigue to pipeline accidents using the Office of Pipeline Safety accident database.
7. Ensuring constant monitoring of an entire pipeline using a computer-based leak detection technology would enhance the controller's ability to detect large spills, increase the likelihood of spill detection, and reduce the response time to large spills.

[NTSB 2002a]

The recommendations based on these conclusions and previous usable security experiments were heavily relied upon to aid in adding realism to the usability experiment in this study [Whitten 2004] [Garfinkel 2005] [Ikuma 2013] [NTSB 2002a].

The goal of the model created was to be as realistic as possible without requiring specific training for test subjects. The pipeline was based on a compilation of maps found for the Sissonville, West Virginia [NTSB 2001], Winchester, Kentucky [NTSB 2002b], Carlsbad, New Mexico [NTSB 2000] and Brenham, Texas pipeline accidents [NTSB 1993]. Two pipelines are shown in the experiment. These pipelines were modeled as interconnected to allow for inclusion of flow redirection due to ‘pigging operations’ and pipeline damage. Pigging operations are when a section of pipeline is closed and a device known as a pig is sent through the pipes to look for any possible structural weaknesses. Both pigging operations and external damage are commonly cited events in creating abnormal operating conditions for which there is no operating guidance. Since training is not typically conducted for this type of event, the training effect is not considered as a solution to the usability problem in this case. This was an attempt to aid in addressing an issue with the test subject pool available for this experiment.

The test subject pool was largely comprised of graduate level engineering students at Auburn University. While it is true that the situations given in the experiment are not commonly trained, it is still believed that trained pipeline operators would perform differently than the naive test subjects. While this is a possible limitation, as the experiment is meant to test an engineering method to improve usability, the SCADA experience and background of the test subjects is believed to have made little difference in the experiment.

The experiment still strived for realistic details, and as such overflow tanks were included to model the salt dome storage operations conducted in the Brenham, Texas incident [NTSB 1993]. Pumping stations were placed along the routes to allow increased pressure for each section of pipe. Valves were placed to control the flow of highly volatile liquid (HVL) into the various sections and outlets along the route. Backwards flow was not allowed. Pressure

indicators were included for each section of pipeline. In keeping with the recommendations made by the NTSB in their report [NTSB 2002a], pressure trends over time were indicated in a separate color coded display and an alarm acknowledgement and management system was provided. The test subject misconfiguration errors were logged using the alarm system. This meant that any misconfiguration that resulted in an alarm resulted in an error being recorded. Misconfiguration errors which did not result in an alarm were not recorded. A black and white screenshot of the pipeline system is shown in Figure 4.1. A larger and more readable version is provided in Figure 6.5.

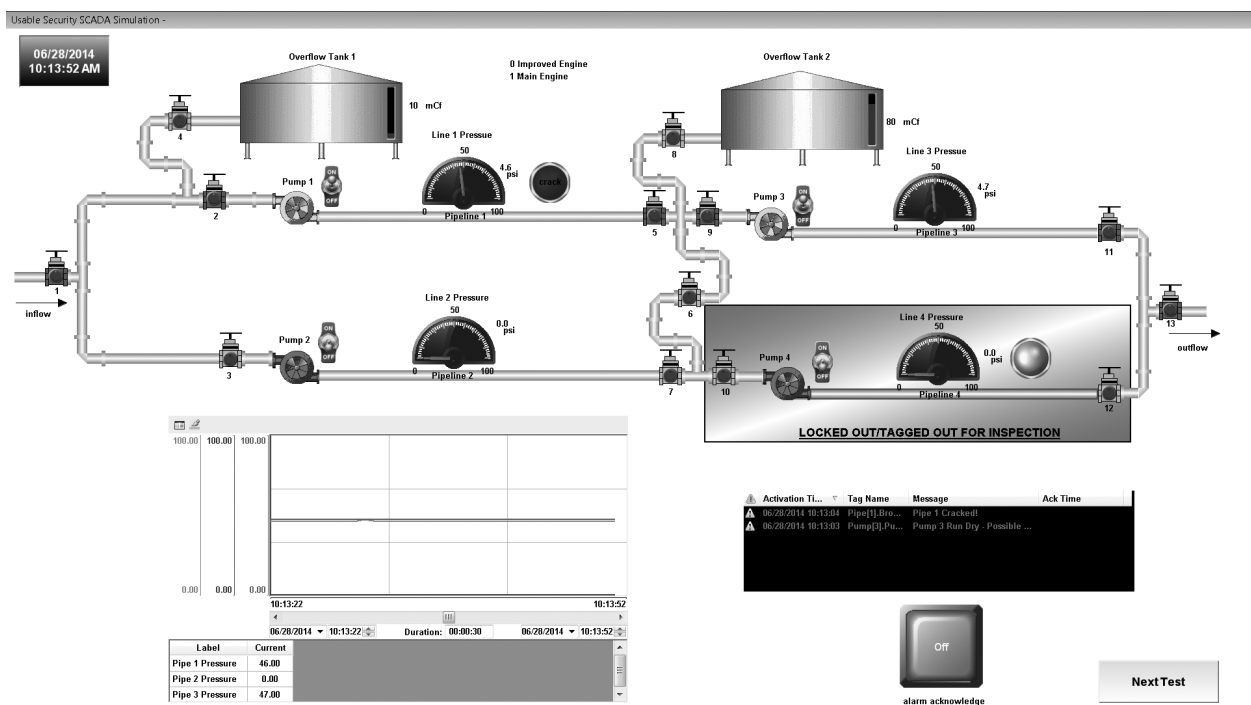


Figure 4.1: Pipeline Usable Security Simulator

A scenario was created which was intended to treat the test subjects as operators of the designed pipeline. The operations were designed to take place under abnormal, and therefore untrained, conditions. It is believed that the test subject, with computer competency to operate graphical software but not expert level knowledge in either computer security or SCADA operations, would be well suited to understanding and operating the model pipeline.

The pipeline was divided into 4 separate sections numbered one to four from the top left section to the lower right section of the pipeline. The valves of the pipeline were labeled as Valves 1 thru Valve 13. Pipeline 4 is closed for ‘pigging operations’, and as such is labelled as logged out and tagged out. Consistent with lock out tag out, Pipeline 4’s controls will not respond to commands. Pipeline 1 has been damaged by contact with a truck, and as such the flow will need to be redirected in order to safely maintain operations. The sections of the pipeline are designed to be operated at a pressure of 4.7 psig. This value is based on liquid natural gas transportation. In keeping with the scenario where Pipeline 1 has been damaged, Pipelines 2-4 fail if the pressure reaches 8.0 psig while Pipeline 1 fails if the pressure reaches 6.4 psig. The overflow tanks rupture or overflow if the tanks are allowed to reach 106 % of their indicated capacity. Pressure relief valves and tank overflows were not included as part of the scenario. As the experiment is trying to test subjects ability to reconfigure a system correctly, these safety devices were viewed as possible distractions to the test subjects. They may be included in future testing.

The experiment was designed using Indusoft’s Web Studio v7.1 tool for developing Human Machine Interfaces. Indusoft markets Web Studio as a collection of automation tools for providing all of the building blocks needed to develop HMIs and SCADA systems for embedded instrumentation solutions. Indusoft’s Web Studio was an ideal program for this experiment because it offers the ability to integrate with real world hardware, is currently used as an industry tool for integrating and controlling SCADA systems, and includes the ability to simulate system reactions for test HMI’s via Visual Basic scripting. The ability to write Visual Basic scripts was integral for mimicking system reactions since we were unable to test on a real world pipeline.

Interestingly, it was possible to test both real world hardware as well as emulated hardware. By using the MicroLogix 1000 PLC, it was possible to integrate ladder logic running on a hardware PLC while simulating system reactions. The mimicked physical reactions are provided via the Visual Basic script included in the appendix. The ability to communicate

with our PLC, since we were restricted to a serial port connection, was provided by using Virtual Serial Port Driver from Eltima Software. As this communication protocol is dated, it was necessary to use a USB Serial Port emulator to connect the PLC and the PLC Emulator, RSLogix Emulator to Indusoft Web Studio. RSLink Classic was used to connect both RSLogix Emulator and the MicroLogix 1000 to the correct communication port.

While the pipeline was being created, it was important to also write the ladder logic which would control the PLC and all of the output devices on the pipeline. Figure 4.2 shows the unimproved experiment ladder logic. Using the method previously described in Chapter 3, the fault tree was created and the fault tree logic was completed with additional system insight.

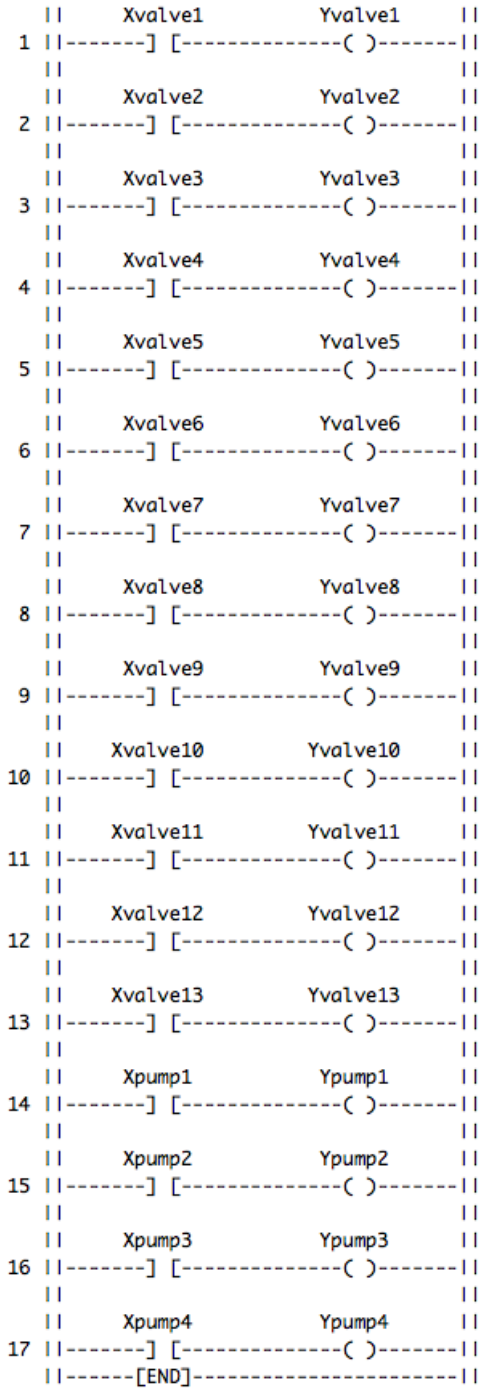


Figure 4.2: Unimproved Pipeline Ladder Logic

The fault tree permutations were calculated to show the interaction between seventeen initiating events coming from thirteen valves and four pumps. The permutations can be

calculated using the Python script in Listing 6.2. Without simplification the possible permutations of fault tree branches would have lead to the creation of 8,191 fault tree branch combinations from the valves alone. Including the pumps increases the number of permutations to 13,171.

In order to simplify, the branches were sorted into event areas which were relevant to each of the components where failure was a concern. The components anticipated to fail were listed as Pipeline 1, Pipeline 2, Pipeline 3, Pipeline 4, Tank 1, and Tank 2. Additionally, logic was built into the scenario to warn users against running pumps dry or from cycling the pumps too often. This was in response to the suspected attack on the Virginia Water Treatment Facility [CNET 2014]. Pipeline 4 was protected because it was shut down and was considered locked out and tagged out from the proposed scenario. The selection method was conducted to allow ‘pruning’ of large number of irrelevant branch permutations to simplify the fault tree. For example, the branch generated for Valve 10 and Valve 12 have no impact on Pipeline 1’s pressure. As such, it would make little sense to include them in the fault tree for Pipeline 1.

This method did not initially include negative system states, or states where it was necessary to think of a valve as closed. Branches needed to be generated to consider all possible states; in this case the states were binary (open or closed versus open, closed, or in an intermediate state). The final improvement to the method was to simplify sets of valves together as either ‘sinks’ or ‘sources’. Sinks to a pipeline would be any complete combination of valves that allows for draining from a pipeline. A source is any complete set of valves that would allow flow into a pipeline. For example, Valve 1 and Valve 2 are a complete set of valves which feeds into Pipeline 1. Alternately, Valve 2 and Valve 4 also comprise a complete source from Tank 1 if Tank 1 is not empty. Valve 5, Valve 9, Valve 11 and Valve 13 are a complete sink which allows flow out of Pipeline 1. Additionally, Valve 5 and Valve 8 can be considered a complete sink if Tank 2 is not full. This simplification allowed for checking of states in the fault tree quickly and easily. Figure 4.3 is the final example of the fault tree

generated by this method for Pipeline 1, and was drawn using L^AT_EX. The pipeline was also generated in OpenFTA. Pruning allowed us to reduce the number of total permutations to 29 by using the sources and sinks. The sinks and sources list is provided in Figure 6.1.

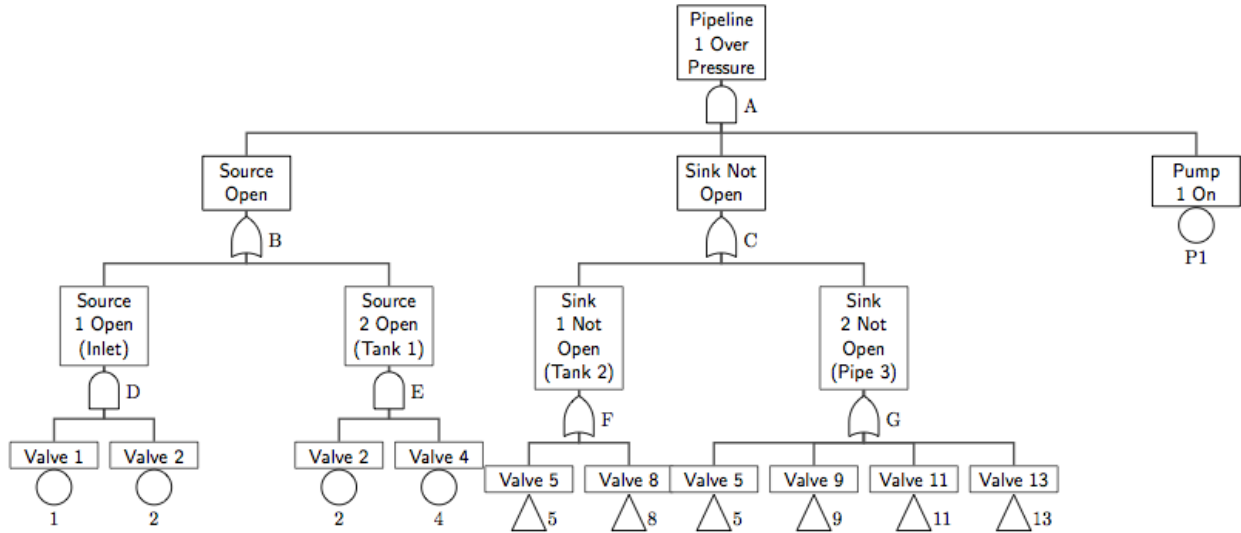


Figure 4.3: Pipeline 1 Fault Tree Example

In order to develop this test case further, it is worthwhile to show the cut set generated by OpenFTA. The cut set given is provided in Table 4.1.

Table 4.1: Cut Set for Pipeline 1

Pump1	TankFull	Valve1	Valve2	Valve4
Pump1	Valve1	Valve2	Valve5	Valve9

The cut set provides a number of means to protect Pipeline 1 from overpressure and possible rupture. The means decided for the experiment in the reconfigured test was to proactively shut off Pump 1 whenever the sink to Pipeline 1 was closed while the source was open. By controlling the pumps, at least in this scenario, it becomes impossible to overpressurize the varying lengths of pipeline. The lengths of pipeline are long enough that the pressure is not maintained far down each line. This is why intermediate pumping stations

would be required in the first place. The change in the ladder logic then becomes what is shown in Figure 4.4.

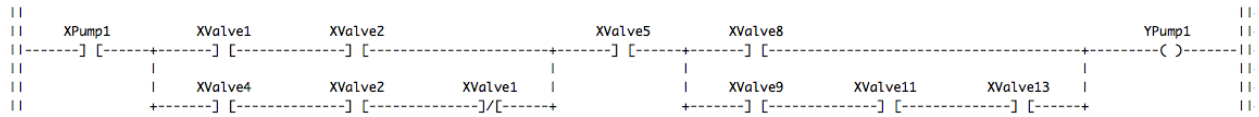


Figure 4.4: Pump 1 Control Rung

This process was conducted until it was believed that no further unsafe system states remained. Figures 6.9 and 6.10 provide the fully improved ladder logic.

4.3.3 Subjects

Based on previously referenced experiments similar to this one and the hypothesis to be tested, it was decided to recruit 25 test subjects required for meaningful results [Whitten 2004] [Whitten and Tygar 1999] [Ikuma 2013] [Garfinkel 2005]. Experimentation was halted after 25 test subjects were tested largely because an initial review of the data gathered showed meaningful results after 16 participants.

Test subjects were recruited via email and flyer after permission was granted from the Institutional Review Board (IRB) at Auburn University. Sample IRB documents are included in the Appendix. Test subjects were screened for computer competency via an online survey. A copy of the survey is located in the Appendix. While most of the test subjects were graduate or undergraduate engineering students, some of the test subjects were professionals working in the local area.

4.3.4 Experimental Apparatus

Test subjects were invited the Shelby Center at Auburn University for the experiment. The experiment was conducted on a desktop computer running Windows 7 that was preloaded with all of the software required. The test computer is shown in Figure 6.3.

Gaze tracking data was captured using the EyeTribe tracking device on 10 of the participants. Additionally, mouse tracking data was gathered using IO graph. Unfortunately, even though gaze tracking data was captured, analysis tools have not yet been made available to analyze the data files collected via the EyeTribe gaze tracker. It is hoped that this data can be analyzed and presented in a follow on publication.

Mouse tracking yielded some unique insight as to how users navigated the HMI screen. Lines indicate where the mouse cursor was moved, while the circles generated indicate mouse clicks. In this case, the larger the circle the more frequently a mouse was clicked in that area. These results are discussed in this section as they are largely speculative.

The mouse movement seems to correlate with both the tasks that users are accomplishing, as well as where they are looking on the screen. Some of the errant mouse clicks are the results of the test switching between the survey screen, the main menu, and the main experiment screen. The seemingly errant clicks probably indicate where the test subject was clicking for the survey before the experiment began. Additional mouse tracking overlays are provided in the Appendix.

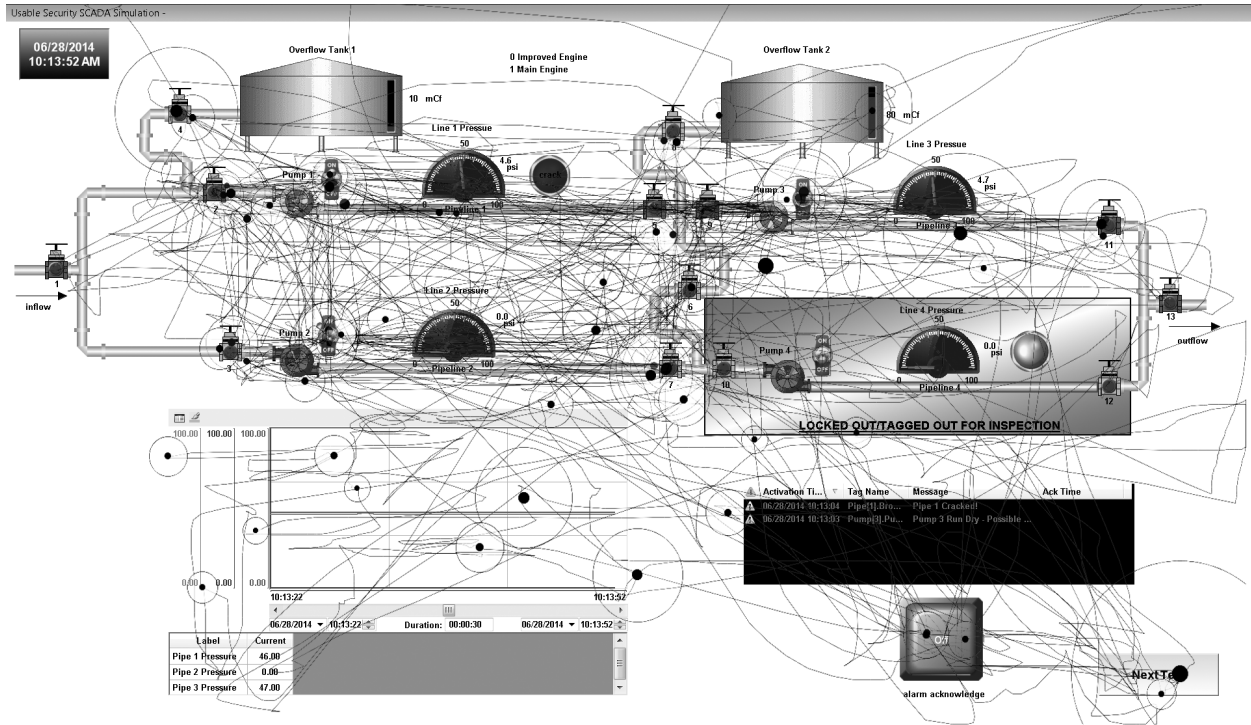


Figure 4.5: IOgraph Mouse Tracking Overlay

4.3.5 Protocol

Subjects were greeted at a time scheduled by the participant and the investigator. Prior to participating in the experiment participants provided informed consent after the investigator fielded any questions or requests for information.

Upon starting the experiment, participants were asked to sit at the computer while any workstation adjustments were made if required for the gaze tracking equipment. The participants were then asked to begin, and were greeted with the screen shown in Figure 4.6.

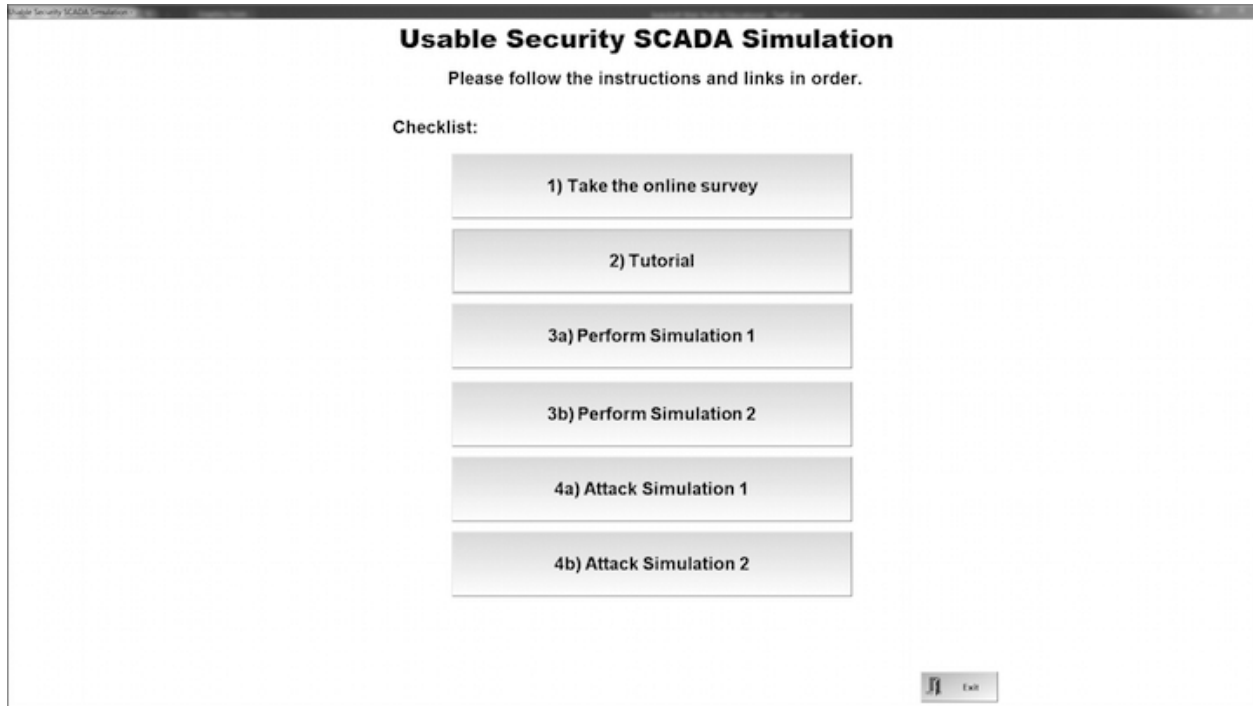


Figure 4.6: Main Menu

The main menu guided participants first to the survey intended to gauge the participants' level of computer competency. The survey was conducted as a control to ensure that participants had a high enough level of computer competency in order to successfully navigate the experiment. No subjects were excluded due to computer competency concerns.

After completing the survey, the main menu screen guided participants to a brief tutorial. A set of sample controls was displayed as shown in Figure 4.7, and participants were prompted by on screen instructions as to their use. Upon completion, participants were returned to the main menu. The main menu guided the participants to button 3a which started the usability experiment. When this window was opened, a visual basic script selected at random between the version of the test that had undergone code analysis and reconfiguration, or the version that had not. The investigator, through a special prompt on the screen, was able to see which version was selected in order to ensure the test was behaving correctly. In each case, the opposite version would be selected for the follow-on test.

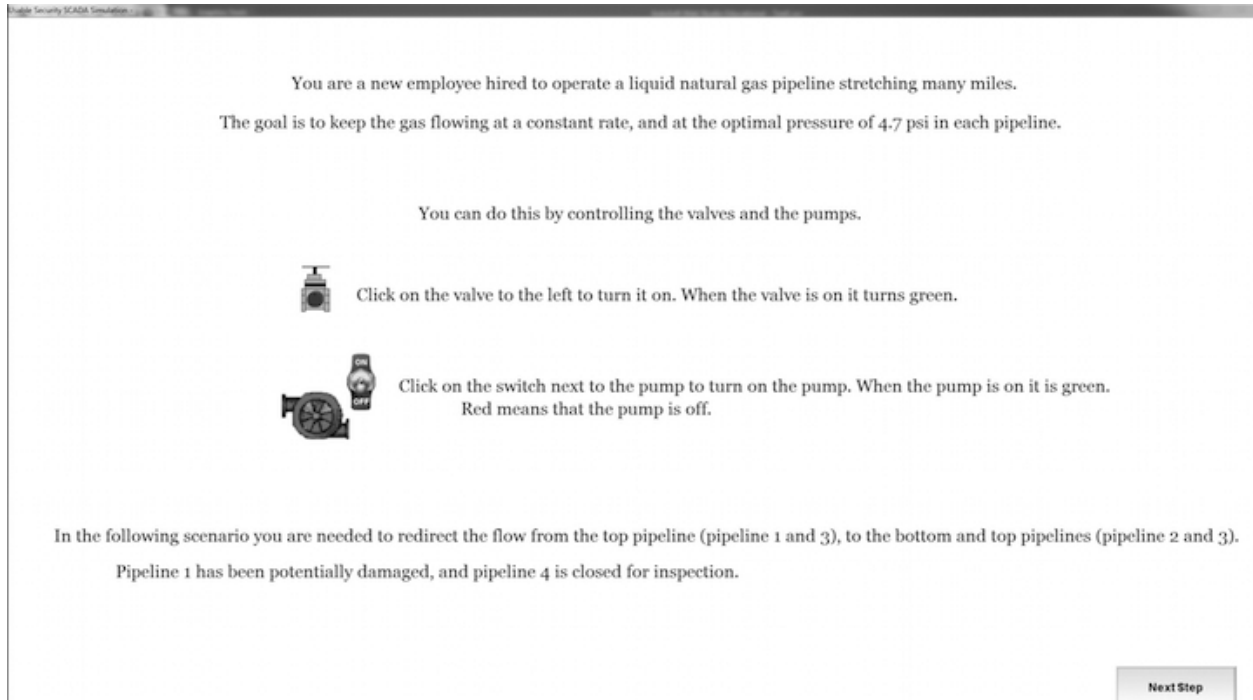


Figure 4.7: Tutorial Screen

The participants were then read the following script after being returned to the main menu.

“You are currently operating a pipeline responsible for transporting highly volatile liquids over a great distance. The pipeline is configured so that all flow is going through pipelines 1 and 3. This is because pipeline 4 is locked out and tagged out for its required bi-annual inspection, or ‘pigging’ operation. The pressure in pipeline 1 and 3 is currently at the ideal 4.7 psig (shown as psi in the HMI). You have just received a phone call stating that a farmer has backed into pipeline 1 and external damage is visible. Further inspection is required. You have been directed to reconfigure the flow of the pipeline from pipelines 1 and 3 to pipelines 2 and 3. Do so without increasing the pressure in any of the pipelines if possible. All steps will be considered complete when the flow has been successfully redirected

and the pressure for pipeline 1 reads 0, pipeline 2 read 4.7, and pipeline 3 reads 4.7. Please begin when you are ready."

Participants were not prompted to ask questions, but if questions were asked the questions were answered until the participant was satisfied. The time to completion, actions, and number and types of errors made were recorded automatically by a data-logger. As a precautionary step the investigator recorded the time the experiment started until the time that the pressures in all pipelines had either reached the appropriate level or until all required steps were taken. If damage was caused to the pipeline, it is possible that the correct pressure might not be reached. The investigator also recorded all errors made and the order in which those errors took place. The data points gathered were time in seconds to task completion, and the number of alarm states that the pipeline experienced.

An error was recorded only when system alarms occurred indicating a warning or mock physical damage. A specific order of operations could be conducted which would not place the pipeline in an unsafe state and would reconfigure the pipeline. Additionally, there were orders of operations which could be completed quickly enough which, even though the pipeline might temporarily be in an unsafe system state, would not result in a system alarm.

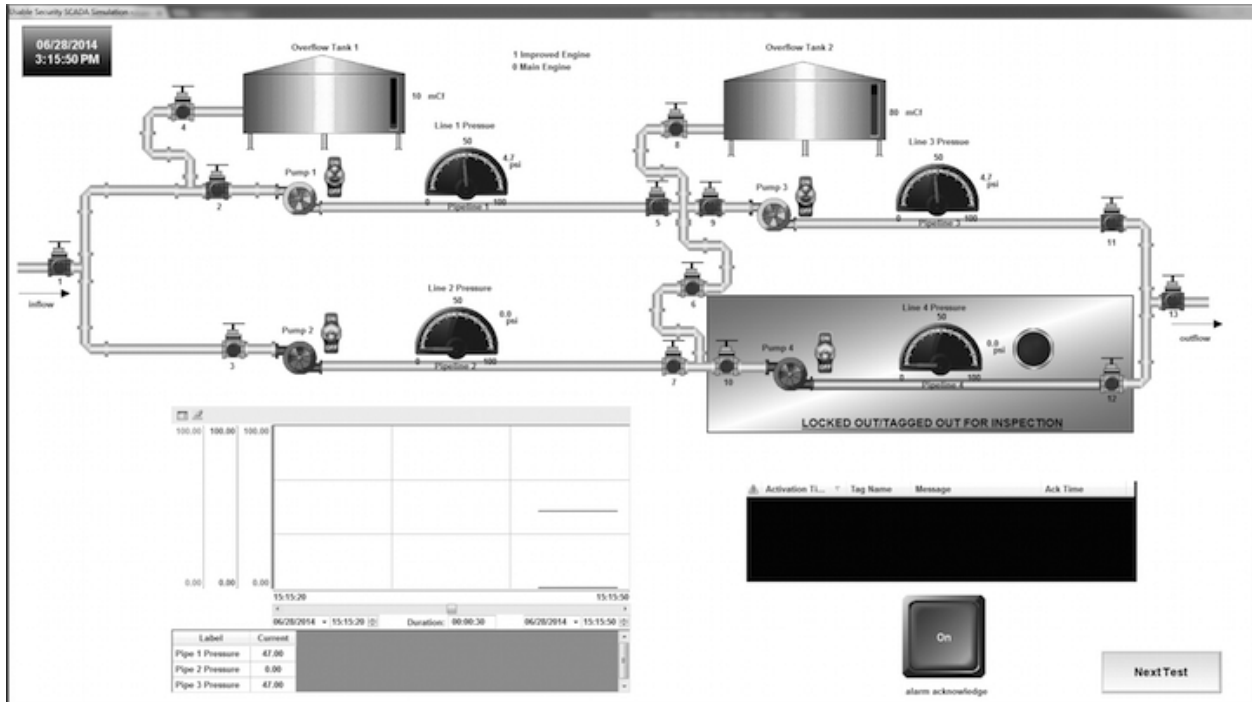


Figure 4.8: Experiment Screen

After the test was completed, the user was returned to the main menu where the next test, 3b, would be selected. The same steps were followed with the same information being recorded. After completing this test the user was returned to the main menu.

From the main menu the same experiment screen was displayed with random selection of which version would be used to drive the experiment. The users were read the following script for this section of the experiment:

"In this scenario you are now an attacker. You have managed to circumvent the security of a major pipeline company. Your goal is to now cause as much damage as possible. You will be allotted five minutes unless you wish to stop before then. If you would like to stop sooner please say so, and if you would like to have more time at the five minute warning please notify the investigator and more time will be allotted."

Again, questions were allowed, but the participants were not prompted. Any questions asked would be answered until the test participant was satisfied. The same model as the usability test was followed. Time to completion, actions, and number and misconfigurations made was recorded automatically by a data-logger. As a precautionary step, the investigator simultaneously recorded the time the experiment started until the participant stated that they were finished. The experiment was conducted one more time with the alternate engine.

4.4 Results

All twenty five subjects completed the experiment. There were 11 possible errors, or successful attacks depending on the stage of the experiment, which could be made by the users. Errors and attacks were logged as any system alarm or warning, and are distinguished by being either accidental or intentional misconfiguration errors. Both of the reconfigured systems had fewer misconfiguration errors. For a normal reconfiguration, on average, users made 1.92 misconfiguration errors per attempt. The results are discussed below with respect to each hypothesis tested.

4.4.1 Time to Completion Difference for Users

The results, as given in Table 4.2, indicated that there is no significant difference in the mean time for completing the Normal and Reconfigured tests. The p-value associated with the paired difference test is 0.7511. A small p-value less than (0.05) indicates that there is a statistically significant difference between the average time to complete each test. Based on the p-value of 0.7511, there is no difference in the average time of completing either the Normal or Reconfigured tests. In looking at the information further, a possible outlier was discovered.

Table 4.2: Descriptive Statistics for Time to Completion on Each Test in Seconds

Test	Mean	Median	Standard Deviation	IQR
Normal: User	142.4	130	111.9	83.5
Reconfigured: User	135.56	143	60.7	118

Subject 9 spent 10 minutes and 13 seconds on the first task that they were presented with. Subject 9, in discussing with the investigator, revealed that they were green color blind and had a difficult time seeing the state of the valves. The valves alternated between light grey (off) and green (on) to indicate their state. Colorblindness was not a factor checked in the survey, and was not considered before testing as a possible limitation. Discussion with the test subject revealed that they had no problem identifying the state of the pump switches as they more visibly moved up and down. As such, Subject 9 was deemed an outlier and his time scores discarded. Table 4.3 provides the new time data with Subject 9’s time removed.

Table 4.3: Descriptive Statistics for Time to Completion on Each Test in Seconds: No Outlier

Test	Mean	Median	Standard Deviation	IQR
Normal: User	122.79	129.5	55.09	78.5
Reconfigured: User	134.75	125	61.88	119

The new paired test gave us the following: p-value=.2659. This p-value is smaller than the original p-value with Subject 9 in the analysis, but shows that there is no significant difference in the times to task completing for the experiment with the code analysis method, and without. No other outliers were found in the any of the data collected.

4.4.2 Errors Made by Users

The number of errors the subjects made on the reconfiguration test were compared. This data is shown in Table 4.4 An error was recorded only when a system alarm sounded signifying theoretical physical damage would have occurred to the pipeline or when a warning was recorded. Examples included rupturing Pipeline 1 or Pipeline 3, over pressuring Pipeline

2, running any of the pumps without the flow of liquid, or overfilling and rupturing the tanks. The p-value associated with this test was less than 0.0001. From this we see that there are significant differences in the number of accidental misconfigurations that the subjects made on the model that had undergone code analysis and the model that had not. The code analysis method, in this experiment for this model pipeline SCADA system, appears to have a significant effect on decreasing the number of errors the subjects made.

Table 4.4: Descriptive Statistics for the Errors in Each Test Based on Test Subject

Test	Mean	Median	Mode	Standard Deviation
Normal: User	1.92	2	2	1.222
Reconfigured: User	0.08	0	0	0.4
Normal: Attack	4.84	5	4.5	1.772
Reconfigured: Attack	0	0	0	0

4.4.3 Successful Attacks by Attackers

The number of errors that the subjects made while ‘attacking’ the model pipeline SCADA system that had undergone code analysis and the model that had not were also compared. This data is shown in Table 4.4 and Table 4.5. The p-value for this test was less than 0.0001. As with the errors for the usability test, there was a significant decrease in the number of intentional misconfigurations that the subjects made on the model that had undergone code analysis in comparison to the model that had not. The code analysis method appears to have a significant effect on decreasing the number of intentional misconfigurations. In this way, there is a strong correlation between improving usability for safety and improving usability for security. By addressing the problem with possible misconfiguration errors by users, it may also be shown the intentional misconfigurations by attackers can be prevented.

Table 4.5: Successful Attacks Made Per Test

Test	Normal: User	Reconfigured: User	Normal: Attack	Reconfigured: Attack
Tank 1 Overflow	1	0	16	0
Tank 2 Overflow	1	0	17	0
Pipe 1 Cracked	7	0	20	0
Pipe 2 Overpressure	5	0	18	0
Pipe 3 Cracked	4	0	12	0
Pump 1 Run Dry	12	0	14	0
Pump 2 Run Dry	11	2	10	0
Pump 3 Run Dry	6	0	12	0
Pump 1 Cycling	1	0	2	0
Pump 2 Cycling	0	0	0	0
Pump 3 Cycling	1	0	0	0
Total	48	2	121	0

4.5 Discussion

Several of the shortcomings of this research have already been mentioned. The static code analysis method described works with the common functions available to both ladder logic and HDL which are easily translatable to the components and logic used in FTA. However, the vocabulary used in FTA can be limiting, while the vocabulary available to HDL and ladder logic are very expressive. An example of this problem came in that few of the FTA editors tested support the ‘NOT’ logical symbol which is commonly used in ladder logic. As such, compromises have had to be used occasionally.

Creating fault trees using this method may also encounter formatting issues as the fault trees generated begin to include more nodes. As the fault trees grow in complexity readability decreases. A vector based graphical solution might be needed for a more complex system in order to view the fault trees from varying levels of abstraction. This brings us to a possible problem with one of the focuses of this study.

One of the focuses of this method is on security. Defense in depth, while previously criticized as possibly proving unsustainable and inadequate to the problem, is still necessary even if this method is successful in identifying and removing possible misconfigurations. Recent SCADA attack research has shown the ability to overwrite PLC logic that would subvert the efforts made in applying this static code analysis method [Queiroz et al. 2009].

An additional limitation of this method is the reliance on a significant level of manual interference and system knowledge to audit the fault trees generated. It is difficult to measure the cost and benefit of applying this method. This is due in large part to the wide variety of tasks for which PLCs are used. There has been a great deal of success in the language used in ladder logic programming in this study, and to mitigate the possibility of encountering a program where this method will fall short, the method has been tried on a variety of ladder logic programs running on hardware and multiple virtualization tools.

The experiment has several limitations that need to be discussed as well. Safety equipment, such as pressure relief valves, was not included in the HMI being tested. This is viewed as a minor shortcoming, but is possibly limiting as to the realism of the test. Validation was conducted as rigorously as possible in regards to the model, but without real world data it is difficult to say how valid the model pipeline was.

Test subjects create another interesting possible shortcoming. The majority of test subjects were graduate level engineering students. A small set of the test subjects were working professionals of varying backgrounds. While the data does not seem to suggest that the mixing of test pools backgrounds had any effect, it is currently unknown if the results would have been more indicative. Additionally, it is recognized that trained pipeline operators would be likely to have different results than the SCADA naive test subjects available.

Finally, the reliance on Visual Basic to provide interaction is a possible limiting factor in the experiment. It is difficult to perfectly model the physical world, and while efforts were

made to ensure experimental realism, without having a real world system to measure against it is difficult to know the level of realism attained.

4.6 Conclusion

Neither pipeline models resulted in any significant difference in regards to task completion time for this experiment. However, the code analysis method in this experiment for this model pipeline SCADA system showed significant reduction in both the number of intentional and accidental misconfiguration that users or attackers could make. Thus, the static code analysis method did help to significantly decrease the number of misconfigurations and possibly unsafe system states that could occur in this experiment, and in this way helped to improve security, safety, and usability while providing a indicator of code correctness.

5.1 Introduction

Static code analysis is used to provide a means of indicating correctness to a programmer with having to execute the program. It is the hypothesized hope that lessons from static code analysis can be used to help provide the same indication with regards to system safety, security, and usability. The method designed and tested in this research is seemingly useful for identifying potentially unsafe system states in ladder logic written for Programmable Logic Controllers, and can potentially be used to improve usability and security. While this study is not foolproof method, the experiments conducted have shown promise in the scope to which it was applied.

5.2 Summary of Findings

A method of static code analysis useful to usable security in SCADA systems was studied, and a collection of usability experiments were conducted in order to explore that method. The findings of the research are summarized below.

1. Completion of tasks for a system which had undergone the previously described code analysis method showed no difference for task completion time in comparison to a nearly identical system where the code analysis method had not been applied.
2. The number of errors made by test subjects was greatly decreased for the experiment conducted when the code analysis method was applied. It appears that misconfiguration errors can be thought of as an engineering design problem, and that identifying

and removing possible misconfiguration states, can be a highly effective means of reducing possible user errors. This may be only true in cases where ladder logic can be analyzed and where the misconfiguration errors can be successfully removed, or a level of control applied.

3. The number of successful attacks made by an intruder can be greatly decreased for the experiment conducted when the code analysis method developed has been applied. It appears that when reducing the possible number of unsafe states resulting from user misconfigurations, the number of unsafe states available to attackers is also reduced. In this way, at least in regards to the experiment conducted, increasing the level of usability by reducing possible misconfiguration states potentially increases the security of a system.

5.3 Limitations of Research

Several of the shortcomings of this research have already been mentioned. The static code analysis method described works with the common functions available to both ladder logic and HDL which are easily translatable to the components and logic used in FTA. However, the vocabulary used in FTA can be limiting, while the vocabulary available to HDL and ladder logic are very expressive. An example of this problem came in that few of the FTA editors tested support the ‘NOT’ logical symbol which is commonly used in ladder logic. As such, compromises have had to be used occasionally.

With the complication in languages comes a problem with the simplicity of the method described. As shown in Figure 3.15 versus Figure 3.16, the fault trees generated using this method can be more complex than necessary, and may need to be simplified. This leads to an additional issue with complexity.

Creating fault trees using this method may also encounter formatting issues as the fault trees generated begin to include more nodes. As the fault trees grow in complexity readability decreases. A vector based graphical solution might be needed for a more complex system in

order to view the fault trees from varying levels of abstraction. This brings us to a possible problem with one of the focuses of this study.

One of the focuses of this method is on security. Defense in depth, while previously criticized as possibly proving unsustainable and inadequate to the problem, is still necessary even if this method is successful in identifying and removing possible misconfigurations. Recent SCADA attack research has shown the ability to overwrite PLC logic that would subvert the efforts made in applying this static code analysis method [Queiroz et al. 2009].

An additional limitation of this method is the reliance on a significant level of manual interference and system knowledge to audit the fault trees generated. It is difficult to measure the cost and benefit of applying this method. This is due in large part to the wide variety of tasks for which PLCs are used. There has been a great deal of success in the language used in ladder logic programming in this study, and to mitigate the possibility of encountering a program where this method will fall short, the method has been tried on a variety of ladder logic programs running on hardware and multiple virtualization tools.

The experiment has several limitations that need to be discussed as well. Safety equipment, such as pressure relief valves, was not included in the HMI being tested. This is viewed as a minor shortcoming, but is possibly limiting as to the realism of the test. Validation was conducted as rigorously as possible in regards to the model, but without real world data it is difficult to say how valid the model pipeline was.

Test subjects create another interesting possible shortcoming. The majority of test subjects were graduate level engineering students. A small set of the test subjects were working professionals of varying backgrounds. While the data does not seem to suggest that the mixing of test pools backgrounds had any effect, it is currently unknown if the results would have been more indicative. Additionally, it is recognized that trained pipeline operators would be likely to have different results than the SCADA naive test subjects available.

Finally, the reliance on Visual Basic to provide interaction is a possible limiting factor in the experiment. It is difficult to perfectly model the physical world, and while efforts were made to ensure experimental realism, without having a real world system to measure against it is difficult to know the level of realism attained.

5.4 Recommendations for Future Research

- The first step in future research is to analyze the eye tracking data against the mouse tracking data as soon as the tools are available. This data could help to lend additional information as to how operators are viewing the system, and where their focus is being held.
- The static code analysis method described in this study could be made more useful through automation. It is recommended that this area be explored further to identify the level of complexity which is capable of being analyzed. The automation can also benefit from the inclusion of tools to create cuts sets and to simplify the fault trees created. OpenFTA is recommended for this as it is open source and available for possible extension.
- Given the large vocabulary available to ladder logic and HDL there is benefit to be gained from attempting to replicate that capability using the logic available to FTA. In this way, the method can be extended to include increasingly complex programs. It may also be necessary to examine petri nets to further extend this method.
- More work needs to be conducted into improving the readability of increasingly complex fault trees. It is recommended to study vector graphics approaches, and to integrate with tools that allow varying levels of abstraction.
- Additional experiments can be conducted based on this method which include the safety hardware excluded from the current model.

Bibliography

- [Adams and Sasse 1999] Anne Adams and Martina Angela Sasse. 1999. Users are not the enemy. *Commun. ACM* 42, 12 (1999), 40–46.
- [Adams and Sasse 2001] Anne Adams and Martina Angela Sasse. 2001. Privacy in multimedia communications: Protecting users, not just data. *PEOPLE AND COMPUTERS* (2001), 49–64.
- [Allen-Bradley 1997] Allen-Bradley. 1997. *MicroLogix 1000 Programmable Controllers User Manual*. Allen-Bradley.
- [Auvation 2014] Auvation. 2014. Open FTA. (2014). <http://www.openfta.com/> 00002.
- [Bardas 2010] Alexandru G Bardas. 2010. Static Code Analysis. *Journal of Information Systems & Operations Management* 4, 2 (2010), 99–107.
- [Black and Garner 1999] Henry Campbell Black and Bryan A Garner. 1999. *Black’s law dictionary*. West Publishing Company.
- [Bloomberg 2014] Bloomberg. 2014. UglyGorilla Hack of U.S. Utility Exposes Cyberwar Threat - Bloomberg. (2014). <http://www.bloomberg.com/news/2014-06-13/uglygorilla-hack-of-u-s-utility-exposes-cyberwar-threat.html> 00000.
- [Bolton 2009] William Bolton. 2009. *Programmable logic controllers*. Newnes.
- [Broadwin 2014] Broadwin. 2014. WebAccess SCADA Node - Web enabled SCADA software for industrial. (2014). <http://broadwin.com/SCADA.htm> 00000.
- [Brumley et al. 2011] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *Computer Aided Verification*. Springer, 463–469.
- [Byres et al. 2004] Eric J Byres, Matthew Franz, and Darrin Miller. 2004. The use of attack trees in assessing vulnerabilities in SCADA systems. In *Proceedings of the International Infrastructure Survivability Workshop*.
- [CERT 2014] CERT. 2014. ICSB-11-327-01 - ILLINOIS WATER PUMP FAILURE REPORT | ICS-CERT. (2014). <http://ics-cert.us-cert.gov/tips/ICSB-11-327-01> 00000.
- [CERT/CC 2013] CERT/CC. 2013. CERT/CC. *From Internet: URL cert.org* (2013).

- [Chan 2012] Chee-Sing Chan. 2012. Complexity is the Enemy of Security @ONLINE. (2012). https://www.computerworld.com/s/article/9234815/Complexity_the_worst_enemy_of_security
- [Clemens 2002] PL Clemens. 2002. Fault tree analysis. *JE Jacobs Severdurup* (2002).
- [Clemens and Simmons 1998] PL Clemens and RJ Simmons. 1998. System safety and risk management: A guide for engineering educators. *NIOSH Instruction module. CDC, US Dept Health and Human Services VIII-1–VIII-8* (1998).
- [CNET 2014] CNET. 2014. Hacker says he broke into Texas water plant, others - CNET. (2014). <http://www.cnet.com/news/hacker-says-he-broke-into-texas-water-plant-others/> 00000.
- [Collins et al. 1997] Jason A Collins, Jim E Greer, Vive S Kumar, Gordon I McCalla, Paul Meagher, and Ray Tkatch. 1997. Inspectable user models for just-in-time workplace training. *COURSES AND LECTURES-INTERNATIONAL CENTRE FOR MECHANICAL SCIENCES* (1997), 327–338.
- [Daneels and Salter 1999] Axel Daneels and Wayne Salter. 1999. What is SCADA. In *International Conference on Accelerator and Large Experimental Physics Control Systems*. 339–343.
- [DeLong 1970] Thomas W DeLong. 1970. *A fault tree manual*. Technical Report. DTIC Document.
- [DHS 2009] NCSD DHS. 2009. Recommended Practice: Improving Industrial Control Systems Cybersecurity with Defense-In-Depth Strategies. (2009).
- [Dix 2004] Alan Dix. 2004. *Human computer interaction*. Pearson Education.
- [Edge 2007] Kenneth S Edge. 2007. *A framework for analyzing and mitigating the vulnerabilities of complex systems via attack and protection trees*. Technical Report. DTIC Document.
- [Erickson 1996] L Erickson. 1996. Programmable logic controllers. *Potentials, IEEE* 15, 1 (1996), 14–17.
- [Falco et al. 2002] Joe Falco, Keith Stouffer, Albert Wavering, and Frederick Proctor. 2002. *IT security for industrial control systems*. Citeseer.
- [Fernandez and Fernandez 2005] John D Fernandez and Andres E Fernandez. 2005. SCADA systems: vulnerabilities and remediation. *Journal of Computing Sciences in Colleges* 20, 4 (2005), 160–168.
- [Finkle 2012] Jim Finkle. 2012. US probes cyber attack on water system. (2012).
- [Fowler 1999] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

- [Garfinkel 2005] Simson Garfinkel. 2005. *Design principles and patterns for computer systems that are simultaneously secure and usable*. Ph.D. Dissertation. Massachusetts Institute of Technology.
- [Geer 2006] David Geer. 2006. Security of critical control systems sparks concern. *Computer* 39, 1 (2006), 20–23.
- [Guido van Rossum 2013] et al. Guido van Rossum. 2013. PEP8. <http://www.python.org/dev/peps/pep-0008/#introduction>. (2013). Accessed: 2013-11-22.
- [Hildick-Smith 2005] Andrew Hildick-Smith. 2005. Security for critical infrastructure scada systems. *SANS Reading Room, GSEC Practical Assignment, Version 1* (2005).
- [Ikuma 2013] Laura Ikuma. 2013. Interfacing with Safety. *Industrial Engineer* 45 (2013), 34–37. Issue 10.
- [Isograph 2014] Isograph. 2014. Fault Tree Analysis - Isograph. (2014). <http://www.isograph.com/software/reliability-workbench/fault-tree-analysis/> 00000.
- [ItemSoftware 2014] ItemSoftware. 2014. Fault Tree Software from ITEM Software. (2014). <http://www.itemsoft.com/faulttree.html> 00000.
- [Leveson 2011] Nancy G Leveson. 2011. *Engineering a safer world: Systems thinking applied to safety*. MIT Press.
- [Leveson et al. 1991] Nancy G. Leveson, Stephen S. Cha, and Timothy J. Shimeall. 1991. Safety verification of ada programs using software fault trees. *Software, IEEE* 8, 4 (1991), 48–59.
- [Logilab 2013] Logilab. 2013. Pylint. <http://www.pylint.org/>. (2013). Accessed: 2013-11-22.
- [Louridas 2006] Panagiotis Louridas. 2006. Static code analysis. *Software, IEEE* 23, 4 (2006), 58–61.
- [Mank 1992] Bradford C Mank. 1992. Preventing Bhopal: Dead Zones and Toxic Death Risk Index Taxes. *Ohio St. LJ* 53 (1992), 761.
- [Mannan and Lees 2005] Sam Mannan and Frank P Lees. 2005. *Lee’s loss prevention in the process industries: hazard identification, assessment, and control*. Vol. 1. Elsevier.
- [Martensen and Butler 1987] Anna L Martensen and Ricky W Butler. 1987. *The fault-tree compiler*. National Aeronautics and Space Administration, Langley Research Center.
- [McCabe 2013] Tom McCabe. 2013. Why Static Analysis Isn’t Good Enough. <https://www.youtube.com/watch?v=T6IKMBPMCL8>. (2013). Accessed: 2013-11-22.
- [Mil Std 882 2002] DoD Mil Std 882. 2002. 882. *System Safety Program Requirements* (2002).

- [Moon 1994] Il Moon. 1994. Modeling programmable logic controllers for logic verification. *Control Systems, IEEE* 14, 2 (1994), 53–59.
- [Moon et al. 1992] Il Moon, Gary J Powers, Jerry R Burch, and Edmund M Clarke. 1992. Automatic verification of sequential control systems using temporal logic. *AICHE Journal* 38, 1 (1992), 67–75.
- [Moore 2010] Robert Moore. 2010. *Cybercrime: Investigating high-technology computer crime*. Elsevier.
- [Moriarty 1990] Brian Moriarty. 1990. *System safety engineering and management*. Wiley.com.
- [Mozina 2005] Charles J Mozina. 2005. Power Plant Horror Stories. *BECKWITH ELECTRIC CO, INC* (2005).
- [Murphy-Hill and Black 2010] Emerson Murphy-Hill and Andrew P Black. 2010. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*. ACM, 5–14.
- [Murphy-Hill and Black 2012] Emerson Murphy-Hill and Andrew P Black. 2012. Seven habits of a highly effective smell detector. *RSSE* 8 (2012), 36–40.
- [NERC 2008] Standard NERC. 2008. Cyber Security, CIP-002 through CIP-009. (2008).
- [Newsome and Song 2005] James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *Carnegie Mellon Research Showcase* (2005).
- [Nielsen 1994] Jakob Nielsen. 1994. *Usability engineering*. Access Online via Elsevier.
- [NRC 2002] NRC. 2002. *Making the nation safer: the role of science and technology in countering terrorism*. National Academies Press.
- [NTSB 1993] NTSB. 1993. *Highly Volatile Release from Underground Storage Cavern and Explosion, Mapco Natural Gas Liquids, Inc, Accident Report, PAR-93-01*. Technical Report. NTSB.
- [NTSB 2000] NTSB. 2000. *Natural Gas Pipeline Rupture and Fire Near Carlsbad, New Mexico, Accident Report, NTSB/PAR-03/01*. Technical Report. NTSB.
- [NTSB 2001] NTSB. 2001. *Columbia Gas Transmission Corporation Pipeline Rupture, Sissonville, West Virginia, Accident Report, NTSB/PAR-14/01*. Technical Report. NTSB.
- [NTSB 2002a] NTSB. 2002a. *Supervisory Control and Data Acquisition (SCADA) in Liquid Pipelines, Safety Study, NTSB/SS-05/02*. Technical Report. NTSB.
- [NTSB 2002b] NTSB. 2002b. *Supervisory Control and Data Acquisition (SCADA) in Liquid Pipelines, Safety Study, NTSB/SS-05/02*. Technical Report. NTSB.

- [Panetta 2012] Leon Panetta. 2012. Remarks by Secretary Panetta on Cybersecurity to the Business Executives for National Security. *Transcript, Intrepid Sea, Air and Space Museum, New York, NY* (2012).
- [Queiroz et al. 2009] Carlos Queiroz, Abdun Mahmood, Jiankun Hu, Zahir Tari, and Xinghuo Yu. 2009. Building a SCADA security testbed. In *Network and System Security, 2009. NSS'09. Third International Conference on*. IEEE, 357–364.
- [Ralston et al. 2007] PAS Ralston, JH Graham, and JL Hieb. 2007. Cyber security risk assessment for SCADA and DCS networks. *ISA transactions* 46, 4 (2007), 583–594.
- [Reed 2007] Thomas Reed. 2007. *At the abyss: an insider's history of the Cold War*. Random House LLC.
- [Reifer 1979] Donald J Reifer. 1979. Software failure modes and effects analysis. *Reliability, IEEE Transactions on* 28, 3 (1979), 247–249.
- [Sasse 2003] M Angela Sasse. 2003. Computer security: Anatomy of a usability disaster, and a plan for recovery. In *Proceedings of CHI 2003 Workshop on HCI and Security Systems*. Citeseer.
- [Schneier 1999] Bruce Schneier. 1999. Attack trees. *Dr. Dobbs's Journal* 24, 12 (1999), 21–29.
- [Simpson et al. 1989] John Andrew Simpson, Edmund SC Weiner, and others. 1989. *The Oxford english dictionary*. Vol. 2. Clarendon Press Oxford.
- [Small 2011] PE Small. 2011. Defense in Depth: An Impractical Strategy for a Cyber World. *SANS Institute, Bethesda* (2011).
- [Song et al. 2008] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Information systems security*. Springer, 1–25.
- [Swearingen et al. 2014] Michael Swearingen, Steven Brunasso, Joe Weiss, and Dennis Huber. 2014. What You Need to Know (and Don't) About the AURORA Vulnerability | POWER Magazine. (2014). <http://www.powermag.com/what-you-need-to-know-and-dont-about-the-aurora-vulnerability/?pagenum=3>
- [Uzam and Jones 1998] M Uzam and AH Jones. 1998. Discrete event control system design using automation Petri nets and their ladder diagram implementation. *The International Journal of Advanced Manufacturing Technology* 14, 10 (1998), 716–728.
- [Weiss 2014] Joe Weiss. 2014. Misconceptions about Aurora: Why Isn't More Being Done. (2014). <http://www.infosecisland.com/blogview/20925-Misconceptions-about-Aurora-Why-Isnt-More-Being-Done.html> 00000.

- [Whitten 2004] Alma Whitten. 2004. *Making security usable*. Ph.D. Dissertation. Princeton University.
- [Whitten and Tygar 1999] Alma Whitten and J Doug Tygar. 1999. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Proceedings of the 8th USENIX Security Symposium*, Vol. 99. McGraw-Hill.
- [Winter 1995] Mathias W Winter. 1995. *Software Fault Tree Analysis of an Automated Control System Device Written in Ada*. Technical Report. DTIC Document.
- [Wixon and Wilson 1997] Dennis Wixon and Chauncey Wilson. 1997. The usability engineering framework for product design and evaluation. *Handbook of human-computer interaction 2* (1997), 653–68.
- [Zeller 2011] Mark Zeller. 2011. Myth or reality: Does the Aurora vulnerability pose a risk to my generator?. In *Protective Relay Engineers, 2011 64th Annual Conference for*. IEEE, 130–136.
- [Zifferer 1994] S.C. Zifferer. 1994. Graphical interfaces for monitoring ladder logic programs. (June 14 1994). <http://www.google.com/patents/US5321829> US Patent 5,321,829.
- [Zoubek et al. 2003] Bohumir Zoubek, Jean-Marc Roussel, Martha Kwiatkowska, and others. 2003. Towards automatic verification of ladder logic programs. *Proceedings of IMACS-IEEE'CESA'03': 'Computational Engineering in Systems Applications'* (2003).
- [Zurko and Simon 1996] Mary Ellen Zurko and Richard T Simon. 1996. User-centered security. *Proceedings of the 1996 workshop on New security paradigms* (1996), 27–33.

Chapter 6
Appendices

6.1 IRB

6.1.1 IRB Approved Consent Form and Recruitment Documents

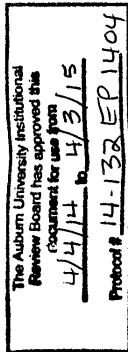


SIBBY CENTER FOR
ENGINEERING TECHNOLOGY
SUITE 3301
AUBURN, AL 36849-5346

TELEPHONE
334-844-4340

FAX
334-844-1381

www.auburn.edu



INFORMED CONSENT for a Research Study entitled
"Identifying Usable Security in Information Systems via Binary Analysis"

You are invited to participate in a research study to test identifying usability errors related to security via binary analysis. The study is being conducted by Christopher Perr, under the direction of Dr. Jerry Davis in the Auburn University Department of Industrial and Systems Engineering. You were selected as a possible participant because you are familiar with computer use and are age 19 or older.

What will be involved if you participate? If you decide to participate in this research study, you will be asked to participate in a simulation. Your total time commitment will be approximately 1 hour.

Are there any risks or discomforts? The risks associated with participating in this study are similar to web surfing for approximately 30 minutes. To minimize these risks, we will ensure proper computer setup and allow for breaks as necessary.

Are there any benefits to yourself or others? If you participate in this study, there may be an opportunity for compensation dependent on funding. I cannot promise you that you will receive any or all of the benefits described.

Are there any costs? If you decide to participate there will be no costs to you.

If you change your mind about participating, you can withdraw at any time during the study. Your participation is completely voluntary. If you choose to withdraw, your data can be withdrawn as long as it is identifiable. Your decision about whether or not to participate or to stop participating will not jeopardize your future relations with Auburn University, the Department of Industrial and Systems Engineering or any person involved in this study.

Your privacy will be protected. Any information obtained in connection with this study will remain confidential. Information obtained through your participation may be used to fulfill an educational requirement, published in a professional journal, presented at a professional meeting, etc.

If you have questions about this study, please ask them now or contact Christopher Perr at cwperr@gmail.com or Dr. Jerry Davis at davisga@auburn.edu. A copy of this document will be given to you to keep.

If you have questions about your rights as a research participant, you may contact the Auburn University Office of Research Compliance or the Institutional Review Board by phone (334)-844-5966 or e-mail at IRBadmin@auburn.edu or IRBChair@auburn.edu.

Participant's initials _____

Page 1 of 2



SHELBY CENTER FOR
ENGINEERING TECHNOLOGY
SUITE 5301
AUBURN, AL 36849-5346

TELEPHONE
334-844-4340

FAX
334-844-1381

www.auburn.edu

The Auburn University Institutional
Review Board has approved this
recruitment for use from
4/2/14 to 4/3/15
Protocol # 14-132 EP 1404

HAVING READ THE INFORMATION PROVIDED, YOU MUST DECIDE WHETHER OR NOT YOU WISH TO PARTICIPATE IN THIS RESEARCH STUDY. YOUR SIGNATURE INDICATES YOUR WILLINGNESS TO PARTICIPATE.

Participant's signature Date Investigator obtaining consent Date

Printed Name

Printed Name

Co-Investigator Date

Printed Name

Usable Security Research Study
Would you like to be a part of a Computer Security and Usability Study?

Are you interested in computer security? Are you interested in usability? What about SCADA systems?

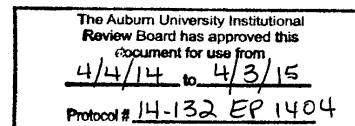
If you answered **YES** to any of these questions, you may be eligible to participate in a study to help improve usable security.

The purpose of this research study is to determine the effectiveness of a new method to identify usable security issues through dynamic binary analysis and static code analysis.

Any adult with basic computer literacy is eligible.

This study is being conducted by the Industrial and Systems Engineering Department at Auburn University.

Please contact Christopher Perr at cwp0002@tigermail.auburn.edu for more information.



6.1.2 IRB Research Protocol Review Form and Submitted Documents

**AUBURN UNIVERSITY INSTITUTIONAL REVIEW BOARD for RESEARCH INVOLVING HUMAN SUBJECTS
RESEARCH PROTOCOL REVIEW FORM**

For Information or help contact THE OFFICE OF RESEARCH COMPLIANCE, 115 Ramsay Hall, Auburn University
Phone: 334-844-5966 e-mail: hsubject@auburn.edu Web Address: <http://www.auburn.edu/research/vpr/ohs/>

Revised 03.26.11 -- DO NOT STAPLE, CLIP TOGETHER ONLY.

Save a Copy

1. PROPOSED START DATE of STUDY: Apr 1, 2014

PROPOSED REVIEW CATEGORY (Check one): FULL BOARD EXPEDITED EXEMPT

2. PROJECT TITLE: Improving Usable Security and Systems Safety via Path Mapping and Software Behavior Analysis

3. Christopher W. Perr
PRINCIPAL INVESTIGATOR TITLE INSY DEPT 4195122124 PHONE cwp0002@tigermail.auburn.edu AU E-MAIL

2061 Felicity Lane, Auburn, AL 36830
MAILING ADDRESS

FAX ALTERNATE E-MAIL

4. SOURCE OF FUNDING SUPPORT: Not Applicable Internal External Agency: _____ Pending Received

5. LIST ANY CONTRACTORS, SUB-CONTRACTORS, OTHER ENTITIES OR IRBs ASSOCIATED WITH THIS PROJECT:

6. GENERAL RESEARCH PROJECT CHARACTERISTICS

6A. Mandatory CITI Training	6B. Research Methodology								
<p>Names of key personnel who have completed CITI: Christopher W. Perr</p> <hr/> <hr/> <hr/> <p align="center">CITI group completed for this study: <input checked="" type="checkbox"/> Social/Behavioral <input type="checkbox"/> Biomedical</p> <p align="center">PLEASE ATTACH TO HARD COPY ALL CITI CERTIFICATES FOR EACH KEY PERSONNEL</p>	<p>Please check all descriptors that best apply to the research methodology.</p> <p>Data Source(s): <input checked="" type="checkbox"/> New Data <input type="checkbox"/> Existing Data</p> <p>Will recorded data directly or indirectly identify participants? Yes <input type="checkbox"/> No <input checked="" type="checkbox"/></p> <p>Data collection will involve the use of:</p> <p><input type="checkbox"/> Educational Tests (cognitive diagnostic, aptitude, etc.)</p> <p><input checked="" type="checkbox"/> Interview / Observation</p> <p><input type="checkbox"/> Physical / Physiological Measures or Specimens (see Section 6E.)</p> <p><input checked="" type="checkbox"/> Surveys / Questionnaires</p> <p><input checked="" type="checkbox"/> Internet / Electronic</p> <p><input type="checkbox"/> Audio / Video / Photos</p> <p><input type="checkbox"/> Private records or files</p>								
6C. Participant Information	6D. Risks to Participants								
<p>Please check all descriptors that apply to the participant population.</p> <p><input checked="" type="checkbox"/> Males <input checked="" type="checkbox"/> Females <input checked="" type="checkbox"/> AU students</p> <p>Vulnerable Populations</p> <p><input type="checkbox"/> Pregnant Women/Fetuses <input type="checkbox"/> Prisoners</p> <p><input type="checkbox"/> Children and/or Adolescents (under age 19 in AL)</p> <p>Persons with:</p> <p><input type="checkbox"/> Economic Disadvantages <input type="checkbox"/> Physical Disabilities</p> <p><input type="checkbox"/> Educational Disadvantages <input type="checkbox"/> Intellectual Disabilities</p> <p>Do you plan to compensate your participants? Yes <input type="checkbox"/> No <input checked="" type="checkbox"/></p>	<p>Please identify all risks that participants might encounter in this research.</p> <table border="0"> <tr> <td><input type="checkbox"/> Breach of Confidentiality*</td> <td><input type="checkbox"/> Coercion</td> </tr> <tr> <td><input type="checkbox"/> Deception</td> <td><input type="checkbox"/> Physical</td> </tr> <tr> <td><input type="checkbox"/> Psychological</td> <td><input type="checkbox"/> Social</td> </tr> <tr> <td><input checked="" type="checkbox"/> None</td> <td><input type="checkbox"/> Other:</td> </tr> </table> <hr/> <hr/> <hr/> <p>*Note that if the investigator is using or accessing confidential or identifiable data, breach of confidentiality is always a risk.</p>	<input type="checkbox"/> Breach of Confidentiality*	<input type="checkbox"/> Coercion	<input type="checkbox"/> Deception	<input type="checkbox"/> Physical	<input type="checkbox"/> Psychological	<input type="checkbox"/> Social	<input checked="" type="checkbox"/> None	<input type="checkbox"/> Other:
<input type="checkbox"/> Breach of Confidentiality*	<input type="checkbox"/> Coercion								
<input type="checkbox"/> Deception	<input type="checkbox"/> Physical								
<input type="checkbox"/> Psychological	<input type="checkbox"/> Social								
<input checked="" type="checkbox"/> None	<input type="checkbox"/> Other:								
<p>Do you need IBC Approval for this study? <input checked="" type="checkbox"/> No <input type="checkbox"/> Yes - BUA # _____ Expiration date _____</p>									

FOR OHSR OFFICE USE ONLY

DATE RECEIVED IN OHSR: _____ by _____ PROTOCOL # _____

DATE OF IRB REVIEW: _____ by _____ APPROVAL CATEGORY: _____

DATE OF IRB APPROVAL: _____ by _____ INTERVAL FOR CONTINUING REVIEW: _____

COMMENTS:

7. PROJECT ASSURANCES

PROJECT TITLE: Improving Usable Security and Systems Safety via Path Mapping and Software Behavior Analysis

A. PRINCIPAL INVESTIGATOR'S ASSURANCES

1. I certify that all information provided in this application is complete and correct.
2. I understand that, as Principal Investigator, I have ultimate responsibility for the conduct of this study, the ethical performance of this project, the protection of the rights and welfare of human subjects, and strict adherence to any stipulations imposed by the Auburn University IRB.
3. I certify that all individuals involved with the conduct of this project are qualified to carry out their specified roles and responsibilities and are in compliance with Auburn University policies regarding the collection and analysis of the research data.
4. I agree to comply with all Auburn policies and procedures, as well as with all applicable federal, state, and local laws regarding the protection of human subjects, including, but not limited to the following:
 - a. Conducting the project by qualified personnel according to the approved protocol
 - b. Implementing no changes in the approved protocol or consent form without prior approval from the Office of Human Subjects Research
 - c. Obtaining the legally effective informed consent from each participant or their legally responsible representative prior to their participation in this project using only the currently approved, stamped consent form
 - d. Promptly reporting significant adverse events and/or effects to the Office of Human Subjects Research in writing within 5 working days of the occurrence.
5. If I will be unavailable to direct this research personally, I will arrange for a co-investigator to assume direct responsibility in my absence. This person has been named as co-investigator in this application, or I will advise OHSR, by letter, in advance of such arrangements.
6. I agree to conduct this study only during the period approved by the Auburn University IRB.
7. I will prepare and submit a renewal request and supply all supporting documents to the Office of Human Subjects Research before the approval period has expired if it is necessary to continue the research project beyond the time period approved by the Auburn University IRB.
8. I will prepare and submit a final report upon completion of this research project.

My signature indicates that I have read, understand and agree to conduct this research project in accordance with the assurances listed above.

Christopher W. Perr

Printed name of Principal Investigator


Principal Investigator's Signature
(SIGN IN BLUE INK ONLY)

Mar 7, 2014

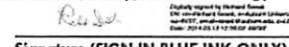
Date

B. FACULTY ADVISOR/SPONSOR'S ASSURANCES

1. By my signature as faculty advisor/sponsor on this research application, I certify that the student or guest investigator is knowledgeable about the regulations and policies governing research with human subjects and has sufficient training and experience to conduct this particular study in accord with the approved protocol.
2. I certify that the project will be performed by qualified personnel according to the approved protocol using conventional or experimental methodology.
3. I agree to meet with the investigator on a regular basis to monitor study progress.
4. Should problems arise during the course of the study, I agree to be available, personally, to supervise the investigator in solving them.
5. I assure that the investigator will promptly report significant adverse events and/or effects to the OHSR in writing within 5 working days of the occurrence.
6. If I will be unavailable, I will arrange for an alternate faculty sponsor to assume responsibility during my absence, and I will advise the OHSR by letter of such arrangements. If the investigator is unable to fulfill requirements for submission of renewals, modifications or the final report, I will assume that responsibility.
7. I have read the protocol submitted for this project for content, clarity, and methodology

Richard F. Sesek

Printed name of Faculty Advisor / Sponsor


Signature (SIGN IN BLUE INK ONLY)

3/7/2014

Date

C. DEPARTMENT HEAD'S ASSURANCE

By my signature as department head, I certify that I will cooperate with the administration in the application and enforcement of all Auburn University policies and procedures, as well as all applicable federal, state, and local laws regarding the protection and ethical treatment of human participants by researchers in my department.

Jorge Valenzuela

Printed name of Department Head


Signature (SIGN IN BLUE INK ONLY)

3/7/2014

Date

**8. PROJECT OVERVIEW: Prepare an abstract that includes:
(400 word maximum, in language understandable to someone who is not familiar with your area of study):**

- I.) A summary of relevant research findings leading to this research proposal:
(Cite sources; include a "Reference List" as Appendix A.)
- II.) A brief description of the methodology,
- III.) Expected and/or possible outcomes, and,
- IV.) A statement regarding the potential significance of this research project.

This project seeks to build upon the usability and security work completed by Whitten, as well as work completed by Garfinkle [71,30]. Both projects sought to reduce user errors in programs dependent on the application of cryptography. The methodology behind this research is to improve identification of usable security problems in information technology systems by applying binary analysis tools to identify possibly dangerous pathways through a program. In order to protect the system and the user from making misconfiguration errors, the graphical user interface of a program will be improved to prevent typical user errors. In order to test the ability to detect and solve usability issues related to security, user studies need to be conducted to gauge the level of improvement made. The basic model is to ask a user to perform a series of tasks on an insecure information technology system, and to perform the same tasks on an improved system. The time to task completion, number of mistakes made, and the type of mistakes will be recorded. The expected outcome is that the method will enable identification of usability issues related to security. The improved program should have fewer errors and time to task completion will be reduced. Given that it is estimated that misconfiguration errors account for approximately 90% of all security cases [12], the potential significance of this research could lead to a method of reducing the number of vulnerabilities created by user misconfiguration errors.

9. PURPOSE.

- a. Clearly state all of the objectives, goals, or aims of this project.

To better identify usability related security problems in information technology systems by applying dynamic code analysis techniques to current usability and security practices.

- b. How will the results of this project be used? (e.g., Presentation? Publication? Thesis? Dissertation?)

Presentation, Publication, and Dissertation. All data will be presented in aggregate form with no identifiers.

10a. KEY PERSONNEL. Describe responsibilities. Include information on research training or certifications related to this project. CITI is required. Be as specific as possible. (Attach extra page if needed.) All non AU-affiliated key personnel must attach CITI certificates of completion.

Principle Investigator Christopher W. Perr Title: Doctoral Candidate E-mail address cwp0002@tigermail.auburn.edu
Dept / Affiliation: INSY/Graduate Research Assistant

Roles / Responsibilities:

Recruitment and consent of participants
Data collection and analysis

Individual: Richard Seseck Title: PhD E-mail address seseck@auburn.edu
Dept / Affiliation: INSY/Assistant Professor

Roles / Responsibilities:

Advisor

Individual: Jerry Davis Title: PhD E-mail address davisga@auburn.edu
Dept / Affiliation: INSY/Associate Professor

Roles / Responsibilities:

Advisor

Individual: _____ Title: _____ E-mail address _____
Dept / Affiliation: _____

Roles / Responsibilities:

Individual: _____ Title: _____ E-mail address _____
Dept / Affiliation: _____

Roles / Responsibilities:

Individual: _____ Title: _____ E-mail address _____
Dept / Affiliation: _____

Roles / Responsibilities:

11. LOCATION OF RESEARCH. List all locations where data collection will take place. (School systems, organizations, businesses, buildings and room numbers, servers for web surveys, etc.) Be as specific as possible. Attach permission letters in Appendix E. (See sample letters at <http://www.auburn.edu/research/vpr/rohs/sample.htm>)

Human Factors Usability Lab, Shelby Center, Auburn University, Auburn, AL

12. PARTICIPANTS.

a. Describe the participant population you have chosen for this project.

Check here if there is existing data; describe the population from whom data was collected & include the # of data files.

Any computer literate person familiar with graphical user interface (GUI) type computing NOT an expert in computer security or supervisory control and data acquisition (SCADA) systems who is over 19 years old

b. Describe why is this participant population is appropriate for inclusion in this research project. (Include criteria for selection.)

This population should be representative of typical computer operator susceptible to making common misconfiguration errors.

c. Describe, step-by-step, all procedures you will use to recruit participants. Include in Appendix B a copy of all e-mails, flyers, advertisements, recruiting scripts, invitations, etc., that will be used to invite people to participate.

(See sample documents at <http://www.auburn.edu/research/vpr/ohs/sample.html>.)

Word of mouth through the INSY and CS departments at Auburn University. Those with interest will be given contact information for Christopher Perr. A poster and e-mail request have also been created.

What is the minimum number of participants you need to validate the study? ²⁰ _____

Is there a limit on the number of participants you will recruit? No Yes – the number is _____

Is there a limit on the number of participants you will include in the study? No Yes – the number is _____

d. Describe the type, amount and method of compensation and/or incentives for participants.

(If no compensation will be given, check here .)

Select the type of compensation: Monetary Incentives

Raffle or Drawing incentive (Include the chances of winning.)

Extra Credit (State the value)

Other

Description:

13. PROJECT DESIGN & METHODS.

- a. Describe, step-by-step, all procedures and methods that will be used to consent participants.
(Check here if this is "not applicable"; you are using existing data.)

Participants will be informed of the procedures, and read the letter of consent. By signing this document they will be consenting to participation.

- b. Describe the procedures you will use in order to address your purpose. Provide a step-by-step description of how you will carry out this research project. Include specific information about the participants' time and effort commitment. (NOTE: Use language that would be understandable to someone who is not familiar with your area of study. Without a complete description of all procedures, the Auburn University IRB will not be able to review this protocol. If additional space is needed for this section, save the information as a .PDF file and insert after page 6 of this form.)

Step 1: Use word of mouth, e-mail and flyers to identify interested parties.

Step 2: Speak with interested individuals to explain procedures.

The following steps will be randomized to eliminate the potential effects of a learning response.

Step 3/4: Improved TEST - Users will be asked to complete a series of tasks on a improved simulated graphical Human Machine Interface controlling a power generation and distribution SCADA system. The test should take between 15-30 minutes. The session will be screen captured, and the time for task completion and errors made will be recorded. A survey will also be included to gauge the participant's level of computer security expertise.

Step 4/3: Unimproved TEST - Users will be asked to complete a series of tasks on a unimproved simulated graphical Human Machine Interface controlling a power generation and distribution SCADA system. The test should take between 15-30 minutes. The session will be screen captured, and the time for task completion and errors made will be recorded. A survey will also be included to gauge the participant's level of computer security expertise.

When possible test participants in the lab setting will be using a computer with eye tracking enabled. No images of the users face will be recorded. The eye tracking is only used to record where a subject is focusing their gaze. No additional requirements will be added to the user, and the eye tracking system used does not require the participant to wear any additional equipment.

13e. List all data collection instruments used in this project, in the order they appear in Appendix C. (e.g., surveys and questionnaires in the format that will be presented to participants, educational tests, data collection sheets, interview questions, audio/video taping methods etc.)

Computer User Background Survey
EyeTribe eye tracking system
SCADA User Evaluation Software (randomized use between unimproved and improved)

d. Data analysis: Explain how the data will be analyzed.

Statistical analysis will be performed to analyze the difference between the improved and unimproved version of the user evaluation software in order to identify the level of improvement in reduction of user errors and task times based on the method for identifying usable security faults.

14. RISKS & DISCOMFORTS: List and describe all of the risks that participants might encounter in this research. If you are using deception in this study, please justify the use of deception and be sure to attach a copy of the debriefing form you plan to use in Appendix D. (Examples of possible risks are in section #8D on page 1.)

Breach of confidentiality
Minimal risk associated with computer use not beyond a brief period similar to Internet browsing.

- 15. PRECAUTIONS.** Identify and describe all precautions you have taken to eliminate or reduce risks as listed in #14. If the participants can be classified as a "vulnerable" population, please describe additional safeguards that you will use to assure the ethical treatment of these individuals. Provide a copy of any emergency plans/procedures and medical referral lists in Appendix D.

No personally identifiable information will be collected in the survey materials.

Computing time will be limited to two 15 minute sessions with a brief break in between sessions. The level of effort required will be similar to a brief web surfing session.

If using the Internet to collect data, what confidentiality or security precautions are in place to protect (or not collect) identifiable data? Include protections used during both the collection and transfer of data. (These are likely listed on the server's website.)

Not applicable

16. BENEFITS.

- a. List all realistic direct benefits participants can expect by participating in this specific study.

(Do not include "compensation" listed in #12d.) Check here if there are no direct benefits to participants.

Participation in this study may help to create a process for identifying usable security issues, and to reduce possible misconfiguration errors in a variety of systems. There will be no benefit beyond the knowledge of a contribution to possibly improving computer security.

- b. List all realistic benefits for the general population that may be generated from this study.

Participation in this study may help to create a process for identifying usable security issues, and to reduce possible misconfiguration errors in a variety of systems. The main benefit comes in improving security and safety of utilities and other public systems which may be vulnerable to cyber attacks.

17. PROTECTION OF DATA.

- a. Will data be collected as anonymous? Yes No *If "YES", skip to part "g".*
(*Anonymous* means that you will not collect any identifiable data.)
- b. Will data be collected as confidential? Yes No
(*Confidential* means that you will collect and protect identifiable data.)
- c. If data are collected as confidential, will the participants' data be coded or linked to identifying information?
 Yes (if so, describe how linked.) No

d. Justify your need to code participants' data or link the data with identifying information.

e. Where will code lists be stored? (Building, room number?)

f. Will data collected as "confidential" be recorded and analyzed as "anonymous"? Yes No
(If you will maintain identifiable data, protections should have been described in #15.)

g. Describe how and where the data will be stored (e.g., hard copy, audio cassette, electronic data, etc.), and how the location where data is stored will be secured in your absence. For electronic data, describe security. If applicable, state specifically where any IRB-approved and participant-signed consent documents will be kept on campus for 3 years after the study ends.

Electronic data will be stored in its original file format on an encrypted laptop. Hard copy data, while anonymous, will be protected by being kept in a home office. After hard copy materials have been scanned or otherwise transferred electronically, they will be shredded.

h. Who will have access to participants' data?

(The faculty advisor should have full access and be able to produce the data in the case of a federal or institutional audit.)

No individual personal participant data will be recorded. The only data recorded will be survey responses pertaining to computer knowledge, screenshots of errors, completion time and types of errors of testing, and anonymous eye tracking results. All data will be aggregated for analysis, and will be anonymous.

i. When is the latest date that confidential data will be retained? (Check here if only anonymous data will be retained. ✓)

j. How will the confidential data be destroyed? (NOTE: Data recorded and analyzed as "anonymous" may be retained indefinitely.)

PROTOCOL REVIEW CHECKLIST

All protocols must include the following items:

1. Research Protocol Review Form (All signatures included and all sections completed)

(Examples of appended documents are found on the OHSR website: <http://www.auburn.edu/research/vpr/ohs/sample.htm>)

2. Consent Form or Information Letter and any Releases (audio, video or photo) that the participant will sign.
3. Appendix A, "Reference List"
4. Appendix B if e-mails, flyers, advertisements, generalized announcements or scripts, etc., are used to recruit participants.
5. Appendix C if data collection sheets, surveys, tests, other recording instruments, interview scripts, etc. will be used for data collection. Be sure to attach them in the order in which they are listed in # 13c.
6. Appendix D if you will be using a debriefing form or include emergency plans/procedures and medical referral lists (A referral list may be attached to the consent document).
7. Appendix E if research is being conducted at sites other than Auburn University or in cooperation with other entities. A permission letter from the site / program director must be included indicating their cooperation or involvement in the project.
NOTE: If the proposed research is a multi-site project, involving investigators or participants at other academic institutions, hospitals or private research organizations, a letter of IRB approval from each entity is required prior to initiating the project.
8. Appendix F - Written evidence of acceptance by the host country if research is conducted outside the United States.

FOR FULL BOARD REVIEW, NUMBER ALL PAGES, INCLUDING APPENDICES

Usable Security Research Study
Would you like to be a part of a Computer Security and Usability Study?

Are you interested in computer security? Are you interested in usability? What about SCADA systems?

If you answered **YES** to any of these questions, you may be eligible to participate in a study to help improve usable security.

The purpose of this research study is to determine the effectiveness of a new method to identify usable security issues through dynamic binary analysis and static code analysis.

Any adult with basic computer literacy is eligible.

This study is being conducted by the Industrial and Systems Engineering Department at Auburn University.

Please contact Christopher Perr at cwp0002@tigermail.auburn.edu for more information.

E-MAIL INVITATION FOR EXPERIMENT

(This should be a brief version of the consent document.)

I am a graduate student in the Department of Industrial and Systems Engineering at Auburn University. I would like to invite you to participate in my research study to identify usability and security issues in information technology systems. You may participate if you are an adult of 19+ years old.

Participants will be asked to participate in two 30-minute sessions completing tasks on a simulated SCADA system.

While no compensation will be given, this is an opportunity to learn about SCADA systems and usable security issues.

If you would like to know more about this study, an information letter can be obtained by email from cwp0002@tigermail.auburn.edu. If you decide to participate after reading the letter, you can access the survey from a link in the letter.

If you have any questions, please contact me at cwp0002@tigermail.auburn.edu or my advisor, Dr. Jerry Davis, at davisga@auburn.edu.

Thank you for your consideration,
/signed/

C.W. Perr

**COLLABORATIVE INSTITUTIONAL TRAINING INITIATIVE (CITI)
COURSE IN THE PROTECTION HUMAN SUBJECTS CURRICULUM COMPLETION REPORT
Printed on 12/20/2013**

LEARNER	christopher perr (ID: 1033235) 2061 Felicity Lane Auburn AL 36830 United States
DEPARTMENT	INSY
PHONE	4195122124
EMAIL	cwp0002@tigermail.auburn.edu
INSTITUTION	Auburn University
EXPIRATION DATE	09/28/2013

SOCIAL/BEHAVIORAL RESEARCH COURSE : Choose this group to satisfy CITI training requirements for Investigators and staff involved primarily in biomedical research with human subjects.

COURSE/STAGE:	Basic Course/1
PASSED ON:	09/29/2008
REFERENCE ID:	2162658

REQUIRED MODULES	DATE COMPLETED	SCORE
Belmont Report and CITI Course Introduction	09/28/08	3/3 (100%)
Students in Research	09/29/08	9/10 (90%)
History and Ethical Principles - SBE	09/29/08	7/7 (100%)
Defining Research with Human Subjects - SBE	09/29/08	5/5 (100%)
Assessing Risk - SBE	09/29/08	5/5 (100%)
Informed Consent - SBE	09/29/08	4/4 (100%)
Privacy and Confidentiality - SBE	09/29/08	4/4 (100%)
Research with Children - SBE	09/29/08	5/5 (100%)
Internet Research - SBE	09/29/08	5/5 (100%)
Auburn University	09/29/08	No Quiz

For this Completion Report to be valid, the learner listed above must be affiliated with a CITI Program participating institution or be a paid Independent Learner. Falsified information and unauthorized use of the CITI Program course site is unethical, and may be considered research misconduct by your institution.

Paul Braunschweiger Ph.D.
Professor, University of Miami
Director Office of Research Education
CITI Program Course Coordinator

**COLLABORATIVE INSTITUTIONAL TRAINING INITIATIVE (CITI)
COURSE IN THE PROTECTION HUMAN SUBJECTS CURRICULUM COMPLETION REPORT**
Printed on 12/20/2013

LEARNER christopher perr (ID: 1033235)
2061 Felicity Lane
Auburn
AL 36830
United States

DEPARTMENT INSY

PHONE 4195122124

EMAIL cwp0002@tigermail.auburn.edu

INSTITUTION Auburn University

EXPIRATION DATE 12/19/2018

SOCIAL/BEHAVIORAL RESEARCH COURSE : Choose this group to satisfy CITI training requirements for Investigators and staff Involved primarily in biomedical research with human subjects.

COURSE/STAGE: Refresher Course/2
PASSED ON: 12/20/2013
REFERENCE ID: 10702401

REQUIRED MODULES	DATE COMPLETED	SCORE
SBE Refresher 1 – Defining Research with Human Subjects	12/20/13	2/2 (100%)
SBE Refresher 1 – Privacy and Confidentiality	12/20/13	2/2 (100%)
SBE Refresher 1 – Assessing Risk	12/20/13	2/2 (100%)
SBE Refresher 1 – Research with Children	12/20/13	2/2 (100%)
SBE Refresher 1 – International Research	12/20/13	2/2 (100%)
Biomed Refresher 2 - Instructions	12/20/13	No Quiz
SBE Refresher 1 – History and Ethical Principles	12/20/13	2/2 (100%)
SBE Refresher 1 – Federal Regulations for Protecting Research Subjects	12/20/13	2/2 (100%)
SBE Refresher 1 – Informed Consent	12/20/13	2/2 (100%)
SBE Refresher 1 – Research with Prisoners	12/20/13	2/2 (100%)
SBE Refresher 1 – Research in Educational Settings	12/20/13	2/2 (100%)
SBE Refresher 1 – Instructions	12/20/13	No Quiz
Biomed Refresher 2 – History and Ethical Principles	12/20/13	3/3 (100%)
Biomed Refresher 2 – Regulations and Process	12/20/13	2/2 (100%)
Biomed Refresher 2 – Informed Consent	12/20/13	3/3 (100%)
Biomed Refresher 2 – SBR Methodologies in Biomedical Research	12/20/13	4/4 (100%)
Biomed Refresher 2 – Genetics Research	12/20/13	2/2 (100%)
Biomed Refresher 2 – Records-Based Research	12/20/13	3/3 (100%)
Biomed Refresher 2 – Research Involving Vulnerable Subjects	12/20/13	1/1 (100%)
Biomed Refresher 2 – Vulnerable Subjects – Prisoners	12/20/13	2/2 (100%)
Biomed Refresher 2 – Vulnerable Subjects – Children	12/20/13	3/3 (100%)
Biomed Refresher 2 – Vulnerable Subjects – Pregnant Women, Human Fetuses, Neonates	12/20/13	2/2 (100%)
Biomed Refresher 2 – Conflicts of Interest in Research Involving Human Subjects	12/20/13	3/3 (100%)
Auburn University	12/20/13	No Quiz

For this Completion Report to be valid, the learner listed above must be affiliated with a CITI Program participating institution or be a paid Independent Learner. Falsified information and unauthorized use of the CITI Program course site is unethical, and may be considered research misconduct by your institution.

Paul Braunschweiger Ph.D.
Professor, University of Miami
Director Office of Research Education
CITI Program Course Coordinator

6.2 Indusoft Educational Agreement



www.InduSoft.com

EDUCATIONAL LICENSE AGREEMENT

Between InduSoft, Ltd. and Christopher W. Peav

This agreement is made and entered into on 15 April 2014 by and between InduSoft, Inc., a Texas corporation with principal offices at 11044 Research Blvd., Suite A-100, Austin, Texas 78759, USA (hereinafter "InduSoft") and Christopher W. Peav (hereinafter "Educational Partner"), whose principal place of business is Auburn, AL.

1.0 Purpose

The purpose of this agreement is to establish and define the relationship between InduSoft and Educational Partner in regards to the licensing of a valuable copyrighted software, InduSoft Web Studio, for use as a teaching and learning tool in a defined educational curriculum. The software must be used in a teaching environment. It may not be used for commercial purposes.

2.0 InduSoft Obligations Under This Agreement

InduSoft will issue either soft key or hard key licenses for the current version of InduSoft Web Studio as well as supporting documentation to the Educational Partner. InduSoft does not charge a fee for either the soft key or hard key. However, upon receipt of a hard key, a soft key, or some combination of the two, Educational Partner shall acknowledge such as a gift, in writing, and send such acknowledgement to InduSoft. InduSoft will also provide Educational Partner training materials for the InduSoft Web Studio course.

3.0 Educational Partner Obligations Under This Agreement

The Educational Partner agrees that all software provided by InduSoft under this agreement is provided to them for teaching and learning purposes *only*. All other uses must be approved by InduSoft prior to such use.

The Educational Partner promises and agrees not to allow use of the software for commercial uses or other non-academic purposes. Licenses granted under this agreement are issued only for computer hardware owned by the Educational Partner or in the control of the Educational Partner.

The Educational Partner agrees not to modify, alter, change, reproduce, reverse engineer, decompile or disassemble the software provided by InduSoft.

The Educational Partner may allow InduSoft the use of Educational Partner's facilities for educational seminars and/or presentations with the prior approval of the Educational Institution.

The Educational Partner gives permission for any educational materials submitted by Educational Partner to be used in InduSoft marketing materials and efforts. Any material that CAN NOT be shared outside of InduSoft should be designated as such before submitting to InduSoft.

4.0 Technical Support

Licenses are issued for the most recent version of the software at the time of signing the contract. InduSoft reserves the right to offer an update to your software for the most recent version at any time.

InduSoft will provide Educational Partner technical support under the terms of this agreement.

5.0 Point of Contract

InduSoft and the Educational Partner designate the following individuals as points of contract for the business relationship InduSoft and Educational Partner define in this agreement:

InduSoft:

Marcia Gadbois
InduSoft, Ltd.
11044 Research Blvd., Suite A100
Austin, TX 78759
Phone: 877-INDUSOFT (877-463-8763) or (512) 349-0334
Fax: (512) 349-0375
Email: marcia.gadbois@indusoft.com

Educational Partner:

PRIMARY CONTACT NAME Christopher W. Pew
EDUCATIONAL PARTNER Christopher W. Pew
DEPARTMENT INSY
ADDRESS 2061 Felicity Lane
ADDRESS 2 _____
City, State Zip Auburn, AL 36830
PHONE: 419 512 2124
Email: ewp0002@tigermail.auburn.edu

Technical Support:

TECHNICAL CONTACT NAME _____
EDUCATIONAL PARTNER _____
DEPARTMENT _____
ADDRESS _____
ADDRESS 2 _____
CITY, STATE, ZIP _____
PHONE: _____
Email: _____

6.0 No Agency

This Agreement does not establish either party as an agent of the other. Neither party may bind the other to any other contract or obligation whatsoever. At all times the parties agree that they are independent contractors of one another and shall not act on behalf of the other party in any capacity.

7.0 Warranty

INDUSOFT MAKES NO WARRANTY OF ANY KIND WITH RESPECT TO ITS SOFTWARE, EXPRESSED OR IMPLIED, INCLUDING WITHOUT LIMIT ANY WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT WILL INDUSOFT BE LIABLE FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE EDUCATIONAL PARTNER'S USE OF OR INABILITY TO USE THE INDUSOFT SOFTWARE OR FOR ANY CLAIM BY ANY OTHER PARTY IN SUCH A CAPACITY.

InduSoft warranties are described in the InduSoft License Agreement.

8.0 Non-Assignment

This agreement shall not be assigned or transferred by the Educational Partner under any circumstances unless written permission for transfer is obtained from an authorized representative of InduSoft.

9.0 Trademark

Educational Partner must seek prior approval before using InduSoft Trademarks. For purposes of this agreement, the term "Trademark(s)" shall mean these names and designations by which any of the InduSoft products are known, including but not limited to the name "InduSoft Web Studio" (IWS or any abbreviation thereof), "InduSoft CEView", "CEView", or "InduSoft."

10.0 Indemnification

Educational Partner agrees to indemnify and hold harmless InduSoft and its officers, directors, employees, and agents from and against any and all claims, demands, costs, and liabilities (including all reasonable attorney's fees) of any kind whatsoever, arising directly or indirectly out of any action or omission by Educational Partner, including without limitation, Educational Partner's performance or failure to perform under this agreement. However, when an Educational Partner is prevented from incurring such liability as a result of state or federal law, Educational Partner is exempt from this Clause.

11.0 Arbitration Clause

All disputes, claims, and controversies between the parties, whether individual, joint, or class in nature, arising from, or relating to this Agreement or any other relationship between the parties, including without limitation contract and tort disputes, shall be arbitrated pursuant to the Rules of the American Arbitration Association, to the extent they are not inconsistent with the provisions of this Section, upon request of either party. Judgment upon any award rendered by any arbitrator may be entered in any court having jurisdiction. Nothing in this Agreement shall preclude any party from seeking equitable or injunctive relief from a court of competent jurisdiction. The statute of limitations, estoppels, waivers, laches, and similar doctrines which would otherwise be applicable in an action brought by a party shall be applicable in any arbitration proceeding, and the commencement of an arbitration proceeding shall be deemed the commencement of an action for these purposes. The Federal Arbitration Act shall apply to the construction, interpretation, and enforcement of this arbitration provision. Discovery may be conducted in any such arbitration in accordance with the Federal Rules of Civil Procedure. The site of the arbitration shall be Austin, Texas.

12.0 Term of Agreement

This agreement is valid from the date of signing. Thereafter, the agreement shall remain in full force and effect unless either party notifies the other, providing 30 days notice of its intent to terminate. The right to terminate this agreement shall be available to either party at any time. This agreement shall be automatically terminated if any of the obligations set forth in this agreement are not fulfilled as agreed. Upon termination, all software shall be uninstalled and all copies of InduSoft Web Studio software and supporting materials distributed to the Educational Partner under this agreement shall be returned to InduSoft and no copies will be retained by the Educational Partner.

This agreement will continue in force for a period of twelve months. Annual revival of this agreement shall be automatic unless terminated by either party.

13.0 Entire Agreement

This agreement sets forth the entire agreement of understanding between InduSoft and the Educational Partner as to the use of the InduSoft Web Studio software distributed to

the Educational Partner, and the requirements which must be met in exchange for InduSoft providing these materials. InduSoft reserves the right to periodically audit Educational Partner's commitment to and fulfillment of the requirements of this agreement. This supersedes all prior agreements, negotiations, commitments, and discussions between the parties as to the subject matter herein.

14.0 Authorization

Both InduSoft and Educational Partner have caused this Agreement to be executed by their duly authorized representatives, signed and dated below:

EDUCATIONAL PARTNER

INDUSOFT INC.


Accepted By

Accepted By

Christopher W. Penn
Printed Name

Printed Name

Research Assistant
Title

Title

15 April 2014
Date

Date

6.3 Test Data

Table 6.1: Number of Errors for Each Test Organized by Subject

Subject	Normal: User	Reconfigured: User	Normal: Attack	Reconfigured: Attack
1	2	0	8	0
2	2	0	4	0
3	4	0	2	0
4	5	0	6	0
5	2	0	5	0
6	3	0	4	0
7	2	0	6	0
8	1	0	8	0
9	3	0	2	0
10	2	0	5	0
11	0	0	8	0
12	1	0	7	0
13	1	0	6	0
14	2	0	5	0
15	1	0	5	0
16	1	0	5	0
17	1	0	4	0
18	2	0	4	0
19	2	0	5	0
20	1	0	2	0
21	3	0	3	0
22	4	0	3	0
23	1	0	4	0
24	2	0	4	0
25	0	2	6	0
Total	48	2	121	0

6.4 Fault Tree Generation

6.4.1 Sinks and Sources Permutations

Listing 6.1: Sinks and Sources Permutations

Sources :

Pipeline 1

V1 V2

V4 V2

Pipeline 2

V1 V3

V4 V3

Pipeline 3

V1 V2 V5 V9

V1 V3 V7 V6 V9

V4 V2 V5 V9

V8 V9

Pipeline 4

V1 V2 V5 V6 V10

V1 V3 V7 V10

V4 V2 V5 V6 V10

V8 V6 V10

Tank 1

V1 V4

Tank 2

V1 V2 V5 V8

V1 V3 V7 V6 V8

Sinks :

Pipeline 1

V5 V8

V5 V9 V11 V13

Pipeline 2

V7 V10 V12 V13

V7 V6 V9 V11 V13

V7 V6 V8

Pipeline 3

V11 V13

Pipeline 4

V12 V13

Tank 1

V4 V2 V5 V9 V11 V13

V4 V3 V7 V10 V12 V13

V4 V2 V5 V6 V10 V12 V13

V4 V3 V7 V6 V9 V11 V13

V4 V2 V5 V8

Tank 2

V8 V9 V11 V13

V8 V6 V10 V12 V13

6.5 Test Hardware

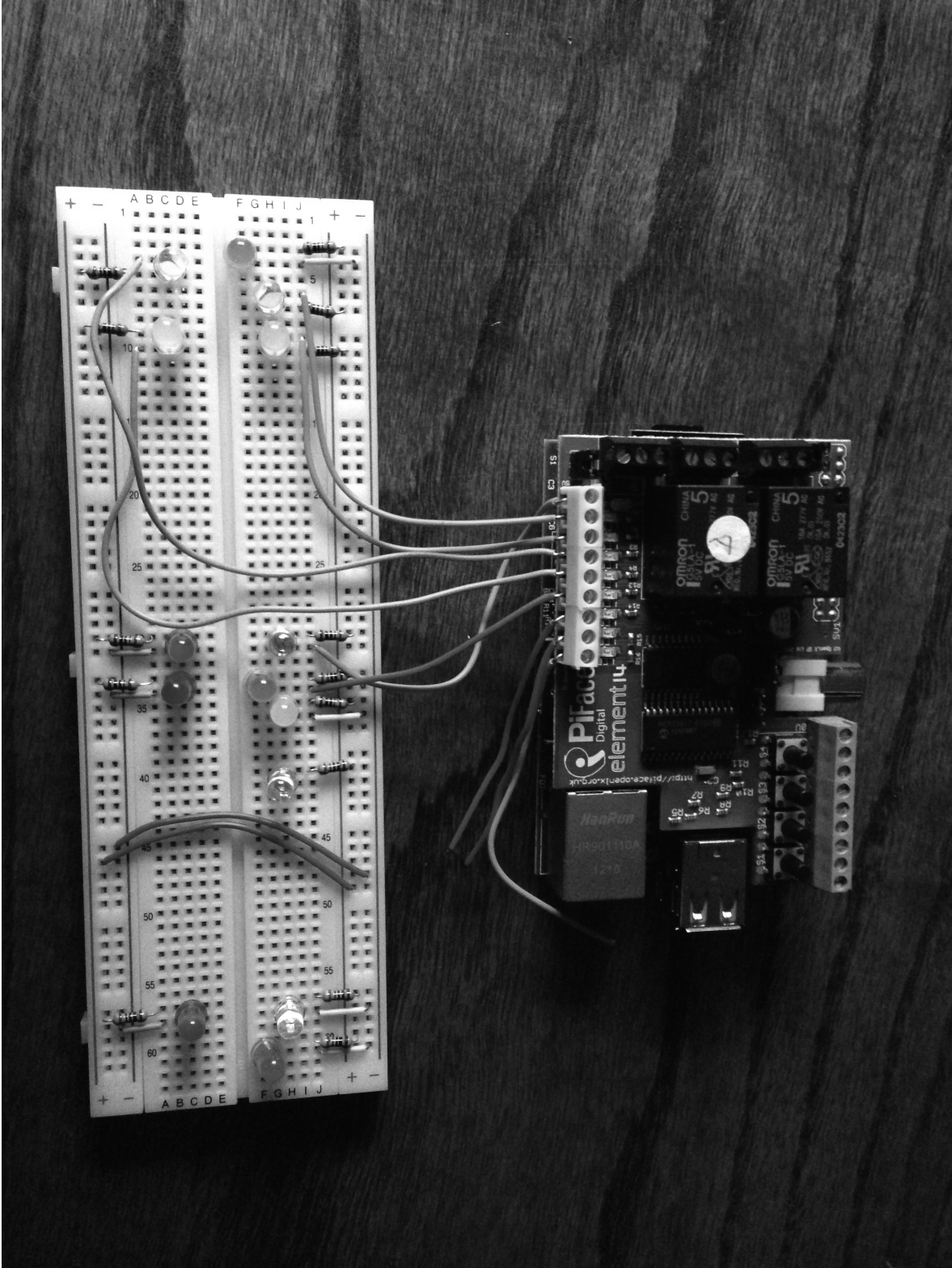


Figure 6.1: Raspberry Pi with PiFace



Figure 6.2: MicroLogix 1000



Figure 6.3: Test Computer

6.6 Screenshots

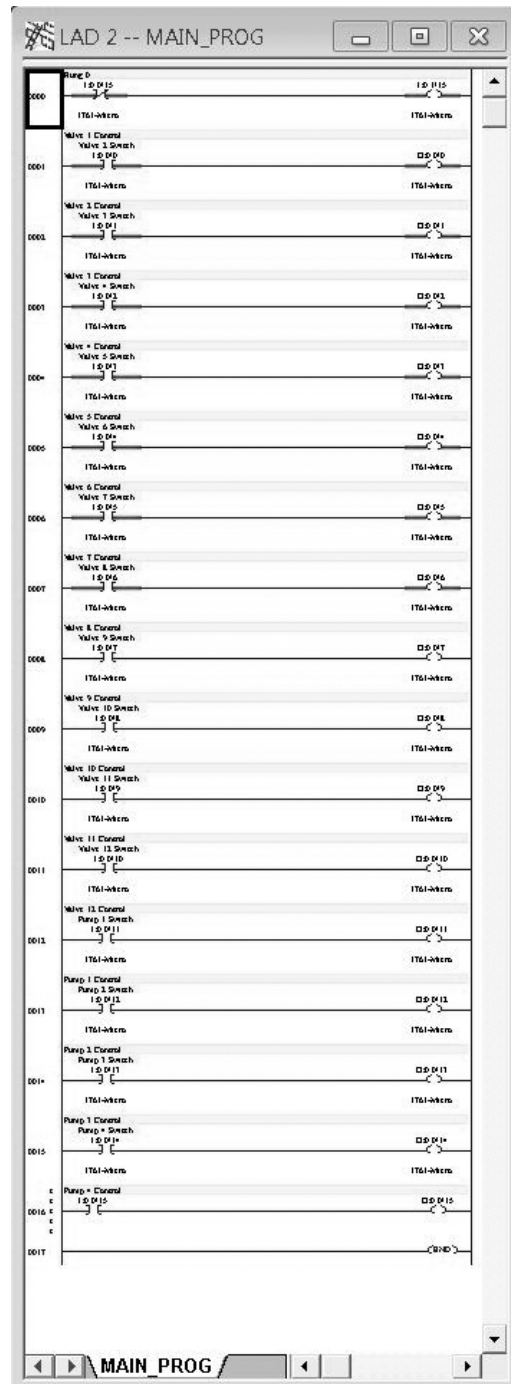


Figure 6.4: RSLogix Running Simple Test PLC Code

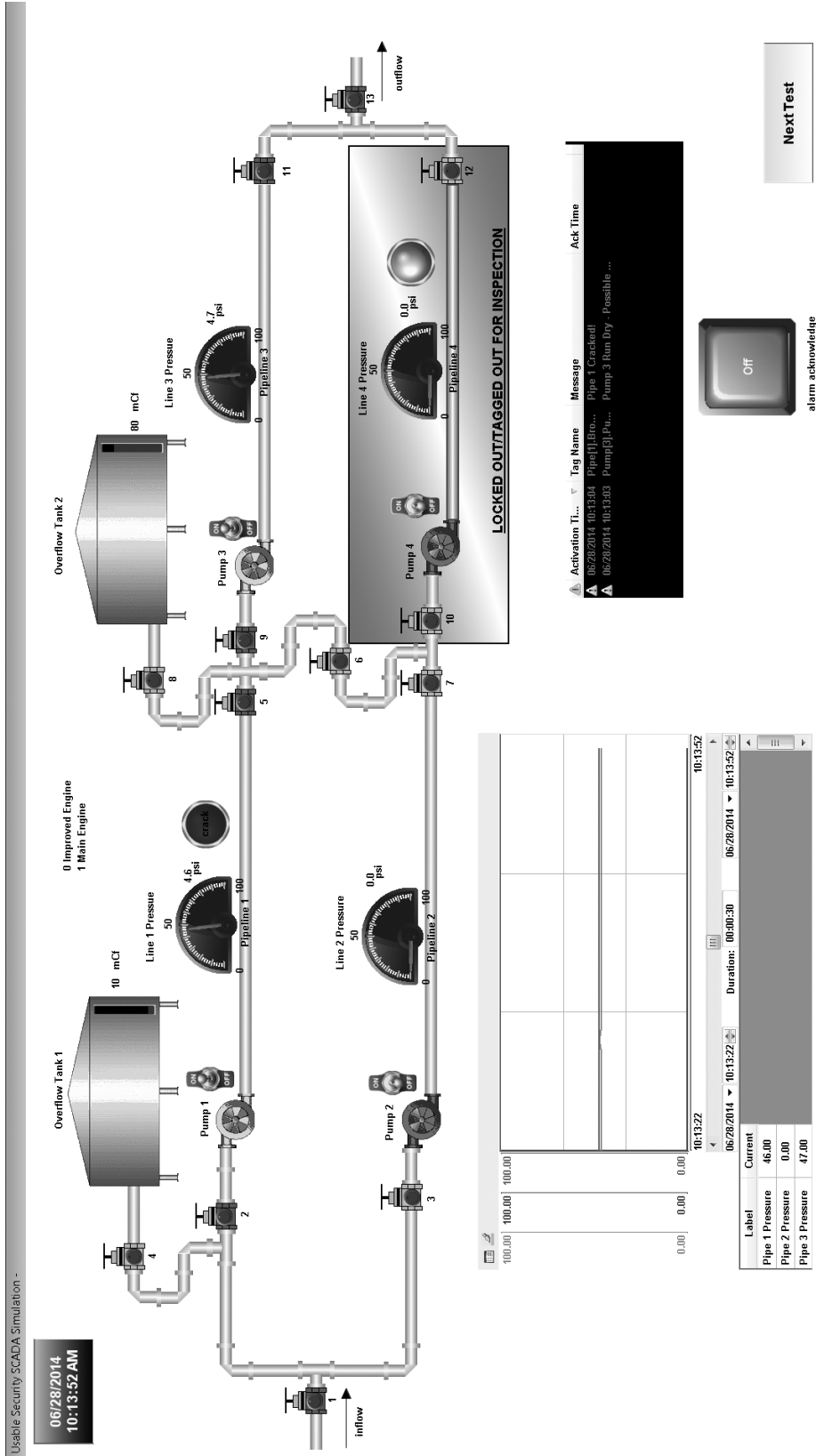


Figure 6.5: Large Version Pipeline Usable Security Simulator

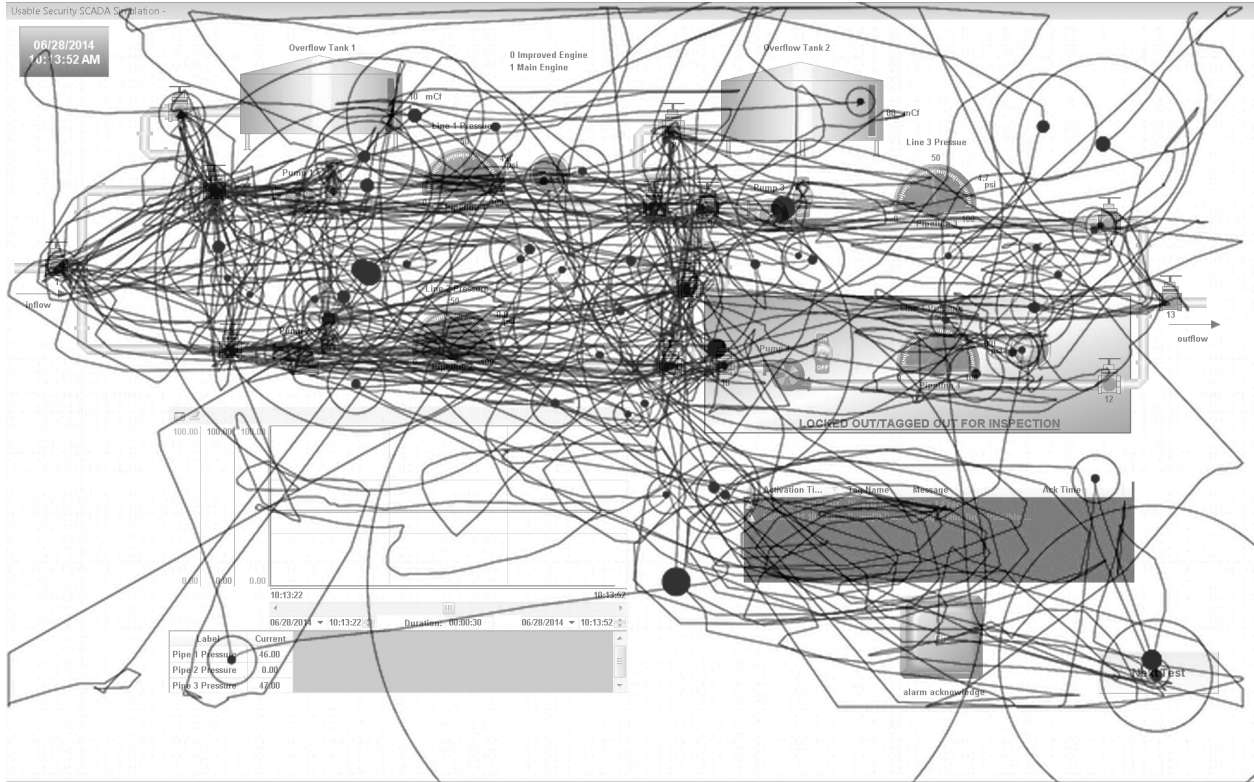


Figure 6.6: IOgraph Mouse Tracking Overlay 1

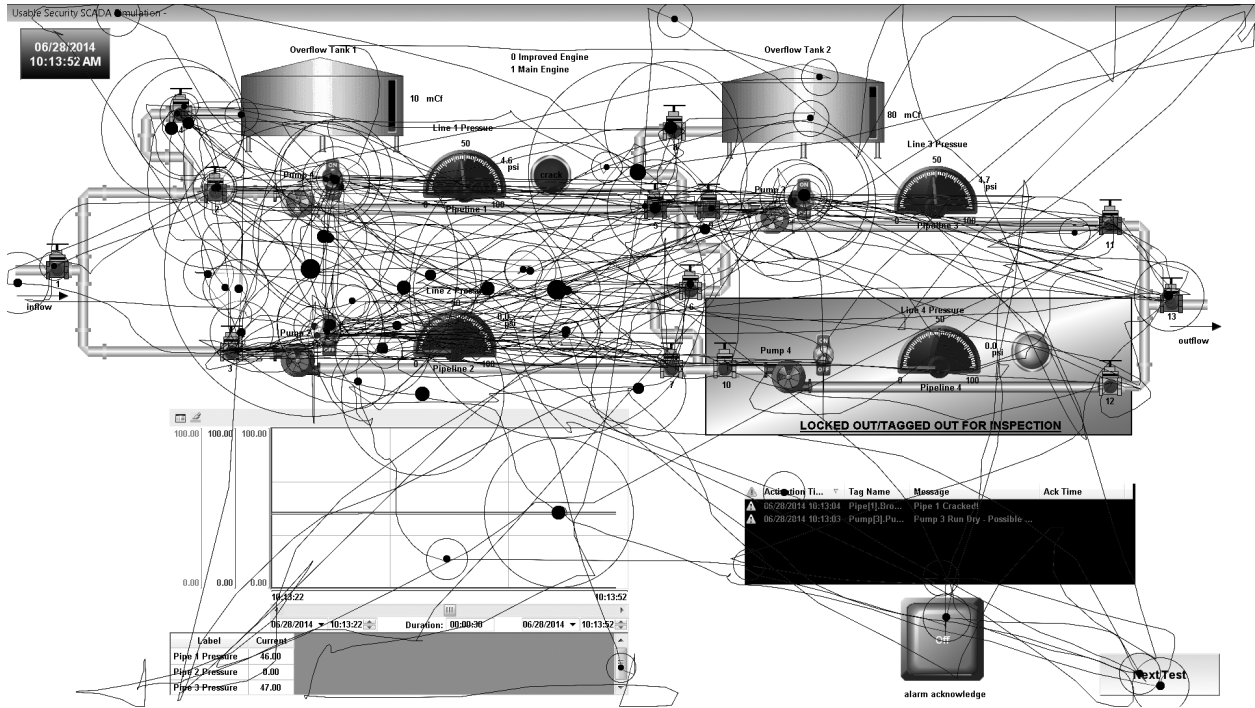


Figure 6.7: IOgraph Mouse Tracking Overlay 2

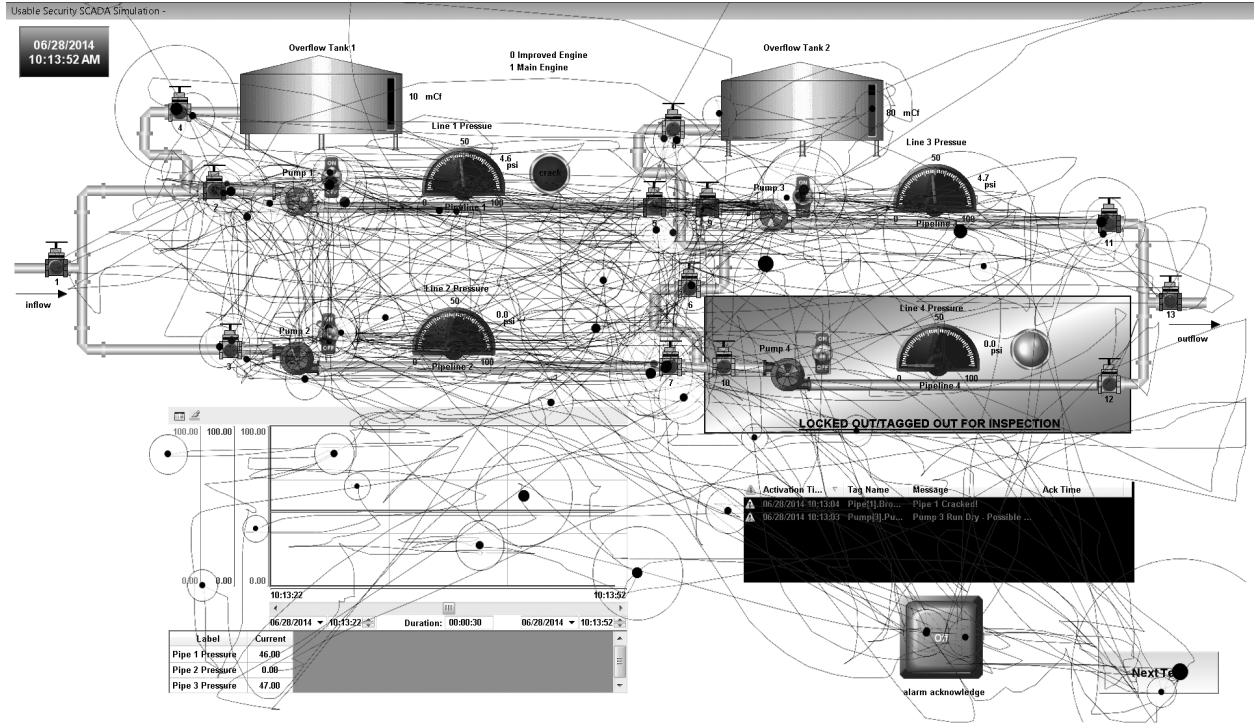


Figure 6.8: IOgraph Mouse Tracking Overlay 3

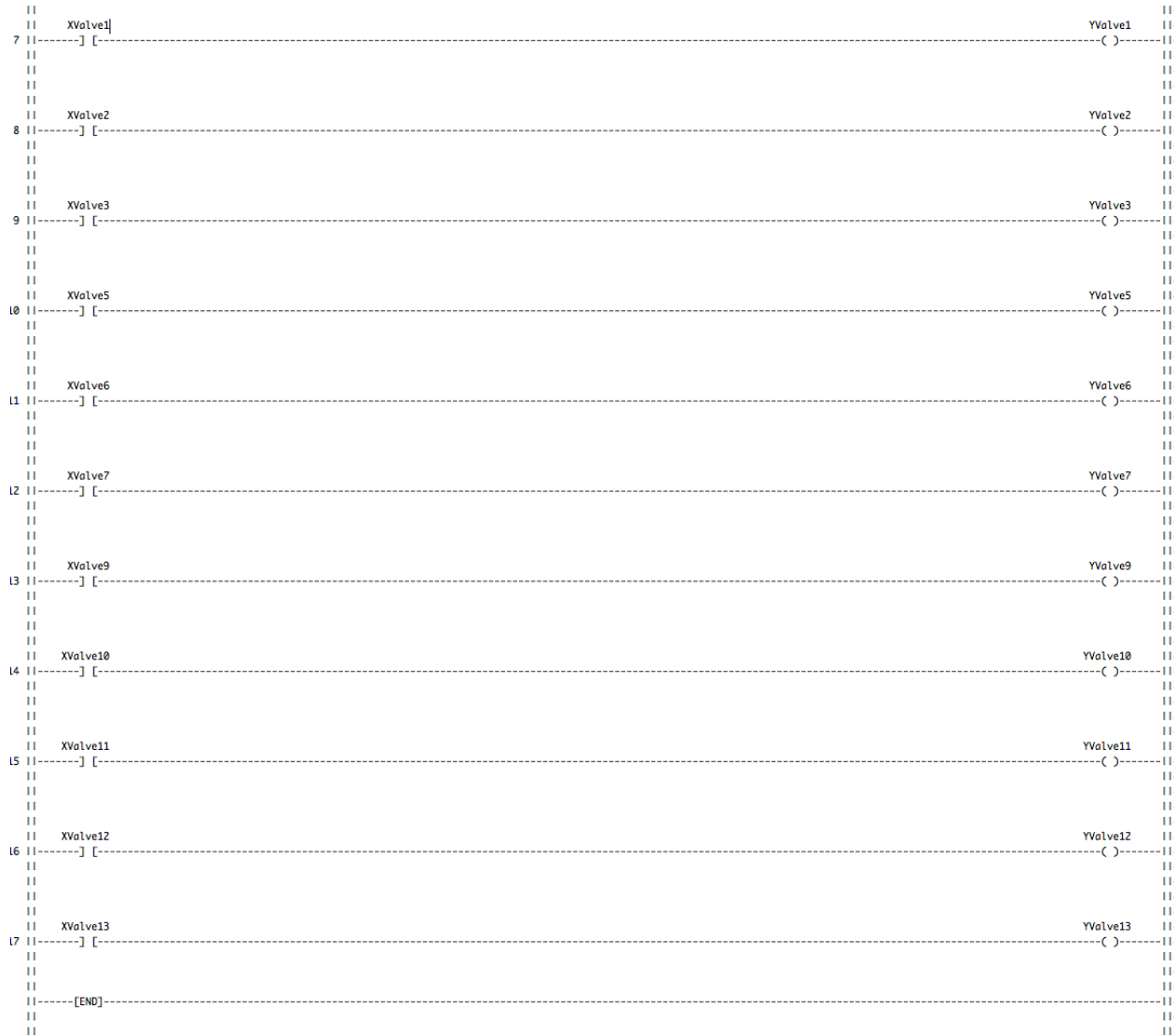


Figure 6.10: Improved Ladder Logic for Pipeline Model Part 2

Listing 6.2: Permutations.py

```
import itertools

products = ['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7', 'v8', 'v9',
            'v10', 'v11', 'v12', 'v13', 'p1', 'p2', 'p3', 'p4']

permutations = 0

for L in range(1, len(products)+1):
```

```
for subset in itertools.combinations(products, L):
    print(subset)
    permutations = permutations+1

print permutations
```


6.7.1 VisualBasic Engine for InduSoft Web Studio 1

Listing 6.3: VisualBasic Engine for InduSoft Web Studio 1

```
'Variables available only for this group can be declared here.
    'Loop variables
Dim i
Dim j
Dim k
Dim l
Dim m
Dim n
Dim o
Dim p

'-----
'-----Set Initial States-----
'-----

'Count up for pump cycle states
For n = 1 To 4
    If $Pump[n].PumpCmd = True And $Pump[n].PumpState = False
        Then
            $Pump[n].PumpCycles = $Pump[n].PumpCycles +1
        End If
    Next
```

```
'The code configured here is executed while the condition  
    configured in the Execution field is TRUE.
```

```
'Set states for valves
```

```
For i = 1 To 13
```

```
    $Valve[i].State = $Valve[i].Command
```

```
Next
```

```
'Set states for the pumps
```

```
For j = 1 To 4
```

```
    $Pump[j].PumpState = $Pump[j].PumpCmd
```

```
Next
```

```
'check for pump burnout
```

```
'Set Pump Burnout
```

```
For m = 1 To 4
```

```
    If $Pump[m].PumpCycles > 35 Then
```

```
        If $Pump[m].PumpState = True Then
```

```
            $Pump[m].PumpCmd = False
```

```
        End If
```

```
    End If
```

```
Next
```

```
'check for pump burnout
```

```
'Set Pump Burnout
```

```
For m = 1 To 4
```

```
    If $Pump[m].PumpCycles > 35 Then
```

```
        If $Pump[m].PumpState = True Then
```

```

                $Pump[m].PumpCmd = False
            End If
        End If
    Next

'-----Safety Features-----

'pipe overpressure
For k = 1 To 4
    If $Pipe[k].Sink = False Then
        $Pump[k].PumpCmd = False
        $Pump[k].PumpState = False
    End If
Next

'tank 1 overflow
If $Tank[1].Level > 100 Then
    $Valve[4].Command = False
    $Valve[4].State = False
End If

'tank 2 overflow
If $Tank[2].Level > 100 Then
    $Valve[8].Command = False
    $Valve[8].State = False
End If

```

```

'pipe 3 overpressure
If $Pipe[3].Pressure > 70 Then
    $Pump[3].PumpCmd = False
    $Pump[3].PumpState = False
    $Pump[1].PumpCmd = False
    $Pump[1].PumpState = False
    $Pump[2].PumpCmd = False
    $Pump[2].PumpState = False
End If

'-----
'Set Sources and Sinks for Tanks and Pipes
'-----

'Test for whether sources and sinks are true

'Tank 1 Source
If $Valve[1].State = True And $Valve[4].State = True Then
    $Tank[1].Source = True
Else
    $Tank[1].Source = False
End If

'Tank 1 Sink
'To Pipe 1
If $Valve[4].State = True And $Valve[2].State = True Then

```

```

        $Tank[1].Sink = True
    'To Pipe 2
ElseIf $Valve[4].State = True And $Valve[3].State = True Then
        $Tank[1].Sink = True
Else
        $Tank[1].Sink = False
End If

'Tank 2 Source
'Pipe 1
If $Valve[8].State = True And $Valve[5].State = True And $Valve
    [2].State = True And $Valve[1].State = True Then
        $Tank[2].Source = True
'Pipe 2
ElseIf $Valve[8].State = True And $Valve[6].State = True And
    $Valve[7].State = True And $Valve[3].State = True And $Valve
    [1].State = True Then
        $Tank[2].Source = True
Else
        $Tank[2].Source = False
End If

'Tank 2 Sink
'To Pipe 3
If $Valve[8].State = True And $Valve[9].State = True And $Valve
    [11].State = True And $Valve[13].State = True Then
        $Tank[2].Sink = True

```

```

'To Pipe 4
ElseIf $Valve[8].State = True And $Valve[6].State = True And
    $Valve[10].State = True And $Valve[12].State = True And $Valve
    [13].State = True Then
    $Tank[2].Sink = True
'To Tank 1 via Pipe 2
ElseIf $Valve[8].State = True And $Valve[6].State = True And
    $Valve[7].State = True And $Valve[3].State = True And $Valve
    [4].State = True Then
    $Tank[2].Sink = True
'To Tank 1 via Pipe 1
ElseIf $Valve[8].State = True And $Valve[5].State = True And
    $Valve[2].State = True And $Valve[4].State = True Then
    $Tank[2].Sink = True
Else
    $Tank[2].Sink = False
End If

'Pipe 1 Source
'From input
If $Valve[1].State = True And $Valve[2].State = True Then
    $Pipe[1].Source = True
'From Tank 1
ElseIf $Valve[4].State = True And $Valve[2].State = True And $Tank
    [1].Level > 0 Then
    $Pipe[1].Source = True
Else

```

```

        $Pipe[1].Source = False
End If

'Pipe 1 Sink
'To Pipe 3
If $Valve[9].State = True And $Valve[5].State = True Then
    $Pipe[1].Sink = True
'To Pipe 4
ElseIf $Pipe[4].Source = True And $Valve[5].State = True And
    $Valve[6].State = True Then
    $Pipe[1].Sink = True
'To Tank 2
ElseIf $Valve[5].State = True And $Tank[2].Source = True Then
    $Pipe[1].Sink = True
Else
    $Pipe[1].Sink = False
End If

'Pipe 2 Source
'From input
If $Valve[1].State = True And $Valve[3].State = True Then
    $Pipe[2].Source = True
'From Tank 1
ElseIf $Valve[4].State = True And $Valve[1].State = True And
    $Valve[3].State = True And $Tank[1].level > 0 Then
    $Pipe[2].Source = True

```

```

Else
    $Pipe[2].Source = False
End If

'Pipe 2 Sink
    'To Pipe 3
If $Valve[7].State = True And $Valve[6].State = True And $Valve
[9].State = True Then
    $Pipe[2].Sink = True
    'To Pipe 4
ElseIf $Valve[7].State = True And $Pipe[4].Source = True Then
    $Pipe[2].Sink = True
    'Else False
Else
    $Pipe[2].Sink = False
End If

'Pipe 3 Source
'From Pipe 1
If $Valve[9].State = True And $Pipe[1].Sink = True Then
    $Pipe[3].Source = True
'From Pipe 2
ElseIf $Valve[9].State = True And $Pipe[2].Sink = True And
    $Valve[6].State = True Then
    $Pipe[3].Source= True
'From Tank 2

```



```

ElseIf $Valve[9].State = True And $Tank[2].Sink = True And $Tank
    [2].Level > 0 Then
    $Pipe[3].Source = True
Else
    $Pipe[3].Source = False
End If

```

```

'Pipe 3 Sink
If $Valve[11].State = True And $Valve[13].State = True Then
    $Pipe[3].Sink = True
Else
    $Pipe[3].Sink = False
End If

```

```

'Pipe 4 Source
'From Tank 2
If $Tank[2].Sink = True And $Valve[6].State = True And $Valve
    [10].State = True Then
    $Pipe[4].Source = True
'From Pipe 1
ElseIf $Pipe[1].Sink = True And $Valve[6].State = True And
    $Valve[10].State = True Then
    $Pipe[4].Source = True
'From Pipe 2
ElseIf $Pipe[2].Sink = True And $Valve[10].State = True Then
    $Pipe[4].Source = True
Else

```

```

        $Pipe [4]. Source = False
End If

'Pipe 4 Sink
If $Valve [12]. State = True And $Valve [13]. State = True Then
    $Pipe [4]. Sink = True
Else
    $Pipe [4]. Sink = False
End If

'-----
'Set Pressures and Levels
'-----

'Tank 1
If $Tank [1]. Source = True And $Tank [1]. Sink = False And $Tank [1].
    Level < 106 Then
    $Tank [1]. Level = $Tank [1]. Level + .5
ElseIf $Tank [1]. Sink = True And $Tank [1]. Source = False And $Tank
    [1]. Level > 0 Then
    $Tank [1]. Level = $Tank [1]. Level - .5
ElseIf $Tank [1]. Sink = True And $Tank [1]. Source = True And $Tank
    [1]. Level > 0 Then
    $Tank [1]. Level = $Tank [1]. Level - .1
End If

'Tank 2

```

```

If $Tank[2].Source = True And $Tank[2].Sink = False And $Tank[2].
  Level < 106 Then
    $Tank[2].Level = $Tank[2].Level + .5
ElseIf $Tank[2].Sink = True And $Tank[2].Source = False And $Tank
  [2].Level > 0 Then
    $Tank[2].Level = $Tank[2].Level -.5
ElseIf $Tank[2].Sink = True And $Tank[2].Source = True And $Tank
  [2].Level > 0 Then
    $Tank[2].Level = $Tank[2].Level -.1
End If

```

```

'Pipe 1

```

```

If $Pipe[1].Sink = True And $Pipe[1].Source = True Then
  If $Pump[1].PumpState = True And $Pipe[1].Pressure < 47
    Then
      $Pipe[1].Pressure = $Pipe[1].Pressure + 1
    ElseIf $Pump[1].PumpState = False And $Pipe[1].Pressure <
      10 Then
        $Pipe[1].Pressure = $Pipe[1].Pressure + 1
      ElseIf $Pipe[1].Pressure > 10 And $Pump[1].PumpState =
        False Then
          $Pipe[1].Pressure = $Pipe[1].Pressure - 1
        ElseIf $Pipe[1].Pressure > 47 And $Pump[1].PumpState =
          True Then
            $Pipe[1].Pressure = $Pipe[1].Pressure - 1
          End If
    ElseIf $Pipe[1].Sink = False And $Pipe[1].Source = True Then

```

```

If $Pump[1].PumpState = True And $Pipe[1].Pressure < 65
    Then
        $Pipe[1].Pressure = $Pipe[1].Pressure + 1
    ElseIf $Pump[1].PumpState = False And $Pipe[1].Pressure <
        10 Then
            $Pipe[1].Pressure = $Pipe[1].Pressure + 1
    ElseIf $Pipe[1].Pressure > 10 And $Pump[1].PumpState =
        False Then
            $Pipe[1].Pressure = $Pipe[1].Pressure - 1
    End If
Else
    If $Pipe[1].Pressure > 0 Then
        $Pipe[1].Pressure = $Pipe[1].Pressure - 1
    End If
End If

'Pipe 2
If $Pipe[2].Sink = True And $Pipe[2].Source = True Then
    If $Pump[2].PumpState = True And $Pipe[2].Pressure < 47
        Then
            $Pipe[2].Pressure = $Pipe[2].Pressure + 1
        ElseIf $Pump[2].PumpState = False And $Pipe[2].Pressure <
            10 Then
                $Pipe[2].Pressure = $Pipe[2].Pressure + 1
        ElseIf $Pipe[2].Pressure > 10 And $Pump[2].PumpState =
            False Then
                $Pipe[2].Pressure = $Pipe[2].Pressure - 1

```

```

ElseIf $Pipe[2].Pressure > 47 And $Pump[2].PumpState =
    True Then
        $Pipe[2].Pressure = $Pipe[2].Pressure - 1
    End If
ElseIf $Pipe[2].Sink = False And $Pipe[2].Source = True Then
    If $Pump[2].PumpState = True And $Pipe[2].Pressure < 65
        Then
            $Pipe[2].Pressure = $Pipe[2].Pressure + 1
        ElseIf $Pump[2].PumpState = False And $Pipe[2].Pressure <
            10 Then
                $Pipe[2].Pressure = $Pipe[2].Pressure + 1
            ElseIf $Pipe[2].Pressure > 10 And $Pump[2].PumpState =
                False Then
                    $Pipe[2].Pressure = $Pipe[2].Pressure - 1
            End If
        End If
    Else
        If $Pipe[2].Pressure > 0 Then
            $Pipe[2].Pressure = $Pipe[2].Pressure - 1
        End If
    End If

```

'PIPE 3

'correct for sources being closed.

```

If $Pipe[3].Sink=True And $Pipe[3].Source = True Then

```

```

    'Pipe 1

```

```

If $Pipe[1].Sink = True And $Pipe[1].Source = True And
    $Pipe[2].Source = False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure <10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure >10 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
        $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[1].Sink = True And $Pipe[1].Source = True And
    $Pipe[2].Sink= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1

```

```

ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
    $Pump[3].PumpState = True Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
End If
ElseIf $Pipe[1].Sink = True And $Pipe[1].Source = True And
    $Pipe[2].Sink= False And $Pipe[2].Source= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
        $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False
    Then
    If $Pipe[3].Pressure > 0 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure -1
    End If
End If
'Pipe 2

```

```

If $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Source = False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[2].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10.3 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[2].Pressure And
        $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Sink = False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[2].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > $Pipe[2].Pressure And
        $Pump[3].PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1

```



```

ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
    PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Sink= False And $Pipe[1].Source= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure Then
            $Pipe[3].Pressure = $Pipe[3].Pressure + 1
        ElseIf $Pump[3].PumpState = False And $Pipe[3].
            Pressure < 10 Then
                $Pipe[3].Pressure = $Pipe[3].Pressure + 1
            ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
                PumpState = False Then
                    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
            ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
                $Pump[3].PumpState = True Then
                    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
            End If
        ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False
            Then
                If $Pipe[3].Pressure > 0 Then
                    $Pipe[3].Pressure = $Pipe[3].Pressure -1
                End If
            End If
        End If
'PIPE 1 AND 2

```

```

If $Pipe[1].Source = True And $Pipe[1].Sink = True And
    $Pipe[2].Source = True And $Pipe[2].Sink = True Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure + $Pipe[2].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < $Pipe[1].Pressure + $Pipe[2].
        Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure +
        $Pipe[2].Pressure And $Pump[3].PumpState =
        False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure +
        $Pipe[2].Pressure And $Pump[3].PumpState = True
        Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
End If

ElseIf $Pipe[3].Sink = False And $Pipe[3].Source = True Then
    'Pipe 1
    If $Pipe[1].Sink = True And $Pipe[1].Source = True And
        $Pipe[2].Source = False Then
        If $Pump[3].PumpState = True And $Pipe[3].Pressure
            < 65 Then
            $Pipe[3].Pressure = $Pipe[3].Pressure + 1

```

```

ElseIf $Pump[3].PumpState = False And $Pipe[3].
    Pressure <10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
ElseIf $Pipe[3].Pressure >10 And $Pump[3].
    PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
    $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
End If
ElseIf $Pipe[1].Sink = True And $Pipe[1].Source = True And
    $Pipe[2].Sink= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < 65 Then
            $Pipe[3].Pressure = $Pipe[3].Pressure + 1
ElseIf $Pump[3].PumpState = False And $Pipe[3].
    Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
    PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
    $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
End If
ElseIf $Pipe[1].Sink = True And $Pipe[1].Source = True And
    $Pipe[2].Sink= False And $Pipe[2].Source= False Then

```

```

If $Pump[3].PumpState = True And $Pipe[3].Pressure
    < $Pipe[1].Pressure Then
    $Pipe[3].Pressure = $Pipe[3].Pressure + 1
ElseIf $Pump[3].PumpState = False And $Pipe[3].
    Pressure < 10 Then
    $Pipe[3].Pressure = $Pipe[3].Pressure + 1
ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
    PumpState = False Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
    $Pump[3].PumpState = True Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
End If
ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False
    Then
    If $Pipe[3].Pressure > 0 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure -1
    End If
End If
'Pipe 2
If $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Source = False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < 65 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then

```

```

        $Pipe [3]. Pressure = $Pipe [3]. Pressure + 1
    ElseIf $Pipe [3]. Pressure > 10.3 And $Pump [3].
        PumpState = False Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
    ElseIf $Pipe [3]. Pressure > $Pipe [2]. Pressure And
        $Pump [3]. PumpState = True Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
    End If
ElseIf $Pipe [2]. Sink = True And $Pipe [2]. Source = True And
    $Pipe [1]. Sink= False Then
    If $Pump [3]. PumpState = True And $Pipe [3]. Pressure
        < 65 Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure + 1
    ElseIf $Pump [3]. PumpState = False And $Pipe [3].
        Pressure < 10 Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure + 1
    ElseIf $Pipe [3]. Pressure > $Pipe [2]. Pressure And
        $Pump [3]. PumpState = False Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
    ElseIf $Pipe [3]. Pressure > 10 And $Pump [3].
        PumpState = True Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
    End If
ElseIf $Pipe [2]. Sink = True And $Pipe [2]. Source = True And
    $Pipe [1]. Sink= False And $Pipe [1]. Source= False Then
    If $Pump [3]. PumpState = True And $Pipe [3]. Pressure
        < $Pipe [1]. Pressure Then

```

```

        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
        $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False
    Then
    If $Pipe[3].Pressure > 0 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure -1
    End If
    End If
'PIPE 1 AND 2
If $Pipe[1].Source = True And $Pipe[1].Sink = True And
    $Pipe[2].Source = True And $Pipe[2].Sink = True Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < 75 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < $Pipe[1].Pressure + $Pipe[2].
        Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1

```

```

ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure +
    $Pipe[2].Pressure And $Pump[3].PumpState =
    False Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure +
    $Pipe[2].Pressure And $Pump[3].PumpState = True
    Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
End If

End If

'ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False Then
'    If $Pipe[3].Pressure > 0 Then
'
'        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
'
'    End If
Else
    If $Pipe[3].Pressure > 0 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
End If

'Pipe 4
'Not used - could be coded for future use - copy pipe 3

'Broken Pipe Overpressure Pipe 1
If $Pipe[1].Pressure >= 64 Then
    $Pipe[1].Broken = True
End If

```

```
If $Pipe[1].Broken = True Then
    If $Pipe[1].Pressure > 0 Then
        $Pipe[1].Pressure = $Pipe[1].Pressure - 1
    End If
End If
```

```
'Broken Pipe Overpressure Pipe 2
If $Pipe[2].Pressure >= 80 Then
    $Pipe[2].Broken = True
End If
```

```
If $Pipe[2].Broken = True Then
    If $Pipe[2].Pressure > 0 Then
        $Pipe[2].Pressure = $Pipe[2].Pressure - 1
    End If
End If
```

```
'Broken Pipe Overpressure Pipe 3
If $Pipe[3].Pressure >= 80 Then
    $Pipe[3].Broken = True
End If
```

```
If $Pipe[3].Broken = True Then
    If $Pipe[3].Pressure > 0 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
End If
```



```

'Set Tank 1 Rupture
If $Tank[1].Level > 106 Then
    $Tank[1].Rupture = True
End If

If $Tank[1].Rupture = True Then
    If $Tank[1].Level > 0 Then
        $Tank[1].Level = $Tank[1].Level - 1
    End If
End If

'Set Tank 2 Rupture
If $Tank[2].Level > 106 Then
    $Tank[2].Rupture = True
End If

If $Tank[2].Rupture = True Then
    If $Tank[2].Level > 0 Then
        $Tank[2].Level = $Tank[2].Level - 1
    End If
End If

\end{verbatim}

```

```

\subsection{Visual Basic Engine for Indusoft Web Studio 2}
\begin{lstlisting}[breaklines=true]

```

```

'Variables available only for this group can be declared here.
    'Loop variables
Dim i
Dim j
Dim k
Dim l
Dim m
Dim n

'-----
'-----Set Initial States-----
'-----

'Count up for pump cycle states
For n = 1 To 4
    If $Pump[n].PumpCmd = True And $Pump[n].PumpState = False
        Then
            $Pump[n].PumpCycles = $Pump[n].PumpCycles +1
        End If
    Next
Next

'The code configured here is executed while the condition
    configured in the Execution field is TRUE.
'Set states for valves
For i = 1 To 13
    $Valve[i].State = $Valve[i].Command
Next

```

```

'Set states for the pumps
For j = 1 To 4
    $Pump[j].PumpState = $Pump[j].PumpCmd
Next

'check for pump burnout
'Set Pump Burnout
For m = 1 To 4
    If $Pump[m].PumpCycles > 35 Then
        If $Pump[m].PumpState = True Then
            $Pump[m].PumpCmd = False
        End If
    End If
Next

'check for pump burnout
'Set Pump Burnout
For m = 1 To 4
    If $Pump[m].PumpCycles > 35 Then
        If $Pump[m].PumpState = True Then
            $Pump[m].PumpCmd = False
        End If
    End If
Next

'


---


'Set Sources and Sinks for Tanks and Pipes

```

```
'Test for whether sources and sinks are true

'Tank 1 Source
If $Valve[1].State = True And $Valve[4].State = True Then
    $Tank[1].Source = True
Else
    $Tank[1].Source = False
End If

'Tank 1 Sink
'To Pipe 1
If $Valve[4].State = True And $Valve[2].State = True Then
    $Tank[1].Sink = True
'To Pipe 2
ElseIf $Valve[4].State = True And $Valve[3].State = True Then
    $Tank[1].Sink = True
Else
    $Tank[1].Sink = False
End If

'Tank 2 Source
'Pipe 1
If $Valve[8].State = True And $Valve[5].State = True And $Valve
    [2].State = True And $Valve[1].State = True Then
    $Tank[2].Source = True
```

```

'Pipe 2
ElseIf $Valve[8].State = True And $Valve[6].State = True And
    $Valve[7].State = True And $Valve[3].State = True And $Valve
    [1].State = True Then
    $Tank[2].Source = True
Else
    $Tank[2].Source = False
End If

'Tank 2 Sink
'To Pipe 3
If $Valve[8].State = True And $Valve[9].State = True And $Valve
    [11].State = True And $Valve[13].State = True Then
    $Tank[2].Sink = True
'To Pipe 4
ElseIf $Valve[8].State = True And $Valve[6].State = True And
    $Valve[10].State = True And $Valve[12].State = True And $Valve
    [13].State = True Then
    $Tank[2].Sink = True
'To Tank 1 via Pipe 2
ElseIf $Valve[8].State = True And $Valve[6].State = True And
    $Valve[7].State = True And $Valve[3].State = True And $Valve
    [4].State = True Then
    $Tank[2].Sink = True
'To Tank 1 via Pipe 1
ElseIf $Valve[8].State = True And $Valve[5].State = True And
    $Valve[2].State = True And $Valve[4].State = True Then

```

```

        $Tank[2].Sink = True
Else
        $Tank[2].Sink = False
End If

'Pipe 1 Source
'From input
If $Valve[1].State = True And $Valve[2].State = True Then
        $Pipe[1].Source = True
'From Tank 1
ElseIf $Valve[4].State = True And $Valve[2].State = True And $Tank
[1].Level > 0 Then
        $Pipe[1].Source = True
Else
        $Pipe[1].Source = False
End If

'Pipe 1 Sink
'To Pipe 3
If $Valve[9].State = True And $Valve[5].State = True Then
        $Pipe[1].Sink = True
'To Pipe 4
ElseIf $Pipe[4].Source = True And $Valve[5].State = True And
$Valve[6].State = True Then
        $Pipe[1].Sink = True
'To Tank 2
ElseIf $Valve[5].State = True And $Tank[2].Source = True Then

```

```
        $Pipe [1]. Sink = True
Else
        $Pipe [1]. Sink = False
End If
```

```
'Pipe 2 Source
```

```
'From input
```

```
If $Valve [1]. State = True And $Valve [3]. State = True Then
```

```
        $Pipe [2]. Source = True
```

```
'From Tank 1
```

```
ElseIf $Valve [4]. State = True And $Valve [1]. State = True And
```

```
        $Valve [3]. State = True And $Tank [1]. level > 0 Then
```

```
        $Pipe [2]. Source = True
```

```
Else
```

```
        $Pipe [2]. Source = False
```

```
End If
```

```
'Pipe 2 Sink
```

```
        'To Pipe 3
```

```
If $Valve [7]. State = True And $Valve [6]. State = True And $Valve  
[9]. State = True Then
```

```
        $Pipe [2]. Sink = True
```

```
        'To Pipe 4
```

```
ElseIf $Valve [7]. State = True And $Pipe [4]. Source = True Then
```

```
        $Pipe [2]. Sink = True
```

```
        'Else False
```

```

Else
    $Pipe [2]. Sink = False
End If

'Pipe 3 Source
'From Pipe 1
If $Valve [9]. State = True And $Pipe [1]. Sink = True Then
    $Pipe [3]. Source = True
'From Pipe 2
ElseIf $Valve [9]. State = True And $Pipe [2]. Sink = True And
    $Valve [6]. State = True Then
    $Pipe [3]. Source= True
'From Tank 2
ElseIf $Valve [9]. State = True And $Tank [2]. Sink = True And $Tank
    [2]. Level > 0 Then
    $Pipe [3]. Source = True
Else
    $Pipe [3]. Source = False
End If

'Pipe 3 Sink
If $Valve [11]. State = True And $Valve [13]. State = True Then
    $Pipe [3]. Sink = True
Else
    $Pipe [3]. Sink = False
End If

```



```

'Pipe 4 Source
'From Tank 2
If $Tank[2].Sink = True And $Valve[6].State = True And $Valve
    [10].State = True Then
    $Pipe[4].Source = True
'From Pipe 1
ElseIf $Pipe[1].Sink = True And $Valve[6].State = True And
    $Valve[10].State = True Then
    $Pipe[4].Source = True
'From Pipe 2
ElseIf $Pipe[2].Sink = True And $Valve[10].State = True Then
    $Pipe[4].Source = True
Else
    $Pipe[4].Source = False
End If

```

```

'Pipe 4 Sink
If $Valve[12].State = True And $Valve[13].State = True Then
    $Pipe[4].Sink = True
Else
    $Pipe[4].Sink = False
End If

```

```

'Set Pressures and Levels

```

```

'Tank 1

```

```

If $Tank[1].Source = True And $Tank[1].Sink = False And $Tank[1].
    Level < 106 Then
    $Tank[1].Level = $Tank[1].Level + .5
ElseIf $Tank[1].Sink = True And $Tank[1].Source = False And $Tank
    [1].Level > 0 Then
    $Tank[1].Level = $Tank[1].Level -.5
ElseIf $Tank[1].Sink = True And $Tank[1].Source = True And $Tank
    [1].Level > 0 Then
    $Tank[1].Level = $Tank[1].Level -.1
End If

```

'Tank 2

```

If $Tank[2].Source = True And $Tank[2].Sink = False And $Tank[2].
    Level < 106 Then
    $Tank[2].Level = $Tank[2].Level + .5
ElseIf $Tank[2].Sink = True And $Tank[2].Source = False And $Tank
    [2].Level > 0 Then
    $Tank[2].Level = $Tank[2].Level -.5
ElseIf $Tank[2].Sink = True And $Tank[2].Source = True And $Tank
    [2].Level > 0 Then
    $Tank[2].Level = $Tank[2].Level -.1
End If

```

'Pipe 1

```

If $Pipe[1].Sink = True And $Pipe[1].Source = True Then
    If $Pump[1].PumpState = True And $Pipe[1].Pressure < 47
        Then

```

```

        $Pipe [1]. Pressure = $Pipe [1]. Pressure + 1
    ElseIf $Pump [1]. PumpState = False And $Pipe [1]. Pressure <
        10 Then
        $Pipe [1]. Pressure = $Pipe [1]. Pressure + 1
    ElseIf $Pipe [1]. Pressure > 10 And $Pump [1]. PumpState =
        False Then
        $Pipe [1]. Pressure = $Pipe [1]. Pressure - 1
    ElseIf $Pipe [1]. Pressure > 47 And $Pump [1]. PumpState =
        True Then
        $Pipe [1]. Pressure = $Pipe [1]. Pressure - 1
    End If
ElseIf $Pipe [1]. Sink = False And $Pipe [1]. Source = True Then
    If $Pump [1]. PumpState = True And $Pipe [1]. Pressure < 65
        Then
        $Pipe [1]. Pressure = $Pipe [1]. Pressure + 1
    ElseIf $Pump [1]. PumpState = False And $Pipe [1]. Pressure <
        10 Then
        $Pipe [1]. Pressure = $Pipe [1]. Pressure + 1
    ElseIf $Pipe [1]. Pressure > 10 And $Pump [1]. PumpState =
        False Then
        $Pipe [1]. Pressure = $Pipe [1]. Pressure - 1
    End If
Else
    If $Pipe [1]. Pressure > 0 Then
        $Pipe [1]. Pressure = $Pipe [1]. Pressure - 1
    End If
End If

```

```

'Pipe 2
If $Pipe[2].Sink = True And $Pipe[2].Source = True Then
    If $Pump[2].PumpState = True And $Pipe[2].Pressure < 47
        Then
            $Pipe[2].Pressure = $Pipe[2].Pressure + 1
        ElseIf $Pump[2].PumpState = False And $Pipe[2].Pressure <
            10 Then
                $Pipe[2].Pressure = $Pipe[2].Pressure + 1
            ElseIf $Pipe[2].Pressure > 10 And $Pump[2].PumpState =
                False Then
                    $Pipe[2].Pressure = $Pipe[2].Pressure - 1
            ElseIf $Pipe[2].Pressure > 47 And $Pump[2].PumpState =
                True Then
                    $Pipe[2].Pressure = $Pipe[2].Pressure - 1
        End If
    ElseIf $Pipe[2].Sink = False And $Pipe[2].Source = True Then
        If $Pump[2].PumpState = True And $Pipe[2].Pressure < 65
            Then
                $Pipe[2].Pressure = $Pipe[2].Pressure + 1
            ElseIf $Pump[2].PumpState = False And $Pipe[2].Pressure <
                10 Then
                    $Pipe[2].Pressure = $Pipe[2].Pressure + 1
            ElseIf $Pipe[2].Pressure > 10 And $Pump[2].PumpState =
                False Then
                    $Pipe[2].Pressure = $Pipe[2].Pressure - 1
        End If
    End If

```

Else

 If \$Pipe[2].Pressure > 0 Then

 \$Pipe[2].Pressure = \$Pipe[2].Pressure - 1

 End If

End If

'PIPE 3

'correct for sources being closed.

If \$Pipe[3].Sink=True And \$Pipe[3].Source = True Then

 'Pipe 1

 If \$Pipe[1].Sink = True And \$Pipe[1].Source = True And

 \$Pipe[2].Source = False Then

 If \$Pump[3].PumpState = True And \$Pipe[3].Pressure
 < \$Pipe[1].Pressure Then

 \$Pipe[3].Pressure = \$Pipe[3].Pressure + 1

 ElseIf \$Pump[3].PumpState = False And \$Pipe[3].

 Pressure <10 Then

 \$Pipe[3].Pressure = \$Pipe[3].Pressure + 1

 ElseIf \$Pipe[3].Pressure >10 And \$Pump[3].

 PumpState = False Then

 \$Pipe[3].Pressure = \$Pipe[3].Pressure - 1

 ElseIf \$Pipe[3].Pressure > \$Pipe[1].Pressure And

 \$Pump[3].PumpState = True Then

 \$Pipe[3].Pressure = \$Pipe[3].Pressure - 1

 End If

```

ElseIf $Pipe[1].Sink = True And $Pipe[1].Source = True And
    $Pipe[2].Sink= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
        $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[1].Sink = True And $Pipe[1].Source = True And
    $Pipe[2].Sink= False And $Pipe[2].Source= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1

```

```

ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
    $Pump[3].PumpState = True Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
End If
ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False
Then
    If $Pipe[3].Pressure > 0 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
End If
'Pipe 2
If $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Source = False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[2].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10.3 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[2].Pressure And
        $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If

```

```

ElseIf $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Sink= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[2].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > $Pipe[2].Pressure And
        $Pump[3].PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
        PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Sink= False And $Pipe[1].Source= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1

```



```

ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
    $Pump[3].PumpState = True Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
End If
ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False
Then
    If $Pipe[3].Pressure > 0 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure -1
    End If
End If
'PIPE 1 AND 2
If $Pipe[1].Source = True And $Pipe[1].Sink = True And
    $Pipe[2].Source = True And $Pipe[2].Sink = True Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure + $Pipe[2].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < $Pipe[1].Pressure + $Pipe[2].
        Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure +
        $Pipe[2].Pressure And $Pump[3].PumpState =
        False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure +
        $Pipe[2].Pressure And $Pump[3].PumpState = True
        Then

```

```

                $Pipe[3].Pressure = $Pipe[3].Pressure - 1
            End If
        End If
    ElseIf $Pipe[3].Sink = False And $Pipe[3].Source = True Then
        'Pipe 1
        If $Pipe[1].Sink = True And $Pipe[1].Source = True And
            $Pipe[2].Source = False Then
            If $Pump[3].PumpState = True And $Pipe[3].Pressure
                < 65 Then
                $Pipe[3].Pressure = $Pipe[3].Pressure + 1
            ElseIf $Pump[3].PumpState = False And $Pipe[3].
                Pressure <10 Then
                $Pipe[3].Pressure = $Pipe[3].Pressure + 1
            ElseIf $Pipe[3].Pressure >10 And $Pump[3].
                PumpState = False Then
                $Pipe[3].Pressure = $Pipe[3].Pressure - 1
            ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
                $Pump[3].PumpState = True Then
                $Pipe[3].Pressure = $Pipe[3].Pressure - 1
            End If
        ElseIf $Pipe[1].Sink = True And $Pipe[1].Source = True And
            $Pipe[2].Sink= False Then
            If $Pump[3].PumpState = True And $Pipe[3].Pressure
                < 65 Then
                $Pipe[3].Pressure = $Pipe[3].Pressure + 1
            ElseIf $Pump[3].PumpState = False And $Pipe[3].
                Pressure < 10 Then

```

```

        $Pipe [3]. Pressure = $Pipe [3]. Pressure + 1
    ElseIf $Pipe [3]. Pressure > 10 And $Pump [3].
        PumpState = False Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
    ElseIf $Pipe [3]. Pressure > $Pipe [1]. Pressure And
        $Pump [3]. PumpState = True Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
    End If
ElseIf $Pipe [1]. Sink = True And $Pipe [1]. Source = True And
    $Pipe [2]. Sink = False And $Pipe [2]. Source = False Then
    If $Pump [3]. PumpState = True And $Pipe [3]. Pressure
        < $Pipe [1]. Pressure Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure + 1
    ElseIf $Pump [3]. PumpState = False And $Pipe [3].
        Pressure < 10 Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure + 1
    ElseIf $Pipe [3]. Pressure > 10 And $Pump [3].
        PumpState = False Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
    ElseIf $Pipe [3]. Pressure > $Pipe [1]. Pressure And
        $Pump [3]. PumpState = True Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
    End If
ElseIf $Pipe [1]. Source = False And $Pipe [2]. Source = False
    Then
        If $Pipe [3]. Pressure > 0 Then
            $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1

```

```

        End If
    End If
'Pipe 2
If $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Source = False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < 65 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10.3 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[2].Pressure And
        $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Sink = False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < 65 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1

```

```

ElseIf $Pipe[3].Pressure > $Pipe[2].Pressure And
    $Pump[3].PumpState = False Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
    PumpState = True Then
    $Pipe[3].Pressure = $Pipe[3].Pressure - 1
End If
ElseIf $Pipe[2].Sink = True And $Pipe[2].Source = True And
    $Pipe[1].Sink= False And $Pipe[1].Source= False Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < $Pipe[1].Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < 10 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > 10 And $Pump[3].
        PumpState = False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure And
        $Pump[3].PumpState = True Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False
    Then
    If $Pipe[3].Pressure > 0 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure -1
    End If

```

```

End If
'PIPE 1 AND 2
If $Pipe[1].Source = True And $Pipe[1].Sink = True And
    $Pipe[2].Source = True And $Pipe[2].Sink = True Then
    If $Pump[3].PumpState = True And $Pipe[3].Pressure
        < 75 Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pump[3].PumpState = False And $Pipe[3].
        Pressure < $Pipe[1].Pressure + $Pipe[2].
        Pressure Then
        $Pipe[3].Pressure = $Pipe[3].Pressure + 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure +
        $Pipe[2].Pressure And $Pump[3].PumpState =
        False Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    ElseIf $Pipe[3].Pressure > $Pipe[1].Pressure +
        $Pipe[2].Pressure And $Pump[3].PumpState = True
        Then
        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
    End If
End If
'ElseIf $Pipe[1].Source = False And $Pipe[2].Source = False Then
'    If $Pipe[3].Pressure > 0 Then
'        $Pipe[3].Pressure = $Pipe[3].Pressure - 1
'    End If
Else
    If $Pipe[3].Pressure > 0 Then

```

```

                                $Pipe [3]. Pressure = $Pipe [3]. Pressure - 1
        End If
End If
'Pipe 4
'Not used – could be coded for future use – copy pipe 3

'Broken Pipe Overpressure Pipe 1
If $Pipe [1]. Pressure >= 64 Then
    $Pipe [1]. Broken = True
End If

If $Pipe [1]. Broken = True Then
    If $Pipe [1]. Pressure > 0 Then
        $Pipe [1]. Pressure = $Pipe [1]. Pressure - 1
    End If
End If

'Broken Pipe Overpressure Pipe 2
If $Pipe [2]. Pressure >= 80 Then
    $Pipe [2]. Broken = True
End If

If $Pipe [2]. Broken = True Then
    If $Pipe [2]. Pressure > 0 Then
        $Pipe [2]. Pressure = $Pipe [2]. Pressure - 1
    End If
End If

```

'Broken Pipe Overpressure Pipe 3

If \$Pipe[3].Pressure >= 80 Then

 \$Pipe[3].Broken = True

End If

If \$Pipe[3].Broken = True Then

 If \$Pipe[3].Pressure > 0 Then

 \$Pipe[3].Pressure = \$Pipe[3].Pressure - 1

 End If

End If

'Set Tank 1 Rupture

If \$Tank[1].Level > 106 Then

 \$Tank[1].Rupture = True

End If

If \$Tank[1].Rupture = True Then

 If \$Tank[1].Level > 0 Then

 \$Tank[1].Level = \$Tank[1].Level - 1

 End If

End If

'Set Tank 2 Rupture

If \$Tank[2].Level > 106 Then

 \$Tank[2].Rupture = True

End If


```

If $Tank[2].Rupture = True Then
    If $Tank[2].Level > 0 Then
        $Tank[2].Level = $Tank[2].Level - 1
    End If
End If

If $Pump[1].PumpState = True And $Pipe[1].Source = False Then
    $Pump[1].PumpRunTime = 100
End If

If $Pump[2].PumpState = True And $Pipe[2].Source = False Then
    $Pump[2].PumpRunTime = 100
End If

If $Pump[3].PumpState = True And $Pipe[3].Source = False Then
    $Pump[3].PumpRunTime = 100
End If

```

6.7.2 Test Reconfiguration Visual Basic Script for InduSoft Web Studio

```

'Variables available for all Script groups from the Script task
    can be declared and initialized here.

Dim i

'Procedures available for all Script groups from the Script task
    can be implemented here.

$Main_Engine = True

```

`$Improved_Engine = False`

`$Valve [1].Command = True`

`$Valve [2].Command = True`

`$Valve [5].Command = True`

`$Valve [9].Command = True`

`$Valve [11].Command = True`

`$Valve [13].Command = True`

`$Valve [3].Command = False`

`$Valve [4].Command = False`

`$Valve [6].Command = False`

`$Valve [7].Command = False`

`$Valve [8].Command = False`

`$Valve [10].Command = False`

`$Valve [12].Command = False`

`$Tank [1].Level = 10`

`$Tank [2].Level = 80`

`$Pump [1].PumpCmd = True`

`$Pump [3].PumpCmd = True`

`$Pump [1].PumpState = True`

`$Pump [3].PumpState = True`

`$Pump [2].PumpCmd = False`

```
For i =1 To 4
    $Pump[ i ]. PumpCycles = 0
    $Pump[ i ]. PumpRunTime = 0
Next

$Experiment_Reset = True
```