The Pennsylvania State University
The Graduate School

SPECIFICATION-BASED ATTACKS AND DEFENSES IN SEQUENTIAL CONTROL SYSTEMS

A Dissertation in
Computer Science and Engineering
by
Stephen Elliot McLaughlin

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2014

The dissertation of Stephen Elliot McLaughlin was reviewed and approved* by the following:

Patrick D. McDaniel
Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Trent R. Jaeger
Professor of Computer Science and Engineering

Thomas F. LaPorta
Professor of Computer Science and Engineering

Seth Blumsack
Associate Professor of Earth and Mineral Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Cyber physical systems, such as control systems use general purpose computation to govern the behavior of physical systems in the manufacturing, transportation, and energy sectors. These systems are increasingly vulnerable to software-based exploits that have physical consequences. In modern control systems, Programmable Logic Controllers (PLCs) drive the physical machinery in a plant according to control logic programs. For ease of modification, control logic is uploaded to the PLC from the local network, the Internet, or other external network, making them vulnerable to malicious code injection. PLCs are unique within the control system, in that they form the last step of computation between the computer and the physical infrastructures. If they are compromised by an adversary, the entire physical system will be under malicious control. If they can be regulated by defenders, then adversarial influence over any other part of the system is nullified. In short, whoever owns the PLC owns the critical infrastructure.

In this work, we look at a novel approach to both attacking and securing automated cyber physical control systems: *specification-based attacks and defenses.* The vast body of existing work in computer security focuses on protecting information. In a control system, on the other hand, the most critical asset is not information, but the physical machinery, processes, and personnel, whose safety must be guaranteed. Due to the sheer complexity of modern information processing systems, it is impossible to completely secure the computer and network perimeters of automated control systems. Thus, instead of trying to harden perimeter security, we instead aim at directly regulating the physical behavior of the control system through behavioral specifications.

This dissertation covers three different systems that demonstrate specification-based attacks and defenses. The attack, called SABOT, allows adversaries to attack control systems knowing only their physical behavior. SABOT takes the adversary's *idea* of how the victim physical system works, and automatically instantiates it into runnable PLC code that executes the desired attack measures. The second system is a Trusted Safety Verifier (TSV). TSV prevents any malicious code from being uploaded to PLCs. This is done by performing a novel analysis technique on the PLC code to determine if it violates any engineer-supplied safety properties. Finally, we consider a Controller Controller (C^2). C^2 monitors the commands sent from PLCs to physical machinery to block any malicious device usage. C^2 is the most minimal of all these security mechanisms, as even the PLC can be fully compromised, and the security guarantees will still hold. These works are built on early experiences in directed penetration testing of smart electric meters, which found vulnerabilities based on the adversarial goals they achieved. Throughout, the case will be made that specification-based control of physical system behavior is a promising approach for securing control systems in modern critical infrastructure.

Table of Contents

List of Figures vii

List of Tables ix

Acknowledgments x

Chapter 1
Introduction 1

1.1 Using Attack Trees to Reason about Control System Attacks . . . 3
1.2 Automating Attacks against Control Systems. . . . 4
1.2.1 Indiscriminate Attacks against Control Systems . . . 5
1.2.2 Specification-based Attacks against Control Systems . . . 7
1.3 Specification-based Defenses . . . 8
1.4 Related Work . . . 9
1.4.1 Attacks against Control Systems . . . 9
1.4.2 Formal Verification of Control Systems . . . 11
1.5 Thesis Statement . . . 13

Chapter 2
Preliminary Concepts 15

2.1 The Advanced Metering Infrastructure . . . 15
2.1.1 Smart Meter Architectures . . . 16
2.1.2 Meter Networks and Utility-Side Management . . . 17
2.1.3 AMI Security Concerns . . . 17
2.2 Control Systems . . . 18
2.2.1 Programmable Logic Controllers . . . 19
2.2.2 Control System Implementations . . . 20
2.2.3 PLC Instruction Set Architectures . . . 21
2.3 Model Checking . . . 23
2.3.1 Computational Tree Logic . . . 23
2.3.2 The NuSMV 2 Model Checker . . . 25
2.3.3 Symbolic Execution . . . 27

Chapter 3
Multi-Vendor Penetration Testing **28**
3.1 Methodology . . . 28
3.2 Archetypal Attack Trees . . . 30
3.2.1 Energy Fraud . . . 31
3.2.2 Denial of Service . . . 33
3.2.3 Targeted Disconnect of Electrical Service . . . 34
3.3 Systems Under Test . . . 34
3.3.1 S1 Specifics . . . 35
3.3.2 S2 Specifics . . . 36
3.4 Concrete Attack Trees . . . 36
3.4.1 Energy Fraud in S1 . . . 37
3.4.2 Denial of Service in S2 . . . 38
3.4.3 Targeted Disconnect in S1 . . . 38
3.5 Results . . . 39
3.5.1 Energy Fraud by Forged Usage Data . . . 39
3.5.2 Denial of Service Against Command Execution . . . 40
3.5.3 Targeted Disconnect . . . 41

Chapter 4
Automating Attacks against Control Systems **42**
4.1 Dynamic Payloads . . . 42
4.1.1 Payload Goals . . . 43
4.1.2 Process Analysis . . . 43
4.1.3 Payload Generation . . . 44

Chapter 5
Specification-based Attacks on Sequential Control Systems **47**
5.1 SABOT . . . 47
5.1.1 Problem Formulation . . . 48
5.2 Decompilation . . . 50
5.3 VTDM Recovery . . . 51
5.3.1 Mapping Sketches to Models . . . 52
5.3.2 Unambiguous Sketches . . . 53
5.4 Payload Construction . . . 54
5.5 Evaluation . . . 56
5.5.1 Accuracy . . . 59
5.5.2 Differentiation . . . 61
5.5.3 Adaptability . . . 61
5.5.4 Performance . . . 61

Chapter 6
Static Specification Checking **63**
6.1 Threat Model . . . 64
6.2 System Overview . . . 64
6.3 The Instruction List Intermediate Language . . . 66
6.3.1 ILIL Symbolic Execution . . . 70
6.3.2 ILIL Type System . . . 71

6.3.3 ILIL Semantics 74
6.3.4 ILIL Instruction Lifting 75
6.4 Model Checking 77
6.5 Evaluation 77
6.5.1 Implementation 77
6.5.2 Control System Case Studies 78
6.5.3 Example Safety Violation 80
6.5.4 Performance 81
6.5.5 Scalability 82
6.6 Related Work 83

Chapter 7
Dynamic Specification-based Behavioral Enforcement **85**
7.1 Background and Threat Model 86
7.1.1 Control Systems 86
7.1.2 Threat Model 87
7.2 C^2 Policy 89
7.2.1 Device Policies 90
7.2.2 Combining Devices 91
7.3 C^2 Policy Enforcement 94
7.3.1 Deny Disciplines 94
7.3.2 PLC Code Instrumentation 95
7.4 Evaluation 96
7.4.1 Testbed 97
7.4.1.1 Test Systems 97
7.4.1.2 Test Policies 97
7.4.1.3 Implementation 99
7.4.2 Safety 99
7.4.3 Security 100
7.4.4 Performance 101
7.4.4.1 C^2 Performance 101
7.4.4.2 Instrumented PLC Code Performance 101

Chapter 8
Conclusions **102**
8.1 More Robust Specification-based Attacks 104
8.2 Hybrid Static and Dynamic Enforcement 104
8.3 Automating the Specification Writing Process 106
8.3.1 Response Policies 108

Bibliography **109**

List of Figures

1.1 Concrete Attack Tree . . . 4
1.2 Archetypal Attack Tree . . . 12

2.1 Connectivity of meters to utilities given two configurations of meter LANs. . . . 17
2.2 Example Sequential Control System . . . 20
2.3 Syntax of Computational Tree Logic. . . . 24

3.1 Example energy fraud attack tree. The three subgoals beneath the root are labeled as (*a*), (*b*), and (*c*) for reference purposes. . . . 29
3.2 Grafting concrete trees for two different systems ($S1$ and $S2$) onto an archetypal attack tree for a specific adversarial goal. . . . 30
3.3 Archetypal tree for Denial of Service. . . . 31
3.4 Archetypal tree for targeted disconnect. . . . 32
3.5 The two SUTs used in our experiments. In $S1$ (A), the collector also functions as a meter, and relays data from a wireless mesh LAN to a telephone network backhaul. $S2$ (B) uses a dedicated device as a collector to relay data between a PLC network and an Internet connection to the utility. . . . 33
3.6 The concrete trees for energy fraud in $S1$. . . . 35
3.7 The concrete trees for DOS in $S2$. . . . 35
3.8 The concrete trees for targeted disconnect $S1$. . . . 37

4.1 The basic steps for constructing a dynamic payload based on observations taken from within a process control system. . . . 43
4.2 Dependency graph for traffic light control. . . . 46

5.1 SABOT: Generating a malicious payload for a target control system based on process sketch, the PLC code, and generic malcode. . . . 48
5.2 An unambiguous sketch with conflict tables for the pH neutralization system. . . . 55

6.1 TSV Architecture. . . . 65
6.2 Simplified ILIL Grammar. . . . 67
6.3 ILIL Code example (IL in Comments). . . . 68
6.4 Basic ILIL type rules. . . . 71
6.5 Operational semantics of ILIL instructions. . . . 73
6.6 Operational semantics of ILIL expressions. . . . 74
6.7 Simplified IL grammar for lifting. . . . 75
6.8 Lifting rules for IL terms to ILIL terms. . . . 76
6.9 Performance Analysis of the Traffic Light Control System. . . . 78

6.10 Generated Temporal Execution Graph (model checking bound = 4). 79
6.11 Time Requirements for All Case Studies on Raspberry Pi. 81
6.12 State Space Size for All Case Studies on Raspberry Pi. 82
6.13 Scalability Analysis for Various Predicates 83

7.1 Control System Model. Entities marked by $\times$ are considered untrusted in our threat model. 87
7.2 Example continuous device policy. 90
7.3 Figure 7.1 modified to include C^2. Untrusted entities are marked by $\times$. 94
7.4 `sawmill.sslang` (excerpt) 98

8.1 Execution of a combined static and dynamic enforcement monitor using shallow TEGs. 105

List of Tables

1.1 Comparison of TSV and C^2 8

2.1 Representative Boolean logic instructions from the Siemens statement list (STL) language. 22

3.1 Summary of concrete attacks and discovered vulnerabilities for each adversarial goal. 39

5.1 Example accumulation of a constraint. 50
5.2 Constructing $\mathcal{M}$ from constraints C. 51
5.3 Control system variants (omitted shaded). 56
5.4 Per-device accuracy results. Empty cell: Correct mapping, '`p`': false positive mapping, '`n`':false negative (no mapping), Shaded cell: experiment omitted (see description). 60
5.5 Failed traffic signal and neutralization tests from Table 5.4 repeated with safeguards in place. 60
5.6 Running time (s) / number of calls to the model checker for each system and case. 62

6.1 Comparison of analyzed features with related work. Related approaches are abbreviated: SAT=SAT Solving, Thm=Theorem Proving, Mod=Model Checking . 83

7.1 C^2 Policy Summary 88
7.2 Safety Test Results. Columns headed by "Not." combine the **Notify** deny discipline. 99
7.3 C^2 policy check times on an embedded Raspberry Pi computer. 101

Acknowledgments

It is said that the PhD is an exercise in depth. For me, nothing could be further from the truth. My experiences during my PhD have transformed nearly ever aspect in the whole breadth of my life. This transformation comes largely through the relationships with the diverse and brilliant group of people I have met during my time working in the Systems and Internet Infrastructure Security (SIIS) Lab and the computer science and engineering department at Penn State, where I spent my grad student career.

At the top of the SIIS lab are the directors, Patrick McDaniel, and Trent Jaeger. I had met both Patrick and Trent not too far apart, both while I was still an undergraduate student at Penn State. It was while taking my first class with Trent (I would go on to take three more!) that I decided computer security was one option I was interested in for graduate study. Indeed, without Trent's deft interweaving of security's deep history and its current frontiers, I wouldn't be the researcher I am today. At this point, my fate was half sealed. The rest of the job was done at a talk given by Patrick at a student ACM meeting. Not only did Patrick describe some of the most exciting systems research I had been exposed to at that time, he also opened a door, as he would many times to come, by inviting undergraduate students to meet with him to discuss the prospects of graduate school. I followed up on his invitation, and as they say, the rest is history. I began working with Patrick, and shortly thereafter, began my PhD with him at Penn State. Having Patrick as an advisor not only had a marked effect on my approach to research, but also on my understanding of life and self respect. For this I am sincerely grateful.

I can still vividly remember the crew of SIIS lab 1.0, who had already bootstrapped the whole operation when I had arrived. Patrick Traynor was the lead grad student of the lab, and he lead by his example in running a tight ship. Will Enck always knew the best tricks for solving systems problems, and he often didn't wait for you to ask for help before offering it up. I remember when I inherited Boniface Hicks' desk, he asked Will, "What will I do without you to keep me honest?" I learned what he meant shortly thereafter! Boniface was himself a skilled and dedicated researcher who had dedicated his life to an even higher calling. If there is one thing I learned from him, it is that we must believe in something bigger and more important than our own lives if we are to be truly happy. Kevin Butler, my intellectual mentor, often surprised me with his ability to reason about the often blurred lines between the subjective and objective aspects of research. Nobody has had a bigger impact on my efforts towards becoming more well rounded.

Along the way, I met many other lab members who joined both before and after myself. I remember Josh Schiffman's uncanny skill for wordplay and puns, as well as the late nights with he and Divya Muthukumaran working on implementing all forms of semantics, big, small, and every machine in between. Divya was as cool a labmate as one is ever likely to encounter. I always knew the fun things happening in town thanks to Divya. Mike Lin has a genuine appreciation for many unrelated areas of computer science. I am always amazed at his awareness of big theoretical problems as well as the intricacies of lazy evaluation in Python. Richard Burhans was a lifesaver in Data Structures and Algorithms, where we frequently collaborated on homework, often at the end of the day in the few spare hours we had, sacrificing sleep to translate our solutions into LaTeX. Dave King writes amazing code, some of which I still cannot comprehend. Raju Kumar was often my only lab mate during late nights and wee hours in those days. As a more senior student, I would come to wonder how he could still pull off such late nights even in his final year.

Anytime we are embattled and dug deep into life's trenches, we inevitably meet some truly special people; people who are so inexplicably and overwhelmingly nicer to us than we are to them, that we can only assume we are truly blessed for having met them. Sandra Rueda was my desk neighbor in the lab for over four years. During that time, she was not only a friend and confidant, but a nearly inexhaustible source of good will. Her thoughtfulness about the well-being of those around her is a lesson to us all. Machigar Ongtang is without doubt, the most patient person I have ever met. Countless times I shared my troubles both in and outside of the lab with her, and she would always be prepared with both her cheerful attitude and advice so deeply philosophical, that it changed some of my beliefs to this day. Thanks, Om. Perhaps no one else is operating on as close a wavelength to my own as Haywardh Vijayakumar. I still remember the first time Hayawrdh asked me if I was going to walk downtown to grab dinner. When I said yes, he said something I wasn't expecting. "OK, I'll go." "What a convenient way to invite oneself along," I thought. Thus began the ritual that has led to the majority of insights that I've had in graduate school. Throughout my many, many discussions with Haywardh, I would like to think that we have gained more insight on life and computer science than we ever could have by ourselves.

So many others have also left their mark on my graduate school experience. Dmitry Podkuiko was my first partner when I was working alone in a closed-off little room at the end of a hall. His frenetic approach to his work took us in directions I had never anticipated, and his various interests, including his encyclopedic knowledge of drug cartels, was a source of endless fascination. Dmitry and I were eventually joined by Sergei Miadzvezhanka, and Adam Delozier, both of whom brought an even more fraternal atmosphere to the project, while also making it more successful. I couldn't have asked for a better team to work with. I was also very lucky to have met two people from my hometown of Altoona, Tom Moyer, and Adam Bergstein. Tom was as great a labmate as one can ask for. As the first person in every morning, he would always have a pot of coffee brewing along with some of the most interesting and useful news bits about security, and Altoona. Adam, whom I met a few years later, was another great friend with whom I frequently enjoyed discussions about Altoona. There is something about the deeper meaning in a conversation with someone from my hometown that always puts a smile on my face.

Arguably, one of the best reasons for going to grad school is the opportunity to meet people from many diverse backgrounds and outlooks. I met many such people during my time at Penn State. Damien Octeau is far too humble for being such a dedicated and precise worker, and his extremely thorough paper reviews have greatly improved the submission quality of my work. He could also probably come up with a comment about that last sentence that would be wry and hilarious. Jil Verdol was always interesting to hear from when we got the chance to go on lab lunches. Devin Pohly, my roommate for most of my time in graduate school has more systems skills then almost anyone else in the world, and he's also quite good at wordplay in a ways I can barely comprehend. Eunjung Yoon was fun to chat with during the short time we got to work together. Matt Dering has always been fascinating to speak with about political and social issues. I was amazed to meet anyone as interested in such things as he is. Yuqiong Sun has spurred several deeply interesting conversations about computer security and how to get work done as a grad student. Xinyang Ge as a certainty about the world that I admire, and he frequently makes me question my understanding of the English language. Srikar Tati has been a great friend over the years, teaching me much about India and the finer points of linear systems, while also having some most amusing and enlightening opinions in our various debates. A frequent cohort with Srikar and I, Wenhui Hu, has taught me more about China than anyone else, and more about Finland as well. Go figure! Nirupama Talele has been my desk neighbor for the last few years, and I am in awe of her ability to laugh off the most blood pressure raising of situations. Diana and Phil Koshy are just two way awesome people in every conceivable dimension. Additionally, Pablo Mayrgundter, my intern host during my summer at Google, greatly helped to shape how I approach new problems and how I measure progress.

Along with my many great influences in the SIIS lab, I have had the fortune of working with two other highly skilled professors. Seth Blumsack, and Saman Zonouz. I met Seth at a reception where I was presenting some of my research. Much to my surprise, not only was he familiar with my area of study, but he also knew of my advisor! Seth was extremely helpful in making introductions and allowing me to present my work to audiences that I would otherwise never have had access to. His energy and positive attitude have always made me want to work harder when he's around. Like Seth, Saman introduced himself after I gave a presentation on some of my work. His enthusiasm and good spirit were immediately apparent, and his skill at organizing projects and keeping people on track led to me working on problems that otherwise would have not been possible. His abilities to quickly pick up new research areas and identify hard problems are traits I continue to attempt to emulate.

Finally, I owe an impractically large debt of gratitude to the two people who have put up with me longer than anyone else: my parents, Richard and Kathy McLaughlin. To attempt to explain the role they had in my success is like attempting to explain the role the earth's gravity plays in the massive commercial success of waterfalls, skydiving, and yo-yos. They set everything I've done in motion, even if their forces cannot be seen by the naked eye. Through their efforts not only in helping me, but in enriching the lives of countless homeless men in Altoona, Pennsylvania for over two decades, they have set an example that I will be eternally working to reach. Thank you, Mom and Dad.

Looking at the profound impact others have had I my life, I begin to wonder, "Have I really done anything?" Many would answer, "Yes, you did the work!" But why? Why did I do the work? At various points throughout my student career, I have made various decisions that resulted in work being done. But does this really have anything to do with, "me?" If I could go back to the times at which I made those decisions, and all circumstances were the same, and my memories were the same, then would I not make the exact same decisions? It seems correct to say yes, but if so, then what credit can I, or indeed anyone, take for making such decisions? If there is some randomness involved in the process, then can any more credit be taken? Or, should I just instead give recognition to noise instead of deterministic causality? In either case, the best available answer is that we share very little, if any, of the responsibility for our actions.

One of the most important concepts in ethics is that of responsibility. However, it seems that the notion of responsibility, as most of us intuitively understand it, has become outdated. Arguably, the best thing we can do for the future is to have a moment of honesty in which we admit that our circumstances are the true authors of our actions. In doing so, we will be able to see each other in an objective light of compassion that has not been possible throughout a history defined by human pride and blame. I humbly submit that this is the single most fundamental mechanism underlying genuine social progress, and I think we would all do well to embrace it.

Dedication

To Mom and Dad and all that came before.

Chapter 1

Introduction

Process control systems are increasingly vulnerable to software-based exploits that have physical consequences. The overwhelming majority of critical infrastructure automation in the world is governed by control systems. With these control systems comes the general-purpose computing and network connectivity that for the first time opens critical infrastructure to electronic attacks from adversaries around the globe. Control systems have proved vulnerable to attacks from traditional computer viruses [1, 2, 3], remote break-ins [4], insider attacks [5], and strategic, targeted attacks [6]. Industries affected by attacks against control systems have included nuclear power and refinement of fissile material [3, 6], transportation [7, 4], electric power delivery [8], manufacturing [1], building automation [5], and space exploration [2].

Furthermore, industrial penetration testers in the last decade have demonstrated the lack of practical security measures in real world control systems and their components. In one study over ten years of penetration test results from industrial control systems, it was found that on average vulnerabilities are not patched for three months after the initial path release [9]. In another study, it was found that over 50% of *reported* attacks against control systems have caused financial losses greater than $1 million [10]. It was also found in the same study that 41% of reported attacks resulted in a "loss of production," and 29% reported a "loss of ability to view or control the plant." In a survey done by power industry security experts, it was found that energy providers make common misconceptions about the security of their own infrastructure, such as believing that their control systems were isolated from public networks, and that using obscure protocols will prevent misuse by external adversaries [11].

Security researchers are also discovering new types of attacks against control systems. Most recently, the control and data acquisition systems in the *smart grid* have come under such scrutiny. Researchers have found vulnerabilities in smart electric meters allowing for theft of energy and denial of power delivery [12, 13]. Flaws have also been found in the basic state estimation functions of power grids allowing for meter based attacks on entire transmission and distribution systems [14], and have generalized such attacks to all linear time-invariant control [15]. Another project showed that under the right conditions, an adversary with only network access to a power control system could physically damage and ultimately destroy a diesel generator [8].

Researchers have also been focusing heavily on the Programmable Logic Controllers (PLCs). PLCs are general-purpose, ruggedized computers that connect directly to the physical machinery in a *plant*. The PLC can be thought of as any other computer except that its input signals come from sensors instead of keyboards and mice, and its output signals go to actuate mechanical devices instead of to monitors. Besides this, PLCs can be programmed in Turing complete languages, can be networked, and, as will be seen, can be compromised like any other computer. Recently, exploit code for vulnerabilities in both Supervisory Control and Data Acquisition (SCADA) systems [16], as well as PLCs [17], and human machine interfaces [18, 19] have been published. Simple attacks given the attacker complete privileges to modify the PLC have also been identified against some of the most common PLC manufacturers including Siemens [20], and against PLCs used in correctional facilities [21].

Compounding these existing problems is the opening of control systems to standard protocols, programming languages, and design practices in the last two decades (since the early 1990s). While "security by obscurity" is known to be insufficient for meaningful security guarantees, the uniqueness of each individual control system's hardware and software at least forced adversaries to expend a modicum of effort to understand a control system. The standardization of control systems now means that Kerckhoff's principal that one should assume that the implementation of a system is known must apply to their security.

Most attempts to address these problems in control system security aim at improving *perimeter defenses* [22]. Perimeter security measures aim to "harden" the IT infrastructure against attacks using intrusion detection, virus scanning, firewalls, access control policies, and software patches, among others. However, for reasons mentioned above, perimeter security measures simply are not sufficient for guaranteeing the safe behavior of physical processes in control systems [10, 9, 20, 11]. Furthermore, it is difficult to analyze the impact of such security measures on process behavior. For example, the creation of a firewall rule does little to guarantee the safe behavior of a variable speed motor at some arbitrary point in the future.

The shortcomings of perimeter security can be explained by the following fact. Traditional security measures, including access control, intrusion detection, and software security, are aimed at making guarantees about confidentiality and integrity of *data*. However, the safe behavior of control systems depends on a completely different set of properties such as stability and observability. Unfortunately, there are currently no mechanisms that enable us to make guarantees about these properties in the face of an adversary. The goal of this dissertation is to examine several promising candidates for such mechanisms. These are based on the concept of *physical behavioral specifications*, an encoding of process engineers expectations about the safe behavior of the plant. The specification is then enforced through the mechanisms presented below. First, we consider how specifications can be leveraged by attackers to reason about attacks against control systems without having to also consider implementation details. We will first discuss attack tree-driven penetration testing of smart electric meters, followed by a framework for automating attacks against control systems. This is followed by our static and dynamic approaches to physical specification-based behavioral enforcement.

1.1 Using Attack Trees to Reason about Control System Attacks

We begin by describing a methodology for attacking a control systems by evaluating and comparing attacks from each of the above three categories, namely *attack trees*. A derivative of fault trees [23], attack trees aim to break down the possible steps and sub-steps for attacking a system in the hopes that the system can then be better defended [24]. They were first used to reason about vulnerabilities in SCADA system by Byres et al. [25]. In this approach attacks are planned against the Modbus protocol as used in the petroleum and power systems.

McLaughlin et al. used attack trees to model different means of achieving adversarial goals in Advanced Metering Infrastructure (AMI) systems [13]. An example attack tree is shown in Figure 1.2. The root of this tree is the ultimate goal of the attack, namely, to tamper with the reported electricity usage data in order to commit energy theft. The three subtrees immediately below the root show the three overarching strategies for achieving this. The first is to tamper with the measurement itself (data acquisition), the second is to tamper with the stored data (data representation), and to tamper with the data in flight (data communication). As can be seen, McLaughlin's model closely mirrors the three areas of control system attacks described above.

The attack tree shown in the example is limited in its specificity. The leaves of the tree describe types of attacks that are system-independent. For example, "Intercept Communications" does not tell how to intercept the communications in any specific system, only that intercepting the communication is a prerequisite for one path to tampering usage data. This was termed as an *archetypal attack tree*. Archetypal trees can be used to reason about systems at the architectural level only. That is, they can be used to reason about systems based on their behavior, but not their specific implementations. To do the latter, *concrete attack trees* are used. The concrete trees for the energy theft tree are shown in Figure 1.1.

The roots of concrete trees are the leaves of an the archetypal tree. As can be seen in the figure, the two trees correspond to two of the prerequisites for the data communication attack (c) from the archetypal tree. The leaves of the attack trees represent attacks that should be attempted as part of penetration testing. Using this method of archetypal and concrete attack trees, attacks were designed and executed for energy theft, denial of meter reading, and targeted disconnect of customer power. The limitation of the attack tree method is that it places the burden of sorting out implementation details on the adversary. Details of our full work in attack tree directed pen testing can be found in Chapter 3.

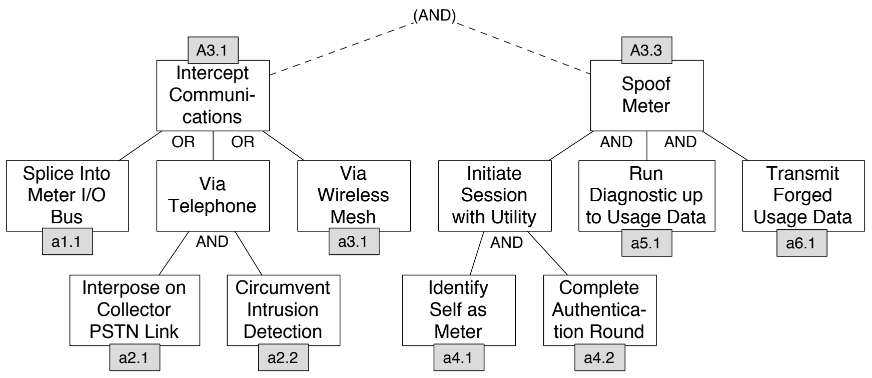

(AND)
A3.1
Intercept Communi-cations
OR
OR
Splice Into Meter I/O Bus
a1.1
Via Telephone
AND
Via Wireless Mesh
a3.1
Interpose on Collector PSTN Link
a2.1
Circumvent Intrusion Detection
a2.2
A3.3
Spoof Meter
AND
AND
Initiate Session with Utility
AND
Run Diagnostic up to Usage Data
a5.1
Transmit Forged Usage Data
a6.1
Identify Self as Meter
a4.1
Complete Authentica-tion Round
a4.2

Figure 1.1. Concrete Attack Tree
Example concrete attack trees for energy theft.

1.2 Automating Attacks against Control Systems.

There is one layer of obscurity for which researchers have yet to identify an attack. While the protocols and programming languages used by modern control systems are converging to a relatively small set of standards, the exact purpose of each control system remains unique. For example, two chemical process may both use Siemens PLCs, communicate using the Modbus protocol, and be programmed in the Ladder Logic language, but the actual behavior of the two systems will vary greatly depending on the specifics of each chemical process. This raises the key question that we hope to answer in the remainder of this work: *Can adversaries with no knowledge of a control system's implementation carry out a meaningful and potentially stealthy attack against it?*

If the answer to this question is a "yes," then we will have significantly lowered the bar for the sophistication needed by an adversary to attack control systems, and perhaps even national critical infrastructure. If the answer is "no," (a much harder conclusion to come by!) then we will have established that adversaries must either have prior knowledge of control system implementation or must do additional reconnaissance on the target system.

To gain some understanding of this problem consider an adversary that has just penetrated the network perimeter of a control system that is believed to govern a waste water treatment process. Furthermore, assume that the adversary wishes to perform a targeted and possibly stealthy attack as opposed to just vandalizing the system. (An example of the former would be allowing partially contaminated water to drain from the system at certain times of day, while an example of the latter would be simply shutting the system down or causing random, unpredictable behavior.) Our adversary has several challenges ahead. First, it must be determined if the PLCs and other nodes in the penetrated system indeed are related to a waste water treatment process. Second, if there are multiple PLCs or other points of control, the subprocess controlled by each must be determined. Finally, it must be known what special locations in the PLCs memory are actually assigned to control specific devices that are usually present in waste water treatment facilities.

Without extensive manual inspection of the code running on each PLC, it will be hard if not impossible for the adversary to answer any of these three questions, let alone construct the desired payload. But is it possible for an automated analysis to reveal the necessary implementation details of the process to make payload construction possible? Furthermore, can a payload be constructed for the target system automatically? Such behavior would be advantageous for a number of reasons. First, as was the case with the Stuxnet attack, an air gap may exist between public networks and the target PLC. Such a gap must be spanned either by propagation through removal storage such a USB drive [26], or with the aid of an insider. We note that for non-critical facilities, this is less of an issue as Internet-enabled PLCs are now entering the market [27, 28]. A second reason that PLC malware must act autonomously is that it significantly reduces the sophistication needed on the part of its author. By assuming that PLC malware may be written in a generic form by a few skilled individuals and widely disseminated to unskilled insiders or script kiddies, we obtain a stronger adversary model for when we later consider mitigations against PLC malware. Finally, as will be shown later, one scenario for the use of PLC malware is the indiscriminate attacking of any PLC onto which it happens to propagate. This obviously cannot be done with any specific goal in mind, and thus an attack must be derived on the fly.

1.2.1 Indiscriminate Attacks against Control Systems

In this section, we preview our work in constructing payloads for control systems without knowledge of their behavior or their implementation. Such attacks make sense for indiscriminately propagating Internet worms or when adversaries simply have no knowledge of even basic control system behavior.

In an automated, indiscriminate against a control system, it is assumed that the adversary has gained access to the PLC (this entails both reading and writing of the PLC's configuration data and code), but does not know how to proceed to construct a payload. This could be for example, because the adversary cannot read the PLC code, does not know which memory locations to modify in the PLC to alter real world devices, or does not know what the target plant does, or how it does it. Can such an adversary reliably cause harm to the target plant? To begin answering this question, we can first observe what the adversary can do without any additional help.

Perhaps the most basic attack would be to simply wipe the PLC's memory. This could be potentially very damaging if certain physical elements in the plant continued running in their course awaiting orders from the PLC to stop. For example, if a valve used to fill a tank were to remain open until the PLC sent a "close" signal. There is however, no guarantee that such a result would happen. Furthermore, this attack provides no stealth as it will be immediately apparent to engineers that something has gone wrong in the plant. It will also be clear that something is wrong with the PLC itself as wiping its memory would halt the transmission of process data from the PLC to the control room. A similar attack would be to write random instructions to the PLC's program memory, but this suffers from the same issues as the memory wipe attack.

Given the failure of these first two attempts, we now give our adversary one additional piece of information. The adversary (or a program) can read and decompile the code running on the PLC. From this, the adversary can see (1) A set of variables that contain values of sensor data, (2) a set of variables that can be written to control plant devices, and (3) the control logic governing the behavior of the variables. Alternatively, this can be thought of as a state machine of the target plant with (2) being the state variables and (3) being the transition function on inputs from (1). Note that the adversary does not know what sensors or devices are actually mapped to the variables in (1) and (2), only how the logic in (3) relates the two. Worsening the adversary's problem, PLC's normally do not have helpful variables names like "*KilnHeatSensor*" or "*FloodGateClose*." Instead they have names like x_1 for inputs and y_2 for outputs.

What good is the knowledge of this program code to the adversary? If it is unknown whether a variable y_i controls a valve, pneumatic arm, alarm light, stepper motor, or variable speed drive, then how can the adversary reliably assign a value to this variable to cause malicious behavior in the plant? The answer to this question lies in the fact that the underlying state machine for the process will only enter a certain limited set of states in the whole state space. Furthermore, there will be certain states for which the controller code will contain explicit checks to never enter. These checks are known as *safety interlocks*, and are present in virtually all control code regardless of the process. By observing the safety interlocks, the adversary need not know to what devices some variables y_1 and y_2 are tied, only that the control code explicitly forbids $y_1 \wedge y_2$. Thus the adversary can upload code to the PLC that intentionally sets both y_1 and y_2 to be on simultaneously with the knowledge that the process engineer did not want this to ever occur.

We will further refine this notion in Chapter 4, but for now, let us consider this safety interlock extraction method in light of the earlier suggested methods to see if we have made an improvement. In the memory wipe and random instruction attacks, the adversary had no way of knowing whether the payload would cause undesired behavior in the plant, and furthermore, this behavior would likely be quickly noticed from the control room. In the safety interlock violation attack, the adversary not only knows that the payload is violating some rule that the process engineer did not want violated, but also knows how to spoof proper plant behavior back to the control room. For example, if the payload is to set y_1 and y_2 both on simultaneously, then it could also contain code to send messages back to the control room stating that only y_1 or y_2 was on, thus prolonging the duration of the attack. At this point, we can safely say that the safety interlock extraction attack is an improvement on previous more naive attacks. However, as we will see in the following section, if the adversary has one additional piece of information, namely plant *behavior*, even greater improvements can be made.

1.2.2 Specification-based Attacks against Control Systems

In the previous section, we previewed a means for automatically generating a payload for a PLC in a completely unknown control system. This relied on the fact that in any control system, the definition of "safe" behavior is encoded directly into the PLC's control logic. We made the preliminary judgement that this approach gives the adversary more certainty in the efficacy of the attack than does the two naive methods: PLC memory wiping, and random PLC instructions. However, the adversary still lacks control over the exact outcome of the attack. For example, it is not clear why the adversary should choose to violate one safety interlock over another.

In this section, we introduce our tool Sabot (Specification-based Attacks against Boolean Operations and Timers) that solves the primary problem still facing the adversary: recovering the semantics of variables in PLC control logic. That is, given a list of devices believed to be in the target plant with a description of those devices *behavior*, the method presented here searches for a Variable To Device Mapping (VTDM) that tells the adversary which variables map to which specific plant devices. Give a VTDM, the adversary can make more precise decisions about how to construct a PLC payload to achieve a highly-specific goal. Furthermore, Sabot allows for malware to be targeted against a control system with exact attack specifications to be carried out. For example, the adversary can encode a generic attack payload like, "Leave $valve_1$ open for an extra second," and leave it up to Sabot to automatically determine which PLC variable actually controls $valve_1$. But how does Sabot know which variable does this?

The adversary assists Sabot in answering this by adding one additional piece of information: a behavioral *sketch* of how $valve_1$ is believed to behave in the target process. For example, if it only opens after a button b is pressed and then closes when $valve_2$ opens, the adversary would encode the following into the sketch:

$$b \wedge \neg valve_2 \Rightarrow \text{AX } valve_1 \tag{1.1}$$

We describe the specifics of the sketch input language in Chapter 2, and discuss effective sketch writing in Chapter 5. For now, it is enough to know that Sabot checks the behavior of devices in the sketch against the behavior of variables in the PLC's control logic. If a variable is found with behavior that matches that of one of the devices in the sketch, the Sabot assumes that the variable does indeed control that device (unless there is a later conflict with another variable, in which case additional behavioral checks are done as described in Chapter 5, Section 5.3.1).

An attack using Sabot proceeds as follows:

1. The adversary encodes their understanding of the target system into a *sketch* of the plant behavior. The sketch may not be completely accurate, but it must capture the core process behavior. The sketch is provided to Sabot.
2. Sabot downloads the existing control logic from the victim PLC and finds a mapping between the sensor and devices in the sketch, and the variables in the control logic.
3. Sabot uses this mapping to instantiate a malicious payload and upload it to the victim

Table 1.1. Comparison of TSV and C^2

	TSV	$\mathbf{C^2}$
Method of enforcement	Static	Dynamic
Time of enforcement	Load Time	Run Time
Advanced warning	Yes	No
Development time impact	Yes	No
PLC must be trusted	Yes	No
Physically intrusive	No	Yes
Blacklisting possible	No	Yes
Specification language	LTL	`sslang`
Control-specific language	No	Yes
Checkable complexity	Bounded	Unbounded

PLC.

The contributions of this work are grounded in these steps. First, we develop techniques for constructing a logical model of the control system solely from its bytecode. Here, we disassemble the PLC instructions into an intermediate set of constraints on sensor and device variables. Second, we develop an automated approach to extract the exact one-to-one correspondence between control logic variables and physical devices, known as the *variable-to-device mapping* (VTDM). We use a model checker to map the sketch onto the systems of constraints derived in the previous step. Lastly, we develop techniques for instantiating a generic attack vector for the control system implementation using the identified VTDM. Further discussion of SABOT can be found in Chapter 5.

1.3 Specification-based Defenses

Security properties such as integrity and confidentiality are ideal when the most critical asset being protected is pure information. In control systems, especially those used in critical infrastructure, this is not the case. For them, the most critical asset is the physical machinery, raw materials, and personnel that must be protected from harm at all times. Given that we want to directly regulate the most critical assets, we will need properties other than integrity and confidentiality in order to make guarantees about the safe behavior of physical systems. In this dissertation, these properties take the form of *specifications*. In any matter of regulation, there are two components: policy and mechanism. For physical specification-based behavioral enforcement, the policy is a set of safety properties designed by process engineers. The mechanisms, however, are not as clear. The way in which a policy is enforced depends on a number of factors including the architecture of the system in which the enforcer must function, the nature of the policy to be enforced, the regulated system's performance requirements, the threat model and assumed adversary capabilities, and many others. For these reasons, the design of an appropriate mechanism enforcing behavioral specifications is an important and difficult question.

In this work, we posit two answers to these questions. The two approaches to specification-

based enforcement are based on a number of design decisions and trade offs summarized in Table 1.1. The first solution is the Trusted Safety Verifier (TSV) (See Chapter 6). TSV is a static enforcer that checks all PLC-bound code for the potential to cause malicious behavior. The other approach is a dynamic Controller Controller (C^2) (See Chapter 7). C^2 uses a novel policy language designed to assist in writing succinct descriptions of desired physical behavior. There are a number of important contrasts between the two. Because TSV statically checks controller code, it can given a highly advanced warning when potential malicious behavior is detected. Thus, preventing the bad code from running in the first place. On the other hand C^2 has the advantage in that it does not impose significant overhead on process development time, as TSV does in the case of accidental safety property violations. Counting against C^2 is the fact that it must be interposed on all wires connecting PLCs to physical devices, whereas TSV is not so physically intrusive, requiring only a single network interposition.
Important Note. It is important to note that when the words *static* or *dynamic* are used in this dissertation, it does not mean in the program analysis sense, i.e., static analysis vs dynamic analysis (such as symbolic execution). Instead, these terms are used in the sense of computability classes [29]. Thus, we are not implying that TSV's use of symbolic execution is a type of static program analysis, but that it represents a static enforcement mechanism because it must accept or reject a program after examining it for a finite number of steps.

1.4 Related Work

1.4.1 Attacks against Control Systems

Attacks against control systems can occur at essentially three distinct places within the system [30, 12]. We now describe each and give examples of real world and research attacks against each.

1. **Data Acquisition:** At this step, sensors collect measurement data from the physical system. Attacks against Data Acquisition involve either physically tampering with sensors, or compromising the security of the sensor network. In either case, this type of attack is most commonly known as a *false data injection attack*.

2. **Data Communication:** At this step, the aggregated data is transmitted to a control room or more centralized level of control. Attacks against data communication typically leverage security flaws in outdated control system protocols to inject false data.

3. **Data Presentation and Control:** At this step, the aggregated data is both presented to human operators who make control decisions (which is the main function of a SCADA system), or is used to automatically control a physical process (which is the main function of distributed control systems). Attacks against this step, especially those against controllers themselves, differ from the previous three steps in that they can directly modify the control

signals sent to plant devices. An adversary able to execute such an attack thus has complete control over the plant, thus making them the primary focus of this work.

Data Acquisition. A recent attack against data acquisition was described by Liu et al. was against state estimation in electric grids [14]. State estimators take measurements of voltage and current amplitude and phase from around the distribution network to estimate the actual load on any given substation. Liu's attack relied on the fact that while these state estimators are robust against random errors, an adversary with knowledge of grid topology can craft a set of malicious measurements that can bypass the error detection in the state estimator, thus giving grid operators an incorrect impression of grid state. This can lead to poor decision making about the n-1 recoverability of the grid given potential failure points.

This attack is actually a specific instance of a more general attack against Linear Time Invariant (LTI) control systems [15]. These are systems that make a linear approximation of plant behavior, and assume that the coefficients in the linear model remain constant over time. In this attack, an unstable eigenvalue for the control matrix is found. An unstable eigenvalue is one that when multiplied with a scalar will produce yet another unstable eigenvalue. If the eigenvector that corresponds with the unstable eigenvalue is in the reachable state space of the control system, then malicious inputs to the control system are crafted to push it into the state described by that eigenvector. This then allows the system to be pushed arbitrarily far away from its desired state without detection from a Kalman filter-based state estimator.

Data Communication. An attack against data communication have similar effect as attacks against acquisition, that the final data read by the control room is forged. Communication-based attacks however typically leverage security flaws in the protocols used for data communications between sensors, PLCs, and the control room. These attacks can occur by spoofing one endpoint, e.g., spoofing a PLC or sensor, or by performing a man in the middle attack. Because many SCADA and DCS systems have no authentication whatsoever for these entities [9, 31], it is trivial to inject forged sensor measurements into the data stream. A striking example of this was the 2008 derailment of a train in the city of Lodz Poland [7]. In this case, a wireless infrared (IR) communications system was used between train control and the actual track sensors and switches. A Polish teenager was able to reverse engineer the IR control protocol, which had no security measures in place, and construct a home-made controller for the protocol out of a television remote. Using this device to tamper with the wireless control system ultimately caused a derailment, leaving 12 injured.

In another case, a vulnerability was demonstrated in smart electric meters that allowed for a meter spoofing attack [12]. While correctly implemented cryptographic algorithms [32] were put in place to authenticate smart meters back to the centralized meter data management system (MDMS), there was a single flaw in the implementation. As a part of the mutual authentication, the meter invents and transmits a *nonce*, a unique one-time use value, back to the MDMS. This nonce is then used to add *freshness* to the authentication to prevent a replay attack of an old session. In the particular implementation of the protocol, however, the nonce was not stored by the MDMS, and thus an adversary can spoof a meter using a laptop computer to replay a previous authentication round.

Data Presentation and Control. Attacks of this nature are arguably the most dangerous as they allow an adversary to not only completely control the behavior of all devices under the domain of the controller, but also do hide their behavior from human operators. This was made most clear with the discovery of the Stuxnet attack in 2010 [26]. The Stuxnet virus had both a sophisticated delivery mechanism, and a sophisticated payload. The delivery mechanism used a number of zero day exploits and other novel techniques to penetrate networks of computers running specific versions of the Windows operating system. Arguably even more impressive was that Stuxnet had a precompiled PLC payload believed to be responsible for manipulating centrifuge drives used in the refinement of nuclear fissile material [6]. Stuxnet also was able to manipulate values in the PLC's memory such that only correct seaming measurements were reported to the control room.

The main limitation of this approach is that to posses the precompiled PLC payload, Stuxnet's designers would of have to had advanced knowledge of the implementation details of the target plant. Thus, an attack imitating the Stuxnet model could only be executed by nation states or other entities with sufficient resources to espionage. The attacks suggested in this work would remove this limitation, while giving the adversary nearly the same degree of confidence in the attack that a precompiled payload would give.

Many of the security flaws that give adversaries access to PLCs have been codified into standardized tools usable by unsophisticated attackers. Exploit frameworks like Metasploit [33] use collections of known exploits to partially automate the task of vulnerability testing. These frameworks have been extended to attacks against SCADA and control systems [34]. While, exploitation of control systems is however not new [10], the rate of release of new SCADA and PLC exploits has been accelerating [9, 35]. SABOT extends automated exploitation to automatic generation and delivery of payloads for systems with obscure implementations.

1.4.2 Formal Verification of Control Systems

In many cases, control software is so safety critical that it must be completely and formally verified before entering a production environment. A formal verification of a hardware or software system consists of first stating some properties that the system should maintain, and then proving mathematically that the system maintains these properties. By far the two most common methods of formal verification are automated theorem proving and model checking. In automated theorem proving, the system is encoded into a set of axioms, and then an attempted is made to automatically prove the specified properties given that set of axioms. In model checking, the system is encoded as a finite state machine, and the specified properties are encoded as constraints on the execution of this state machine. The overwhelming majority of literature on formal verification of sequential control systems deals with model checking-based verification.

One of the applications of sequential control for which model checking has been most widely and successfully applied is that of railway interlocking. Railway interlocking systems essentially

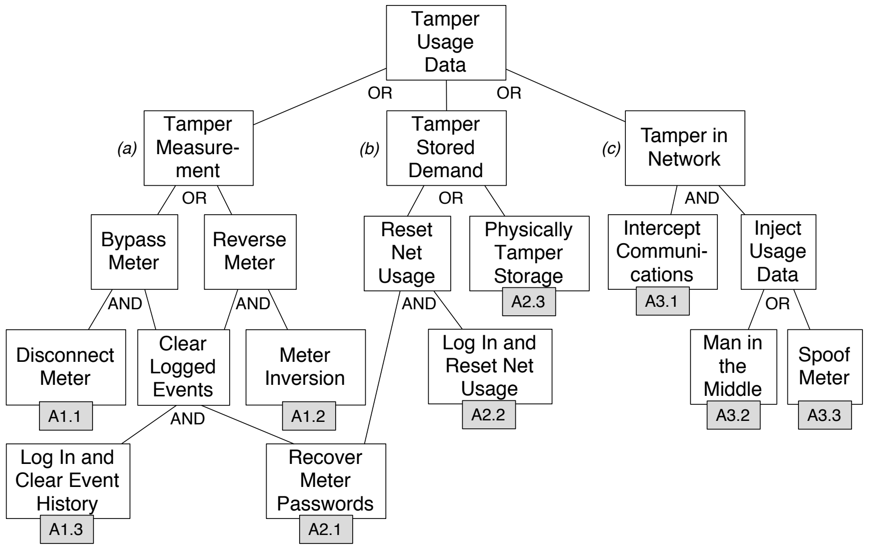


Figure 1.2. Archetypal Attack Tree
An example archetypal attack tree for energy theft.

enforce a set of rules over parts of a rail system. Examples of such rules are: "No two trains should ever simultaneously enter the same portion of track" or "If the switch is active for a given track junction, then a the corresponding signal must be on to notify the engineer of the track switch." Of course, given hundreds of trains traveling thousands of track segments per day, the rules can grow very large and must be verified via machine. One of the earliest such efforts used bounded SAT-based model checking to verify properties of a railway interlocking system represented as a system of Boolean equations [36]. While this work was able to guarantee that the interlocking system exhibited certain properties, at the level of individual scan cycles, it could not verify any real time guarantees as done in [37]. Later works extended this verification to subway systems that included variable speed codes broadcasted at each track segment [38]. None of these methods are able to verify that a system maintains a property without knowledge of the VTDM.

As described above, railways are not the only systems that maintain software-based safety interlockings. Sachdev et al. describes a method for automatically generating a set of safety interlockings for an electrical substation to avoid faults while switching [39, 40]. Park used binary integer programming to verify that safety properties were maintained by sequential chemical plant processes encoded as systems of Boolean equations [41]. More generally, efforts have also been made to verify properties of programs written in special-purpose PLC programming languages. Zoubek et al. converted *ladder logic* programs into timed automata that were then checked for real time properties. Ladder logic, which is a high-level language for control logic programming, aims to model PLC programs visually as relay circuits. In contrast the Instruction List (IL) language is essentially an assembly language for PLC programming. Huck. specified a formal semantics for IL programs and showed that it can be used to verify some basic properties using theorem proving [42].

An alternative to verifying that a system correctly implements a set of specified properties, one can start with a set of properties and *synthesize* a system that is guaranteed to implement them. In sequential control systems, the seminal and most long lasting work towards controller synthesis is the supervisory control theory of Ramadge and Wonham [43]. In this theory, the system designer specifies one or more discrete event process that describe a *plant*. The plant generates events one after another, and it is the job of the controller or *supervisor* to regulate the behavior of the plant such that it only visits a series of safe states, and ultimately makes progress towards a *marked state*. (This latter property is known as a non-blocking supervisor.) The goal of RW control theory is to automatically synthesize such a supervisor based on the specified properties. In the worse case for generating supervisors for a plant made up of three or more parallel discrete event systems, this synthesis is NP complete [44].

As a final note, we mention that there is related work in the area of specification-driven control systems, and it is deferred to the appropriate sections below so as to allow for fine-grained contrasting with the presented work.

1.5 Thesis Statement

We now turn to the principle contribution of this work. Specifically, we claim that adversaries and defenders can attack and defend control systems based on specifications of physical behavior. In each case, the practical use of the specification is very different, however, the underlying reason for using it remains the same: the physical resources and process are the most critical assets in a control system. The claim is broken down into two parts. The first describes the key insight of this dissertation, that we need new types of guarantees and methods of enforcement for control systems. The second part is more pragmatic. It makes the claim that specification-based methods offer promise for real-world adoption, and that they merit further study. We now state the thesis of this work, followed by pointers to the supporting evidence throughout the dissertation.

(i) Attacks against and defenses for cyber physical control systems can be substantially simplified using specifications of physical behavior instead of traditional approaches. (ii) Both are achievable within the constraints of real-world control systems.

To back up these claims, this thesis explores on example of a specification-based attack, and two examples of defenses. The attack is described in Chapter 5. Here, we see that the adversary's need for specification lies in a layer of semantic obfuscation present in every PLC. Namely, even if an attacker knows how the victim control system operates, it is likely that the exact method by which the PLC achieves this is unknown. The tool presented in that chapter allows the adversary to focus solely on writing behavioral specifications. The first of two defenses is examined in Chapter 6. This chapter evaluates the challenge of preventing bad code from being loaded onto PLCs. This is done through a static enforcement monitor that verifies PLC-bound code does not violate engineer-supplied safety properties. Finally, we consider the second specification-based defense mechanism in Chapter 7. This mechanism runs in real time at the last step of the control system between computation and physical machinery. Thus, any malicious action originating throughout the system can be blocked by this method. Also introduced in that chapter is a language `sslang` for the encoding policies regarding the usage of physical machinery in control systems. We give closing observations and directions for future study in Chapter 8.

Publications Related to This Work:

- Stephen McLaughlin, Dmitry Podkuiko, and Patrick McDaniel. **Energy Theft in the Advanced Metering Infrastructure**. *4th International Workshop on Critical Information Infrastructure Security (CRITIS 2009)*, Bonn, Germany. September, 2009.
- Stephen McLaughlin, Dmitry Podkuiko, Sergei Miadzvezhanka, Adam Delozier, and Patrick McDaniel. **Multi-vendor Penetration Testing in the Advanced Metering Infrastructure**. *26th Annual Computer Security Applications Conference (ACSAC 2010)*, Austin, TX, USA. December 2010.
- Stephen McLaughlin, **On Dynamic Malware Payloads Aimed at Programmable Logic Controllers**. *6th USENIX Workshop on Hot Topics in Security*, San Francisco, CA. August, 2011.
- Stephen McLaughlin and Patrick McDaniel. **SABOT: Specification-based attacks on Sequential Control Systems**. *18th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, USA. October 2012.
- Stephen McLaughlin **Stateful Policy Enforcement for Control System Device Usage**. *29th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, USA. December 2013.
- Stephen McLaughlin, Devin Pohly, Patrick McDaniel, and Saman Zonouz, **A Trusted Safety Verifier for Process Controller Code**. *ISOC Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, USA. February 2014.

Chapter 2

Preliminary Concepts

There are two basic concepts that form the basis for our work in automatic payload construction for control systems: (1) the role of Programmable Logic Controllers (PLCs) in control systems, and (2) the formal verification technique of model checking. We begin discussion of these concepts by defining the purpose and function of closed loop control systems, and explaining where sequential control systems fit within this domain. PLCs are then introduced as the basic unit of control at the boundary between computation and the physical machinery in the plant. A formal and widely used model of a control system and PLC control logic is then given, and used in the following section on model checking. An example is also presented in this model. There are also several practical concerns relating to PLCs, namely, their interfaces to other computers and machinery, and the ways in which they are programmed. Both of these details are important in implementing PLC malware and evaluating its potential efficacy in the real world.

The second section, on model-checking based formal verification of sequential control systems, places emphasis first on the basic theory of model checking, and second, on how both models and property specifications can be encoded for input to model checking tools. In specific, we are interested in the input language and capabilities of the NuSMV model checker. We conclude this second section with examples of how our example control system from the first section can be encoded into the NuSMV input language, a task that will be automated in later chapters.

2.1 The Advanced Metering Infrastructure

AMI may be divided into utility-side management, smart electric meter deployments, and the networks that connect these two. This section describes these three along with AMI security concerns. At the edge of the AMI resides its main component: smart electric meters. A smart meter is a digital equivalent of a stand-alone electromechanical meter. Their most distinguishing characteristic is the use of two-way network communication with utilities. Smart meters have evolved from early Automated Meter Reading (AMR) systems [45] to allow for automatic updates of dynamic pricing information [46] and curtailment of individual loads when the grid is under stress [47]. Internal storage is used to keep time of day measurements for Time of Use (TOU) pricing schemes [48] and logs for both power outages [49] and potential intrusions, the latter of which is further explored in section 3.3.1.

2.1.1 Smart Meter Architectures

A smart meter is a networked embedded system equipped with a special apparatus for sensing electrical currents flowing through wires. In this section, we tease out the details of this definition, starting with the individual computing platform and finishing with the network. Unless otherwise specified, the features described in this section are present in the vast majority of commercial smart meters.

Meters that are kept outside, such as those in the US, reside in protective socket enclosures, while those kept inside, which is common in the EU, often do not require a socket. The meter's internals are further protected by its cylindrical housing which consists of a base and a removable cover. To detect tampering by removal of the cover, a "flag" style aluminum tamper seal connects the cover to the base. This inexpensive seal consists of a stem which must be broken to remove the cover and a flag with a stamped identifier for the seal. As one might expect, there are no restrictions preventing the purchase of the seal with whatever flag marking is desired, making the removal of a seal for the purposes of physical tampering inconsequential.

The activities of a smart meter are coordinated by its Microcontroller Unit (MCU). The majority of work done by the MCU involves retrieving energy measurements from the low-level *meter engine* and storing them in flash memory for later transmission to the utility. Smart meter storage, however, is not used for electrical measurements alone. Like any general-purpose system, smart meters maintain logs of event histories and operating conditions. While the set of logged events varies between meter vendors, we cover the logs relevant to our security analysis later.

For flexibility of installation, smart meters within the same deployment can communicate over a number of different network mediums and topologies. Thus, meter firmware is designed to support a generic communication interface, leaving the specifics of a given network to a pluggable Network Interface Card (NIC). The meter exports a generic serial interface to communicate with the NIC, leaving the processing of specific network communication to the NIC.

If a meter is out of network communication with the utility and configurations or repairs are needed, it can be controlled locally through a standard infrared optical port located on its front panel. These ports are accessed via a small optical probe consisting of an LED and a photo-sensor at the range of less than one inch. While most meter vendors follow the physical layer standard for this port [50], the application layer is often proprietary. Typically, the optical ports transmit all data in the clear including passwords for user authentication. This includes the meter's administrator password.

One final component that deserves attention is the remote disconnect switch. If a utility wishes to disconnect a customer's power, it may do so remotely by transmitting a request to the

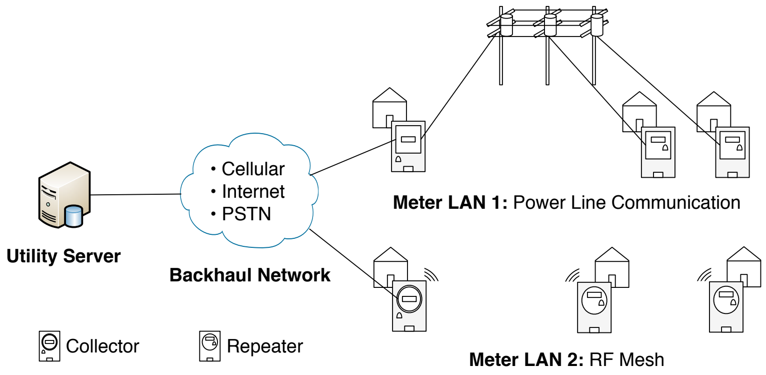


Figure 2.1. Connectivity of meters to utilities given two configurations of meter LANs.

meter to open the switch. The request is received by the digital portion of the meter, which issues the signal to the switch to break the circuit for the power flowing through the meter.

2.1.2 Meter Networks and Utility-Side Management

Given the sheer size of a utility's customer base, achieving networking connectivity with a meter at each individual home is a serious logistical challenge. Given the near impossibility of placing each individual meter on a public network, smart meters are designed to form their own LANs, each of which relies on a gateway device for communication between the LAN and public network. Some common choices of LAN and public network configurations are shown in Figure 2.1.

In the most common meter LANs, meters are connected in an adaptive wireless mesh network. Each meter in the mesh is a *repeater* that propagates data through the LAN to a *collector*. In some cases, the collector may itself be a meter. Power Line Communication networks piggyback signalling over power distribution lines to form a star network topology that directly connects each meter in the LAN with the collector. The collector connects to the utility via a *backhaul* network such as the cellular or landline phone network, or the Internet.

On the utility end of the meter network resides a PC or server machine responsible for performing all regularly scheduled interactions with the meter. This machine runs a commodity OS, e.g. Microsoft Windows, a database server and the proprietary meter server software. If the utility server is compromised, the entire meter deployment is compromised.

2.1.3 AMI Security Concerns

Since smart meters have first come under scrutiny, concerns have been raised regarding their accuracy, reliability, security and privacy [51]. Academic and industrial pen-testing efforts have found flaws in smart meter hardware [52], firmware [53] and network protocols [12]. Recently, Pacific Gas and Electric (PG&E) has experienced problems with measurement accuracy and meter network connectivity in their 5 million meter deployment, one of the largest in the US [54]. The addition of networks of such large numbers of devices to the uncontrolled Internet has been known to leave systems vulnerable to Denial of Service (DoS) attacks stemming from incompatibilities between their rigid proprietary designs and the Internet's open architecture [55, 56]. It will later be shown that this is the case for one of our pen-tested systems.

In addition to basic cyber security concerns, the advanced measurement capabilities of smart meters makes them a potential threat to privacy if used in an unrestricted manner. This is due to their ability to implement Non-Intrusive Load Monitoring (NILM), which can disaggregate the loads exerted by the individual appliances in a house from the net load recorded at the electric meter [57]. Hart posited NILM's use as a means of surveillance over activities that are normally considered within the sanctity of the home [58]. More recently, Lisovich et al. showed that the appliance information extracted by NILM is useful to recover some information about occupant behavior [59]. While this paper is limited to AMI related concerns, we mention that attacks on sensors in the grid's core distribution network have also been considered [14], along with the necessary conditions for such attacks to lead to large scale cascading failures [60].

2.2 Control Systems

A closed-loop control system is a means for regulating the value of one or more *state variables* in a physical system by repeatedly monitoring and modifying the system *control variables.* For example, consider the task of mixing the contents of a tank only after it has filled to a certain level. If the volume and fill rate of the tank are known precisely, then only open loop control is necessary. For example, one could calculate the exact time needed for the tank to fill, and have the mixing begin only after that time has expired. But what if the flow rate to the tank is variable? In this case, a closed loop control is needed to constantly monitor whether the tank has reached the desired level, and subsequently, begin the mixing process.

In this simple example, the *state variable* in question is the level of the tank's contents. In practice, this could be a real valued continuous variable, but more realistically, it would be a Boolean variable designating whether the tank contents have reached the desired level (true) or have not yet reached the desired level (false). The *control variable* in the example is the mixer, which has a state that can be directly modified by the control system. Similar to the state variable, the mixer may be in some bounded real, integer, or Boolean domain. Most often, it will be Boolean, but could in some cases also be in the integer domain in the case of a mixer with a variable speed drive with several settings.

More generally, control systems come in three basic flavors, continuous, sequential, and hybrid. Continuous control systems have state and control variables in the continuous domain, usually, the real or complex numbers. Sequential control systems have variables in the Boolean domain, though in theory, any bounded countable domain will suffice. Hybrid systems contain both continuous and sequential elements.

In this work, we restrict ourselves to sequential (or sometimes called discrete or digital) control. This is done for several reasons. First, continuous control is most often used for maintaining *setpoints*, desired steady states in a continuous system. Thus, they will be inflexible and special purpose, utilizing techniques such as PID control, Linear state estimation, Kalman Filter state estimation, and Linear-quadratic-Guassian (LQG) controllers. Such systems have been previously attacked by false data injection attacks [15], but are less interesting from the point of view of data presentation and control attacks. The second reason we focus on sequential control is that it often forms the "supervisory control" element of the control system. That is, while continuous control is very reactive to current plant state, sequential control will actively drive the plant through a sequence of states, making it more akin to procedural computer programming. Thus, sequential systems tend to provide general-purpose computing environments that are more open to abuse. Finally, a large body of existing work in model checking sequential systems is at our disposal.

In the remainder of this section, we go into more detail about the design and implementation of sequential control systems, and give an example of a control system for a basic chemical mixing process, placing emphasis on the role of the PLC. We then detail the low-level instruction sets used by PLCs. Knowledge of these is instrumental in writing tools for automated payload generation.

2.2.1 Programmable Logic Controllers

A Programmable Logic Controller (PLC) is a digital, multi-input multi-output computer used for real-time automation of physical machinery. They are used in virtually every control application from assembly lines to nuclear power plants. The PLC sits in a tight closed loop with the physical system it controls. Many times per second, the PLC reads sensor measurements, calculates the necessary change to the system, and sends commands to physical machinery to make the changes. The PLC uses a modifiable software program to perform the second step.

A PLC's program is executed continuously as long as the PLC is running. Each execution of the program is called a *scan cycle*, and typically lasts several milliseconds. On each scan cycle, three steps occur. (*i.*) The sensor inputs are buffered into the *input memory* (`I`). (*ii.*) The PLC program is executed to perform calculations based on the input memory and state from the previous scan cycle. (*iii.*) The result of the PLC program is buffered in the *output memory* (`Q`), from where it is transmitted to the plant machinery. In addition to the sensor and machine interfaces, PLCs have a separate *programming interface*, e.g., Ethernet or RS-232 for uploading of new code and data by process engineers and plant operators. TSV's job is to efficiently check any code coming over this interface for safety properties specific to the plant machinery.

In addition to the typical features found in most instruction set architectures PLCs employ a number of special features that TSV must handle.

- **Function Blocks.** PLCs execute code in discrete segments called *function blocks* with fixed entry and exit points. Each function block has a local memory that only it can address. This is different from stack memory in that each function block's local memory exists in the same absolute address space, i.e., not relative to a stack pointer.
- **Timers.** PLCs support hardware timers that evaluate to a Boolean value. A timer starts

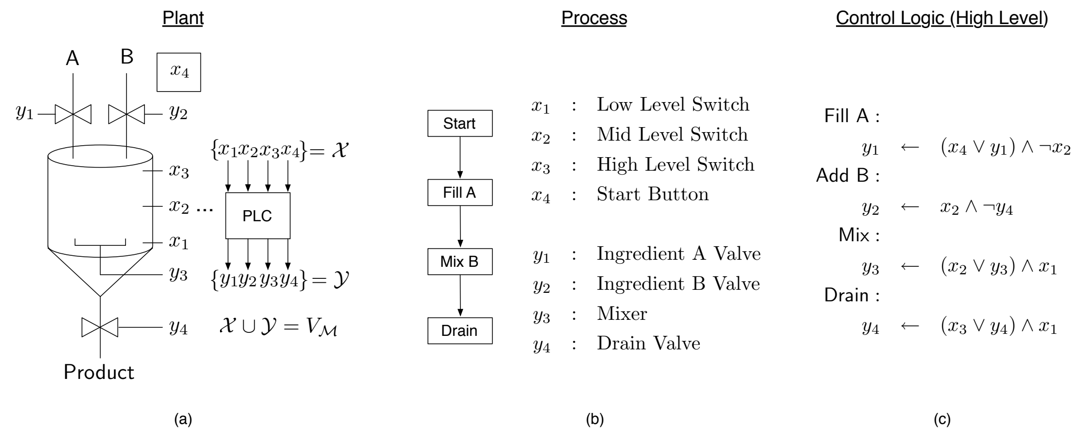

Plant
A
B
x_4
y_1
y_2
$\{x_1 x_2 x_3 x_4\} = \mathcal{X}$
x_3
x_2 ...
PLC
x_1
y_3
$\{y_1 y_2 y_3 y_4\} = \mathcal{Y}$
y_4
$\mathcal{X} \cup \mathcal{Y} = V_{\mathcal{M}}$
Product
(a)
Process
Start
Fill A
Mix B
Drain
x_1 : Low Level Switch
x_2 : Mid Level Switch
x_3 : High Level Switch
x_4 : Start Button
y_1 : Ingredient A Valve
y_2 : Ingredient B Valve
y_3 : Mixer
y_4 : Drain Valve
(b)
Control Logic (High Level)
Fill A :
$y_1 \leftarrow (x_4 \vee y_1) \wedge \neg x_2$
Add B :
$y_2 \leftarrow x_2 \wedge \neg y_4$
Mix :
$y_3 \leftarrow (x_2 \vee y_3) \wedge x_1$
Drain :
$y_4 \leftarrow (x_3 \vee y_4) \wedge x_1$
(c)

Figure 2.2. Example Sequential Control System
An example sequential control system.

when its input experiences an edge transition. Once the timer has reached a preset time, its own output goes from low to high.

- **Counters.** A counter is a value in PLC memory that is incremented each time a specified instruction causes a value to go from low to high. This is useful for counting events like the number of times an input wire receives a high signal.
- **Master Control Relays.** A Master Control Relay (MCR) defines a section of PLC code which behaves differently depending on the value of a specific input wire. If the MCR input is false, the code executes normally. If it is true, certain instructions will output a zero value. This is done to halt any machinery in case of an emergency condition.
- **Data Blocks.** PLCs retrieve configuration information about the physical process from blocks of persistent storage called *data blocks* (DBs). Each DB has a unique integer used to qualify any addresses to its data. A special kind of DB, called an *instance* DB is also used to pass parameters to function blocks.
- **Edge Detection.** Certain PLC instructions will only execute after a specific memory value goes from low to high. This requires the CPU to check for low to high transitions before any such instruction executes.

Given an understanding of the features provided by a PLC, we can move now describe how they are used in the implementation of full control systems.

2.2.2 Control System Implementations

An example control system for a simplified chemical mixer is shown in Figure 2.2. The plant (Figure 2.2(a)) is a single mixer with valves to dispense two ingredients, A and B, a mixing element, and a valve for draining the tank. The valves are controlled by the output variables y_1 and y_2 respectively, the mixer by y_3, and the drain by y_4. A device is ON when its corresponding output variable is set to $\top$ (true) and OFF when it is $\perp$ (false). Three level sensors are used to detect when the tank is at three levels–low, half full, and full (corresponding to inputs x_1, x_2, and x_3, respectively). A level switch $x_i = \top$ if the contents of the tank are at or above its level. A start signal is sent to the PLC via x_4.

The mixer follows a simple process in which ingredient A is added until the tank is half full, ingredient B is mixed with A until the tank is full, and the result is drained. The specification, shown in Figure 2.2(b), details the control system implementation:

1. Initially, all inputs and outputs are off until the Start button is pressed ($x_4 = \top$).
2. At this point the process enters state Fill A ($y_1 = \top$), and the tank is filled with ingredient A until the tank is filled midway ($x_2 = \top$), at which point the valve for A is closed ($y_1 = \perp$).
3. Next, the system transitions to the state Mix B, in which ingredient B is added (valve B is opened) until $x_3 = \top$. The mixer y_3 is also started in this state.
4. At this point, the system closes the value for B and enters the Drain state ($y_4 = \top$) until the tank is empty (detected by the "low" sensor $x_1 = \perp$). At this point, the mixer is stopped.

The mixing process is an example of a common class of control systems, called sequential control systems. Sequential control systems drive a physical plant through a process consisting of a sequence of discrete steps. Sequential control is used in industrial manufacturing (automotive assembly, QA testing), building automation (elevators, lighting), chemical processing (process control), energy delivery (power management), and transportation automation (railway switching, traffic lights), among others.

Not shown in the example, a *timer* is a special control system primitive that introduces a preset time delay between when an input becomes true, and when a subsequent output becomes true. A timer only sets the specified output to $\top$ when the input has been set to $\top$ for the duration of its preset delay value. Timers can be used to replace other sensors. For example, the level sensors above could be replaced with timers, presuming the flow and drain rates of the apparatus were known and fixed.

2.2.3 PLC Instruction Set Architectures

A logic program must be compiled to a PLC's native instruction set architecture (ISA) before being uploaded to the PLC. While ISAs vary between PLC vendors, many closely follow the IEC 61131-3 standard for the Instruction List (IL) programming language [61]. For our implementation of SABOT, we use one of the most popular commercial implementations of IL, the Siemens Statement List (STL) language, which is used on the S7 line of PLCs. Once the logic equations have been compiled into STL, they are then assembled into a vendor-specific bytecode (MC7 in the case of Siemens systems).

Instruction	Description
	Bit-logic
`AND x`	The current bit in the accumulator is logical ANDed with `x`.
`OR x`	The current bit in the accumulator is logical ORed with `x`.
`XOR x`	The current bit in the accumulator is logical XORed with `x`.
`NOT`	Negate the value in the accumulator.
`SET`	Set the accumulator to $\top$.
`CLR`	Set the accumulator to $\bot$.
	Nested-logic
`AND(`	Saves the accumulator and the `AND` function code to the stack.
`OR(`	Saves the accumulator and the `OR` function code to the stack.
`XOR(`	Saves the accumulator and the `XOR` function code to the stack.
`)`	Pops a previous accumulator value and a function code from the stack, and uses the specified function to combine the current accumulator with the one from the stack.
	Timers/Register Load
`L v`	Load `v` into the accumulator.
`SP x`	Start a timer that can be checked at address `x` using the time value in the accumulator.
	Output/Flip-Flip
`= x`	Assign the accumulator to the output or variable at address `x`.
`S x`	Assign $\top$ to `x` if the accumulator is $\top$.
`R x`	Assign $\bot$ to `x` if the accumulator is $\top$.

Table 2.1. Representative Boolean logic instructions from the Siemens statement list (STL) language.

STL contains a full set of instructions for numerical calculations, access to data storage, and Boolean logic among others. For the purposes of infecting sequential logic programs, we are interested specifically in STL's bit-logic subset, which is summarized in Table 2.1. For each instruction in the table, there is also a "not" version (e.g. `AN`, `ON`) which negates the variable input, as well as XOR versions of each instruction type. Note that use of instructions outside of the bit-logic subset in the victim logic program is still allowed as we only need analyze the sequential logic portions for locations in to which the rootkit hooks its malicious payload and mimicry routines. For example, in a hybrid control system, the conrol logic might first calculate the amount of time for a particular step to complete, and execute the step for the particular length of time before moving to the next step. We are not interested specifically in the process of calculating the timer value, but instead in tampering with the logical sequence that relies on the timer value.

The STL compilation of the example control logic is shown in Figure 2.2(d). Informally STL is an accumulator-based language in which each instruction has the form `OPCODE [OPERAND]`, and performs the specified operation on the operand and the current accumulator value. For the Boolean subset, the logic accumulator ϕ, starts with a value of $\top$ (true) at the beginning of each new logical branch. Thus, if the first instruction in a branch is `A x`, then $\phi = \top \ \wedge$ `x` after executing the instruction. There are also nesting logic instructions `A(`, `O(`, etc, and `)`, which are used to start a new logical branch in the circuit. This is done as follows. The instruction

`OPCODE(` first pushes (`OPCODE`, ϕ) onto the *nesting stack* ν. The accumulator is then reset to $\phi = \top$. Execution then continues until the next `)` instruction, at which point the (`OPCODE`,ϕ_1) are popped from ν, and $\phi \leftarrow \phi \circ \phi_1$, where $\circ$ is the Boolean operator corresponding to `OPCODE`. For a logic program to be well formed, no stores can occur inside nested logic, i.e., the nesting stack must be empty upon executing a `=` instruction. We only consider well-formed logic programs.

Additionally, STL also has instructions to simulate a flip-flop circuit element. The set instruction `S x` is equivalent to `x` $\leftarrow$ `x` $\vee \phi$, setting `x` to $\top$ only when the accumulator is $\top$. Similarly, the reset instruction `R x` is equivalent to `x` $\leftarrow$ `x` $\wedge \phi$, clearing `x` only when $\phi = \bot$.

Timer instructions are used to set the initial values of timers, start different types of timers, and to check if a timer has expired. The value of a timer can be set by using the normal `LD` instruction to load a numerical value in s or ms into the integer register. One of two instructions `SP` or `SF` can be used to start a timer either when its input becomes true or false respectively. The timer value can be checked as a regular Boolean variable, where the timer true until expiring.

STL also contains instructions for numerical calculations. While Sabot does not directly analyze the calculations, it does support the setpoint attack, which is concerned only with manipulating the calculated value immediately before its final assignment. Other instructions in STL are also used by Sabot to efficiently pack its own runtime data into bit vectors in memory.

2.3 Model Checking

Model checking is one of the two most common forms of program verification (with automated theorem proving being the other). The basic task in model checking is to take a model of a software or hardware system and verify that it will maintain one or more properties over either an infinite or bounded execution trace. Formally, this can be stated as checking the truth of the following proposition for a model $\mathcal{M}$ and property ϕ.

$$\mathcal{M} \models \phi \tag{2.1}$$

This can be read, "The property ϕ holds true under the model $\mathcal{M}$ for all initial state assignments of $\mathcal{M}$." More specifically, $\mathcal{M}$ is a Labeled Transition System (LTS) or Finite State Machine (FSM), and ϕ is a logical proposition. The connection between $\mathcal{M}$ and ϕ is the names used for state variables in $\mathcal{M}$, which are also used in ϕ to specify the properties that should hold over those state variables. Because model checking is most useful for checking that properties hold over certain periods of *time*, ϕ is stated using a *temporal* logic.

2.3.1 Computational Tree Logic

Computational Tree Logic (CTL) is a branching time logic used for reasoning about properties over all or some branches of a program's execution [62]. Model checkers using CTL as a specification language have been used extensively in proving safety and liveliness properties of control systems [38, 63, 41]. CTL itself is insufficient to fully specify a property for verification.

$$\begin{aligned}\phi ::= &\top \mid \bot \mid p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \\ &\mathrm{AX}\phi \mid \mathrm{AF}\phi \mid \mathrm{AG}\phi \mid \mathrm{EX}\phi \mid \mathrm{EF}\phi \mid \mathrm{EG}\phi \mid \\ &\mathrm{A}[\phi \text{ U } \phi] \mid \mathrm{E}[\phi \text{ U } \phi]\end{aligned}$$

Figure 2.3. Syntax of Computational Tree Logic.

First, verification properties (propositions) must be stated as atomic formulas in a set Atoms. For our purposes, we let Atoms be a set of Boolean formulas. CTL then provides the temporal connectives necessary to describe when the formulas from Atoms must hold.

For example, consider that we wish to verify a control system (modeled by $\mathcal{M}$) in which two mechanical arms can potentially occupy the same space. Let the variables arm_1, and arm_2 denote whether each arm is in the space. Thus, $arm_i = true$ means that arm_i is in the space. We may wish to verify two properties: *i. Mutual Exclusion* - only one arm may occupy the space at the time, and *ii. Nonblocking* - one arm cannot occupy the space indefinitely. Property i can be stated strictly in Boolean logic, specifically: $\neg arm_1 \vee \neg arm_2$. Property ii on the other hand cannot be stated in Boolean logic. Instead, we can state it in CTL with $\mathsf{Atoms} = \{arm_1, arm_2, \neg arm_1, \neg arm_2\}$. Specifically:

$$arm_1 \Rightarrow \mathrm{AF}\neg arm_1 \wedge arm_2 \Rightarrow \mathrm{AF}\neg arm_2 \tag{2.2}$$

This can be read as "If $arm_1 = \top$, then *A*lways execution will *F*inally reach a program state in which $\neg arm_1 = \top$, and the same for arm_2."

The syntax of a CTL formula is as follows, where $p \in \mathsf{Atoms}$.
Intuitively, a formula AXϕ (*A*lways ne*X*t ϕ) is true iff ϕ holds on the next state of the program for all computational branches from the current state. Similarly, EXϕ (*E*xists ne*X*t ϕ) is true iff ϕ holds on the next state of some computational branch of the program. AFϕ (*A*lways *F*inally ϕ) and EFϕ (*E*xists *F*inally ϕ) are true iff ϕ eventually holds for one state on all branches and some branch respectively. AGϕ (*A*lways *G*lobal ϕ) and EGϕ (*E*xists *G*lobal ϕ) are true iff ϕ holds for all states on all branches and some branches respectively. Finally, A[ϕUψ] (*A*lways [ϕ *U*ntil ψ]) is true iff on all branches, ϕ holds at least until ψ holds after some finite number of steps. Similarly, E[ϕUψ] (*E*xists [ϕ *U*ntil ψ]) is true if ϕ holds until ψ holds on at least one branch from the current state.

As an example, in a control system in which a stop button named *stop* cuts the power to the motor named *mot*, the CTL property AG(*stop* $\Rightarrow$ AX$\neg$*mot*) holds. Additionally, the property EG(*mot* $\Rightarrow$ A[*mot* U *stop*]) also holds.

The exact definition of $\mathcal{M} \models \phi$ can be defined inductively over ϕ. If $\mathcal{M}$ is a Kripke Structure, then the temporal connectives in ϕ can be defined in terms of traversals of the unwinding of $\mathcal{M}$. This inductive definition is as follows:

$$\begin{aligned}
\mathcal{M}, s &\models \top \\
\mathcal{M}, s &\not\models \bot \\
\mathcal{M}, s \models \neg\phi &\quad\Leftrightarrow\quad \mathcal{M}, s \not\models \phi \\
\mathcal{M}, s \models \phi_1 \wedge \phi_2 &\quad\Leftrightarrow\quad \mathcal{M}, s \models \phi_i \wedge \mathcal{M}, s \models \phi_2 \\
\mathcal{M}, s \models \phi_1 \vee \phi_2 &\quad\Leftrightarrow\quad \mathcal{M}, s \models \phi_i \vee \mathcal{M}, s \models \phi_2 \\
\mathcal{M}, s \models \phi_1 \Rightarrow \phi_2 &\quad\Leftrightarrow\quad \mathcal{M}, s \models \neg\phi_1 \vee \phi_2 \\
\mathcal{M}, s \models \phi_1 \Leftrightarrow \phi_2 &\quad\Leftrightarrow\quad \mathcal{M}, s \models \phi_1 \wedge \phi_2 \vee \mathcal{M}, s \models \neg\phi_1 \wedge \neg\phi_2 \\
\mathcal{M}, s \models \mathrm{AX}\phi &\quad\Leftrightarrow\quad \forall s \rightarrow s_x, \mathcal{M}, s_x \models \phi \\
\mathcal{M}, s \models \mathrm{EX}\phi &\quad\Leftrightarrow\quad \exists s \rightarrow s_x, \mathcal{M}, x_x \models \phi \\
\mathcal{M}, s_1 \models \mathrm{AF}\phi &\quad\Leftrightarrow\quad \forall s_1 \rightarrow s_2 \rightarrow \ldots, \exists i \ s.t.\mathcal{M}, s_i \models \phi \\
\mathcal{M}, s_1 \models \mathrm{EF}\phi &\quad\Leftrightarrow\quad \exists s_1 \rightarrow s_2 \rightarrow \ldots, \exists i \ s.t.\mathcal{M}, s_i \models \phi \\
\mathcal{M}, s_1 \models \mathrm{AG}\phi &\quad\Leftrightarrow\quad \forall s_1 \rightsquigarrow s_n, \mathcal{M}, s_n \models \phi \\
\mathcal{M}, s_1 \models \mathrm{EG}\phi &\quad\Leftrightarrow\quad \exists s_1 \rightsquigarrow s_n, \mathcal{M}, s_n \models \phi \\
\mathcal{M}, s_1 \models \mathrm{A}[\phi_1 \mathrm{U} \phi_2] &\quad\Leftrightarrow\quad \forall s_1 \rightarrow s_2 \rightarrow \ldots, \exists i s.t.\mathcal{M}, s_i \models \phi_2 \wedge \forall j < i \mathcal{M}, s_j \models \phi_1 \\
\mathcal{M}, s_1 \models \mathrm{E}[\phi_1 \mathrm{U} \phi_2] &\quad\Leftrightarrow\quad \exists s_1 \rightarrow s_2 \rightarrow \ldots, \exists i s.t.\mathcal{M}, s_i \models \phi_2 \wedge \forall j < i \mathcal{M}, s_j \models \phi_1
\end{aligned}$$

Here, the notation $\mathcal{M}, s \models \phi$ means that starting from the state s in the state space of $\mathcal{M}$, the property ϕ holds. In the case of non-temporal connectives, e.g. $\wedge$, only the current state s is considered. In the case of temporal connectives, e.g., AX, the graph notations $s_1 \rightarrow s_2$ and $s_1 \rightsquigarrow s_n$ are used to denote that s_1 is a predecessor to s_2 and is contained on a path to s_n respectively in the state space of $\mathcal{M}$. For example, the induction rule for $\mathcal{M}, s_1 \models \mathrm{AF}\phi$ can be interpreted as "On all paths starting at s_1, there exists a node s_i such that phi holds at state s_i in $\mathcal{M}$.

2.3.2 The NuSMV 2 Model Checker

In practice, model checking software is chosen based on a particular set of verification requirements. Some model checkers can check whether two programs are equivalent, e.g., under bisimulation. Others perform better for systems with large numbers of variables that can have the state space explosion problem. Others can check programs written in traditional programming languages such as C. And finally, most model checkers have their own unique input language designed to facilitate the user in easily and correctly encoding the state transition system that defines the model $\mathcal{M}$.

In this work, we use the NuSMV 2 model checker [64]. NuSMV was chosen for its maturity and reliability, as well as its prior use in sequential control systems [38]. NuSMV, which is the second generation of the SMV model checker uses an input language that allows the system of Boolean equations resulting from the decompilation of a PLC program to be directly encoded into the model. Furthermore, the properties that we wish to verify, e.g., ϕ, can be stated in CTL as well as several other specification input languages that NuSMV understands. As CTL has already been covered, in the remainder of this section, we explore the basic elements of the NuSMV model input language, and defer further details to the full documentation [65].

Recall that a model $\mathcal{M}$ is a Kripke Structure. Formally, a Kripke Structure is a 5-tuple: $\langle S, I, AP, \rightarrow, L\rangle$. Here, S is the set of states, I is a set of initial states, AP is a set of Atomic Propositions (see Section 2.3.1), $\rightarrow$ is a transition relation such that $\rightarrow \subseteq S \times S$, and L is a state labeling function with $L : S \rightarrow 2^{AP}$, where 2^{AP} is the power set of AP. Of course, for complex systems, it is impractical to directly encode the model as a Kripke Structure. Instead, the model is stated implicitly in a special modeling language provided by the model checker. The NuSMV language supports most of the typical datatypes found in traditional programming languages as well as typical control structures. However, unlike a functional or strictly procedural language the NuSMV modeling language describes state transitions of an FSM.

Specifically, the state is defined as the state of individual variables of well defined types. These types include `Boolean`, `Integer`, `Enum`, `Word`, `Word`, `Array`, and `Set`. For the purposes of modeling a sequential control system, only the `Boolean` type is necessary for tracking the ON/OFF state of each device. Transitions between states in the FSM are defined using one of two methods. The first is to define the transitions explicitly using the `ASSIGN` keyword followed by the `init(`*var*`)` and `next(`*var*`)` keywords used to define the initial and next states of *var* respectively. (This is the method used in this work.) The alternative to defining explicit transitions is to use an implicit constraint style of programming in which the next state is determined by solving a system of constraints on the state variables under the current state.

NuSMV allows variables to be initialized or updated with nondeterministic values. For example, consider a Boolean variable x declared as: `VAR` x `: boolean`. Typically, an initialization of this variable would look like: `ASSIGN init(`x`) := TRUE;`. However, what if we need to consider a system in which the initial state is not known? In this case, it can be initialized as follows: `ASSIGN init(`x`) :=` {`TRUE, FALSE`}. This will cause the model checker to check both alternative initial assignments. The same nondeterministic assignment can be used for the `next(`$\cdot$`)` keyword. The nondeterministic assignments can also be used with control flow structures. For example, NuSMV has a ternary operator *cond* `?` e_1 `:` e_2 where e_1 is evaluated if $cond = \top$ ad e_2 is evaluated if $cond = \bot$. In this case, both e_1 and e_2 may contain nondeterministic values.

A final point of the NuSMV modeling language that we will cover is the `FAIRNESS` constraint. The `FAIRNESS` constraint can be used to limit the branches branches explored in the FSM. The syntax of the `FAIRNESS` constraint is: `FAIRNESS` e, where e is a Boolean expression. Under this constraint, the model checker will only evaluate executions paths in which e is true infinitely often. This concludes the NuSMV modeling language features used in this work. Exhaustive documentation can be found in [65].

2.3.3 Symbolic Execution

A prerequisite step to model checking used in several of the research contributions below is *symbolic execution*. A symbolic execution of a program is an execution in which some or all variables do not contain concrete values. Instead, they contain a symbolic representation of a value, e.g., "$x + 7$". Symbolic execution is done starting from some specified point in a program, usually the beginning or a place where user input is received, and then terminates at a specified point. Upon termination, each symbolic variable will contain an expression for its current value. The advantage of symbolic execution is that this expression represents *all possible values* that could be placed into that variable. Thus, we can reason about what values that variable may take on, or what must have happened earlier in the program to give it a particular value.

As an example of a symbolic execution, consider the following program segment:

```
x_2 = True
a = x_2 || y_1
b = x_1 && a
```

A symbolic execution of this program would proceed as follows. First, `x_2` would be a concrete variable, because it is assigned the value `True`. Next, the variable `y_1` will be symbolic, because it receives no concrete value. The variable `a` will be symbolic, because it is assigned the value of an expression containing at least one symbolic variable (namely `y_1`). Similarly, `b` is a symbolic variable for the same reason. Upon termination, `b` will have the symbolic value `(True || y_1) && a` which is the same as `a`.

This is a very simple example, and does not account for a number of challenges. Two of the most important challenges in symbolic execution arise from the use of loops and pointers. When the symbolic execution reaches a branch in the program, e.g., an conditional *if* statement or loop condition, it must decide whether to follow the branch. Given the possibility of infinite loops, the symbolic execution engine must decide for how many iterations to follow the loop. Often the use of a Satisfiability Modulo Theory (SMT) solver can be used to test for loop termination. However, as will be seen in Chapter 6, the operational constraints of control systems can simplify this problem. Pointers pose an additional problem, as they can result in program jumps to arbitrary locations or inclusions of unknown symbolic values in expressions. Once again, SMT or SAT solvers are often used to try and address this problem, but there are control-system specific features that make it more tractable for us.

Chapter 3

Multi-Vendor Penetration Testing

Traditionally, one wishing to attack a system must first study an instance of the system to understand potential points of vulnerability. In this section, we review our methodology for structured penetration testing of smart metering systems. As opposed to immediately attempting to reason about attacking a specific instance of the system, we instead focus first on devising attack strategies against the general architecture to which that system belongs. For example, in this section, we consider the general architecture of the Advanced Metering Infrastructure (AMI). In contrast to the sections to follow, here we consider a high level way of *manually* searching for *vulnerabilities*. In the latter sections we describe means for *automatically* constructing *payloads*. These two are complementary as the exploitation of a vulnerability is a necessary prerequisite to the execution of a payload.

3.1 Methodology

An attack tree is a structure for enumerating the kinds of attacks that achieve a particular adversarial goal [24]. It does this by recursively breaking down a goal into finer- and finer-grained subgoals and finally to a set of attacks that achieve the original goal. An example attack tree that formed the genesis of this work [12] is shown in Figure 3.1. The root specifies the end goal, committing energy fraud by forging the energy usage information reported to the utility. The internal nodes (those with parents and children) describe the different combinations of conditions that must be met to commit fraud. Finally, the leaves of the tree are the attacks necessary for energy fraud. The final attribute of the tree is the conjunctions (AND/OR) between each layer of child nodes. These specify whether all or just one of the child branches must be followed to reach the goal in the parent node.

What we notice about this example is that the attacks at the leaves of the tree are fairly general, and seem applicable to most smart metering systems. This suggests that this type of tree is a widely applicable tool. However, because it lacks details about any specific system,

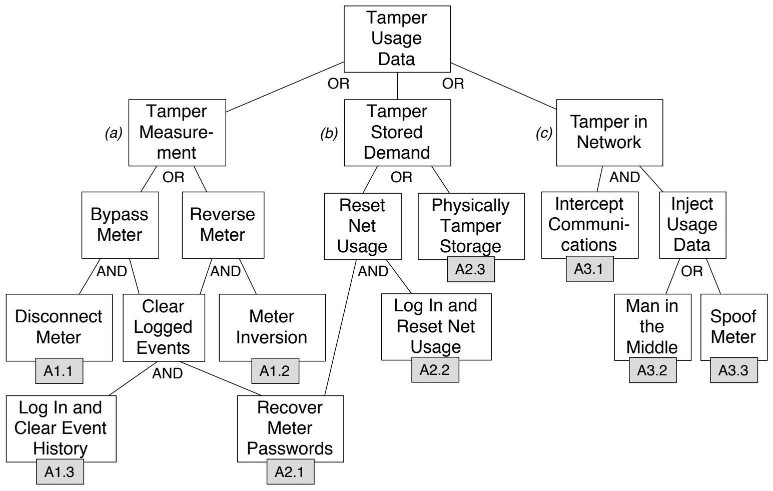


Figure 3.1. Example energy fraud attack tree. The three subgoals beneath the root are labeled as (*a*), (*b*), and (*c*) for reference purposes.

its usefulness is limited in finding concrete vulnerabilities. Because we are pen-testing multiple commercially available metering systems, we will want to further specify the details of each attack in this generic tree. Thus, as we learn about the individual systems, we extend this generic tree with vendor-specific attack strategies. These ideas can be refined into two types of attack trees: *archetypal* and *concrete*.

The process of grafting a concrete tree to an archetypal tree is shown in Figure 3.2. For a given adversarial goal, one may define an *archetypal tree* that enumerates strategies for reaching the goal against any system of a given architecture. In the case of the example above, the goal is forged energy demand and the architecture is smart metering. Each leaf of an archetypal tree is an *archetypal attack*. A concrete tree then refines an archetypal attack with respect to a specific vendor's system. The subgoals in the concrete tree sensitive to the security mechanisms present in the system, and thus define the exact conditions under which the root goal can be achieved. The leaves of the concrete tree are the *concrete attacks* which ultimately allow an adversarial goal to be achieved. For the purposes of our study, we use penetration testing to determine the feasibility of each concrete attack.

Our method is similar to that originally used for attack patterns [66, 67]. An attack pattern is a parameterized description of an attack, e.g. an injection attack, that is generic until its parameters are instantiated. Attack patterns may be described in terms of attack trees. When considering a particular attack against a particular *instance* of a system, e.g. a company's network, its parameters are instantiated with the specific details of that system. The concept of attack trees is based on that of fault trees, which were originally use to model the dependencies between potential faults in aviation and nuclear power systems [23, 68].

Attack trees by themselves are useful as a guide for penetration testing. However, once the knowledge of system interfaces has been exhausted and the concrete attacks are developed, we resort to standard pen-testing techniques such as reverse engineering [69], fuzz testing [70], and

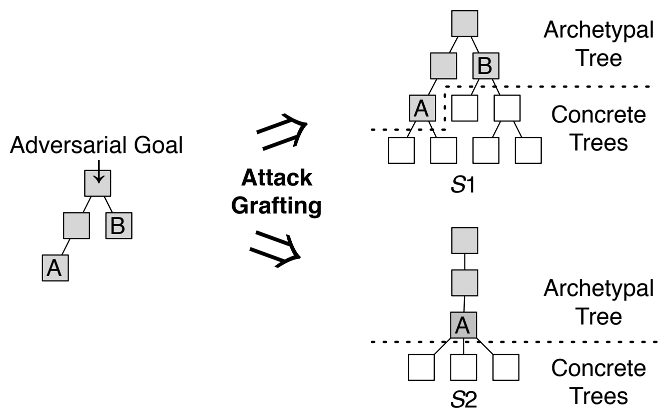


Figure 3.2. Grafting concrete trees for two different systems ($S1$ and $S2$) onto an archetypal attack tree for a specific adversarial goal.

the construction of custom attack tools. For example, we later examine an energy fraud attack based on a meter spoof program written in Python.

Documented throughout, our methodology for directing penetration testing includes:

1. **Capture architectural description:** Elicit the features of a general architecture for target domain (see Section 2.1).
2. **Construct archetypal tree:** Given the architectural description, design a comprehensive archetypal tree for each adversarial goal (see Section 3.2).
3. **Capture vendor-specific description:** Identify the security mechanisms present the Systems Under Test (SUTs) that may thwart a given archetypal attack (see Section 3.3).
4. **Construct concrete trees:** Graft the vendor-specific goals to an archetypal goal to form concrete trees (see Section 3.4).
5. **Perform Penetration Testing:** Attempt to achieve the concrete goals by performing penetration testing on the SUT (see Section 3.5).

3.2 Archetypal Attack Trees

Having reviewed the general architecture of smart metering systems, we may now construct archetypal trees that describe attacks in a broad sense that is applicable to any system within the architecture. An archetypal tree is an attack tree that is general enough to be applicable to all systems of a given architecture. As with a regular attack tree, the root of an archetypal tree is a single adversarial goal. This goal is repeatedly broken down into subgoals that describe the individual conditions that must exist to reach the root goal. Unlike a regular attack tree, the leaf nodes of the archetypal tree are not targeted at a specific system. Instead, the leaves constitute the points to which concrete trees are grafted. It is thus critical that they be selected to clearly define the boundary between broad architectural goals and vendor-specific goals. While

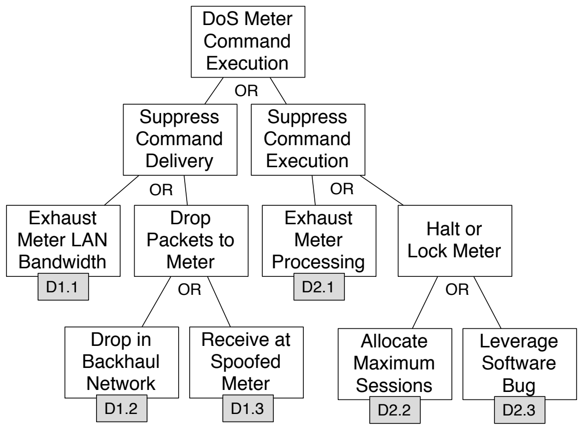


Figure 3.3. Archetypal tree for Denial of Service.

this is somewhat of an art rather than a science, we have devised a set of criteria to aid us in differentiating between archetypal and concrete goals. If any of the following are true of a goal during the construction of an archetypal tree, then it becomes a leaf node, to which a concrete tree can be grafted.

1. *The goal targets a component whose implementation is vendor-specific.* An example of such a component is the meter LAN. While an archetypal tree can prescribe an attack on a meter LAN, the attack can not be specific to any particular LAN media.
2. *The goal may be hindered by the presence of a vendor-specific protection mechanism.* The addition of any subgoals for circumventing vendor-specific protection mechanisms is by definition not archetypal. Such details must be described in the concrete tree. An example of this can be seen in the following section on energy fraud (Section 3.2.1), where nothing general is known about the protection mechanisms present at the collector's link to the backhaul network.

If a subgoal does not meet these conditions, it is broken down. In the following sections, we provide justification for extending or terminating a given subgoal where instructive.

3.2.1 Energy Fraud

For our initial pen-testing efforts [12], we constructed an archetypal tree for energy fraud (shown in Figure 3.1). It is described here so that it may be instantiated later. We define energy fraud as any tampering with the metering infrastructure that leads to a customer not being billed for some energy consumed. (Note that in this particular archetypal tree, we do not consider using energy fraud to artificially inflate a victim's bill.) In AMI, fraud may be committed in the field by modifying the recorded energy usage before it is read by the utility. Known methods for fraud in electromechanical meters include interfering with the meter's sensors using magnets and rewinding usage gauges by inverting the meter in the socket (thereby reversing current flow through the meter).

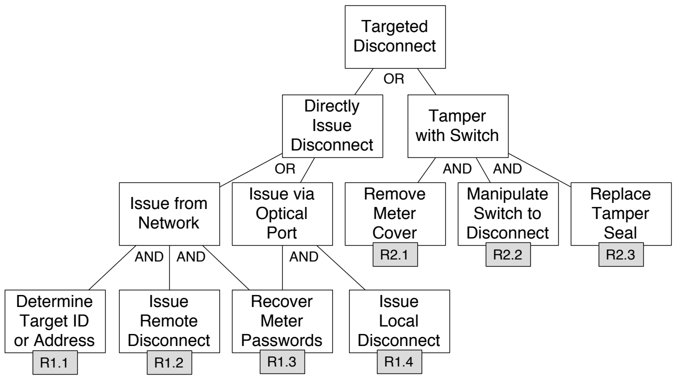


Figure 3.4. Archetypal tree for targeted disconnect.

Smart meters, present new opportunities for tampering with usage data. As shown in the first level of subgoals in the example tree, this can be done in three places (a) in the meter's low-level components, (b) the meter's long-term storage, and (c) in transmission to the utility. The archetypal attacks in this tree, as in the others, are labeled as $TX.Y$, where T is a letter specific to the tree, X is the index of the subtree below the root to which the attack belongs, and Y is the index of the attack within that subtree. Starting with the physical attacks in subtree a, there are two means to interrupt a smart meter's physical measurement of usage. A1.1 simply requires that the meter is removed from the path of current flow, and A1.2 that it be reversed in its socket. As described in section 2.1.1, virtually all smart meters will log and report both of these events (power cycle and reverse energy flow respectively). Thus, in the archetypal level, we already recognize that the log messages will need to be cleared of these events. As a final note on physical attacks, because obtaining physical access to the meter is specific to a *particular installation*, we do not consider this prerequisite in either the archetypal or concrete trees. This does not matter for the case of fraud because it is assumed that the adversary already has access to her own meter.

Modifying logs and usage in meter storage is the goal of subtree b. This can be achieved in one of two ways. Either the meter's administrator password can be obtained and used to clear the log files: A2.1 AND A2.2, or the physical storage device may be tampered without interfering with the meter. As this is an archetypal tree, the implementation of the storage is left unmentioned.

The strategies for forging usage data on the wire are shown in subtree c. The interception of network communications is assumed to be necessary both for the purposes of understanding the meter's protocol stack, assuming it is non-standard, and for interposing one's self in the communication path with the utility. In the archetypal tree, we ignore over which network (meter LAN or backhaul) the interception occurs, as well as any potential protection mechanisms. Along with A3.1, the adversary must either hijack a session between the meter and utility (A3.2) or impersonate a meter for the entire session (A3.3).

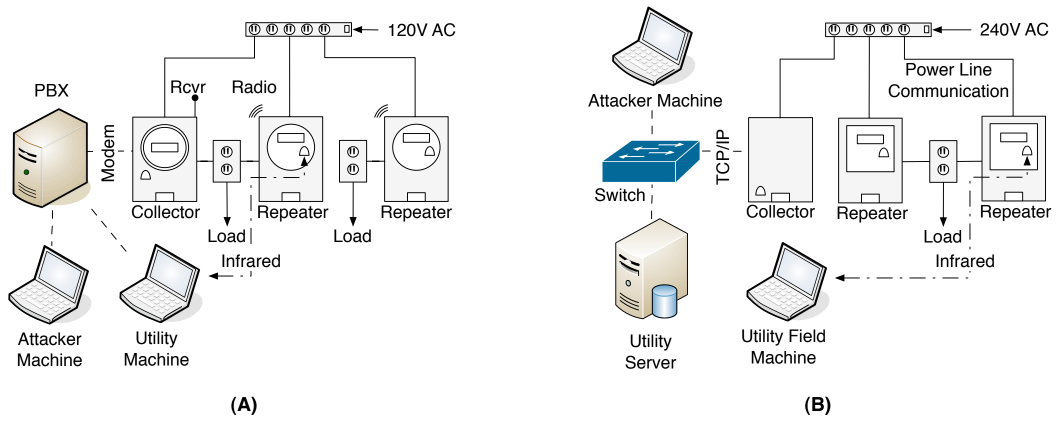


Figure 3.5. The two SUTs used in our experiments. In $S1$ (A), the collector also functions as a meter, and relays data from a wireless mesh LAN to a telephone network backhaul. $S2$ (B) uses a dedicated device as a collector to relay data between a PLC network and an Internet connection to the utility.

3.2.2 Denial of Service

This section considers DoS attacks that prevent meters from acting on commands such as usage queries, firmware upgrades, and remote disconnects. This is a realistic adversary goal. For example, if the retrieval of meter log files can be prevented for a sufficient period of time, a suspicious event such as a meter power cycle can be erased when the logs roll over with benign events.

The archetypal tree for meter DoS against meter command execution is shown in Figure 3.3. The adversary has two choices for a general strategy, either prevent the command from reaching the meter, or prevent its execution on the meter. The former can be achieved either through network resource exhaustion, or by tampering with the routing of packets away from the meter. As the LAN media is system specific, we do not break this subgoal down any further in the archetypal tree. A potentially more practical strategy is to drop traffic destined for the meter. This may either be done at a link or routing layer (D1.2) or at the transmission layer (D1.3). The latter seems like the more reasonable method, as dropping a packet at an intermediate hop will result in a retransmission by a higher layer.

The second strategy for command DoS prevents the meter from executing a command once it is received. An extremely simplistic method for doing this is to exhaust the meter's input processing capability (D2.1). This could be done either from the backhaul network or meter LAN. While effective, this type of attack is not covert, and cannot guarantee the command will fail. A more failsafe approach would be to put the meter into an unresponsive state. This may be done through interactions that exhaust a particular system resource, e.g. allocating and maintaining the maximum allowed number of open connections (D2.2), or by leveraging a firmware bug causing a system hang (D2.3).

3.2.3 Targeted Disconnect of Electrical Service

Most meter vendors include remote disconnect functionality in their meters. The ability to disconnect a target's power can cause at best, inconvenience and in worse scenarios, financial or physical harm depending on the setting. As described earlier, remote disconnect systems consist of a physical switch that breaks the current flowing to the house, and a set of remote commands to operate this switch. The archetypal tree for this attack is shown in Figure 3.4.

The ideal case for an adversary would be to issue the disconnect command remotely. Doing this requires at least that the ID be known for the target device (R1.1), and that its administrator password has been recovered (R1.3). Notice that this is the second archetypal tree with a leaf node requiring meter passwords to be recovered. This illustrates a secondary usefulness of attack trees: they act as a reference for quickly mapping security flaws to the adversarial goals they enable.

We reason that the disconnect functionality will be accessible through the optical ports on most systems because optical port functionality needs to contain at least the network functionality to allow the meter to function in the event that it is not network accessible, e.g. the meter's network card is malfunctioning. This is the basis of archetypal attacks R1.3 and R1.4.

Finally, physical access to a meter may also be useful for manipulating the disconnect switch, be it by mechanical or electrical means (R2.2). From experience, we have found that virtually all smart meters use the same tamper seal [71]. We have contacted the manufacturer of these seals and confirmed that there are no limitations on the text which we could have embossed on the flag.

3.3 Systems Under Test

This section details the two Systems Under Test (SUTs) that have been the subject of our penetration testing[1]. We will denote the two systems as $S1$ and $S2$. Besides the meters themselves, this section covers the additional components needed to run utility-end software and to network meters with the utilities. In describing the two systems, we will refer to the *utility machine* or *utility server* to mean a Microsoft Windows-based PC or laptop computer running software for meter management. We found that Windows by far the most common choice of utility-end operating system across vendors. The *attacker machine* is used to represent our machine used for various pen-testing purposes. In practice, this could be any machine within network reachability of a meter that is controlled by an adversary.

The general environment for both systems is identical. Both SUTs consist of several repeaters and a single collector, the main difference being that in $S2$, the collector does not function as a meter itself. We constructed sockets to allow the meters in our lab to function using wall socket power. The meters in $S1$ are able to run on 120V AC at 60 Hz, while the meters in

[1]We do not reveal vendor identities here, as we are already in contact with them, and both SUTs are already deployed in the US and Europe.

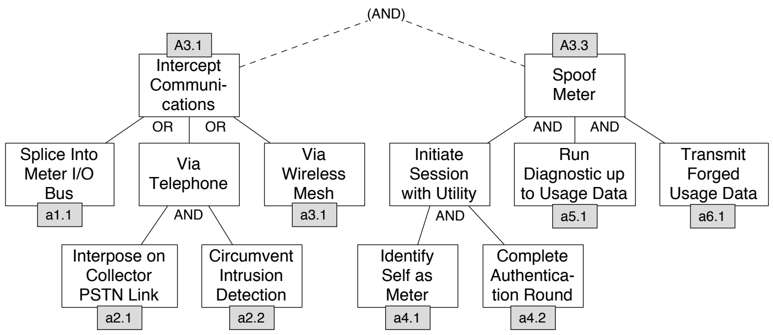

(AND)
A3.1
Intercept Communi-cations
OR
OR
Splice Into Meter I/O Bus
a1.1
Via Telephone
AND
Interpose on Collector PSTN Link
a2.1
Circumvent Intrusion Detection
a2.2
Via Wireless Mesh
a3.1
A3.3
Spoof Meter
AND
AND
Initiate Session with Utility
AND
Identify Self as Meter
a4.1
Complete Authentica-tion Round
a4.2
Run Diagnostic up to Usage Data
a5.1
Transmit Forged Usage Data
a6.1

Figure 3.6. The concrete trees for energy fraud in $S1$.

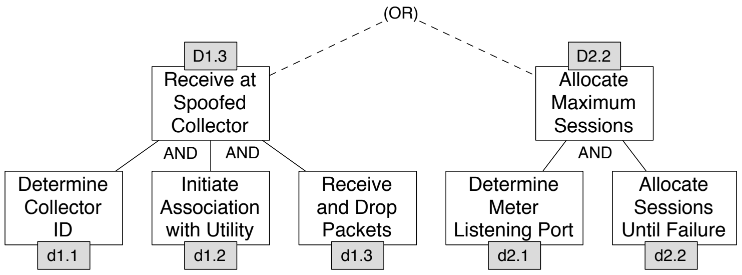

(OR)
D1.3
Receive at Spoofed Collector
AND
AND
Determine Collector ID
d1.1
Initiate Association with Utility
d1.2
Receive and Drop Packets
d1.3
D2.2
Allocate Maximum Sessions
AND
Determine Meter Listening Port
d2.1
Allocate Sessions Until Failure
d2.2

Figure 3.7. The concrete trees for DOS in $S2$.

$S2$ require a 240V step up transformer. A simple load was exerted by a small synchronous motor and measured to check the proper installation of each meter.

3.3.1 S1 Specifics

An overview of $S1$ is given in Figure 3.5.A. In $S1$, utilities communicate with meters via Public Switched Telephone Service. For obvious security reasons, we were unable to directly connect our collector to the telephone network. Instead, an Asterisk [72] based private branch exchange (PBX) on an x86 Linux machine provided call routing between the collector and utility machine. The PBX routes calls according to a table called the *dial plan*. The attacker machine sits on the PBX along with the meter and utility machine. Calls to the meter can be routed to the attacker machine by modifying the PBX dial plan. The ability to perform such rerouting using a commodity system was instrumental in our instantiation of the energy fraud attack for $S1$.

For all communication, the utility machine initiates communication with collector meters, with the exception of alarm conditions such as outage management or potential intrusions, in which case the meter preemptively contacts the utility. We augmented the utility machine with a modem monitor for analyzing the telephone protocol. What we quickly found is that it largely conforms to the ANSI C12.21 standard for telephone modem communication with meters. After this, the monitor was only needed to understand the occasional deviations from the standard.

The PSTN backhaul link at the collector is guarded by an “intrusion detection” mechanism. The purpose of this mechanism is to prevent both active and passive attacks from telephony devices connected on the same link as the collector, i.e. via a line splitter. The intrusion detection mechanism will immediately terminate a call from the utility if another device on the line goes off the hook. When a device goes off the hook, it receives a dial tone and voltage via an onboard component called the Foreign Exchange Office (FXO). All endpoint devices in a telephone network use an FXO. The dial tone and voltage are supplied from the other end of the line by the Foreign Exchange Service (FXS), usually implemented by the phone company. Because the meter can detect when another device is receiving a voltage and dial tone, it can terminate its current call.

The main operation of concern in $S1$ is the diagnostic protocol between the meter and utility. This protocol can perform many functions from a simple meter reading, to a full check of every parameter set in the meter. For the purposes of energy fraud we are mainly concerned with how this protocol performs energy usage readings. The utility initiates a diagnostic by calling a collector, resulting in the collector responding with an identification message. An authentication round is then carried out according to the default scheme specified by ANSI C12.21 (ANSI X3.92-198) [32]. If authentication is successful, the utility will probe the meter for some variable number of parameters, after which the current net usages are read. This is the point in the protocol where a usage forgery must occur. The remainder of the protocol consists of potentially more parameter queries, and finally a goodbye message. The meter LAN, a wireless mesh operating in the 900 MHz band, is currently under evaluation.

3.3.2 S2 Specifics

Our testbed for $S2$ is shown in Figure 3.5.B. The main differences from $S1$ are the backhaul and meter LAN protocols, and the collector, which does not function as a meter in $S2$. Upon initial inspection, one notices that $S2$ is more accessible to remote attacks due to the use of an Internet-based backhaul. This fact becomes useful when instantiating a concrete tree for DoS against meter command execution. The meter LAN uses a proprietary protocol that requires special equipment to analyze.

Though the application layer protocol between the utility and collector is proprietary, two thing are clear from initial inspection. First, an initial association between the two is started by the collector, and each subsequent command execution is started by the utility. This suggests that both directions should be considered when designing a concrete DoS attack. Second, in the initial association, the collector transmits its unique ID number and associated network address in the clear to the utility. Thus, knowing this ID for a target collector may be useful in a DoS attack.

3.4 Concrete Attack Trees

Concrete attack trees function as a guide for penetration testing a specific system. As with the archetypal trees, we use basic guidelines to determine when a concrete tree is specific enough.

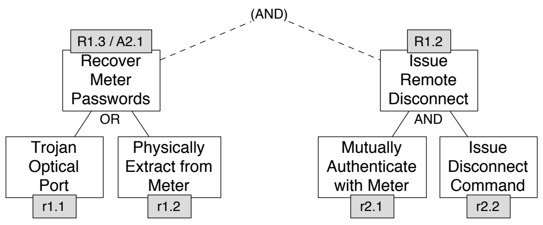


Figure 3.8. The concrete trees for targeted disconnect $S1$.

Any details not elaborated in the concrete tree must either already be known about the system, or must be discovered during pen-testing. In constructing the concrete trees for fraud, DoS, and targeted disconnect, we use the following two rules:

1. *A goal should be a leaf if it is achievable completely by known means in the system.* This is the simplest case as no additional pen-testing is required. Several leaves in the concrete DoS tree are of this type.
2. *A goal should be a leaf if no vulnerability is yet known that would allow it to be executed.* At this point, determining the existence of a vulnerability enabling the goal becomes the job of penetration testing.

We instantiate concrete trees for the three adversarial goals for $S1$ and $S2$ below. The root of each concrete tree shares a reference number with a leaf in one or more archetypal attack trees to which it may be grafted. We instantiate fraud and targeted disconnect for $S1$, and DoS for $S2$[2].

3.4.1 Energy Fraud in S1

The archetypal attack tree for energy fraud presented three broad strategies: tampering with the measurement process, tampering with the recorded usage in meter storage, and tampering with the usage data in transmission. For our first attempt to implement a fraud attack in $S1$, we chose the third strategy because of its relatively low invasiveness and our understanding of the backhaul network operation. This strategy terminated in three archetypal attacks: a mandatory requirement of being interposed on the backhaul link (A3.1), and the option of either performing a man in the middle attack (A3.2) or meter spoofing (A3.3). After evaluating the ANSI C12.21 specification via a trace of $S1$'s telephony-based diagnostic protocol, we determined that meter spoofing was more straightforward. Thus, to complete the goal of fraud in $S1$, we must instantiate and execute concrete trees for archetypal attacks A1.1 and A1.3. Both concrete trees are shown in Figure 3.6.

Archetypal attack A3.1 requires that the adversary be interposed somewhere on the path between the meter's networking interface card (NIC) and the utility. In one extreme end, this may be achieved by directly tampering with the communications bus on which the NIC resides

[2]While there are a large number of attempted attacks, we find the ones described here to be the most instructive.

(a1.1). Two more likely places are the mesh network (a3.1), and the telephone backhaul (a2.1). For the latter, the additional prerequisite of bypassing the "intrusion detection" mechanism is necessary (a2.2).

The second archetypal attack for energy fraud requires meter spoofing. This calls for three steps to successfully deliver forged usage data as part of $S1$'s diagnostic protocol. First, the spoofing device must initiate a new diagnostic session with the utility. This will require first identifying itself as the expected meter (a4.1), and second, completing the authentication round (a4.2). Once the session is established, the spoofing device must answer all diagnostic queries up to the forged demand (a5.1), and finally, insert the forged demand value (a6.1). The remainder of work to realize these attacks is achieved by pen-testing as described in section 3.5.

3.4.2 Denial of Service in S2

Unlike the two concrete trees for energy fraud, the root nodes of the two for DoS are combined by disjunction in the archetypal tree. Thus, fulfilling the requirements of either tree is sufficient for achieving denial of command execution. Recall that there are two options because communication in $S2$ may be initiated by both the collector and the utility at different points in time. The first tree (D1.3) requires another device to spoof the collector node in order to receive any commands destined for meters and drop them en route. This requires first the necessary reconnaissance to determine the collectors network ID (d1.1), and to establish a new session with the utility using that ID (d1.2). Finally, the spoofed collector can receive and drop commands from the utility (d1.3). All three of these are leaves in the concrete tree because they are achievable using known actions within the system.

The other option for DoS against utility command execution is to allocate a maximum number of sessions in the meter (D2.2). First, it must be determined on which port the meter listens for commands (d2.1). If this is possible, an attempt may be made to open multiple sessions on this port in an attempt to exhaust either memory or OS resources in the meter (d2.2). Both concrete attacks are leaf nodes because pen-testing of $S2$ is needed to determine how they may be executed in practice.

3.4.3 Targeted Disconnect in S1

The final concrete attack tree analyzed here is for the disruption of electrical service. As an adversary would ideally want to execute this attack remotely, we chose archetypal attacks R1.1 - R1.3 for instantiation. In $S1$, the meter ID is printed on the front of each meter, making R1.1 achievable by visual inspection. The concrete trees for R1.2 and R1.3 are shown in Figure 3.8.

Two strategies are feasible for meter password recovery in $S1$ (R1.3). If the optical port can be physically monitored, then the password can be obtained upon the next visit by the utility (r1.1). Alternatively, if the contents of meter storage can be extracted, the password may be recoverable, though potentially only in a hashed format (r1.3). As both of these are physical attacks, they may only be used to recover a password from a single meter. This would normally

Table 3.1. Summary of concrete attacks and discovered vulnerabilities for each adversarial goal.

Ref.	Description	Enabling Feature or Vulnerability
	Energy Fraud in $S1$	
a2.1	Interpose between utility and collector	Telephone line may be accessible.
a2.2	Defeat modem intrusion detection	The mechanism cannot detect an FXS.
a4.1	Identify self as meter	A meter's ID is printed on its faceplate.
a4.2	Complete authentication round	Lack of nonce-tracking allows replayed authentication.
a5.1	Run diagnostic up to usage data	Protocol is standardized.
a6.1	Transmit forged usage data	Usage data is not integrity protected.
	Denial of Service in $S2$	
d1.1	Determine collector ID	The ID is transmitted in the clear.
d1.2	Initiate association with utility	Initialization uses a simple `init` message.
d1.3	Receive and drop packets	The utility uses the IP address of the initiator of the most recent association.
d2.1	Determine meter listening port	The collector is responsive to port scanning.
d2.2	Allocate sessions until failure	The collector does not handle many sessions robustly.
	Targeted Disconnect in $S1$	
r1.2	Physically extract passwords	Passwords are stored in the clear in EEPROM storage.
r2.1	Mutually authenticate with meter	The encryption key is derived from passwords.
r2.2	Issue disconnect command	Administrative software is commercially available.

be a limiting factor in the impact of an attack against $S1$, but we observe that its architecture encourages utilities to use the same password for a large number of meters. In the administrative utility-end software, a single password set (consisting of a read-only and administrative user) is chosen for a template program that is pushed to the meters at configuration time. This makes it very tedious to create a different program template for each meter. A brute force guessing attack is not considered, as the maximum length of a password in $S1$ is well over ten bytes. The final archetypal attack needed is the issuance of the command to the target meter. This requires that the known password be used in the mutual authentication round (the same as that used in $S1$'s diagnostic protocol) (r2.1). Once authenticated, the command can be issued (r2.2).

3.5 Results

We now turn to the results of the penetration testing to achieve each goal as summarized in Table 3.1.

3.5.1 Energy Fraud by Forged Usage Data

The energy fraud attack in $S1$ works as follows. First, an adversarial device is interposed on the PSTN link from a collector (a2.1) so as not to trigger the intrusion detection mechanism (a2.2). This was achieved by interposing our PBX on the line. Recall that the purpose of the intrusion detection feature is to protect meter communication in situations where the link to the PSTN is shared with that already present in a house. The PBX is used to route incoming calls to the meter to a laptop computer that impersonates the meter using a Python program we wrote. This is sufficient for circumventing the intrusion detection mechanism for two reasons. First, routing a call to the laptop need not involve the meter at all. Second, if the PBX is used for the purposes of eavesdropping on communication between the meter and utility, it cannot be detected by the intrusion detection mechanism that can only sense other FXOs on the line (as described in section 3.3.1). Thus two requirements of A3.1 are satisfied.

Once the adversarial laptop has been contacted by the utility, it must identify itself as the target meter (a4.1) and complete the authentication round (a4.2). This was possible without knowing the meter's password, which is used to derive the key for the authentication protocol. Spoofing meter identification only required using the ID which was printed on the meter's nameplate. Completing the authentication round without knowing the password required one observation about the protocol: the meter generates the nonce used for mutual authentication, but nonces are not tracked by the utility's server. Thus, a replayed nonce is sufficient for replaying the remainder of the authentication protocol. What was not initially obvious was that the meter places the nonce in a special field as part of the identification round. Thus, replaying both the identification and authentication rounds of ANSI C12.21 is sufficient for spoofing the meter during a diagnostic.

The remaining protocol up to forged demand insertion may also be replayed in this manner, satisfying (a5.1). The final task towards energy fraud is inserting a forged net usage value into the diagnostic. This requires adding two additional pieces of information along with the numerical usage value. First, a one byte checksum of the value is placed in the application-layer header, and second a CRC is placed in the MAC layer header, again as specified in ANSI C12.21.

3.5.2 Denial of Service Against Command Execution

Two concrete attack trees were previously introduced for Denial of Service against the execution of utility commands by meters in $S2$. The first assumed that an association could be formed between the utility and a device impersonating a collector (d1.1,d1.2). At this point, the fake collector could simply drop all commands issued by the utility (d1.3). The initial association with the utility is initiated by an `init` sent by the collector. This message, which is transmitted in the clear, contains the unique serial number used to identify the collector. The utility assumes that the source IP address of the `init` message is the collector. Any device may submit an `init` message to the utility, but will not be able to establish a secure channel without knowing the collector's symmetric key. This does not prevent the DoS attack however, as receiving the `init` message causes the utility to drop its previous association with the real collector. After this, the collector will only attempt to create a new association if it is rebooted or if some alarm condition occurs such as a power outage or potential physical tampering. A subsequent second forged `init` would suffice to immediately break this association.

The other concrete attack tree in this category is based on the idea that the collector has a maximum number of sessions which can be reached (D2.2). In practice, finding the port on which a collector listens for utility requests (d2.1) is done using the `nmap` [73] utility to perform a port scan of the collector. What we found was that while attempting to open many concurrent TCP connections on the collector's listening port, the collector would become unresponsive after fewer than ten such connections. If continual attempts at establishing new connections were made at the rate of once per ten seconds, the collector remains unresponsive, and the utility-end server is unable to complete any commands on that collector, thus satisfying d2.2.

Under the category of DoS, we do have one result that was found completely independently of the methodology presented in this paper. The use of a software fuzz tester [74] found that the collector was vulnerable to crashing while processing malformed packets. While we had not planned on systematically exploring methods for leveraging software bugs (D2.3), the use of fuzz testing would make a viable addition to the archetypal tree.

3.5.3 Targeted Disconnect

The final result we explore is the application of concrete trees R1.2 and R1.3 to disrupting electric service at a target meter by subverting its remote disconnect feature. We were unable to verify the efficacy of this attack due to fact that our $S1$ meters do not include the optional physical disconnect switch. However, we reason by inspection that the attack is possible. The first step needed to issue the remote disconnect command is password recovery (R1.3). After some experimentation, we found that concrete attack (r1.2) is possible. By desoldering a small SPI-based EEPROM memory chip from the $S1$ collector's radio card, we were able to extract the plaintext password. While this is a potentially dangerous operation, given that the same password may be used throughout a deployment, the payoff is high. Upon discovering that the meter passwords may be extracted from memory, a check of the archetypal trees reveals that A2.1 from Figure 3.1 is also satisfied, enabling several alternate strategies for energy fraud. This demonstrates the usefulness of archetypal attack trees in mapping newly discovered vulnerabilities to adversarial goals.

Once the password is recovered, it must be used to perform the default C12.21 authentication function with the target meter (r2.1). This authentication is a keyed hash based on the DES cipher, and thus requires a DES key. An internet search revealed that one distributor of $S1$ had placed the manual for the utility-end software in a publicly accessible directory. The manual revealed the fact that the first eight bytes of the password are used to derive a DES key. This is done using an unknown obfuscation method. The easiest procedure to use the recovered passwords for authenticating to the target meter would be to obtain the utility-end software, which can be purchased from third party distributors, and provide it the password to issue the disconnect command (r2.2). Otherwise, a degree of reconnaissance and reverse engineering will be necessary to determine the obfuscation method.

Chapter 4

Automating Attacks against Control Systems

This chapter provides the first steps towards the work in the following chapter. Here, we describe a scenario in which a physical control system may be manipulated through malicious control of its PLC. In such a scenario, the adversary will have penetrated the control system and accessed the PLC, and will wish to execute a *payload* (malicious control logic) on the PLC. We restrict this chapter to discussions of *dynamic payloads*, that is, payloads that are derived on the fly either by the adversary using analysis tools or by autonomous malware. We assume that payloads must be derived on the fly as it is not always practical for an adversary to have *a priori* knowledge of the exact internal configuration and layout of a PLC relative to the target plant.

4.1 Dynamic Payloads

Figure 6.1 shows the basic steps PLC malware take to dynamically construct a payload against an unknown process. As with any malware, it must first *infect* (1) one or more hosts before executing its *payload*. Infection may occur via viral propagation, Trojan horse, insiders, or any other attack vector. PLC malware ultimately tries to infect a host that can reach a PLC. Because the details of the process are unknown at this point, a payload cannot yet be directly uploaded. Instead, the PLC's memory contents are read (2) for a step called *process analysis*, which produces a canonical *process representation* (3). This may require the use of an *format library* to decode proprietary binary formats. The process representation is then used by the subsequent *payload generation* step to create a payload that will achieve the *payload goal* in the plant (4). If payload generation is successful, then a payload tailored to the specific process may be uploaded to the PLC and executed.

For the remainder of this section, we describe techniques by which each of the above steps may be achieved.

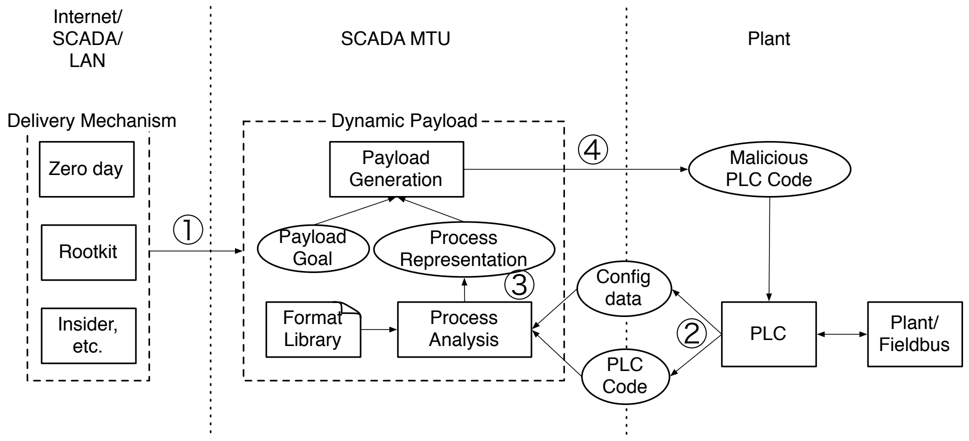


Figure 4.1. The basic steps for constructing a dynamic payload based on observations taken from within a process control system.

4.1.1 Payload Goals

A payload goal specifies the behavior that the adversary wishes to cause in the plant. It may be as simple as "Open all breakers in the electrical substation," or as complex as "Identify all incompatible regions of track and signal two trains to enter a conflicting route." It may also be very broad in scope, e.g. "Identify and violate all safety checks maintained by the PLC." Regardless of the exact goal, the dynamic payload will ultimately be a sequence of one or more assignments to output variables that achieves the goal in the plant. Thus, the payload goal can be thought of as a template for the dynamic payload, with the specifics being filled in by the steps of process analysis and payload generation.

4.1.2 Process Analysis

The job of process analysis is to convert the logic and data read from a PLC into a canonical process representation. A complete process representation should contain both a canonicalization of the PLC code, and the mapping from input and output variables to their corresponding sensors and devices in the plant. While the code can always be obtained by reading the PLC's function blocks, it may not be possible to obtain or infer the device mapping. The challenges associated with each task are described as follows.

Recovering the Boolean equations. The first step towards obtaining the process representation is to recover the set of Boolean equations Φ that represents the logic. Similar to the procedure for reverse engineering a typical computer program, the native code will have to be disassembled into mnemonics and then transformed to the higher-level representation. The disassembling will require one module in the format library for each native binary execution format.

A good candidate for mnemonic language is the IEC 61131-3 Instruction List (IL) language [75]. IL is an assembly language for accumulator-based architectures that is supported at least partially by most popular vendors including Siemens, Rockwell Automation, and ABB. Because IL is accumulator based, it can be converted to a set of Boolean equations via a symbolic execution tracking the logic accumulator. As a proof of concept, we wrote a simple program to recover Boolean equations from Siemens' version of IL (called statement list) in under 200 lines of Standard ML code. Previous investigations have shown that the mapping for disassembly can be reverse engineered with concentrated manual effort [26].

Discovering Plant Devices. Given the capabilities of modern high-end PLCs, it is in some cases possible to extract a description of the plant directly from a PLC's configuration data. A prime example of this is the emerging use of *process fieldbuses* such as Profibus and Profinet, which is quickly becoming a dominant solution for industrial Ethernet [76]. In these protocols, each piece of plant equipment identifies itself by a unique ID number that specifies its vendor and model. By collecting the device IDs, PLC malware may infer higher-level aspects of plant behavior such as which safety properties are most critical. Fieldbus device information may be acquired in two ways: by querying system configuration data in the PLC or by executing a small fieldbus scan on the PLC. The former is stored in *system data blocks* in Siemens PLC's and the latter can be achieved by uploading the scanner program to the PLC from the infected MTU.

The device IDs themselves reveal no semantic information about the nature of a device or its purpose. However, there are Internet databases, e.g. on the Profibus website, that contain pairings of device ID with additional device information. These databases may be scraped by PLC malware authors. One additional complication arises from the fact that device information is collected via a specialized network protocol. Because fieldbus-enabled devices are not connected directly to the PLC's input and output ports, PLC malware must piggyback on existing fieldbus interface code to send commands to devices.

Inferring Plant Device Types. While the target PLC may not support fieldbus communication with devices, it may still be possible to infer the *types* of devices in the plant. This will however require some additional hints to the malware from its author regarding the nature of the plant. For example, if the target plant is a rail yard, then it is known that the two most common devices are signals and switches. One common safety requirement for rail yard automation is that a signal be activated several seconds before a switch is activated [36]. Pairs of signals and switches may be identified by searching for logic variables that are connected by such timing delays. Similar ordering relationships exist in the manufacturing processes, and the operation of switchgear in electrical substations [39].

4.1.3 Payload Generation

The payload generation step produces a PLC program that will execute the payload goal in a specific plant. It should also be recognized if this is not possible given the available information about the process. Payload generation may result with infeasible if either the process description contains insufficient information about the process to instantiate the payload goal, or if the goal is not relevant to the structure of the process.

Inferring Safety Interlocks. A *safety interlock* is a check in a PLC's logic that attempts to ensure the plant never enters some unsafe state. For example, an interlock may be used to make sure that an electrical substation only performs one switching operation at a time [39]. From the perspective of PLC malware, the safety interlocks found in a program offer a definition of how to make the plant perform unsafe operations. It may obtain the set of interlocks by analysis of the set of Boolean equations in the process representation.

Formally, safety interlocks my be stated as relationships among the variables in $\mathcal{O}$. We can illustrate a basic interlock using the pedestrian crossing. The output variables p_r and p_g control the pedestrian's red and green lights respectively, and t_r and t_g control the red and green lights for traffic. A light is turned on only when its variable is true. Without even examining the implementation, one can imagine some interlocking properties. For example the state $p_g \wedge t_g$ should never occur, or else traffic might conflict with pedestrians crossing. To ensure this, an interlock statement such as $p_g \rightarrow \neg t_g$ should be added to the logic. The presence of such a statement is immediately indicative to PLC malware that sending the signals $p_g = 1$ and $t_g = 1$ to the plant will cause an unsafe state, regardless of the meaning of the variables p_g and t_g.

Of course, the property $\neg(p_g \wedge t_g)$ may be implicit in the implementation. For example, it may have been verified via formal methods that all executions preserve the desired property. For such implicit safety properties, it is not guaranteed that PLC malware can always infer an unsafe input to the plant (especially when the number of variables in the property becomes large). A significant body of work exists on property verification for control systems [38, 41, 77, 78, 36], which may be leveraged to find implicit safety properties. It is however important to differentiate between the problem of verifying a property and finding one. Even if the verification problem is tractable, the search space of possible properties can be quite large. Because the verification procedure can be quite difficult in practice, most safety properties are explicitly encoded in the process. Though it is worth noting that an equivalent rewriting of a process that contains only implicit properties could be an effective measure to thwart some dynamic malware payloads.

Inferring Plant Structure and Purpose. Before delivering a payload, PLC malware may want to test that the plant is of a specific class. Causing anomalous but harmless behavior due to misunderstanding of the purpose of the plant is likely to cause suspicion. The plant structure refers to the relationships (e.g. dependencies) between plant devices. One tool for achieving this is the dependency graph. Much like dependency graphs are used to statically find flows between variables in programs, they can be used to identify the flow of work between devices in a plant. Questions that a dependency graph can answer include which sensor inputs affect which devices, and the ordering of devices in the process' sequence of events.

An example of a process dependency graph for a full-featured traffic light control system (taken from [79]) is shown in Figure 4.2. We assume that it is only known if a variable is an input, output, state, or timer, labeled as xi, yi, vi, and ti respectively. There are at least two items of interest in the dependency graph. First, the six timers form a cycle, indicating that the process follows a set sequence of events in repetition. This means that the process is inherently

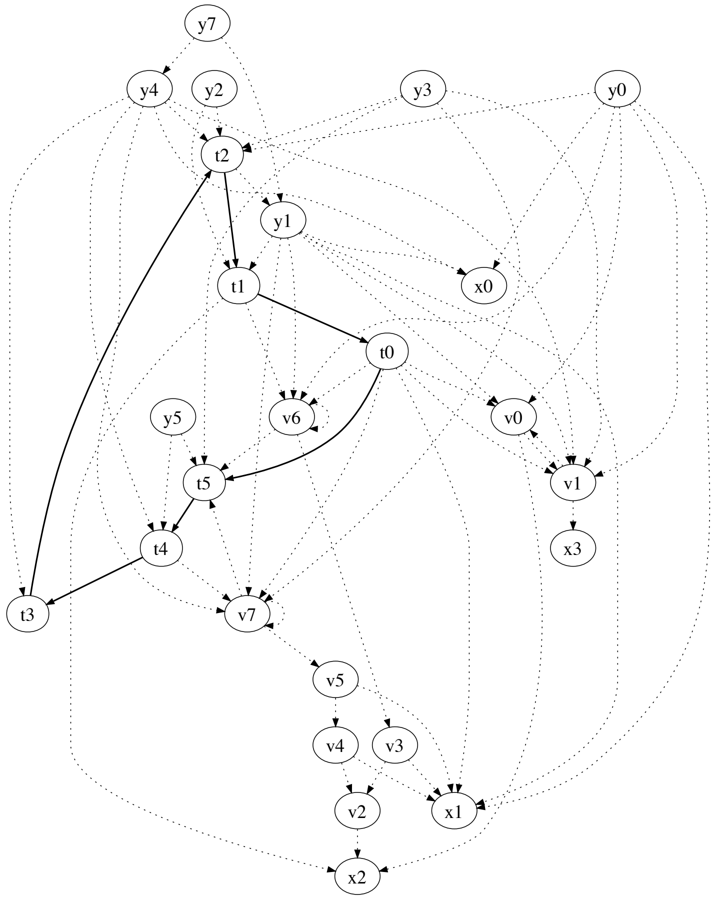


Figure 4.2. Dependency graph for traffic light control.

sequential in nature as opposed to event driven. Second, the output variable $y7$ (top) depends on two other output variables, $y2$ and $y5$, and is not a dependency for any other variable. This suggests that $y7$ is interlocked into $y2$ and $y5$ as a terminal condition in the process. Indeed, inspection of the ladder logic reveals that $y7$ triggers an alarm condition when the two opposing green lights controlled by $y2$ and $y5$ are simultaneously active. We have found that this same pattern reveals alarm states in ladder logic programs for industrial processes.

Compiling The Payload. The malicious payload is a piece of control logic that ultimately assigns values to output variables in order to disrupt proper plant behavior as described by the adversarial goal. In the absence of a specific goal, a measure such as violating all safety interlocks may also prove destructive. If the goal contains assignments to devices for which no variables have been discovered or inferred, then a payload cannot be compiled. Otherwise, a set of Boolean assignments is created, and assembled back into the PLC's native format. (The format libraries for this step are available from most vendors.) As was the case with Stuxnet, the malicious assignment may be embedded within the valid process code and programmed to execute at a certain time or under certain conditions.

Chapter 5

Specification-based Attacks on Sequential Control Systems

In this chapter, we describe Sabot a tool for extracting a key piece of information from a control system's implementation. Sabot recovers the variable to device mapping used by PLCs, thus enabling an adversary to instantiate an arbitrary attack against a control system based solely on PLC access. Unlike the safety interlock attacks described in the previous chapter, Sabot is for performing targeted attacks that have specific purposes, or are stealthy. Thus, the adversary in this chapter is assumed to have knowledge of the behavior of the target control system ahead of time. While this may seem a burdensome requirement, we note that it is actually contradictory to expect an adversary without specific knowledge to execute a specific type of attack. We also note that behavioral information about a control system may be easier to obtain than first thought. Systems such as electrical substations and railway switching operate in plain sight. Low ranking employees can likely gain visibility of operations on a factory floor. Furthermore, plant schematics and parameters can be leaked through vulnerable human machine interfaces [19, 18]. Thus, it is with this model of an adversary with knowledge of plant behavior, but not PLC implementation that we begin this chapter.

5.1 SABOT

Sabot creates malicious payloads for targeted control systems. Depicted in Figure 5.1, the Sabot initially extracts a logical model of the process from the PLC code (see Section 5.2, Decompilation). Next, the model and process sketch is used to create a mapping of physical devices to input and output variables (called the *variable-to-device mapping*, or VTDM, (see Section 5.3, VTDM Recovery). Last, a generic attack is projected onto to the existing model and VTDM to create a malicious payload called the *PLC Malcode* (see Section 5.4, Payload Construction). This malcode is delivered to the victim interface.

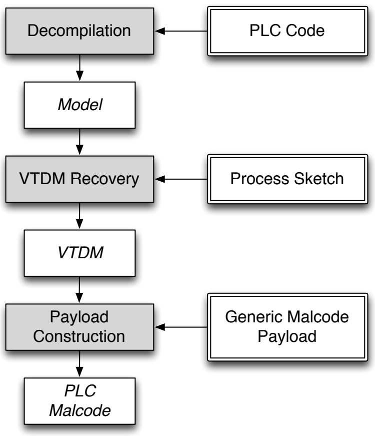


Figure 5.1. SABOT: Generating a malicious payload for a target control system based on process sketch, the PLC code, and generic malcode.

SABOT can be used in a number of scenarios. In a more interactive scenario, the adversary is able to obtain the bytecode for the target process. Using this information they could develop a highly accurate process sketch, targeted payload, and ultimately carefully targeted malcode for one or more specific PLCs. If the adversary cannot directly obtain network access to the PLC, i.e., due to an air gapped control system, SABOT can also function within autonomous malware [26]. In this mode, a malicious program scans a domain for open PLC interfaces, extracts their bytecode control logic, and supplies them to SABOT. SABOT can then test a number of adversary-supplied sketches to determine which is the "best fit" (see Section 5.5.2). Thereafter, the it can select the appropriate malcode for the identified process. For example, a worm with tens of automotive process sketches and malcode payloads released in an automotive factory could inflict substantial damage. We defer investigation of this latter mode to future work.

Note that this work explicitly does not address how the PLC bytecode is obtained from or reprogrammed into the PLC. Also as shown by Stuxnet, such procedures are most frequently accomplished by attacking the IT infrastructure of the facility in which the plant is housed. While sometimes the techniques are sophisticated, recent studies have shown that old attack vectors are often sufficient [9]; control systems expose open interfaces that are not only addressable on an internal network, but also may be reached from the Internet [80, 11].

5.1.1 Problem Formulation

Consider a scenario in which an adversary may wish to cause ingredient A to be omitted from the chemical mixing process described above. A PLC payload for this might look like:

Valve $\mathsf{A} \leftarrow \bot$

Valve $\mathsf{B} \leftarrow$ (Start Button $\vee$ Valve B) $\wedge\, \neg$Drain Valve

Intuitively, this control system never dispenses A, but rather fills the tank with B. Unfortunately, the adversary does not know how to specify to the PLC which device is meant by "Valve A" or

"Start Button." This is because PLCs do not necessarily label their I/O devices with semantically meaningful names like "Drain Valve."[1] Instead, PLCs use memory addresses, e.g., x_1, y_2, to read values from and write values to sensors and physical devices. We refer to this set of address names as $V_{\mathcal{M}}$. The adversary, who does not know the semantics of the names in $V_{\mathcal{M}}$, prefers to use the set of semantically meaningful names $V_\phi = \{$ Start Button, Valve $\mathsf{B}, \ldots\}$.

This raises the problem, *How can an adversary project attack payloads using names in V_ϕ onto a system that uses the unknown memory references $V_{\mathcal{M}}$?* One of SABOT's main tasks is to find a mapping from the names in V_ϕ to those in $V_{\mathcal{M}}$. Here, SABOT requires one additional piece of information from the adversary: a *sketch* of the target behavior.

If the adversary is to write a payload such as the one above for the mixing plant, then it is assumed that he knows some facts about the plant. For example, the adversary can make statements like: "The plant contains at least two ingredient valves, and one drain valve," and "When the start button is pressed, the valve for ingredient A opens." The adversary encodes such statements into a behavioral sketch of the target plant. When SABOT is then given a sketch and control logic from a plant PLC, it will try to locate the device addresses that behave the same under the rules of the logic as the semantically meaningful names in the adversary's sketch.

Like the payload, the sensors and devices specified in the sketch are defined using semantically meaningful names from V_ϕ. Given a control logic implementation, SABOT will construct a model $\mathcal{M}$ from the control logic ($\mathsf{Var}(\mathcal{M}) = V_{\mathcal{M}}$), and perform a model checking analysis to find the Variable To Device Mapping (VTDM) $\mu : V_\phi \rightarrow V_{\mathcal{M}}$. SABOT assumes it has the correct mapping μ when all properties in the sketch hold under the control logic after their names have been mapped according to μ. For example, the above property, "When the start button is pressed, the valve for ingredient A opens," will be checked as, "Under the rules of the control logic, When x_4 is pressed, then y_1 opens," under the mapping $\mu = \{\text{Start Button} \mapsto x_4, \text{Valve } \mathsf{A} \mapsto y_1\}$.

The sketch is written as one or more temporal logic formulas ϕ ($\mathsf{Var}(\phi) \subseteq V_\phi$) with some additional hints for SABOT. For a given mapping μ, the adversary supplied payload or sketch under μ, denoted $\mu/\mathit{payload}$ or μ/ϕ, is identical to the original, except with any names from V_ϕ replaced by names from $V_{\mathcal{M}}$. Thus, to check whether a given mapping μ maps V_ϕ to the devices is correct, SABOT checks:

$$\mathcal{M} \models \mu/\phi$$

Read, "The temporal logic formula ϕ with literal names mapped by μ holds over the labeled transition system $\mathcal{M}$." If these checks are satisfied under a given μ, then sabot instantiates the payload over V_ϕ into a payload over $V_{\mathcal{M}}$.

Desc.	Bytecode	Accumulator α	Stack
		$\top$	-
And x_1	`A` x_1	x_1	-
Nested Or	`O(`	$\top$	$x_1 : \vee$
And x_2	`A` x_2	x_2	$x_1 : \vee$
And y_1	`A` y_1	$x_2 \wedge y_1$	$x_1 : \vee$
Pop stack	`)`	$x_1 \vee (x_2 \wedge y_1)$	-
Store α to y_2	`=` y_2	$\top$	-

$$C \leftarrow C \cup \{y_2 \leftarrow \alpha\}$$
$$V_{\mathcal{M}} \leftarrow V_{\mathcal{M}} \cup \{x_1, x_2, y_1, y_2\}$$

Table 5.1. Example accumulation of a constraint.

5.2 Decompilation

To obtain a process model $\mathcal{M}$, SABOT must first bridge the gap between the bytecode-level control logic, and the model itself. This means decompiling a list of assembly mnemonics that execute on an accumulator-based architecture into a labeled transition system defined over state variables. SABOT performs this decompilation in two steps. (1) The disassembled control logic bytecode is converted to an intermediate set of constraints C on local, output, and timer variables from the PLC. (2) The constraints in C are then translated to $\mathcal{M}$ using the modeling language of the NuSMV model checker [64].

For step 1, the constraints are obtained via symbolic execution of the bytecode. This requires a pre-processing to remove nonstandard instructions not handled by our symbolic execution. The resulting code conforms to the IEC 61131-3 standard for PLC instruction lists [61]. The control flow graph (CFG) of the resulting code is constructed and a symbolic execution is done over the CFG according to a topological ordering. Several register values are tracked, most importantly the logic accumulator α. An example symbolic accumulation of control logic is shown in Table 5.1.

Step 2 translates the set of constraints resulting from step 1 into a control logic model $\mathcal{M}$ that can be evaluated by the NuSMV model checker. NuSMV takes definitions of labeled transitions systems with states consisting of state variables. SABOT uses the `VAR · : boolean` expression to declare a state variable for each name in $V_{\mathcal{M}}$. Each Boolean variable is first initialized using the `init( · )` expression, and updated at each state transition using the `next( · )` expression. A Boolean variable may be initialized or updated to a constant value of $\top$ or $\bot$, another expression, or a nondeterministic assignment $\{\top, \bot\}$, where both transitions are considered when checking a property. For a complete specification of the NuSMV input language, see [65].

As shown in Table 5.2, there are three translation rules. In the case of input variables, a new Boolean variable is declared, initialized to $\bot$, and updated nondeterministically. The nondeterministic update is necessary because all possible combinations of sensor readings must be factored into the model. Output and local variables are initialized to $\bot$ and updated according to the expression α.

[1]Some PLCs such as the Rockwell Controllogix line allow programmers to give names to I/O ports, but these names are still of no use autonomous malware.

Constraint	**NuSMV Model $\mathcal{M}$**
input x	$\texttt{VAR}\ x : \texttt{boolean};$ $\texttt{ASSIGN}$ $\texttt{init}(x)\ := \bot;$ $\texttt{next}(x)\ := \{\top, \bot\};$
output or local y $c = y \leftarrow \alpha$	$\texttt{VAR}\ y : \texttt{boolean};$ $\texttt{ASSIGN}$ $\texttt{init}(y)\ := \bot;$ $\texttt{next}(y)\ := \alpha;$
timer t $c = t \leftarrow \alpha$	$\texttt{VAR}\ t : \texttt{boolean},\ t_p : \texttt{boolean};$ $\texttt{ASSIGN}$ $\texttt{init}(t)\ := \bot;$ $\texttt{next}(t)\ := \alpha \wedge (t_p \vee t)\ ?\ \top\ :\ \bot;$ $\texttt{init}(t_p)\ := \bot;$ $\texttt{next}(t_p)\ := \alpha;$

Table 5.2. Constructing $\mathcal{M}$ from constraints C.

Timer variables require an extra bit of state. Recall that a PLC timer $t = \top$ only when its input expression $\alpha = \top$ continuously for at least t's preset time duration. Otherwise, $\alpha = \bot \Rightarrow t = \bot$. Furthermore, any input variable in the model may change state while the timer is expiring. Thus, for each timer α must hold for two state transitions. The first transition simulates the starting of the timer's countdown, and the second simulates the expiration, allowing the timer to output $\top$.

5.3 VTDM Recovery

Recall that the process sketch is the adversary-supplied description and expected behaviors of some plant devices. SABOT attempts to find a Variable To Device Mapping (VTDM) μ from names in the sketch to names in the control logic model $\mathcal{M}$. If the correct mapping is found, then the semantics are known for each name in $V_{\mathcal{M}}$ mapped to by μ.

A sketch is an ordered list of *specifications*. A specification with name *id* has the following syntax:

id : `<input [input-list]> <output [output-list]>`
    `<INIT [init-input-list]> <UNIQUE>` ϕ

As an example, we can now restate our earlier specification for the plant start button, "When the start button is pressed, the valve for ingredient A opens," as the following specification *sbutton*:

$$\begin{aligned} sbutton\ :\ & \texttt{input}\ start^*\ \texttt{output}\ v_{\mathsf{A}}\ \texttt{INIT}\ start^* \\ & start^* \Rightarrow \mathrm{AX}\ v_{\mathsf{A}} \end{aligned}$$

The only mandatory part of a specification is the property ϕ, which is defined in Computational Tree Logic (CTL). (See the box to the right for a brief introduction to CTL.) The CTL specification is defined over names given after the `input` and `output` keywords, where {`input-list`} $\cup$ {`output-list`} $\subseteq V_\phi$. The SABOT checker will check this ϕ under the control logic model $\mathcal{M}$ in three steps:

1. Choose μ: {`input-list`} $\cup$ {`output-list`} $\rightarrow V_\mathcal{M}$.
2. Apply μ to ϕ by substituting all names in ϕ with their mappings in μ. This is denoted by μ/ϕ, read "The property ϕ under the mapping μ."
3. Check $\mathcal{M} \models \mu/\phi$.

These three steps are applied over all possible mappings for a given specification.

There are two more optional parts to each specification, the list of inputs that will be initially ON `INIT`, and the conflict resolution hint `UNIQUE`. Any names in `init-input-list` will have their initial values set to $\top$ when checking $\mathcal{M} \models \mu/\phi$, while all other inputs will have initial value $\bot$. This is useful for checking plant starting states. The keyword `UNIQUE` declares that the names in `input-list` and `output-list` will not appear in any *conflict mappings.* For a definition and discussion of conflict mappings, see Section 5.3.2.

5.3.1 Mapping Sketches to Models

SABOT searches for a mapping $\mu : V_\phi \rightarrow V_\mathcal{M}$ such that $\mathcal{M} \models \mu/\phi$ for every specification ϕ in the sketch. This is done incrementally, finding a satisfying mapping for each specification before moving to the next. If no satisfying mapping is found for a given specification, the previous specification's mapping is discarded, and it is searched again for another satisfying mapping. If no more satisfying mappings are found for the first specification in the sketch, the algorithm terminates without identifying a mapping. If a satisfying mapping is found for all specifications, the algorithm accepts this as the correct mapping μ_{SAT} Algorithm 1 shows the basic mapping procedure (except for the `UNIQUE` feature).

Algorithm 1: IncMapping

```
  Input  : $\mu$, $sketch$, $V_\mathcal{M}$, $\mathcal{M}$
  Output: The satisfying mapping $\mu_{SAT}$ or none
1 if $sketch = \emptyset$ then
2     $\mu_{SAT} \leftarrow \mu$
3     return $\top$
4 $\phi \leftarrow$Pop($sketch$)
5 foreach $\mu_0$ :Var($\phi$) $\rightarrow V_\mathcal{M}$ do
6     if $\mathcal{M} \models \mu_0/\phi$ then
7         if IncMapping($\mu \cup \mu_0, sketch, V_\mathcal{M} - \mu_0$(Var($\phi$)), $\mathcal{M}$) then
8             return $\top$
9 return $\bot$
```

We use the NuSMV model checker [64] for deciding $\mathcal{M} \models \mu/\phi$. The running time of IncMapping is dependent on the number of false positive mappings for each specification in the sketch bounded below by $\Omega(|sketch|)$ and above by $O(|V_{\mathcal{M}}|^{|V_\phi|})$.

5.3.2 Unambiguous Sketches

Consider the problem of writing a sketch for the chemical mixer process shown in Figure 2.2. First, one must define the names in V_ϕ. Denoting input variable names with an '*', let l_1^*, l_2^*, and l_3^* be the names of the low, mid, and high level switches respectively, and let $start^*$ be the start button. Additionally, let v_{A} and v_{B} be the valves for ingredients A and B, *mixer* be the mixer, and v_d be the drainage valve. As in the figure, $V_{\mathcal{M}} = \{x_1, x_2, x_3, x_4\} \cup \{y_1, y_2, y_3, y_4\}$.

Recall the specification *sbutton* from above. While *sbutton* is an accurate specification of plant behavior, it is also ambiguous. To see this, we first consider the case of the correct (true positive) mapping $\mu_{TP} = \{start^* \mapsto x_4, v_{\mathsf{A}} \mapsto y_1\}$. When the mapping is applied to the CTL specification, we get: $\mu_{TP}/sbutton = \texttt{INIT}\ x_4\ \ x_4 \Rightarrow \text{AX}\ y_1$. μ_{TP} is the correct mapping because, (1) $\mu_{TP}/sbutton$ holds under the control logic (Figure 2.2(c)), and (2) x_4, and y_1 are the names of the control input and output for the devices that the adversary intended.

Consider an exemplary false positive mapping: $\mu_{FP} = \{start^* \mapsto x_2, v_{\mathsf{A}} \mapsto y_2\}$. Judging by the same criteria as above, we can see that (1) $\mu_{FP}/sbutton$ holds under the control logic, but criterion (2) fails because x_2 and y_2 (the mid level switch and the B ingredient valve respectively) are not the names of control variables intended by the adversary. This raises the question: how can the adversary remove this ambiguity from the sketch without *a priori* knowledge of the semantics of x_1, x_2, y_1, and y_2?

While the adversary does not know the semantics of names in $V_{\mathcal{M}}$, he does know the semantics of names in V_ϕ. Thus, the adversary need not know that x_2 is a name for a mid level switch and not a start button, only that there is *some* control variable name that corresponds to a mid level switch. But the adversary already has an abstraction for this, the name l_2^*. The same goes for y_2 and its abstraction, the name v_{B}. Thus, the adversary can reliably remove this ambiguity by checking which names in V_ϕ are in *conflict* with names in the property *sbutton*. For example, consider a sketch that has the same structure as *sbutton* but with different names:

$$
\begin{aligned}
cflict\ :\ & \texttt{input}\ l_2^*\ \texttt{output}\ v_{\mathsf{B}}\ \texttt{INIT}\ l_2^* \\
& l_2^* \Rightarrow \text{AX}\ v_{\mathsf{B}}
\end{aligned}
$$

Because the correct mapping for *cflict* differs from the correct mapping for *sbutton*, we have that l_2^* conflicts with $start^*$, and v_{B} conflicts with v_{A}. If the adversary can remove this conflict, then the false positive mapping μ_{FP} will also be removed, and the ambiguity is resolved. The conflict can be removed by the addition of the following specification.

$$\mathit{sbindep} : \texttt{input}\ \mathit{start}^*\ \texttt{output}\ v_{\mathsf{A}}$$
$$\mathrm{EF}(\neg \mathit{start}^* \wedge v_{\mathsf{A}})$$

To see that the pair *sbutton*, *sbindep* removes the conflict, we can substitute the conflict into *sbindp*, giving: $\mathrm{EF}(\neg l_2^* \wedge v_{\mathsf{B}})$, which does not hold under the control logic. Thus, if the conflicting mapping is initially made when checking *sbutton*, this mapping will fail when checking *sbindep*, which will cause the Sabot checker to go back to *sbutton*, and search for another mapping, in this case, the correct one. Of course, this approach is not guaranteed to make a completely unambiguous sketch (as will be seen in Section 7.4), but it does remove all ambiguities with respect to devices the adversary is aware of.

In larger examples, we keep track of conflicts and resolutions using *conflict tables*. An example set of conflict tables for a sketch of a chemical process is shown in Figure 5.2. Each specification has a (potentially empty) conflict table listing all unmapped names in V_ϕ that satisfy the specification. Given a nonempty conflict table for a specification *spec*, one can make *spec* unambiguous by writing at most one additional specification over the names in *spec* for each entry in its conflict table. This guarantees a finite sketch size bounded in $O(|V_\phi|^2)$ specifications. Significantly fewer are usually necessary in practice.

The specification keyword `UNIQUE` is used to declare that no name appearing in the specification will appear in a conflicting mapping. This is useful because there are some cases where the same name appears in many conflict table entries. (See specification *sbutton* in Figure 5.2). It is never required to use `UNIQUE` to remove conflicts, but it can be useful in reducing the search space.

5.4 Payload Construction

Surprisingly, instantiating payloads into logic code is one of the more straight-forward operations: given a VTDM μ for a victim PLC and sketch, Sabot uses the VTDM to map the names in the adversary's generic payload into those in the control logic. Thereafter, the instantiated payload is recompiled into bytecode and uploaded to the PLC.

Sabot payloads are control logic programs defined over names in V_ϕ. Once the VTDM μ is found, the payload is *instantiated* under μ, producing a payload over names in $V_{\mathcal{M}}$. Using this approach, an adversary can assume the same semantics for names in the payload that are assumed for names in the sketch. As an example, the following payload manipulates the chemical mixing process into omitting ingredient A from the mixture.
Assuming that the correct mapping, i.e. μ_{TP} is found by Sabot, then this payload will execute as expected when uploaded to the mixer PLC. Our results in Section 5.5.1 show that the correct mapping can be recovered the majority of the time, even when the plant has unexpected devices and functionality.

Sketch: Neutralizer

1. *sbutton*
2. *latched*
3. *heateron*
4. *mixeron*
5. *mixerindep*
6. *levelindep*
7. *level*2
8. *valve*2
9. *valve*4
10. *valve*3
11. *tlight*
12. *pHlight*

sbutton : `INIT` $start^*$ `UNIQUE`			
	$start^*$	$\Rightarrow$ AX	v_1
1.	ls_2^*		v_2
2.	ls_2^*		$mixer$
3.	ls_2^*		$heater$
4.	ls_3^*		v_4
5.	ls_3^*		$heater$
6.	ts^*		tl
7.	as^*		al

latched refines *sbutton* :
$\mathrm{EF}(start^* \wedge \mathrm{AX}(\neg start^* \wedge v1^*))$
Empty

heateron : `UNIQUE`				
AG(	ts^*	$\Rightarrow$ AX$\neg$	$heater$	)
1.	ls_2^*		v_1	
2.	ls_3^*		v_2	
2.	as^*		v_2	

mixeron :				
AG($\neg$	ls_1^*	$\Rightarrow$ AX $\neg$	$mixer$	)
1.	ls_1^*		$v3$	
2.	ls_2^*		$v4$	
3.	ls_2^*		tl	
4.	ls_2^*		as	

*valve*2 : `INIT` ls_1^*, ls_2^*
$ls_1^* \wedge ls_2^* \Rightarrow$ AX v_2
Empty

*valve*4 :
$\mathrm{AG}(ls_1^* \wedge ls_2^* \wedge ls_3^* \Rightarrow \mathrm{AX}\ v_4)$
Empty

mixerindep refines *mixeron* :
$\mathrm{EF}(\neg ts^* \wedge \neg mixer \wedge \mathrm{AX}(\neg ts^* \wedge mixer))$
Empty

*valve*3 :
$\mathrm{AG}(\neg as^* \vee \neg ts^* \wedge \neg v_3 \Rightarrow \mathrm{AX}\ \neg v_3)$
Empty

levelindep refines *mixeron* :
$\mathrm{EF}(ls_1^* \wedge v_1 \wedge \mathrm{AX}(ls_1^* \wedge v_1))$
Empty

tlight :
$\mathrm{AG}(\neg ts^* \Rightarrow \mathrm{AX}\ \neg tl)$
Empty

*level*2 :
$\mathrm{AG}(\neg mixer \wedge ls_1^* \wedge ls_2^* \Rightarrow \mathrm{AX}\ mixer)$
Empty

pHlight :
$\mathrm{AG}(\neg as^* \Rightarrow \mathrm{AX}\ \neg al)$
Empty

Figure 5.2. An unambiguous sketch with conflict tables for the pH neutralization system.

Generic Payload	**Instantiated Payload**
$v_{\mathsf{A}} \leftarrow \bot$	$y_1 \leftarrow \bot$
$v_{\mathsf{B}} \leftarrow (start^* \vee v_{\mathsf{B}}) \wedge \neg l_3^*$	$y_2 \leftarrow (x_4 \vee y_2) \wedge \neg x_3$
$mixer \leftarrow (l_2^* \vee mixer) \wedge l_1^*$	$y_3 \leftarrow (x_2 \vee y_3) \wedge x_1$
$v_d \leftarrow (l_3^* \vee v_d) \wedge l_1^*$	$y_4 \leftarrow (x_3 \vee y_4) \wedge x_1$

In the event that Sabot finds a correct VTDM, but the adversary was unaware of some devices in the plant, an instantiated payload can cause *side effects* that may reduce stealthiness or reduce the impact of the attack. E.g., suppressing ingredient A in the chemical mixing process could result in a conspicuous failure later in the process. The impact of side effects may be partially mitigated by the use of low profile attacks such as merely reducing the amount of

System	Baseline	Emergency	Annunciator	Sequential	Parallel
Container Filling					
Motor Control		▨			
Traffic Signaling			▨		
pH Neutralization					
Railway Switching			▨		

Table 5.3. Control system variants (omitted shaded).

ingredient A.

5.5 Evaluation

This section details a validation study of Sabot focusing on three metrics:

- **Accuracy** is the ability of Sabot to correctly map a sketch onto a control system. Related experiments apply reference sketches to five variants of five diverse control systems and measurements are taken (see Section 5.5.1).
- **Differentiation**: is the ability to recognize the intended control system based on the sketch. Sketches are blindly matched against the corpus of control systems and false-matches measured. (see Section 5.5.2)
- **Adaptability** is the ability to recognize different variants of a control system. A canonical sketch is applied to vastly different implementations of a traffic light control and results measured. (see Section 5.5.3)

Another set of tests evaluates run-time costs, see Section 5.5.4. Note that this is the first set of experiments on Sabot, and other broader and larger studies providing further validation and enlightenment are appropriate (and indeed under design). However, the results of this initial study indicate that Sabot works well in practice, and that there remain numerous avenues for improving its results.

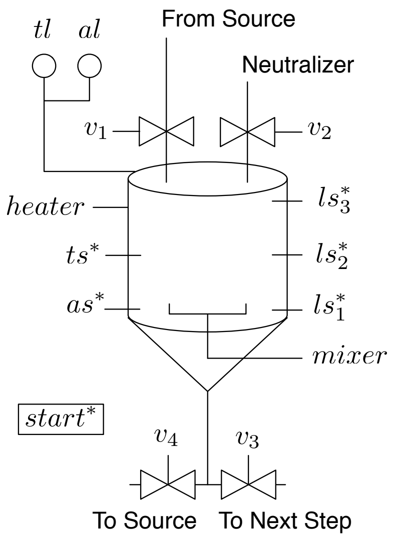
All experiments are conducted using the five variants of five representative process control systems taken from real-world applications described in Table 5.3. Each process description is used to implement a sketch and a control logic. All sketches were created independently of the implementations. Note that control variables are denoted using *emphasis* and input variable are annotated with '*' (e.g., $invar^*$, $outvar$) below. The five applications are:

Container Filling. Consider the filling of product containers on an assembly line, e.g., cereal boxes [81]. In the basic process, a belt *belt* carries an empty container ($belt = \top$) until it is under the fill-bin as indicated by a condition $cond^*$, e.g., a light barrier detects when container is in position. Thereafter, a fill valve $fill$ is opened for a period of time to fill, and the belt is activated to move the next container into place. It may also occur that the fill bin itself is depleted, as indicated by a level sensor low^* and a secondary source bin with valve *source* will be used to replenished the fill bin.

Motor Control. Stepper motors provide accurate positioning by dividing a full rotation into a large number of discrete angular positions. This has many applications in precision machining equipment like lathes and surface grinders, and conveyor belts. Most stepper motors also offer bidirectional operation. A motor controller for a stepper motor will allow for starting the motor in the forward direction fon, reversing the motor ron, such that $\neg fon \vee \neg ron$ always holds. Buttons are used for selecting forward (for^*) and reverse (rev^*) operation, and stopping the motor $stop^*$. Larger motors that require spin down times before changing direction are equipped with timers that do not allow for the motor to be started in its opposite direction until it has been stopped for several seconds.

Traffic Signaling.[2] Consider further a typical 4-way traffic light with red_1, $yellow_1$, and $green_1$ in the North/South direction, and red_2, $yellow_2$, and $green_2$ in the East/West direction. Ignoring inputs from pedestrians or road sensors, the traffic light follows the circular timed sequence: (red_1, red_2), $(red_1, green_2)$, $(red_1, yellow_2)$, (red_1, red_2), $(green_1, red_2)$, $(yellow_1, red_2)$.

pH Neutralization. A pH neutralization system mixes a substance of unknown acidity with a neutralizer until the desired pH level is achieved, e.g., in wastewater treatment. An example process adapted from [82] operates as follows:

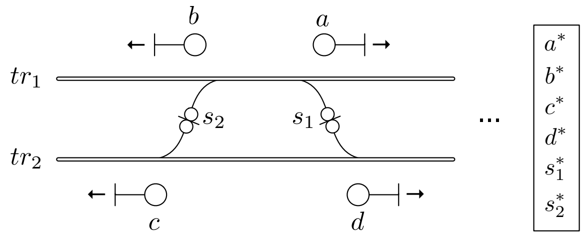


When the tank level is below ls_2^*, valve v_1 is opened to fill the tank with the acidic substance up to level ls_2^*. At this point valve v_2 is opened to dispense the neutralizer until the correct acidity level is indicated by the acidity sensor $as^* = \top$ and the correct temperature of the product has been reached as indicated by temperature sensor $ts^* = \top$. If $ts^* = \bot$, the *heater* is activated. If the desired pH level is not achieved before the tank fills to ls_3^*, v_2 is closed and v_4 is opened to drain the tank back to level ls_2^*. Once the correct pH level and temperature are achieved, and

[2]Traffic signal attacks can cause significant congestion [4].

there is at least ls_2^* liquid in the tank, it is drained to the next stage by v_3. The temperature and acidity lights tl and al are activated when the desired temperature and acidity have been reached respectively, and the tank is at least at level ls_2^*.

Railway Switching. Lastly, consider a process that safely coordinates track switches and signal lights on a real railway segment [38]:



The segment consists of the two tracks (tr_1 and tr_2), two switches (s_1 and s_2) and four signal lights a–d. A switch is said to be in the *normal* state if it does not allow a train to switch tracks and in the *reverse* state if it does allow a train to switch tracks. If a signal is ON, it indicates that the train may proceed and if it is OFF, the train must stop. The direction of each signal is indicated by an arrow.

The signalman can control the state of the lights and switches using the inputs s_1^*, s_2^* and a^*–d^*, where $s_1^* = \top$ puts switch s_1 into reverse, and a^* turns signal a ON. To ensure safe operation, the control logic maintains several invariants over the signal and switching commands from the signalman. (1) Two signals on the same track, e.g., a and b, can never be ON simultaneously. (2) If a switch is in reverse, then at most one signal is allowed to be ON. (3.a) If switch s_1 is in reverse, then signals a and c must be OFF. (3.b)If switch s_2 is in reverse, then signals b and d must be OFF. (4) If both switches are set to reverse, then all signals must be OFF.

Each of the applications is implemented in a **Baseline** control system. Each baseline system is then modified with four variants to introduce plant features not anticipated by the baseline sketches:

Emergency. This case adds an emergency shutdown button named *estop** to each plant. If the emergency shutdown button is pushed, all devices will be immediately turned OFF. One can see how this can cause false negative mappings. For example, a property of the form AG($input^* \Rightarrow$ AX $output$) will no longer hold, because of the case where $input^* \wedge estop^*$ holds, but *output* is forced to $\perp$. The motor controller's stop button acts as a shutdown, so there was no need to add one.

Annunciator. *Annunciator panels* are visual or sometimes audible displays present within the plant itself. For safety reasons, they are wired directly to the PLC, and used to indicate if there is a condition in the plant that should require special precautions by humans on the plant floor. We place a single annunciator light on each input and output in the plant. This light is turned ON by the control logic if the corresponding input or output is ON, and OFF otherwise. We evaluate the plants with annunciator panels because the annunciator variables nearly double the number of control variables that are expected by the adversary. The traffic signal and railway switching processes were not evaluated with annunciator panels, as annunciator functionality is already present in both systems.

Sequential. This case considers a plant with two distinct instances of the process, where the second instance is dependent on the first. For the motor controller, this simply means that the second motor mimics the forward and reverse behavior of the second. The same is true for the traffic lights, where the state of the second mimics the first. The sequential container filling plant simulates a pipelined plant in which the spacing between containers on the conveyor is constant. Thus, the second filling process mimics the first. (The case where spacing on the belt varies is covered in the Parallel case.) The railway switching example is modified to include three tracks, and allow a train to switch from the first track, to a middle second track, and then to a rightmost track. The safety properties are extended to prevent any conflicting routes between the three tracks. Finally, the chemical neutralization process is serialized to two tanks, such that only once a batch is complete and drained from the first tank into the second, is the process at the second tank started.

Parallel. This case models two independent instances of the process executing in parallel on the same PLC. This is expected to occur in production environments where it is more cost effective to add more input and output wires to the same PLC than to maintain distinct PLCs for each parallel instance of the process. A special criterion is added for the parallel case called **Synchronized**, which is true if all mappings where true positives in the same instance of the process. I.e., there is no mixing of mappings between the two independent processes.

5.5.1 Accuracy

Recall that accuracy is a measure of the correctness of the identified mappings between internal devices and the process sketches, i.e., the accuracy of the VTDM. Here we measure correctly mapped devices, incorrect mappings (false positives), and failures to identify any mapping at all (false negatives). The results for accuracy experiments are shown in Table 5.4. To summarize, in three out of five test systems the baseline sketch is sufficient to produce a complete, correct mapping for the control system, and four out of five systems had no false positives.

As expected, the emergency shutdown case caused false negatives in two out of the three control systems. This was due to the failure of specifications of the form $\mathrm{AG}(input^* \Rightarrow output)$ to always hold under any mapping when there is always a state in which $estop^*$ makes the property not hold. The false negatives occur for all devices in both cases, because later specifications contained names that failed to map in earlier specifications, making them uncheckable.

With respect to false positives, the pH neutralization system proved problematic due to its structural complexity and large state space caused by parallel behaviors. In the annunciator panel, the behavior of the valve for the neutralizer (v_2) could not be distinguished from the annunciator light for the mid level switch ls_2^* because $ls_2^* = \top$ implies that both the valve and annunciator light are ON. More broadly, the sequential system false positives were caused by

Control Logic	Container Filling	Start Button	Belt	Fill Valve	Synchronized	Motor Control	Forward Button	Reverse Button	Stop Button	Motor Forward	Motor Reverse	Synchronized	Traffic Signal	Red Light 1	Red Light 2	Green Light 1	Green Light 2	Yellow Light 1	Yellow Light 2	Synchronized
Baseline					▨							▨								▨
Emergency					▨		▨	▨	▨	▨	▨	▨		n	n	n	n	n	n	▨
Annunciator					▨							▨		▨	▨	▨	▨	▨	▨	▨
Sequential					▨							▨								▨
Parallel																				

Control Logic	pH Neutralization	Start Button	Source Valve	Neutralizer Valve	To Source Valve	Product Valve	Heater	Mixer	Temp Sensor	Acidity Sensor	Low Level Switch	Mid Level Switch	High Level Switch	Synchronized	Railway Switching	Signal A	Signal B	Signal C	Signal D	Switch 1	Switch 2	Synchronized
Baseline														▨								▨
Emergency		n	n	n	n	n	n	n	n	n	n	n	n	▨								▨
Annunciator				p										▨		▨	▨	▨	▨	▨	▨	▨
Sequential				p	p	p							p	▨								▨
Parallel				p	p	p	p		p	p			p	p								

Table 5.4. Per-device accuracy results. Empty cell: Correct mapping, '`p`': false positive mapping, '`n`':false negative (no mapping), Shaded cell: experiment omitted (see description).

Control Logic	pH Neutralization	Start Button	Source Valve	Neutralizer Valve	To Source Valve	Product Valve	Heater	Mixer	Temp Sensor	Acidity Sensor	Low Level Switch	Mid Level Switch	High Level Switch	Synchronized	Traffic Signal	Red Light 1	Red Light 2	Green Light 1	Green Light 2	Yellow Light 1	Yellow Light 2	Synchronized
Baseline														▨								▨
Emergency														▨								▨
Annunciator				p										▨		▨	▨	▨	▨	▨	▨	▨
Sequential				p	p	p							p	▨								▨
Parallel				p	p	p	p		p	p			p	p								

Table 5.5. Failed traffic signal and neutralization tests from Table 5.4 repeated with safeguards in place.

devices in the first instance that were mistaken for a device in the second instance. For example, the heater in the first instance of the pH neutralization system was confused with the heater in the second. The false positives in the parallel case were caused by devices in the first instance that were mistaken for non-equivalent devices in the second. For example, the finished product valve (v_3) in the first instance of the pH neutralization system was confused with the mixer in the second instance.

Next we considered a more intelligent adversary by asking the question: Could a smart adversary write a sketch that would map correctly whether an emergency shutdown was present or not? Our solution was to safeguard against the false negatives created by the emergency stop button by slightly weakening each property of the form AG($\psi \Rightarrow \phi$) to allow it to fail on cases where $\psi \wedge estop$ holds. For example, in the pH neutralizer, the property AG($\neg mixer \wedge ls_1^* \wedge ls_2^* \Rightarrow$

AX *mixer*) was changed to AF($\neg mixer \wedge ls_1^* \wedge ls_2^* \Rightarrow$ AX *mixer*), and a fairness constraint `FAIRNESS` *mixer* was added to force the model checker to ignore the path on which *estop* was infinitely ON. For properties of other forms, such as AG($\neg\psi \Rightarrow \phi$), no such weakening is necessary because it still holds when either $\psi \ \wedge$ *estop* or $\psi \ \wedge \ \neg$*estop*.

The safeguard was applied to all properties in the pH neutralization and traffic signal systems, and all five test cases were rerun. The results are shown in Table 5.5. The addition of the safeguards did not negatively affect accuracy in the baseline case, and the number of test cases for which all devices are mapped regardless of plant features is increased to 4 out of 5. Thus, if an adversary has reason to suspect that an emergency stop system may be in place, the use of safeguards can be an effective workaround.

5.5.2 Differentiation

Differentiation is the ability to identify the target control system based on the sketch. Here was ask, "*what if an adversary obtains a different control logic than expected?*" For example, waste water treatment plants have PLCs supervising different sub-processes in the same facility, and an adversary may access control logic from a PLC for an unintended subsystem. Ideally, in such a case SABOT should fail to find a mapping if the wrong sketch is used.

We cross-checked the specification for each of the five test systems against the baseline control logic for each of the five implementations. Only one false positive mapping occurred between the sketch for railway switching, and the motor control logic. It is worth noting that these were the two systems controlled mainly through user inputs. The remaining 19 checks correctly failed to produce a mapping as at least one specification was never satisfied.

5.5.3 Adaptability

Adaptability is the ability to recognize a control system by its behavior, independent of its implementation. Ideally, because SABOT only considers control logic *behavior* and not its *structure*, any implementation conforming to the process description will be handled equivalently. To confirm this is the case, and as an experimental control, a team member not involved in prior analysis was tasked with implementing an alternative traffic signal control program that exhibited the behavior from the description. The team member took an alternative strategy of allowing the light timers to drive the rest of the process, resulting in a significantly different implementation. The same experiments run above were rerun on this new implementation, and the results were identical.

5.5.4 Performance

To gauge runtime costs, we measured the running time and number of calls to the NuSMV model checker of each experiment conducted in Section 5.5.1. Note that SABOT's running time for a given model and sketch is hard to calculate for a given set of inputs because it is highly dependent on the number of incremental mappings attempted.

System	Baseline	Emergency	Annunciator	Sequential	Parallel
Container Filling	0.59/16	0.68/21	1.03/28	1.00/27	1.57/29
Motor Control	0.23/7	-/-	0.26/7	0.45/14	0.86/19
Traffic Signal	5.59/125	6.35/130	-/-	51.76/1040	103.60/1385
pH Neutralization	2.42/70	3.67/101	4.55/100	14.16/228	11.51/179
Railway Switching	0.92/25	1.05/27	-/-	4.00/97	19.95/97

Table 5.6. Running time (s) / number of calls to the model checker for each system and case.

Shown in Table 5.6, over 75% of tests, the mapping is found in less than 10 seconds, and in 90% of tests, the mapping is found in less than 30 seconds. Two tests however deserve particular attention. The test with a running time of 1m43.6s for parallel traffic signaling made the most calls to the model checker of any test. It also represented the greatest increase in calls to the model checker over its own baseline. This can be attributed to the fact that the traffic signal was the only sketch containing specifications that mapped three names at once. This gives it a guaranteed best case running time of $\Omega(|V_{\mathcal{M}}|^3)$.

Also note the comparative running times and number of checker calls for the sequential and parallel railway switching tests. Both required the same number of calls to the checker, but the parallel case has nearly a fivefold increase in running time. The greatly extended running time for each call to the checker was the result of the difference in state space between the two. In the sequential case, there were two systems with one set of inputs, and the second system dependent on the first. In the parallel case, the independent input sets greatly inflated the model's state space.

Chapter 6

Static Specification Checking

The previous chapter discussed specification-based attacks against control systems. These attacks assumed that the adversary could supply a specification that our tool SABOT could compile into a concrete payload for a target control system, once it is able to inspect the PLC's internal workings. We now consider how specifications may be of use to defending control systems against adversaries, including those with SABOT like mechanisms. Specifically, we would like to provide a means by which a set of *safety properties* about the physical behavior of a control system can be provided and enforced. Here, the enforcement occurs by static verification of any code that may be used to manipulate the target control system. In the following chapter we will consider how this enforcement can be done dynamically, but for now, we consider only the static verification case. We will present a tool called the Trusted Safety Verifier (TSV). TSV must verify that control system code will not violate any safety properties, and it must do this from within a fairly minimal package. This is where the *trusted* aspect of TSV comes into play. Along with the PLC's hardware and operating system, TSV forms the Trusted Computing Base (TCB) for the safe physical behavior of the plant.

A description is given of a working prototype of TSV on a Raspberry PI embedded computer to check code for Siemens PLCs, the most widely deployed in the world [83]. We evaluated TSV on six case studies representing a diverse set of PLC and control system functionality. These case studies consist of specifications and code that is runnable on several of the most popular PLC architectures. Our implementation can check for typical safety properties like range violations and safety interlocks in a few minutes. If a safety check fails, a useful counterexample and the offending instructions are produced. By using an intermediate language for our analysis, TSV can be extended in the future to handle check code for other proprietary PLC architectures. Because our prototype checks assembly-level code, it effectively checks any PLC program written in higher-level or graphical languages like relay ladder logic, function block diagram, and structured text.

6.1 Threat Model

Due to the strategic importance and large attack surfaces of most modern control systems, they are becoming attractive targets for attacks leading to physical damage [84, 7]. Most recently, it was revealed that the Stuxnet malware uploaded malicious code to Programmable Logic Controllers (PLCs) to physically damage the centrifuges they controlled [26]. A recent study found that PLC honeypots experienced not only port scanning, but also attempts at modifying control system specific protocols and access to system diagnostics [85]. These vulnerable Internet-connected controllers are exposed by computer search engines such as Shodan [86]. In modern control system networks, a security flaw in almost any component can be leveraged to upload malicious code to a PLC. A clear example of this was the Stuxnet virus, which used many potential vectors, including the program development environment, to propagate to a PLC-connected computer [26]. TSV's aim is to reduce the amount of control system infrastructure needed to guarantee safe behavior of PLC-controlled processes to a single embedded computer and the PLC itself.

It must be possible to notify plant operators when a safety violation is found in some PLC-bound code. If there is a rootkit present on a PLC connected computer, then notifications from TSV may be suppressed at the receiver. To handle this, we assume that there is some narrow, secure interface for notifications. A common way of implementing this in a control environment could be by way of an analog alarm, similar to those sounded when a piece of physical machinery malfunctions. Upon hearing the alarm, plant operators could directly download the safety counterexample, e.g., via serial port. In an emergency situation where immediate PLC access is needed, a physical switch in the plant could be employed to bypass TSV, similar to the emergency stop buttons on most heavy machinery.

We also must assume that the interface for uploading safety properties to TSV is secured. For example, a simple file format could be read from a USB key directly by TSV. While the use of an air-gap may seem to mitigate the advantage of a network connected PLC, safety properties require modification far less frequently than PLC code. Compared to the large numbers of requirements in existing industrial security regulations [87], this is a small additional overhead. We note that TSV is not secure against a privileged insider with physical access to the plant floor.

TSV cannot defend against false data injection attacks, in which a PLC is given forged sensor data. Additional defenses already exist for such attacks based on improved state estimators [88, 89]. Additionally, TSV cannot defend against PLC firmware exploitation, in which case the verified control logic can be completely bypassed by the compromised PLC.

6.2 System Overview

Figure 6.1 shows TSV's architecture. TSV sits as a bump-in-the-wire between system operators and the PLC. Any piece of PLC-bound code is intercepted and checked for safety properties,

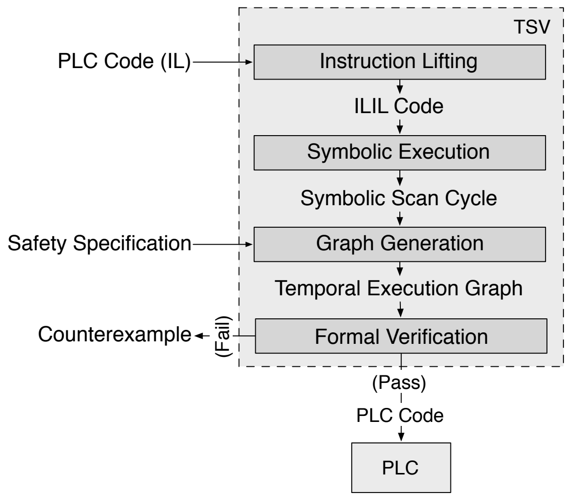


Figure 6.1. TSV Architecture.

previously supplied by process engineers. To test safety properties written in temporal logic, TSV uses model checking for part of the verification. Model checking suffers from state space explosion on numerical inputs. For example, if a PLC program has a single conditional branch depending on an integer value, the model checker will explore all of the possibilities, i.e., 2^{32} states. To prevent state space explosion, TSV first performs a symbolic execution of the program to *lump* together all inputs that produce the same symbolic output. The resulting state machine is many orders of magnitude smaller than the naive approach.

Symbolic execution of the PLC code occurs in two main steps. First, the PLC code is lifted into an intermediate language designed to make the analysis more generic and explicit. The lifted program is then symbolically executed to generate a mapping from path predicates to symbolic outputs. This mapping contains all possible executions of a single scan cycle[1], hence, we refer to it as a *symbolic scan cycle*. Once the symbolic scan cycle is calculated, TSV's model checking component is invoked. The symbolic scan cycle is an important step, as it prevents intra-scan cycle property violations from being rejected by TSV. In other words, if the PLC's variables temporarily violate a safety property at some point during the scan cycle, but the property is not violated when control signals are sent to physical machinery, then TSV will not reject the control program. This is an important distinction from the existing work combining symbolic execution and formal verification.

The formal verification component takes a set of temporal properties, consisting of atomic propositions combined with temporal qualifiers, and verifies they are maintained by a state-based model of the PLC code. We call this model, the *Temporal Execution Graph* (TEG). The TEG is a state machine in which each state transition represents a single scan cycle. Each state in the TEG contains the state and output variables of the PLC program, and a set of Boolean variables for the atomic propositions. Each Boolean represents whether the safety property is true or false

[1]In practice, it contains all executions that are reachable by scan cycle's hard deadline. See Section 6.3.1.

in that state. If any path is reachable in which a property is false, then a safety violation is raised.

One of the major tasks that TSV must accomplish is symbolically executing the PLC code, to provide the symbolic scan cycle to the model checking step. This symbolic evaluation was complicated by the fact that commercially available model checking tools are insufficient for the many PLC-specific features listed in Section 2.2.1. Furthermore, the existing work on program analysis of PLC code also yields no useful solution, as this body of work only considers a small subset of PLC features that is insufficient to statically check programs that will appear in practice. The main tool used to assist in the symbolic execution of PLC code is the Instruction List Intermediate Language (ILIL). Instead of executing raw IL code, the program is first transformed into an equivalent ILIL program, which itself is then executed. The following section details ILIL, including its grammar, type system, formal semantics, and the formalization of the transformation procedure between IL and ILIL.

6.3 The Instruction List Intermediate Language

In this section, we present our intermediate language, used for the binary analysis of PLC code. The Instruction List Intermediate Language (ILIL) is a type of Register Transfer Language (RTL) used to describe the execution of an Instruction List (IL) program on the PLC's hardware. An important reason for using such an intermediate language, is that PLC instructions that normally have side effects (modifications to variables not specified in the parameter list) are made completely explicit. In an ILIL program all PLC registers and system memory locations exist as globally accessible variables. In this section, we describe ILIL in detail. We begin with a motivation and overview for the use of this new language. There are several challenges to using traditional program analysis methods on PLC code.

- **Lack of High-level Languages.** PLCs have traditionally been programmed either in assembly language or in graphical circuit languages like relay ladder logic. The graphical languages are mere sugar used to make the assembly appear like a circuit, and they add no additional semantic information. We are thus forced to do binary or assembly-level analysis.

- **Hierarchical Addresses.** PLC addresses are not just integer values. They are prefixed by an architecturally fixed number of namespace qualifiers. When analyzing indirect addressing, we must not only consider byte address pointers, but also indirect references to different namespace qualifiers.

- **Multi-indexed Memory.** While most hardware memories only support a single size of memory access, e.g., byte or word addressing, some PLC memories can be addressed at the word-, byte-, and bit-level. This should not be confused with loading different sized registers from a byte address. Multi-indexed memory complicates the dynamic taint analysis needs for mixed execution.

$$
\begin{aligned}
\mathit{prog} &::= \mathit{inst}^{*}\mathit{fun}^{*}\\
\mathit{fun} &::= \texttt{ident}(\mathit{var})\{\mathit{inst}^{*}\}\\
\mathit{inst} &::= \texttt{cjmp}\ e, e, e \mid \texttt{jmp}\ e \mid \texttt{label ident} \mid \texttt{ident} := e\\
&\quad \mid \texttt{call ident}(\mathit{var}{=}e) \mid \texttt{ret} \mid \texttt{assert}\ e\\
e &::= \texttt{load}(\texttt{ident}, \mathit{addr}) \mid \texttt{store}(\texttt{ident}, \mathit{addr}, \texttt{int}) \mid e\ \mathit{binop}\ e\\
&\quad \mid \mathit{unop}\ e \mid \mathit{var} \mid \mathit{val} \mid (e)\\
\mathit{binop} &::= +, -, *, /, \texttt{mod}, \&, \&\&, <<, \ldots \quad \text{(And signed versions.)}\\
\mathit{unop} &::= - \text{ (Negate)}, \sim \text{ (Bitwise)}, ! \text{ (Boolean)}\\
\mathit{var} &::= \texttt{ident}\ (:\ \tau)\\
\mathit{val} &::= \mathit{mem} \mid \mathit{addr} \mid \texttt{int}\ (:\ \tau)\\
\mathit{mem} &::= \{\mathit{addr} \mapsto \texttt{int},\ \mathit{addr} \mapsto \texttt{int}, \ldots\}\\
\mathit{addr} &::= [\texttt{int} :: \texttt{int} :: \ \ldots]\\
\tau &::= \texttt{reg1\_t} \ldots \texttt{reg64\_t} \mid \texttt{mem\_t}(\texttt{int}) \mid \texttt{addr\_t}
\end{aligned}
$$

Figure 6.2. Simplified ILIL Grammar.

Even without these challenges, directly analyzing an IL program would be prohibitively difficult. IL syntax and semantics vary widely by vendor, and IL instructions have side effects that can obscure certain control flows. For these reasons, we introduce the IL Intermediate Language (ILIL) as a basis for our analysis. ILIL is based on the Vine Intermediate Language [90] (Hereafter, Vine) used for binary code analysis. We extend Vine to handle several PLC-specific features described below. A simplified grammar for ILIL is shown in Figure 6.2. Vine features such as casts and memory endianness are omitted for space sake. The full ILIL semantics are in Section 6.3.3.

An ILIL program is a set of top-level instructions followed by function definitions. This may seem strange for a binary analysis, but there are two reasons for the distinction. First, PLC code begins execution in an *Organization Block* (OB), akin to the entries in an operating system's interrupt vector. OBs are implemented in top-level code. Second, on some architectures, OBs make additional calls to function blocks. For each function call, additional ILIL code is generated to handle the parameter passing.

ILIL uses the two basic Vine types *registers* and *memories.* A single register variable is used to represent each CPU register in a particular PLC architecture. They are implemented as bit vectors of size 1, 8, 16, 32, and 64 bits. Memories are implemented differently than in Vine. ILIL Memories are mappings from hierarchical addresses (See next paragraph.) to integers. Memory loads return the integer for a given address. Memory stores return a new copy of the memory with the specified location modified.

```
// Initialize PLC state.
mem := {} : mem_t(1);                    // Main memory.
I   := 0;                                // Input memory qualifier.
Q   := 1;                                // Output memory qualifier.
RLO := 1 : reg1_t;                       // Boolean accumulator.
FC  := 0 : reg1_t;                       // System status registers.
STA := 0 : reg1_t;
...

// A I 0.5
STA := load(mem, [I::0::0::0::5]);
cjmp FC == 0 : reg1_t,L1,L2;
label L1;
RLO := STA;
label L2;
RLO := RLO && STA;
FC  := 1 : reg1_t;                       // Side effects.
...

// = Q 0.1
STA := RLO;
mem := store(mem, [Q::0::0::0::1], RLO);
FC  := 0 : reg1_t;                       // Side effects.
...
```

Figure 6.3. ILIL Code example (IL in Comments).

In addition to registers and memories, ILIL adds a third type, *addresses*. In Vine, memories are mappings from integers to integers. This is reasonable as most architectures use 32- or 64-bit address registers. This is not sufficient for PLCs which use *hierarchical addresses*. A hierarchical address has several namespace qualifiers before the actual byte or bit address. For example, in Siemens PLCs, addresses have a single namespace qualifier called a memory area. In Allen Bradley, there are three: rack, group, and slot. ILIL addresses are essentially integer lists where the leftmost n entries represent the n namespace qualifiers. We also extend the memory type to include n. Thus, the ILIL statement:

```
mem := {} : mem_t(1);
```

initializes an empty memory with a single namespace qualifier. In some cases, all or part of an address will be stored in memory. To handle loads and stores of hierarchical addresses, we extend the Vine `cast` expression to convert addresses to byte sequences. Note that the number of namespace qualifiers prefixing an address is architecturally fixed, so the number of types is finite.

ILIL instructions have no side effects, making all control flows explicit. As an example, Figure 6.3 shows the lifted version of the IL instructions:

```
A I   0.5   ;; And input bit 5
```

```
      = Q   0.1   ;; Store at output bit 1
```

First, the machine state is configured to have a single main memory and two memory areas for input and output. Part of the definition of the system status word is also shown. The *And* instruction consists of three parts. The operand is loaded from memory, combined with an accumulator, and one or more status words are updated. The address `[I::0::0::0::5]` is read, "memory area `I`, dword 0, word 0, byte 0, bit 5." This convention of listing offsets of different sizes allows us to canonically address multi-indexed memories.

The PLC features from Section 2.1, as well as several other issues, are handled by IL to ILIL translation as follows.

Timers. For each timer, an unused memory address is allocated. During symbolic execution, an attempt to check the timer value at this address will generate a fresh symbol. In the model checking step, this symbol will be nondeterministic, i.e., it will cause both paths to be explored if used in a branch condition. This is similar to the approach used by SABOT, though TSV's semantics are more flexible in allowing for the case where the timer value changes within a scan cycle, not just between them.

Counters. Counters are implemented in a straightforward manner. For each counter, a memory word is allocated to handle the current value. ILIL instructions are added to check if the counter's condition has transitioned from low to high each time a counter instruction is executed. Once the counter reaches a preset value, attempts to access its address in the counter memory area will return the concrete value true.

Master Control Relays. When an MCR section is reached, a conditional branch is generated depending on the MCR status bit. The false branch contains the original code, and the true branch contains code that modifies instruction results. While the semantics differ by architecture, typically numerical and Boolean instructions all output 0 or false when the MCR is active.

Data Blocks. Data blocks are implemented using the hierarchical address type. When a program opens a data block, a namespace qualifier is created with the index of that data block, e.g., `DB3`. When an access is made into the datablock, the address is prepended with the qualifier, e.g.,, `[DB3::20::1]` for word 41. Each data block is populated with any configuration data blocks accompanying the PLC code.

Edge Detection. For each bit of memory that is checked for a low-to-high edge transition, ILIL code is generated to monitor that bit across scan cycles. If an edge is detected, a separate bit address is set to true. This address is then checked before any dependent instructions are executed.

Flow-sensitive Optimizations. During instruction lifting, additional control flows are added to the program. For example, after an integer addition, an overflow check is added, setting several status registers to either 0 or 1. To prevent additional control flows from leading to path explosion, we only include such checks when a subsequent instruction has an explicit data dependency on the result. For example, if two additions are done in a row followed by a jump that checks an overflow status flag, only the overflow check of the second addition will be included in the lifted code.

Memory Tags. PLCs use strings, sometimes called tags, as human-readable labels on memory locations. A group of tags may be referenced by a single *name*. This leads to a complicated issue with function block parameter passing. If the name of a tag group is passed to a function, the PLC performs a pass-by-value of all tags in the group. As we would like to expose such execution semantics to our analysis, ILIL code is generated to do a pass-by-value of each tagged memory location in the group.

6.3.1 ILIL Symbolic Execution

TSV symbolically executes an ILIL program to produce a mapping from path predicates to symbolic outputs. This mapping, called the *symbolic scan cycle*, describes all possible executions of a single PLC scan cycle. Fresh symbols are allocated the first time a previously unwritten memory location is accessed. Thus, if a sensor input `I0.2` is read, then a new symbol `I_0_0_0_2` will be generated and used each subsequent time that same location is read.

Symbolic execution follows all possible paths through a single scan cycle of the program. An SMT solver is used to ensure only feasible paths are followed. Loops are followed for a constant number of iterations. Because PLC scan cycles are terminated at a hard deadline, this number of iterations can be set high enough to ensure TSV explores all iterations that are reachable by the deadline. PLCs allow function calls by indirect reference, e.g., `call FB` [`MD 0`] where the function block number is stored in `MD 0`. Fortunately, if `MD 0` contains a symbolic value, there is only a small number of possible functions it could resolve to, making the jump successor problem more tractable. Symbolic execution must handle two additional challenges, register type inference and mixed execution.

Register Type Inference. Typically, binary analysis is done on bit vectors using the register sizes of the target architecture. This is sufficient for PLC analysis, except in the common case of real-valued computations. While bit vectors will not work here, we would still like to make some safety assertions about real-valued PLC outputs. TSV relies on opcodes to infer which symbols are real-valued. Initially, all symbols start as uninitialized. The first time an instruction is executed on that symbol, or a variable that symbol propagated to, it is assigned either real or bit vector, depending on the opcode. This has the minor limitation that if both a real-valued and non-real-valued instruction are executed on the same symbol, the symbolic machine gets stuck. This is however, not common. A symbol's type can be changed only by a cast instruction in the original IL code.

Mixed Execution. PLC programs make heavy use of constants as process parameters. Thus, many instructions can be executed on concrete operands instead of symbolically. Like previous tools, such as Rudder [90], TSV performs a mixed symbolic and concrete execution. An expression produces a concrete result iff all its variables are concrete. This requires dynamically tracking

$$\frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}\ \textsf{context}$$

$$\frac{}{\Gamma \vdash v : \tau : \tau}\ \textsf{explicit}$$

$$\frac{}{\Gamma \vdash [\,] : \texttt{addr\_t}}\ \textsf{empty–addr}$$

$$\frac{l \in \mathbb{N}_0 \quad \Gamma \vdash [r_1{::}\cdots{::}r_n] : \texttt{addr\_t}}{\Gamma \vdash [l{::}r_1{::}\cdots{::}r_n] : \texttt{addr\_t}}\ \textsf{addr}$$

$$\frac{n \in \mathbb{N}_0}{\Gamma \vdash \{\} : \texttt{mem\_t}(n)}\ \textsf{empty–mem}$$

$$\frac{\begin{array}{c}\Gamma \vdash a_1 : \texttt{addr\_t} \qquad\qquad n_1 \in \mathbb{N}_0 \\ \Gamma \vdash \{a_2 \mapsto n_2, \ldots, a_m \mapsto n_m\} : \texttt{mem\_t}(n)\end{array}}{\Gamma \vdash \{a_1 \mapsto n_1, a_2 \mapsto n_2, \ldots, a_m \mapsto n_m\} : \texttt{mem\_t}(n)}\ \textsf{mem}$$

$$\frac{\Gamma \vdash e_1 : \texttt{mem\_t}(n) \quad \Gamma \vdash e_2 : \texttt{addr\_t}}{\Gamma \vdash \texttt{load}(e_1, e_2) : \texttt{regn\_t}}\ \textsf{load}$$

$$\frac{\Gamma \vdash e_1 : \texttt{mem\_t}(n) \quad \Gamma \vdash e_2 : \texttt{addr\_t} \quad \Gamma \vdash e_3 : \tau_s}{\Gamma \vdash \texttt{store}(e_1, e_2, e_3) : \texttt{mem\_t}(n)}\ \textsf{store}$$

$$\frac{\Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let x} = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}\ \textsf{let}$$

$$\frac{}{\Gamma \vdash \texttt{cast}(k, \tau, e_1) : \tau}\ \textsf{cast}$$

$$\frac{\Gamma \vdash e_1 : \tau_{reg}}{\Gamma \vdash \diamond_u\, e_1 : \tau_{reg}}\ \textsf{uop}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \diamond \in \{==, <, <=\}}{\Gamma \vdash e_1 \diamond e_2 : \texttt{reg1\_t}}\ \textsf{bop–bool}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_{reg1} \quad \Gamma \vdash e_2 : \tau_{reg2} \quad \tau_{reg1} = \tau_{reg2} \\ \diamond\ \in\ \{+, -, *, /, \&, |, >>, <<, \texttt{mod}\}\end{array}}{\Gamma \vdash e_1 \diamond e_2 : \tau_{reg1}}\ \textsf{bop}$$

Figure 6.4. Basic ILIL type rules.

whether each register and memory word is concrete or symbolic. There is a complication here for multi-indexed memories, which can be accessed at the word-, byte-, or bit-level. To handle this, TSV tracks each bit of memory as either symbolic or concrete. Initially, all memory is concrete, and typical taint propagation rules are used to track symbolic bits. We add an additional rule to allows a bit to become concrete again. If a sequential string of symbolic bits are overwritten by a equal or longer string of concrete bits, then the whole string becomes concrete.

6.3.2 ILIL Type System

Static type rules for values:

ILIL is a dynamically, implicitly typed language. That is, types are determined at run time, though some type labels are used to give hints to the type checker. ILIL has three basic types, registers and memories, which are adapted from Vine, and addresses, which are introduced in

ILIL to handle the hierarchical address schemes, commonly used by PLCs. It may seem strange to have a type checker for a register transfer language, however in this case it is necessary because PLC memory locations cannot simply be integers. Thus, we must separately track the types of integer values, and of addresses.

The basic type rules used by the ILIL type checker are shown in Figure 6.4. Each rule has a consequence of the form: $\Gamma \vdash \alpha : \tau$, where α is an expression or a value as derived from the grammar in the previous section. This consequence can be read, under the type context Γ, the expression or value α will eventually evaluate to a value with type τ. The first six rules are for values. As can be seen, there are two basic ways of determining the type of a value. The context rule allows for a value's type to be determined from the type context. The explicit rule, on the other hand, allows for a static type label to be given as a hint to the type checker. The next two rules are for the inductive definition of hierarchical addresses. An empty address simply has the value `[]`, and a non-empty address has the value `[a ... n]` where `a` through `n` are within the set of natural numbers or zero. The following two rules define the type for memories. An empty memory consists of a single integer parameter. This parameter specifies how many entries are in a word-level hierarchical address into that memory. The second rule then inductively defines the memory as a mapping from addresses to integers.

Similarly, there are type rules for *expressions*. As will be seen in the section on formal semantics, an expression resolves to a value. A `load` instruction takes a memory and an address, and produces a register. Similarly, a store instruction takes a memory, an address, and a register, and returns a memory. In the let expression, the type of expression e_1 is irrelevant to the resulting type, as long as the type of e_2 can be resolved. Casts, being type transformations themselves, are axiomatic. Finally, there are three rules for unary and binary operations. As can be seen Unary operators do not change the type of their argument. Boolean operators produce a `reg1_t`, and other binary operations do not change the types of their arguments, which themselves, must have the same type to prevent a type error. We note that there is one exception here. For the short circuit logical binary operators `||` and `&&`, a comparison between a Boolean value (which is a "type" supported by PLCs), and an integer is allowed. In this case, both are cast to `reg1_t` and compared.

To see how judgements are made with the type system, we can examine two example derivations. The first is a binary operation performed on the casted values of two registers *ACCU1* and *ACCU2*. These are the arithmetic accumulator registers in Siemens PLCs.

$$\dfrac{\dfrac{}{\Gamma \vdash \texttt{cast}(\texttt{low}, \texttt{reg16\_t}, ACCU1) : \texttt{reg16\_t}}\ \textsf{cast} \qquad \dfrac{}{\Gamma \vdash \texttt{cast}(\texttt{low}, \texttt{reg16\_t}, ACCU2) : \texttt{reg16\_t}}\ \textsf{cast} \qquad \diamond = +}{\Gamma \vdash \texttt{cast}(\texttt{low}, \texttt{reg16\_t}, ACCU1) + \texttt{cast}(\texttt{low}, \texttt{reg16\_t}, ACCU2) : \texttt{reg16\_t}}\ \textsf{bop}$$

This derivation does not depend on the current type context, though we should note that both *ACCU1* and *ACCU2* have type `reg32_t` $\in \Gamma$. The type context does not matter, as the `cast` is axiomatic. Note that if we wished to support stronger type checking, we could enforce that a size reducing cast from `regn_t` to `regm_t` could require $\texttt{n} > \texttt{m}$. However, in practice, this is not

$$\dfrac{\Sigma = \cdot \quad \Delta \vdash e \Downarrow v \quad \Delta' = \Delta[\mathtt{x} \leftarrow v] \quad \Pi \vdash p+1 : i}{\Sigma, \Delta, \ell, p, \mathtt{x} := e \ \hookrightarrow\ \Sigma, \Delta', \ell, p+1, i}\ \textsf{assign–g}$$

$$\dfrac{\Sigma \neq \cdot \quad \ell \vdash e \Downarrow v \quad \ell' = \ell[\mathtt{x} \leftarrow v] \quad \Pi \vdash p+1 : i}{\Sigma, \Delta, \ell, p, \mathtt{x} := e \ \hookrightarrow\ \Sigma, \Delta, \ell', p+1, i}\ \textsf{assign–l}$$

$$\dfrac{\Sigma' = \Sigma; (\ell, p+1) \quad \Delta, \ell \vdash e \Downarrow v \quad \Phi \vdash f : p' \quad \Pi \vdash p' : i}{\Sigma, \Delta, \ell, p, \mathtt{call}\ f(\mathtt{x}\ e) \ \hookrightarrow\ \Sigma', \Delta, \{\mathtt{x} : v\}, p', i}\ \textsf{call}$$

$$\dfrac{\Pi \vdash p' : i}{\Sigma; (\ell', p'), \Delta, \ell, p, \mathtt{ret} \ \hookrightarrow\ \Sigma, \Delta, \ell', p', i}\ \textsf{ret}$$

$$\dfrac{\Delta, \ell \vdash e \Downarrow v \quad \Lambda \vdash v : p' \quad \Pi \vdash p' : i}{\Sigma, \Delta, \ell, p, \mathtt{jmp}\ e \ \hookrightarrow\ \Sigma, \Delta, \ell, p', i}\ \textsf{jmp}$$

$$\dfrac{\Delta, \ell \vdash e_1 \Downarrow 1 \quad \Delta, \ell \vdash e_2 \Downarrow v \quad \Lambda \vdash v : p' \quad \Pi \vdash p' : i}{\Sigma, \Delta, \ell, p, \mathtt{cjmp}\ e_1, e_2, e_3 \ \hookrightarrow\ \Sigma, \Delta, \ell, p', i}\ \textsf{cjmp–t}$$

$$\dfrac{\Delta, \ell \vdash e_1 \Downarrow 0 \quad \Delta, \ell \vdash e_3 \Downarrow v \quad \Lambda \vdash v : p' \quad \Pi \vdash p' : i}{\Sigma, \Delta, \ell, p, \mathtt{cjmp}\ e_1, e_2, e_3 \ \hookrightarrow\ \Sigma, \Delta, \ell, p', i}\ \textsf{cjmp–f}$$

$$\dfrac{\Pi \vdash p+1 : i}{\Sigma, \Delta, \ell, p, \mathtt{label}\ s \ \hookrightarrow\ \Sigma, \Delta, \ell, p+1, i}\ \textsf{label}$$

$$\dfrac{\Delta, \ell \vdash e \Downarrow 1 \quad \Pi \vdash p+1 : i}{\Sigma, \Delta, \ell, p, \mathtt{assert}\ e \ \hookrightarrow\ \Sigma, \Delta, \ell, p+1, i}\ \textsf{assert–t}$$

$$\dfrac{\Delta, \ell \vdash e \Downarrow 0}{\Sigma, \Delta, \ell, p, \mathtt{assert}\ e \ \hookrightarrow\ \bullet, \bullet, \{\text{“err”} : \text{“}e\text{”}\}, \bullet, \bullet}\ \textsf{assert–f}$$

Figure 6.5. Operational semantics of ILIL instructions.

possible given the instruction lifting procedure, which only chooses appropriate registers to be substituted into casts, by definitions of the templates. For a full description and formalization of the lifting process, see Section 6.3.4.

As a second example, consider this derivation of a store operation. Unlike the previous derivation, this one is dependent on the types of *mem* and *RLO* in Γ. Specifically, $\Gamma = \{mem : \tau, RLO : \mathtt{reg1\_t}\}$.

$$\dfrac{\dfrac{\Gamma(mem) = \tau}{\Gamma \vdash mem : \tau}\ \textsf{context} \qquad \dfrac{Y \in \mathbb{N}_0 \quad \dfrac{1 \in \mathbb{N}_0 \quad \dfrac{0 \in \mathbb{N}_0 \quad \dfrac{}{\Gamma \vdash [] : \mathtt{addr\_t}}\ \textsf{empty–addr}}{\Gamma \vdash [0] : \mathtt{addr\_t}}\ \textsf{addr}}{\Gamma \vdash [1 :: 0] : \mathtt{addr\_t}}\ \textsf{addr}}{\Gamma \vdash [Y :: 1 :: 0] : \mathtt{addr\_t}}\ \textsf{addr} \qquad \dfrac{\Gamma(RLO) = \mathtt{reg1\_t}}{\Gamma \vdash RLO : \mathtt{reg1\_t}}\ \textsf{context}}{\Gamma \vdash \mathtt{store}(mem, [Y :: 1 :: 0], RLO, \mathtt{reg1\_t}) : \tau}\ \textsf{store}$$

Note that in practice Y would be a constant value hardcoded into an interpreter, as it is architecturally fixed in most PLCs.

$$\frac{\Delta, \ell \vdash e_1 \Downarrow v_1 \quad \Delta, \ell \vdash e_2 \Downarrow v_2 \quad size(v_2) = m \quad \Gamma \vdash v_1 : \texttt{mem\_t}(m) \quad v = v_1[v_2 \ldots v_2 + n]}{\Delta, \ell \vdash \texttt{load}(e_1, e_2) \Downarrow v} \ \textsf{load–bytes}$$

$$\frac{\Delta, \ell \vdash e_1 \Downarrow v_1 \quad \Delta, \ell \vdash e_2 \Downarrow v_2 \quad size(v_2) = m + 1 \quad \Gamma \vdash v_1 : \texttt{mem\_t}(m) \quad v = v_1[v_2]}{\Delta, \ell \vdash \texttt{load}(e_1, e_2) \Downarrow v} \ \textsf{load–bit}$$

$$\frac{\Delta, \ell \vdash e_1 \Downarrow v_1 \quad \Delta, \ell \vdash e_2 \Downarrow v_2 \quad \Delta, \ell \vdash e_3 \Downarrow v_3 \quad size(v_2) = m \quad \Gamma \vdash v_1 : \texttt{mem\_t}(m) \quad v = v_1[v_2 \ldots v_2 + n \leftarrow v_3]}{\Delta, \ell \vdash \texttt{store}(e_1, e_2, e_3) \Downarrow v} \ \textsf{store–bytes}$$

$$\frac{\Delta, \ell \vdash e_1 \Downarrow v_1 \quad \Delta, \ell \vdash e_2 \Downarrow v_2 \quad \Delta, \ell \vdash e_3 \Downarrow v_3 \quad size(v_2) = m + 1 \quad \Gamma \vdash v_1 : \texttt{mem\_t}(m) \quad v = v_1[v_2 \leftarrow v_3]}{\Delta, \ell \vdash \texttt{store}(e_1, e_2, e_3) \Downarrow v} \ \textsf{store–bit}$$

$$\frac{\Delta, \ell \vdash e_1 \Downarrow v_1 \quad \Delta' = \Delta[\texttt{x} \leftarrow v_1] \quad \Delta', \ell \vdash e_2 \Downarrow v}{\Delta, \ell \vdash \texttt{let x} = e_1 \ \texttt{in} \ e_2 \ \texttt{end} \Downarrow v} \ \textsf{let}$$

$$\frac{\Delta, \ell \vdash e_1 \Downarrow v_1 \quad \Delta, \ell \vdash e_2 \Downarrow v_2 \quad v = v_1 \diamond_b v_2}{\Delta, \ell \vdash e_1 \diamond_b e_2 \Downarrow v} \ \textsf{bop}$$

$$\frac{\Delta, \ell \vdash e_1 \Downarrow v_1 \quad v = \diamond_u v_1}{\Delta, \ell \vdash \diamond_u e_1 \Downarrow v} \ \textsf{uop}$$

$$\frac{\Delta, \ell \vdash e \Downarrow v_1 \quad v = \text{higher } \tau_{reg} \text{ bits of } v_1}{\Delta, \ell \vdash \texttt{cast}(\texttt{high}, \tau_{reg}, e) \Downarrow v} \ \textsf{cast–u}$$

$$\frac{\Delta, \ell \vdash e \Downarrow v_1 \quad v = v_1 \text{ sign-extended to } \tau_{reg} \text{ bits}}{\Delta, \ell \vdash \texttt{cast}(\texttt{signed}, \tau_{reg}, e) \Downarrow v} \ \textsf{cast–s}$$

$$\frac{\Delta, \ell \vdash e \Downarrow v_1 \quad v = \textsf{pack}(\textsf{v}_1)}{\Delta, \ell \vdash \texttt{cast}(\texttt{ptr}, \texttt{addr\_t}, e) \Downarrow v} \ \textsf{cast–ptr}$$

$$\frac{\Delta, \ell \vdash e \Downarrow v_1 \quad v = \textsf{unpack}(\textsf{v}_1)}{\Delta, \ell \vdash \texttt{cast}(\texttt{addr}, \texttt{reg32\_t}, e) \Downarrow v} \ \textsf{cast–addr}$$

$$\frac{\ell \vdash \texttt{x} : v}{\ell \vdash \texttt{x} \Downarrow v} \ \textsf{var–local}$$

$$\frac{\Delta \vdash \texttt{x} : v \quad \texttt{x} \notin \ell}{\Delta, \ell \vdash \texttt{x} \Downarrow v} \ \textsf{var}$$

$$\frac{}{\Delta, \ell \vdash v \Downarrow v} \ \textsf{value}$$

Figure 6.6. Operational semantics of ILIL expressions.

6.3.3 ILIL Semantics

The operational semantics of ILIL extend those of the Vine intermediate language to include function blocks, scoped variable resolution, hierarchical addresses, loads and stores to multi-indexed memories, and casts of hierarchical addresses.

Contexts. The ILIL machine state consists of the following contexts.

- Σ - The function call stack.

$$
\begin{aligned}
\mathit{Inst} &::= \mathit{Op}\ \mathit{Arg}\\
\mathit{Op} &::= \text{(Any valid IL instruction.)}\\
\mathit{Arg} &::= \mathtt{n} \mid {}^{*}\mathit{Loc} \mid {}^{*}\mathit{Mem}\ \mathit{Loc} \mid {}^{*}\mathit{Mono}\ \mathtt{n} \mid \mathit{Mem}\ \mathit{Loc} \mid \mathit{Mono}\ \mathtt{n}\\
&\quad \mid [\mathit{Mem}\ \mathtt{n}] \mid \mathit{Mem}\ [\mathit{Mem}\ \mathtt{n}] \mid [\mathtt{AR}, {}^{*}\mathit{Loc}] \mid \varepsilon\\
\mathit{Mem} &::= \mathtt{I} \mid \mathtt{Q} \mid \mathtt{M} \mid \mathtt{L} \mid \mathtt{DB} \mid \mathtt{DI}\\
\mathit{Mono} &::= \mathtt{T} \mid \mathtt{C}\\
\mathit{Loc} &::= \mathit{Size}\ \mathtt{n} \mid \mathtt{n.n}\\
\mathit{Size} &::= \mathtt{D} \mid \mathtt{W} \mid \mathtt{B}
\end{aligned}
$$

Figure 6.7. Simplified IL grammar for lifting.

- Λ - Label to instruction number mapping.
- Π - Instruction pointer to instruction mapping.
- Φ - Function name to entry point mapping.
- Δ - The global variable context.
- ℓ - The local variable context.
- p - The instruction pointer.
- Γ - The type context.

The operational consist of instructions (Figure 6.5) and expressions (Figure 6.6). The consequence of each instruction is of the form $\Sigma, \Delta, \ell, p, i \hookrightarrow \Sigma', \Delta', \ell', p', i'$, meaning the call stack, global and local variable contexts, instruction pointer, and current instruction are transformed from the left hand side to the right hand side after execution of the instruction i. Similarly, The consequence of each expression is of the form $\Delta, \ell \vdash e \Downarrow v$, meaning that under the global and local variable contexts, the expression e evaluates to the value v.

6.3.4 ILIL Instruction Lifting

The instruction lifting procedure transforms an IL program, with side effects, into a side effect free ILIL program. The grammar for IL code that can be lifted is shown in Figure 6.7. During this process, each IL instruction will be transformed into likely, several ILIL instructions. As such, the procedure relies on two parts:

1. **Template library.** The template library consists of templated sequences of ILIL instructions. Each template corresponds to a pair $(\mathit{Op}, \mathit{atype})$ where Op is an IL opcode or assembly mnemonic, and atype is the type of argument to the mnemonic. Each template contains

$$\frac{\mathrm{T} \vdash (Op, \cdot) : t}{Op \;\rightarrow_L\; t}\ \textsf{Op}$$

$$\frac{\mathrm{T} \vdash (Op, \mathsf{n}) : t}{Op\ \texttt{n} \;\rightarrow_L\; t[\texttt{x}/\texttt{n}]}\ \textsf{Op–Num}$$

$$\frac{\mathrm{T} \vdash (Op, \mathsf{p}) : t \quad Loc \rightarrow_L a \quad b = [a]}{Op\ {}^{*}Loc \;\rightarrow_L\; t[\texttt{x}/b]}\ \textsf{Op–Ptr}$$

$$\frac{\mathrm{T} \vdash (Op, \mathsf{p}) : t \quad \mu \vdash Mem : m \quad Loc \rightarrow_L a \quad b = [m{::}a]}{Op\ {}^{*}Mem\ Loc \;\rightarrow_L\; t[\texttt{x}/b]}\ \textsf{Op–Ptr–Base}$$

$$\frac{\mathrm{T} \vdash (Op, \mathsf{p}) : t \quad \mu \vdash Mono : m \quad a = [m{::}\texttt{n}]}{Op\ {}^{*}Mono\ \texttt{n} \;\rightarrow_L\; t[\texttt{x}/a]}\ \textsf{Op–Ptr–Mono}$$

$$\frac{\mathrm{T} \vdash (Op, \mathsf{m}) : t \quad \mu \vdash Mem : m \quad Loc \rightarrow_L a \quad b = [m{::}a]}{Op\ Mem\ Loc \;\rightarrow_L\; t[\texttt{x}/b]}\ \textsf{Op–Mem}$$

$$\frac{\mathrm{T} \vdash (Op, \mathsf{m}) : t \quad \mu \vdash Mono : m \quad a = [m{::}\texttt{n}]}{Op\ Mono\ \texttt{n} \;\rightarrow_L\; t[\texttt{x}/a]}\ \textsf{Op–Mono}$$

$$\frac{\begin{array}{c}\mathrm{T} \vdash (Op, \mathsf{m}) : t \quad \mu \vdash Mem_1 : m_1 \quad \mu \vdash Mem_2 : m_2 \\ a \quad = \quad [m_1]@\texttt{load}(\texttt{mem}, [m_2{::}\texttt{n}])\end{array}}{Op\ Mem_1\ [Mem_2\ \texttt{n}] \;\rightarrow_L\; t[\texttt{x}/a]}\ \textsf{Op–Deref}$$

$$\frac{\mathrm{T} \vdash (Op, \mathsf{m}) : t \quad \mu \vdash Mem : m \quad a = \texttt{load}(\texttt{mem}, [m{::}\texttt{n}])}{Op\ [Mem\ \texttt{n}] \;\rightarrow_L\; t[\texttt{x}/a]}\ \textsf{Op–Deref–Base}$$

$$\frac{\mathrm{T} \vdash (Op, \mathsf{m}) : t \quad Loc \rightarrow_L a \quad b = \texttt{AR} + a}{Op\ [\texttt{AR}, {}^{*}Loc] \;\rightarrow_L\; t[\texttt{x}/b]}\ \textsf{Op–Deref–Arith}$$

$$\frac{}{\texttt{D n} \;\rightarrow_L\; \texttt{n}}\ \textsf{Loc–Double}$$

$$\frac{\texttt{n} = 2 * \texttt{n}_1 + \texttt{n}_2 \quad \texttt{n}_1 \in \mathbb{N}_0 \quad \texttt{n}_2 \in \{0, 1\}}{\texttt{W n} \;\rightarrow_L\; \texttt{n}_1{::}\texttt{n}_2}\ \textsf{Loc–Word}$$

$$\frac{\texttt{n} = 4 * \texttt{n}_1 + 2 * \texttt{n}_2 + \texttt{n}_3 \quad \texttt{n}_1 \in \mathbb{N}_0 \quad \texttt{n}_2, \texttt{n}_3 \in \{0, 1\}}{\texttt{B n} \;\rightarrow_L\; \texttt{n}_1{::}\texttt{n}_2{::}\texttt{n}_3}\ \textsf{Loc–Byte}$$

$$\frac{\texttt{B}\ \texttt{n}_1 \rightarrow_L a \quad \texttt{n}_2 \in \{0 \ldots 7\}}{\texttt{n}_1.\texttt{n}_2 \;\rightarrow_L\; a{::}\texttt{n}_2}\ \textsf{Loc–Bit}$$

Figure 6.8. Lifting rules for IL terms to ILIL terms.

one or more template parameters that are replaced with the argument or arguments to the IL instruction.

2. **Lifting rules.** IL programs support a number of different argument addressing modes, and, as mentioned above, support multi-indexed memories, that can be accessed at the word-, byte-, or bit-level. Because of this, there are several different potential instruction formats in an IL program. To handle these, a set of *lifting rules* define how each format should be transformed to ILIL.

The lifting rules are shown in Figure 6.8. Each lifting rule can be formalized as a judgement of the form $IL \rightarrow_L ILIL$. Meaning that the specified IL instruction corresponds to the specified ILIL code, assuming some set of antecedents are satisfied. There are two main contexts used in the set of lifting rules. The first is the template context: $\mathrm{T} : (Op, \{\mathsf{n}, \mathsf{p}, \mathsf{m}\}) \rightarrow inst^*$. It is used to choose an appropriate instruction template for a given IL instruction. The second is the memory mapping $\mu : Mem : \mathbb{N}_0$ that resolves *memory areas* to natural numbers as part of the hierarchical addressing scheme. A memory area represents a level of address hierarchy. For example, in Siemens PLC systems, typical memory areas are `I` for input buffering and `Q` for output buffering.

6.4 Model Checking

Because PLCs use *stateful* variables, that retain their values across the scan cycles, analysis of a single scan cycle is not sufficient to check all temporal safety properties. The results of symbolic execution are used to first construct the Temporal Execution Graph (TEG) to model the state transitions occurring over a scan cycle. Each node of the TEG is then productized with valuations of the atomic proposition in the safety property. Finally, the symbolic variables are removed from each state to produce an abstract graph, which is fed to the model checker. A full description of the model checking procedure can be found in [91].

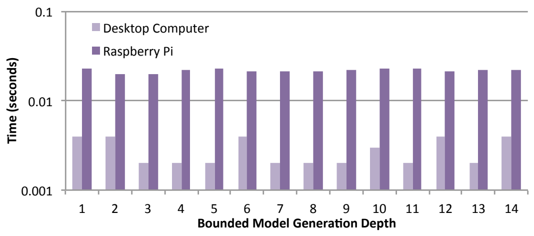
6.5 Evaluation

We now wish to investigate TSV's efficacy in checking typical safety properties against a representative set of PLC programs. In particular, we designed a set of experiments to verify whether TSV can be useful in real-world practical scenarios by answering the following questions empirically: How accurately do the employed model checking techniques in TSV verify whether a given PLC code is compliant with the requirements? How efficiently does TSV complete the formal verification steps for an uploaded PLC code? How well can TSV scale up for complex security requirements? We start by describing the experimentation control system case studies, and then proceed to examine these questions.

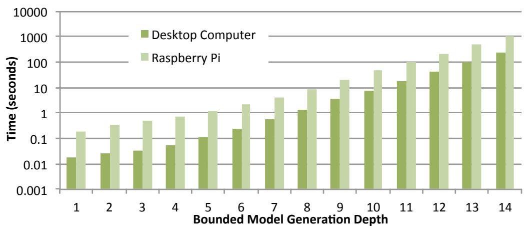
6.5.1 Implementation

We implemented TSV on a Raspberry Pi embedded computer running Linux kernel 3.2.27. The IL[2] lifting is implemented in $2,933$ lines of C++ code, the symbolic execution in $11,724$ lines of C++ code, and the TEG generation in $3,194$ lines of C++ code. In addition, TSV uses the `Z3` theorem prover [92] both for checking path feasibility during symbolic execution, and for simplifying symbolic variable values during TEG construction. `NuSMV` is used for model checking of the refined TEG [93]. In the case of a safety violation, `Z3` is used to find a concrete input for the path predicate corresponding to the offending output.

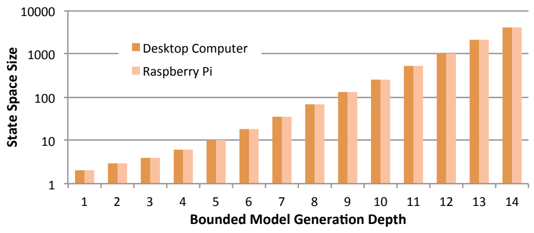
[2]To support other programming languages, a new source code lifter needs to be developed to generate the ILIL code. However, due to the syntactical similarities between most of existing PLC programming languages, the lifter may not be needed to be developed from scratch.



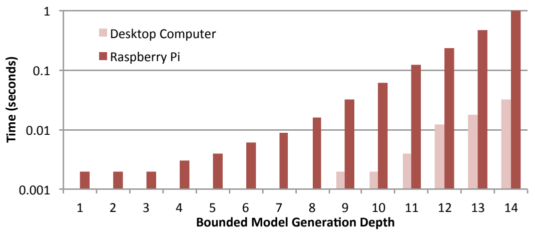
(a) Initial Model Creation



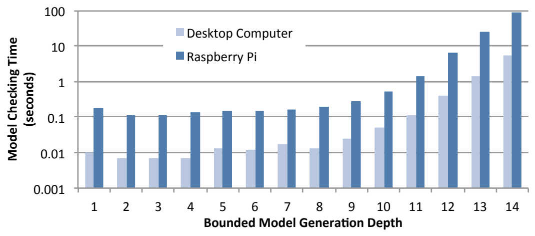
(b) Temporal Execution Graph Generation



(c) Temporal Execution Graph Cardinality



(d) Model Translation



(e) Symbolic Model Checking

Figure 6.9. Performance Analysis of the Traffic Light Control System.

6.5.2 Control System Case Studies

To make sure that TSV can be used for practical safety verification of real-world infrastructures, we deployed TSV on several real-world Siemens PLC programs for different industrial control system settings. Our examples are runnable on several of the most popular PLC architectures[3].

- **PID controller.** (Proportional Integral Derivative) The most common type of controller for real-valued states. A PID controller attempts to minimize the error between the actual state, e.g., the temperature in a room, and a desired state. This is done by adjusting a controlled quantity, e.g., heating element, by a weighted sum of the error, and its integral and derivative.
    - *Safety requirement:* (*i.*) The controlled quantity may not exceed a constant value.

[3]Specifically, the Instruction List samples run on Siemens and Rockwell PLCs accounting for 50% of PLC market share [83].

- **Traffic light.** Traffic lights at a four way intersection are governed by Boolean output signals, with a single Boolean value for each color and direction. Light changes are timer-based.
    - *Safety requirements:* (*i.*) Orthogonal green lights should not be ON simultaneously, i.e., $G\neg(g_1 \wedge g_2)$ where Boolean variable g_i denotes the i-th green light. (*ii.*) A red light should always be proceeded by a yellow light.
- **Assembly way.** Items are moved down an assembly line by a belt. A light barrier detects when an item has arrived at an assembly station, and an arm moves to hold the item in place. A part is then assembled with the item, and the barrier removed. The process repeats further down the line.
    - *Safety requirements:* (*i.*) No arm can come down until the belt stops moving. (*ii.*) The belt should not move while the arm is down.
- **Stacker.** A series of items are loaded onto a platform by a conveyor belt. Once a light barrier detects that the platform is full, the items are combined by a melter, and the resulting product is pushed onto a lift and lowered for removal.
    - *Safety requirements:* (*i.*) The product should never be pushed out while the melter is working. (*ii.*) No more items should be loaded once the platform is full.
- **Sorter.** A conveyor belt passes packages by a barcode scanner. Depending on the scanned code, one of three arms extends to push the package into an appropriate bin.
    - *Safety requirements:* (*i.*) No more than one arm should extend at each time instant. (*ii.*) An arm extends only after the barcode scanning is complete.
- **Rail Interlocking.** As opposed to the other programs, which drive the actions of a system, a railway interlocking checks that a sequence of engineer commands will not cause conflicting routes between two or more trains.
    - *Safety requirements:* (*i.*) There should never be conflicting routes. (*ii.*) No inputs are read after the checking procedure starts execution.

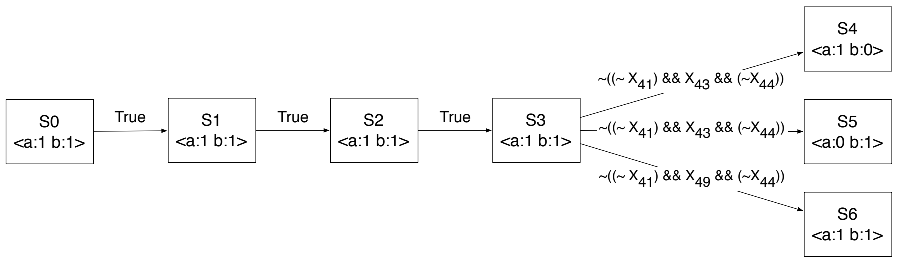


Figure 6.10. Generated Temporal Execution Graph (model checking bound = 4).

6.5.3 Example Safety Violation

To demonstrate the full usage of TSV, we show the steps that occur when attempting to upload code containing a safety violation. For this example, we modified the traffic light controller to switch both directions to green after cycling through the correct states once. Specifically, we appended the following code to the traffic light program.

```
... original program ...

RESET       ;; Reset logic accumulator.
A M 0.5     ;; Check for trigger state.
JC ATTACK   ;; Jump to attack code (if triggered).
JMP END     ;; Skip attack code.
ATTACK:
SET
= Q 0.0     ;; Set first green light.
= Q 0.3     ;; Set second green light.
END: NOP
```

The malicious program was analyzed by TSV against an interlock property prohibiting both lights from being simultaneously green. The model checker produced the concrete counterexample:

Cycle	Timer					
	1	2	3	4	5	6
1	f	f	f	f	f	t
2						f

This states that a violation was detected on the scan cycle where light timers 1-5 are false, and timer 6 switches from true to false. The next step is to identify the line of code where the violation occured. First, the ILIL interpreter preloads the concrete counterexample values for each timer variable. Next, the ILIL version of the program is instrumented with an assertion of the violated property after each line:

```
assert load(mem, [Q::0::0::0::0]) == 0 : reg1_t ||
       load(mem, [Q::0::0::0::3]) == 0 : reg1_t;
```

This simply states that at least one of the output memory locations for green lights must be off. The instrumented program is then executed with the concrete timer values. The assertion fails exactly after the line that stores 1 in `[Q::0::0::0::3]`. If the operator so desired, an execution trace of instructions and memory values leading up to this point could also be produced. Even if the original IL program is obfuscated, the ability to execute on a concrete counterexample will quickly point system operators to the offending instruction.

The example above verifies the state invariants for a simple safety requirement only for presentation clarity. However, as discussed later, TSV verified our case study PLC programs for more complex temporal safety requirements (Section 6.5.2) using the execution history information across input-output scan cycles that was encoded in the generated TEG graph.

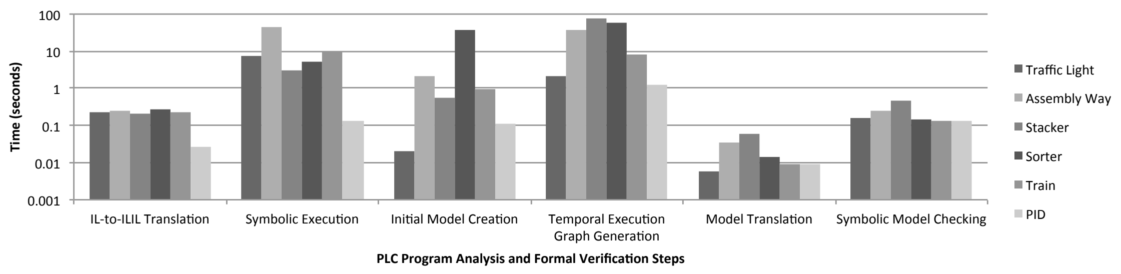


Figure 6.11. Time Requirements for All Case Studies on Raspberry Pi.

6.5.4 Performance

We measured the run times for individual TSV components while checking the safety properties for each test case. Figure 6.9 shows the results for a sample use case (the traffic light control system) for up to 14 steps during bounded model generation. This allows for exploration of control systems with up to 14 consecutive unique state outputs. This is significantly more than Stuxnet's malicious code, which used a state machine with three unique outputs to manipulate centrifuge speed [26][4]. One could imagine an attack that evades detection by counting to 15 before violating a safety property. In this case, any control logic capable of producing a non-repeating chain of more than 14 unique outputs could also be rejected. This bound could be set higher if required for the legitimate plant functionality. The results are shown for running TSV on a desktop computer with a 3.4 GHz processor and Raspberry Pi with a 700 MHz processor.

The initial processing of the symbolic scan cycle is shown in Figure 6.9(a). For all cases, this step requires less than $22ms$. Once TSV creates the initial PLC program models, it starts building the temporal execution graph, which is the main source of overhead. Figure 6.9(b) shows how long TSV needs to complete the graph generation phase. The majority of time in this phase is spent performing recursive exploration of the TEG to set concrete values for atomic propositions. A complete graph generation for 14 input-output scans takes 2 and 17 minutes on a desktop computer and Raspberry Pi respectively. However, as expected, trimming the analysis horizon limit to 10 reduces the graph generation time requirement significantly—down to $<$ 10 seconds on a desktop computer and 1 minute on Raspberry Pi. Figure 6.9(c) shows the corresponding state space sizes for the generated graphs. The reported numbers, only $4K$ states for a full 14 horizon analysis, proves the effectiveness of the usage of symbolic execution at reducing the state space size.

Figure 6.10 shows a sample generated execution graph for the Assembly Way case study with a model checking bound of 4. The safety requirement included two atomic propositions a and b. Thus, each state is assigned with a pair of concrete atomic propositions, and the state transitions are labeled with the path predicates as Boolean expressions in infix order. For readability purposes, we did not include the symbolic variables and their values in each state. The atomic propositions are both true regardless of the input values in states $S0$, $S1$, $S2$, and $S3$.

[4]We are currently working with several parties to obtain a disassembled copy of Stuxnet's PLC code.

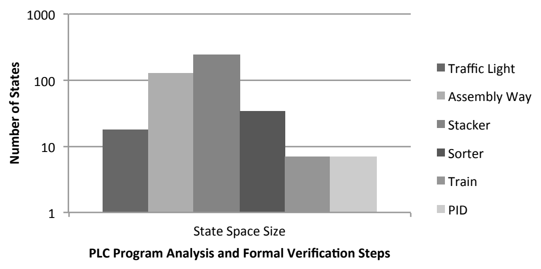


Figure 6.12. State Space Size for All Case Studies on Raspberry Pi.

However, the input values affect the atomic propositions starting in state $S3$. Out of $S3$'s four possible children, $|\{a, b\}|^2 = 4$, three have been created. Only the path condition for $\langle a : 0\ b : 0\rangle$ was not satisfiable.

TSV runs the symbolic model checking engine on the refined and atomic proposition-level abstract temporal execution graph. Figure 6.9(d) shows the run times to translate the abstract TEG into the model checker's syntax, which is not a significant source of overhead. Figure 6.9(e) shows the time requirement results for the symbolic model verification that takes no more than 10 and 90 seconds, on the desktop and Raspberry Pi respectively. In summation, the total average overheads of less than three minutes for checking with bound 10 are within reason for an analysis that is only executed once when new code is uploaded. Of course, in the case of malicious code uploading, this bound does not affect productivity, as safety checks are done independently of plant execution under the previous, legitimate code.

We ran the same experiments for all of our case studies. Figure 6.11 shows how much each analysis step contributes to verification for each case study on the Raspberry Pi with bound 6. Requirements for each step vary due to different factors. The costliest test case for symbolic execution was the AssemblyWay, which explored the most feasible paths. The single costliest operation was construction of the TEG for the train interlocking. This was caused by checking the feasibility of very large path predicates in the symbolic scan cycle. Despite the variance between use cases, it is clear that the net overhead is within reasonable bounds for all case studies. Figure 6.12 shows the state space cardinality for the generated temporal execution graphs for the case studies. It is noteworthy that there is not a direct correlation between the state space size and the overall analysis time requirement, e.g., the Train case study results in the smallest state space and yet requires the largest amount of time to finish the overall analysis.

6.5.5 Scalability

To make sure that TSV can be used for real-world PLC code verifications, it is crucial that it can handle safety properties of realistic sizes, i.e., number of atomic propositions, efficiently. To that end, we investigated typical and frequently-used linear temporal logic-based software specification formula[5] [94], where the largest predicate includes 5 atomic propositions. Figure 6.13 shows the results of our experiments with TSV that can handle requirement predicates with

[5]`http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`.

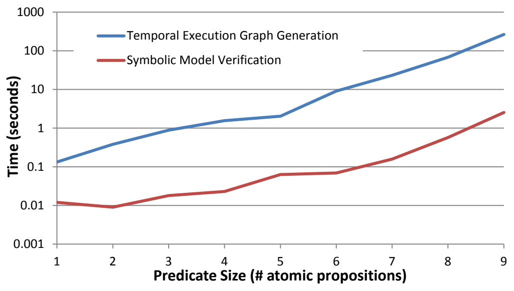


Figure 6.13. Scalability Analysis for Various Predicates

9 atomic propositions within approximately 2 minutes on average. It is noteworthy that handling additional safety properties only requires rerunning the atomic proposition value concretization on the temporal execution graph. Consequently, the time requirement to process every new security predicate is often negligible because the execution graph generation is the dominant factor in TSV's overall performance overhead (see Section 6.5.4).

6.6 Related Work

Table 6.1. Comparison of analyzed features with related work. Related approaches are abbreviated: SAT=SAT Solving, Thm=Theorem Proving, Mod=Model Checking

	Approach	**Boolean Logic**	**Enum**	**Numeric**	**Cond. Branching**	**Function Blocks**	**MCR**	**Nested Logic**	**Timers**	**Counters**	**Pointers**	**Data Blocks**	**Edge Detection**
Park et al. [41]	SAT	✓											
Groote et al. [36]	SAT	✓							✓				
Homer [42]	Thm	✓	✓	✓	✓								
Biha [95]	Thm	✓	✓	✓	✓				✓				
SABOT [96]	Mod	✓							✓				
Canet et al. [75]	Mod	✓	✓		✓								
TSV	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

We now review several previous approaches to safety verification of PLC software. The set of approaches reviewed here represent the most applicable in terms of ability to run directly on PLC code without requiring engineers to author an additional high-level system model. As shown in Table 6.1, our approach can check more features than any previous approach to PLC analysis. Existing tools for binary analysis of general purpose programs are omitted as they do not handle PLC architectural traits like multi-indexed memories.

The most basic approaches are those using SAT-based model checking. Park et al. [41] handle only Boolean logic. This had the advantage of being able to analyze larger sequence-based control systems, but is only narrowly applicable. Groote et al. [36] employs a similar technique, but is able to handle timers by modeling the exact wall clock execution time. This assumes that the approximate time taken for each scan cycle is known, and fails if scan cycle times vary too greatly depending on input. An improved handling of timers can be found in SABOT [96], which models their termination as a nondeterministic Boolean value. Additionally, TSV's improvements over SABOT allow for virtually all PLC programs to be analyzed, as opposed to exclusively Boolean variables and timers, which SABOT is limited to.

The two theorem proving based approaches [42, 95] handle numerical instructions, but not do not implement rules for overflow checks or mixed bit vector and integer arithmetic. The model checking approach used by Canet et al. [75], uses the same modeling as TSV for conditional branches, but does not implement numerical instructions, which lead to state space explosion. Our use of symbolic execution eliminates this explosion problem. TSV's ability to handle more PLC features (in most cases all available features) than previous work is thanks to the use of ILIL, which reduces side effects that would otherwise require many high-level modeling rules into a small set of low-level primitives.

We now review some representative past efforts at securing control systems. Stouffer et al. [97] present a series of NIST guideline security architectures for the industrial control systems that cover supervisory control and data acquisition systems, distributed control systems, and PLCs. Such guidelines are also used in the energy industry [98, 87]. It has, however, been argued that compliance with these standards can lead to a false sense of security [99, 11].

There have also been efforts to build novel security mechanisms for control systems. Mohan et al. [100] introduced a monitor that dynamically checks the safety of plant behavior. A similar approach using model based intrusion detection was proposed in [101]. Goble [102] introduce mathematical analysis techniques to evaluate various aspects, such as safety and reliability, of a given control system including the PLC devices quantitatively. However, the proposed solution focuses mainly on accidental failures and does not investigate intentionally malicious actions.

Compared with existing binary analysis tools, TSV is more apt for verifying temporal properties. For example, platforms such as BitBlaze [90], are aimed mainly at comparing binary programs, identifying malicious behavior, and exploit generation. Additionally, compared with the existing work combining symbolic execution and model checking to reduce state space explosion, TSV is the only solution enabling binary-level analysis.

PLC vendors themselves have included some rudimentary security measures into their solutions. Based on market data by Schwartz et al.[83], we studied the security measures used by PLCs accounting for 74% of market share. This included PLCs from Siemens (31%), Rockwell (22%), Mitsubishi Electric (13%), and Schneider Electric (8%). We found that all four vendors use only password authorization, typically with a single privilege level. Furthermore, password authentication measure can be disabled in all four systems. Additionally, certain Siemens systems use client-side authentication. This allows the attacker to completely bypass authentication by implementing his own client for uploading malicious code.

Chapter 7

Dynamic Specification-based Behavioral Enforcement

In the previous chapter, we evaluated methods of statically (in a finite number of steps) analyzing PLC programs for safe or malicious behaviors. This testing was based on a specification formulated in a temporal logic (linear temporal logic). The advantages of this technique are early detection of potential malicious behaviors, and zero run time overhead. On the other hand, it did increase overheads into the process development life cycle. While these overheads are on the order of minutes, this could add up with processes that undergo significant number of test and revision cycles. Furthermore, there is a limitation in the complexity of the system that can be analyzed. PLC programs take potentially infinitely many inputs. Furthermore, some control systems may undergo more than some small constant bound of unique state transitions in a row (for example, systems that use large counters). Thus, in this chapter, we examine a different approach *dynamic enforcement monitoring*. Dynamic enforcement monitors are known to be strictly more powerful than static monitors [29] in terms of the set of policies they can enforce. However, they introduce run time performance overheads, and offer little advanced warning of malicious behaviors.

We now present C^2 [1], a mediator for the usage of electromechanical control system devices. As opposed to existing control system access control policies, C^2 mediates the usage of the most sensitive resource in a control system: *the physical devices themselves.* Previous approaches have been based on mediating access to logical resources, files, ports, etc. However, these act as mere proxies for the physical system. Instead, C^2 directly mediates control signals sent by human operators and embedded controllers to motors, valves, heating elements, and other devices.

To achieve this, C^2 performs stateful mediation. Instead of making decisions based on the user that initiated an operation, C^2 checks whether its policy allows an operation depending on the state of the plant around the time the operation was issued. This has a number of advantages.

[1] C^2 is a Controller Controller.

First, it allows process engineers, who are the primary experts in the control system's behavior, to write the policy. Second, unlike access control policies, which base privilege levels on the perceived trustworthiness of users, C^2's policy can directly state what behaviors should be allowed to ensure the safety of physical machinery and assets.

In designing and evaluating C^2, we address two basic, novel issues for the mediation of physical systems.

1. *How should a policy for physical behavior be specified?* To answer this question, we define a logical framework to specify allowable behavior for individual devices, and then prune this behavior based on how devices may physically conflict with each other. To encode C^2 policies, we provide a novel state space language called sslang. We then demonstrate the performance and security provided by enforcement of C^2 policies.

2. *What action should be taken if an operation is denied?* Denying intentionally malicious operations is desirable. However, simply dropping operations caused by operator or policy errors could have severe physical consequences. To help keep the physical system stable when an operation is denied, C^2 can take a number of corrective actions. We show that these actions do not lead to new unsafe behaviors in a representative set of test control systems.

Our evaluation of C^2 provides us with several additional novel insights into the mediation of control system devices. First, we find that cooperation between C^2 and embedded controllers is crucial for safe and efficient plant behavior. As such, we provide a means for automatically instrumenting embedded controller code to work with C^2. Second, the control lines connected to physical machinery are the ideal place for an enforcement monitor, as there is no loss of semantics, minimal attack surface, and complete mediation of operations regardless of their origin. Finally, we show that while C^2 policies can be enforced without significant overhead, even tighter integration with embedded controllers can eliminate performance overheads altogether.

7.1 Background and Threat Model

7.1.1 Control Systems

For our purposes, a control system is any system in which one or more electromechanical devices is operated by a computer or human[2]. Usually, these devices manipulate features of a physical substance such as temperature, position, or pH balance. More specifically, the control systems we consider will have two features: human *operators* and Programmable Logic Controllers (*PLCs*). Human operators initiate and monitor processes on a coarse time scale, while PLCs regulate processes in real time. The PLC does this with a rapidly repeating process known as a *scan cycle*. Each scan cycle consists of three steps. (*i.*) The PLC takes a set of *sensor measurements*

[2]This includes the common definitions of Supervisory Control and Data Acquisition (SCADA), and Networked Control Systems (NCS).

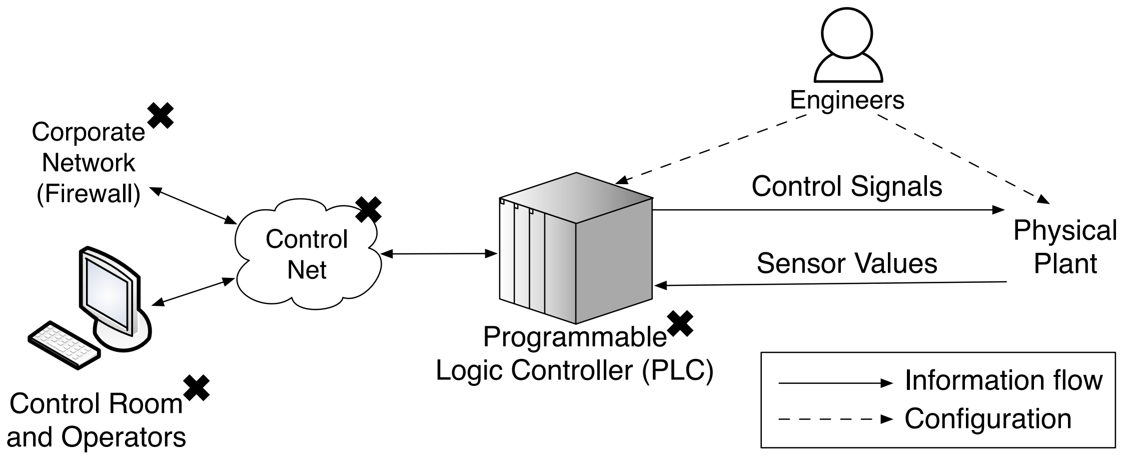


Figure 7.1. Control System Model. Entities marked by × are considered untrusted in our threat model.

of the physical process. (*ii.*) The PLC runs its software logic program to determine how the process should change according to the sensor measurements. (*iii.*) The PLC sends *control signals* to devices to actuate the calculated changes.

The basic architecture of a typical control system is shown in Figure 7.1. The PLC acts as a bridge between the *control network* used for operation and data aggregation and the *physical plant*. The PLC receives both commands and logic programs over its control room interface. Logic programs can also be directly uploaded to the PLC through a handheld programming device. In addition to the control room, the control network is often connected to a *corporate network* [103]. While often separated by a firewall or DMZ, commonly used protocols such as HTTP are often left open to allow for data export from the control network. While operators use control room interfaces to monitor the plant, another type of user *engineers* configure the PLC's logic program and physical plant. As engineers are also the experts on safe plant behavior, they are in charge of writing and uploading C^2's policy.

Plant devices can be classified as either *discrete* or *continuous*. Discrete devices have a finite state space. An example of a discrete device is a valve that is either open or shut. Continuous devices have state spaces that are effectively infinite, e.g., a valve that may be adjusted to achieve a desired flow rate. Hybrid devices, those that exhibit both discrete and continuous states, can be broken down into individual logical discrete and continuous devices for the purposes of C^2's policy.

7.1.2 Threat Model

Due to their once closed-off nature, insufficient progress has been made in securing networked control systems. One study of over a hundred control systems found that the average age of discovered vulnerabilities was nearly a year from disclosure date, and some as many as three years [9]. Arbitrary code execution vulnerabilities have been found in control room interfaces [18, 19], and embedded controllers from most major vendors [20, 17]. Compounding these vulnerabilities,

Table 7.1. C^2 Policy Summary

	Representation	**Check**	**Description**
Operations	(s_1, s_2, t)	-	A transition from device state s_1 to s_2 in time t.
Discrete Devices	(S, E)	dallow	A graph containing transitions in E between device states in S. Only the specified transitions are allowed.
Continuous Devices	$P^{\Uparrow}, P^{\Downarrow} : [s_{\bot}, s_{\top}] \rightarrow \mathbb{R}^n_+$	callow	Continuous functions describing transitions in a continuous device state space $[s_{\bot}, s_{\top}]$. Only transitions in the interval within the space are allowed.
Discrete Conflicts	$\{(s^{\mathsf{CON}}, t^{\mathsf{CON}}), (s, t) \ldots\}$	confree	A discrete device state s^{CON} will physically conflict with another device state* s after time at most t^{CON}. Conflicting transitions are not allowed.
Continuous Conflicts	$\{[s_1^{\mathsf{CON}}, s_2^{\mathsf{CON}}], (s, t) \ldots\}$	confree	A continuous device holding any state in the interval $[s_1^{\mathsf{CON}}, s_2^{\mathsf{CON}}]$ will physically conflict with another device state* s. Conflicting transitions are not allowed.
Sensor Conflicts	$\{v^{\mathsf{CON}}, s \ldots\}$	confree	The plant will conflict with some device state* s when a sensor reading has value v. Transitioning to s is not allowed when the sensor reads v.
Full System	-	check	Check that an operation is allowed for a discrete or continuous device (allow), and will not conflict with other device states or sensor values (confree).

$^*s$ may be a discrete or continuous device state.

machine controllers themselves are now being provided with Internet-friendly interfaces [27]. Many Internet-exposed control system components are locatable by banner search engines such as Shodan [86, 104]. Recently, a study using honeypot PLCs found that attempts were made to log into control-specific functionality [85]. Numerous other studies and reviews how shown that the notion of control system isolation is a myth [10, 80, 11].

Given the level of vulnerability demonstrated throughout control networks, we make no assumptions about their integrity or trustworthiness. Only C^2 itself, as described in Section 7.3, is assumed to be tamperproof. This opens an important question: *Which parties are trusted to create and modify C^2 policies?* To answer this question, we first divide control system users into two groups, *engineers* and *operators*. Engineers are responsible for implementing the physical system, PLC programs, and operational procedures for running the plant. They are trusted by necessity as they are responsible for defining C^2's policy. Operators are responsible for monitoring the plant from the control room and performing maintenance. Operators need not be trusted.

Furthermore, engineers need only be trusted for the time that they have access to C^2 in order to create and modify policies. This rules out the possibility of a disgruntled former employee using their access to mount an attack. In the highly costly Maroochy water system incident, a former engineer used his knowledge of the water system to spill thousands of gallons of sewage into public waterways [84]. Thus, only engineers actively involved with the plant should have access to C^2.

C^2 mitigates all *control channel attacks* against devices. Control channel attacks are those that issue malicious commands or control signals to directly manipulate plant devices [31]. This is in contrast to out-of-band attacks which attempt to disrupt the physical process by rebooting PLCs, flooding network connections, or tampering with control room interfaces. A third class of attack is the false data injection attack, which uses forged sensor data to cause harmful control decisions [14, 15]. C^2 can prevent any unsafe device behavior caused by a false data injection attack, but it cannot detect forged sensor data. Detecting such attacks requires improved state estimators [89, 88].

7.2 C^2 Policy

There are a number of challenges in defining a policy for the behavior of physical devices.

- **Stateful Requirements.** It is not sufficient to state only which operations are allowed on a device. For example, it may be safe to allow a speedup of 30 RPMs on a motor, but a subsequent speedup of another 30 RPMs may not be safe. Thus, all C^2 policies must be stateful.
- **Rates of Change.** The policy must not only specify which states a device is allowed to occupy, but at what rates it is allowed to transition between those states. As physical devices experiences kinetic forces such as momentum and jerk, allowing arbitrarily fast movements will lead to device wear or destruction.
- **Continuous Devices.** Some devices have state spaces that are essentially infinite, e.g., mechanical arms that can be positioned to the millimeter. It must be possible to specify the state space these devices can occupy, as well as their maximum rate of state transition.
- **Plant Trajectory.** It is not sufficient to verify that an operation is allowed in the current plant state, as many plant devices are constantly in motion. Instead, C^2 must check whether an operation will conflict with *future* states of other plant devices.

C^2's policy is summarized for reference in Table 7.1. We explore each item in the following two sections. First, we explain how to specify individual device policies to C^2. This will result in the allow predicate, defined in Section 7.2.1. We then cover policies to ensure two or more devices will not physically interfere with each other. This will result in the confree predicate, defined in Section 7.2.2.

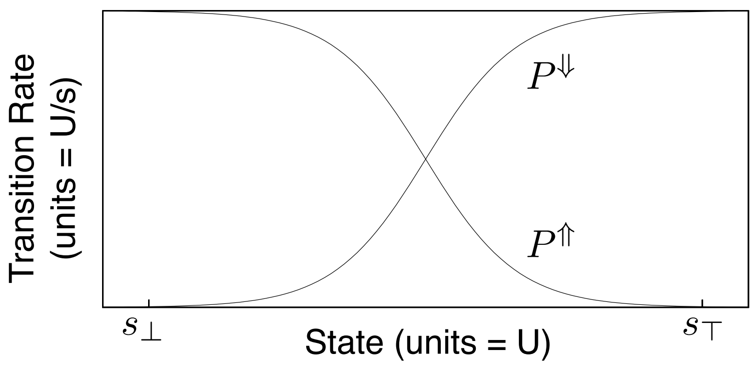


Figure 7.2. Example continuous device policy.

7.2.1 Device Policies

Policy Primitives. All security policies contain three basic primitives: *subjects*, *objects*, and *operations*. The policy then dictates whether each subject is allowed to perform each operation on each object. For the case of control systems, we replace the subject with the system *state*. Thus, C^2 determines whether to allow an operation on an object when the set of all objects is in some state. This is done by necessity as most control systems do not propagate subject information to the PLC or devices. Furthermore, the number of subjects in a control system is typically small, and most or all need to operate under identical privilege levels.

Objects are simply plant devices, and operations are commands to physically manipulate plant devices. Manipulation of a device can be modeled as a state transition from the start state s_1 to a destination state s_2 taking time t. We encode operations as triples (s_1, s_2, t). C^2's main device policy check is the $\mathsf{allow}(s_1, s_2, t)$ statement, which evaluates to true if the operation (s_1, s_2, t) is allowed under the device's policy. Policies are specified differently for devices with continuous and discrete state spaces. For simplicity, we omit the device identifier from operations and device policies.

Discrete State Devices. A device has a discrete state space if it can only take a fixed, finite number of states, e.g., a single speed motor has states $\{$*forward*, *reverse*, and *off*$\}$. Policies for devices with discrete state spaces are encoded as directed graphs $G = (S, E)$. Each node in S represents a physical state, and each edge in E represents an allowed state transition. An edge e may have a label $\ell(e)$, which specifies the minimum time allowed for the transition. If there is no minimum for a particular transition, then $\ell(e) = 0$. The check for an operation (s_1, s_2, t) on a single discrete device is defined as follows.

$$\mathsf{dallow}(s_1, s_2, t) \overset{\text{def}}{=} (s_1, s_2) \in E \wedge \ell((s_1, s_2)) \leq t$$

This states that for an operation to be allowed, the transition must be within the set of allowed transitions E, and the transition must take at least the minimum requred time.

Continuous State Devices. Policies for devices with continuous state spaces are encoded as a pair of continuous, monotonic functions $P^{\Uparrow}(s)$ and $P^{\Downarrow}(s)$ defined over an interval $[s_\perp, s_\top] \subset \mathbb{R}^n_+$, where $s_\perp$ and $s_\top$ are vectors with n entries. In practice, these functions are defined as strings containing arithmetic operators, roots, logs, reals, and typical constants like π and e. $P^{\Uparrow}(s)$ and $P^{\Downarrow}(s)$ define the maximum allowed rate of transition from state s to a higher or lower state respectively. An example continuous device policy is shown in Figure 7.2. At the bounds of the device's allowed range, both $P^{\Uparrow}$ and $P^{\Downarrow}$ go to zero. This prevents the device from making a harsh jerk when approaching a limit. Of course, this will vary according to physical device characteristics. To avoid having separate equations when either $P^{\Uparrow}$ or $P^{\Downarrow}$ may be used, we define $P^{s_1}_{s_2}$.

$$P^{s_1}_{s_2} \overset{\text{def}}{=} \begin{cases} P^{\Uparrow} & : s_1 \leq s_2 \\ P^{\Downarrow} & : s_2 < s_1 \end{cases}$$

We can now give a policy check that is analogous to the discrete case. The target state must be within the device's state space $[s_\perp, s_\top]$, and the transition must take at least the minimal amount of time given the maximum allowed rate of change. For continuous state devices, we wish to derive the minimum time t_{SAFE} in which a given state transition is allowed to occur according to $P^{s_1}_{s_2}$ for a given device. We can then check that $t_{\mathsf{SAFE}} \leq t$, where t is the transition time specified in an operation. For a transition from state s_1 to s_2, the average maximum rate of state change v_{SAFE} is given by:

$$v_{\mathsf{SAFE}} = \frac{1}{|s_1 - s_2|} \int_{s_1}^{s_2} P^{s_1}_{s_2}(s)\,\mathrm{d}s$$

Additionally, the total change d is given by $|s_1 - s_2|$. Substituting v_{SAFE} and d into $v = d/t$ gives us

$$t_{\mathsf{SAFE}} = \frac{|s_1 - s_2|^2}{\int_{s_1}^{s_2} P^{s_1}_{s_2}(s)\,\mathrm{d}s}.$$

Finally, we can put this into predicate form.

$$\mathsf{allow}(s_1, s_2, t) \overset{\text{def}}{=} s_1, s_2 \in [s_\perp, s_\top] \wedge \frac{|s_1 - s_2|^2}{\int_{s_1}^{s_2} P^{s_1}_{s_2}(s)\,\mathrm{d}s} \leq t$$

For brevity's sake, we define a single statement for both the discrete check dallow and the continuous check callow.

$$\mathsf{allow}(s_1, s_2, t) \overset{\text{def}}{=} \begin{cases} \mathsf{dallow}(s_1, s_2, t) & : s_1, s_2 \text{ discrete} \\ \mathsf{callow}(s_1, s_2, t) & : s_1, s_2 \text{ continuous} \end{cases}$$

7.2.2 Combining Devices

Device Conflicts. Thus far, we have only considered policies for individual device behaviors. This leaves open the possibility that two or more devices behaving safely on their own may become unsafe when combined, e.g., by attempting to occupy the same space at the same time. We call these unsafe state combinations, *conflicts*. A conflict is a state assignment to a subset of physical devices. At no time during plant operation should any set of devices take the set of states specified in a conflict. In this section, we show how process engineers may express such conflicts to C^2.

C^2 maintains a set of all declared conflicts C. Each conflict $c \in C$ is a list containing one of two kinds of entries: one for discrete devices, and one for continuous devices. Discrete device entries are pairs $(s^{\mathsf{CON}}, t^{\mathsf{CON}})$, where s^{CON} is a device state, and t^{CON} is the minimum time for any incoming transition to s^{CON} to conflict with any other device in c. Continuous device entries are intervals $[s_1^{\mathsf{CON}}, s_2^{\mathsf{CON}}] \subseteq [s_\bot, s_\top]$. For convenience of implementation and verification, a conflict tuple c may contain at most one entry for each device, though a device may appear in arbitrarily many conflict tuples in C. We use the notation $s \in c$ to mean that a device state s appears in a conflict c. For succinctness, we use the same notation for both the discrete and continuous cases, defined as follows.

$$s \in c \overset{\text{def}}{=} \begin{cases} (s, t^{\mathsf{CON}}) \in c & : s \text{ discrete} \\ s \in [s_1, s_2] \in c & : s \text{ continuous} \end{cases}$$

Additionally, we define a subset of conflicts in C that contain a state s.

$$C[s] \overset{\text{def}}{=} \{c \mid c \in C \land s \in c\}$$

Having now defined the notion of a conflict, we can describe C^2's approach to conflict checking operations.

Conflict Checks. The problem of checking whether an operation might cause a conflict is not as simple as it might first seem. It is not sufficient to verify that a state transition will be conflict free given the current states of other devices. Instead, it must be verified that at no point during a state transition will a device conflict with other devices that are also in transition. This leads us to the following, informal definition of a conflict check.

> **Conflict Check.** *Given the current trajectory of all devices in the system, if a potential state transition would cause the plant to assume a state assignment in some conflict tuple, the conflict check returns true.*

Most important to this definition is the idea of the physical system's *trajectory*. This notion captures the fact that conflicts may occur for infinitesimal moments as multiple devices are in various stages of state transitions. The physical system's trajectory T is the set of all device trajectories. An individual device trajectory is a pair (s, τ), where s is the device's current state and τ describes any state transition the device may be in. If the device is not in a state transition then $\tau = \bullet$. If the device is in transition, then τ is a triple $(s_f, t, t_{\mathsf{START}})$ where s_f is the destination state, t is the transition's duration, and t_{START} is the time at which the transition began. Here, t is from the operation (s_1, s_2, t) that initiated the state transition. To assist in defining the conflict check, we define $\mathsf{rem}(\tau)$ for the amount of time remaining in a device's transition given its trajectory.

$$\mathsf{rem}(s_f, t, t_{\mathsf{START}}) \overset{\text{def}}{=} t + t_{\mathsf{START}} - \text{current time}$$

Given the notions of a conflict c and trajectory (s, τ), we can now ask whether an operation (s_1, s_2, t) will cause the conflict specified by c given (s, τ) for all other devices. The definition breaks down into two conditions that result in a conflict. First, the proposed state transition may conflict with a state that a transitioning device has not fully left (the LHS in the disjunction), or it may conflict with the destination state of a transitioning device (the RHS in the disjunction). The exact definition is as follows.

$$\begin{aligned}\mathsf{conflict}(t, s, \bullet, c) &\overset{\text{def}}{=} \ s \in c\\ \mathsf{conflict}(t, s, \tau, c) &\overset{\text{def}}{=} [s \in c \wedge t < \mathsf{rem}(\tau)] \vee [\tau.s_f \in c]\end{aligned}$$

For discrete devices, the t in the previous definition is replaced by t^{CON}, from c.

Sensor Conflicts. To this point, we have addressed defining and checking for conflicts between devices. This is sufficient to prevent every instance in which two devices will physically conflict with each other *independently of sensor measurements.* There are however, cases where a device may conflict with some physical aspect of the system, e.g., temperature, that may cause physical wear or damage. To handle these cases, we define a second type of conflict, the *sensor conflict.*

A sensor conflict c_s is a pair (v^{CON}, s) where s is a device state, and v^{CON} is a sensor measurement. Sensor conflicts are included in the conflict set C as part of the policy. The current value of each sensor is kept in the system trajectory T, as pairs $(v, \bullet)$. The sensor value v^{CON} may be a discrete state, or an n-dimensional real-valued interval. Thus, the definitions of $v^{\mathsf{CON}} \in c_s$ and $s \in c_s$ are similar to that of $s \in c$. We extend $\mathsf{conflict}$ to handle sensor conflicts as follows.

$$\mathsf{conflict}(t, v, \bullet, c_s) \overset{\text{def}}{=} v \in c_s$$

Given checks for both device and sensor conflicts, we can now define the following check to determine if an operation is conflict free under the current system trajectory.

$$\mathsf{confree}(s_1, s_2, t) \overset{\text{def}}{=} \forall c \in C[s_2], (s, \tau) \in T,\ \neg\mathsf{conflict}(t, s, \tau, c)$$

Finally, we define C^2's complete policy check for an operation o.

$$\mathsf{check}(o) \overset{\text{def}}{=} \mathsf{allow}(o) \wedge \mathsf{confree}(o)$$

The device declarations G, P^{s1}_{s2}, and conflicts C are declared in C^2's policy language called sslang (state space language).

As a final note on C^2's policy, we mention that conflicts may also be caused by out-of-band, e.g., DoS, attacks. If a PLC or control channel can be suppressed, a conflict may arise due to some action the PLC did *not* take. C^2 still detects such conflicts, but requires additional policy statements to return the plant to a safe state. This could be as simple as an annotation to

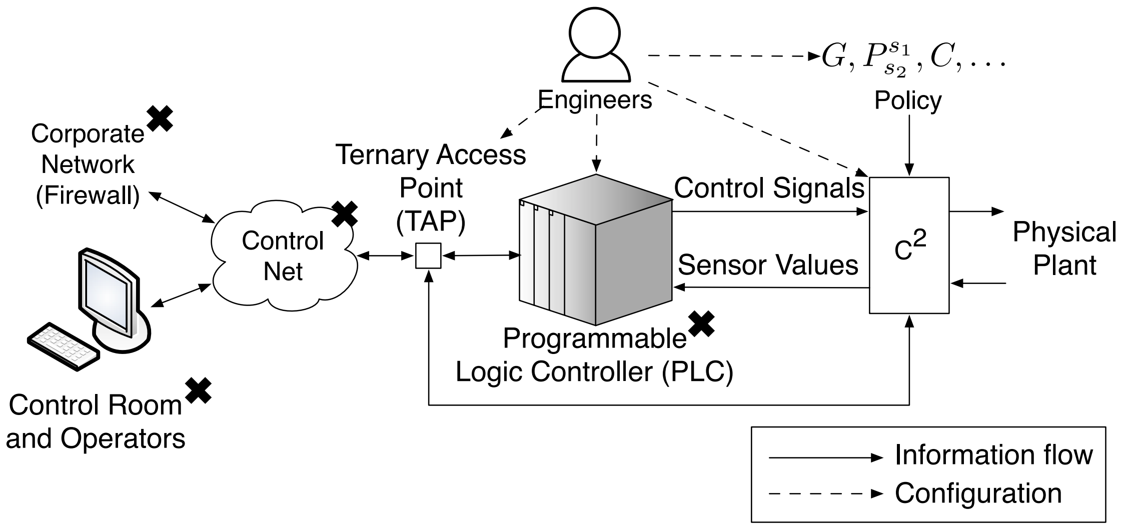

Engineers
$G, P_{s_2}^{s_1}, C, \ldots$
Policy
Corporate Network (Firewall)
Ternary Access Point (TAP)
Control Net
Control Signals
Sensor Values
C^2
Physical Plant
Programmable Logic Controller (PLC)
Control Room and Operators
Information flow
Configuration

Figure 7.3. Figure 7.1 modified to include C^2. Untrusted entities are marked by $\times$.

each conflict tuple stating which device should have its state changed to resolve the conflict. As most of the known real-world attacks on control systems involved the issuing of malicious control signals [84, 7, 4, 26, 8] and not out-of-band attacks, we defer conflict extensions to future work.

7.3 C^2 Policy Enforcement

We now turn to the issue of enforcing the policy described in the previous section. As will be seen, of paramount importance is the action taken when an operation is denied. C^2 enforces its policy from within the control system. Every adjustment the PLC attempts to make to a physical device is checked against the policy. As shown in Figure 7.3, C^2 sits at two positions in the control system. The enforcement monitor itself sits between the PLC and physical plant, where it intercepts both device-bound control signals and sensor measurements. A second part, the Ternary Access Point (TAP), allows C^2 to both monitor and inject traffic between the control room and PLC. The TAP is used to communicate with the operators or PLC if an operation is denied.

7.3.1 Deny Disciplines

C^2 checks all control signals against its policy. If a control operation is allowed then it is forwarded to the correct device. *It is not so clear what is the right course of action if an operation is denied.* Simply dropping a denied operation could be as dangerous as allowing it. For example, a control system operator could have issued a command to accelerate a motor to a velocity only slightly above that allowed by the policy. However, dropping the operation will result in no acceleration, which may or may not be preferable depending on the circumstances. To allow for better handling of denied requests, C^2 implements several possible *deny disciplines.* They are as follows.

1. **Drop.** The most basic action that can be taken is to drop the operation. This is sufficient if the policy, PLC, and system operators are assumed to be error free, and if the plant will not become uncontrollable when an operation is denied. Under any other circumstances, another discipline should be selected.
2. **Approximate.** For continuous state spaces devices, it may be possible to partially fulfil an operation that is beyond the allowed interval $[s_\perp, s_\top]$. If the Approximate discipline is selected, C^2 will calculate the trajectory the denied operation would have caused, and truncate it to the allowed interval. Thus, it achieves the maximum allowable change in the same trajectory as the original operation. A control signal is then issued to execute the truncated operation.
3. **Retry.** In some cases, an operation may have failed only because the plant was temporarily in a conflicting state. To handle such cases, C^2 can check if a violation still holds before the PLC begins its next scan cycle. This is useful if an operation appeared safe to the PLC, but some transient conflicting state caused C^2 to deny it. If the transient conflict has passed in time for the retry, the operation will be executed. For the remainder of this paper, we assume that one half scan cycle time is used for a retry delay.
4. **Notify.** This discipline uses the TAP to notify the PLC that a previous operation was denied. This is important if the PLC code makes assumptions about the states of plant devices, and could suffer errors due to incorrect assumptions. Notify can be combined with any of the other three disciplines, in which case, Notify runs after the other discipline. This allows Notify to inform the PLC of the exact device state calculated by the other discipline. We describe a simple method for automatically instrumenting PLC code to handle notifications in the following section (7.3.2).

It should be emphasized that the above disciplines are only necessary when the controller or human operator initiates an operation that is not allowed under the policy. Under normal circumstances, this should never happen. An operation will only be denied (*i.*) if it was malicious or erroneous, or (*ii.*) if the policy was overly restrictive. In case (*i*) denying the operation is the desired behavior. If the request is erroneous, the Approximate, Retry, or Notify disciplines serve as corrective actions. Case (*ii*) could itself be potentially costly or damaging if the denied request was critical for plant operation. For this reason, testing should be done to ensure that no desired operations are denied.

7.3.2 PLC Code Instrumentation

The **Notify** deny discipline informs the PLC when a device operation was denied. By default, PLC programs are not able to handle such information, and it would place additional burden on engineers to write code that accepts it. Given that economic constraints place the main focus on functionality over security, Notify-compatible PLC code is more feasible if the code can be automatically instrumented. This is possible given the presence of the TAP, which can be configured to modify PLC-bound code to handle Notify messages.

The following code snippet can be prepended to a PLC program to make it Notification-aware. The symbols `"Noti"` and `"Bool"` resolve to memory address that are mapped to inputs from the TAP. The addresses `ID32` and `ID36` contain pointers to the actual device state value, and the PLC's copy of the device state respectively. These are also mapped to TAP inputs.

```
1. A "Noti"          8. BoolN:
2. JCN Rest          9. CLR
3. A "Bool"         10. A I [ID32]
4. JC  BoolN        11. = Q [ID36]
5. L I [ID32]       12. Rest:
6. T Q [ID36]       13. RESET
7. JU Rest          ... rest of program ...
```

The first two lines check for a notification, and skip to the rest of the PLC code if none is available. Line 3 determines if a Boolean or numerical variable needs to be updated. Lines 5 and 6 update the value of a numerical PLC variable, and lines 8-11 update the value of a Boolean PLC variable. Line 12 resets the PLC's status flags before beginning the rest of the scan cycle. This code is compatible with PLCs representing over 50% of world market share, and requires only slight alteration for others. (See Section 7.4.1.3).

As a final note, we consider than an attacker may attempt to cause PLC errors by spoofing notification messages and sending them from the control network. To prevent this, the TAP will drop any Notify messages from the control network. Thus, only notifications originating from C^2 are handled by the PLC. This is not a problem, as no other machine on the control network is aware of the Notify discipline or when operations are denied.

7.4 Evaluation

In evaluating C^2 for its applicability to control systems, three important properties must be considered.

Safety. C^2 must not introduce any new unsafe behaviors into a control system. Under normal operation, this is not a concern, as all operations are executed as dictated by operators and PLCs. However, if an operation is denied, C^2's deny discipline will exert a degree of control over the plant. This control should not cause unsafe plant behavior.

Security. To be an effective enforcement monitor, C^2 must provide certain guarantees about its mediation. Furthermore, these guarantees must hold under all attackers allowed by our threat model in Section 7.1.2. If this is the case, then C^2 provides a stronger level of guarantees than existing safety solutions that only handle failures.

Performance. Control systems must meet real time deadlines. Thus, C^2 must impose minimal overhead to PLC scan cycle time. Similarly, PLC code that has been instrumented with notification handling must also provide little overhead in the common case (no denial), and not overshoot real-time deadlines in the case of a denied request.

Before evaluating C^2 for each of these properties, we introduce our testbed consisting of six example control systems, their policies, and our implementation of C^2.

7.4.1 Testbed

Our evaluation testbed consists of three parts: (*i.*) the code and requirements for six control systems, (*ii.*) C^2 policies for each system, and (*iii.*) our implementation of C^2. We cover each in the following three sections.

7.4.1.1 Test Systems

We evaluate C^2 on the following test systems. The saw mill and sorting arm are taken from the TrySim physical simulation platform used for control system testing[3]. The motor control, neutralization process, assembly line, and traffic light are test cases used in [105]. For each system, the high-level requirements and PLC code are used in the experiments.

- **Motor Control.** This simple system uses a PLC to drive a bi-directional motor from a button panel. The motor can be set to FORWARD, REVERSE, and OFF, with time delays needed for direction changes.
- **Saw Mill.** A mill in which a board is guided across a rotating saw until reaching a light barrier. Boards reaching the barrier are moved by the nearest of three hooks on an overhead chain.
- **Sorting Arm.** A 3-jointed arm, capable of full motion through a 3D hemisphere, collects items from a conveyor and sorts them into bins.
- **pH Neutralization.** A substance of unknown acidity is injected into a tank and mixed with a pH neutralizer at a specified temperature. When the desired pH value is reached, the mixture is passed to another process.
- **Assembly Line.** A conveyor belt moves objects down an assembly line with two stations. At each station, an arm is lowered to secure the object in place. An additional arm then moves to the station to assemble a part.
- **Traffic Light.** A four-way intersection is governed by a timer-based traffic light. Each light color in each direction is controlled by a separate variable.

7.4.1.2 Test Policies

We implemented policies for each test system in sslang. (See Figure 7.4 for an example policy.) Each policy was based on the requirements and PLC code for the test system, both of which

[3]`http://www.trysim.de`

```
// Discrete Device definitions.

// The board loader. (In this case, the loader
// is controlled by PLC output wire Q0.1.)
ddev Loader at 'Q0.1';
edge Loader ON OFF;
edge Loader OFF ON;

// The cutting saw, chain, and transporter are
// similarly defined Boolean devices.

// ... snip ...

// Sensor definitions.

// A light barrier that detects when a board
// has completed passing through the saw.
sensor LightBar at 'I0.0';

// Conflicts.

// The saw should be stopped when the board
// is completed, triggering the light barrier.
conflict { dcon(Saw, ON, 0.0) sdcon(LightBar, ON) };

// The loader should not load another board until
// the previous one has been removed. This means
// that the saw should not be running, and
// the light barrier should be off, meaning there is
// currently no board.
conflict { dcon(Loader, ON, 0.0) dcon(Saw, ON, 0.0) };
conflict { dcon(Loader, ON, 0.0) sdcon(LightBar, ON) };
```

Figure 7.4. `sawmill.sslang` (excerpt)

would be available to a process engineer. While it would be prohibitively long to list all allowed device operations and conflicts here, the salient safety properties enforced by our test policies are as follows.

- **Motor Control.** (*a.*) The motor must switch off before changing directions. (*b.*) At least two seconds must elapse between direction changes for spin down.
- **Saw Mill.** (*c.*) The saw should not switch ON without a board loaded. (*d.*) The loader should not attempt to load a board until the current board is complete.
- **Sorting Arm.** (*e.*) The arm should not leave the quarter hemisphere needed to move objects from the buffer to the bins. (*f.*) The arm should only release an object when positioned over a sorting bin.
- **pH Neutralization.** (*g.*) A product of any pH value besides that specified should not be passed to the next stage. (*h.*) The heating element should be off when the product is above the specified temperature.

Table 7.2. Safety Test Results. Columns headed by "Not." combine the **Notify** deny discipline.

Violated Property		**Deny Discipline**					
		Drop		Approx.		Retry	
		-	Not.	-	Not.	-	Not.
Motor Control	*a.*						
	b.						
Saw Mill	*c.*	**d**		**d**		**d**	
	d.	**d**	**c**				
Sorting Arm	*e.*	**d**		**d**	**c**	**d**	
	f.	**u**		**u**		**u**	
Neutralization	*g.*	**d**		**d**		**d**	
	h.						
Assembly Line	*i.*						
	j.						
Traffic Light	*k.*	**d**	**c**	**d**	**c**	**d**	**c**
	l.	**d**	**c**	**d**	**c**	**d**	**c**

- **Assembly Line.** (*i.*) A station arm should only lower when a light barrier signals an item arrival. (*j.*) The conveyor belt motors must never rotate counterclockwise, which would reverse the belt.
- **Traffic Light.** (*k.*) Green lights for intersecting lanes should not be on simultaneously. (*l.*) Yellow lights must remain on for at least 4 seconds before transitioning to red.

7.4.1.3 Implementation

We implemented C^2 in 2,240 lines of C++ and Java for compatibility with different libraries.The ANTLR3 parser generator [106] was used for sslang policies, and GNU Flex and Bison for PLC code. PLC code instrumentation is supported for the IEC Instruction List (IL) standard [61], as well as the Siemens Statement List (STL/AWL) variant [107]. Vendors supporting these two languages account for over 50% of the world PLC market share [83]. To optimize policy checks, all device policies, conflicts, and device trajectories were pinned to memory to prevent overhead due to swapping. All experiments were run on a single core at 2.8 GHz, except for the performance tests performed on a Raspberry Pi embedded computer running at 700 MHz.

7.4.2 Safety

The deny disciplines described in Section 7.3.1 should not introduce any new unsafe behaviors into a control system. To evaluate the effects of deny disciplines on system behavior, we inject violating operations into each test system, and observe how the different deny disciplines affect the plant. After each deny discipline is applied, we manually inspect the state of the plant, and classify it in one of four ways: (**u**) unsafe, (**d**) safe, but diverged from its normal course of action, (**c**) safe, and converges back to normal behavior after a brief detour, and (**n**) no change from behavior without violation. The state of each plant before violation was chosen at random from the set of valid plant states.

The results of injecting operations that violated our 12 safety properties (a–l) are shown in Table 7.2. As no change (**n**) was the common case, we represent it with a blank cell. The only unsafe behavior was caused by violation of policy f. for the sorting arm when notifications were disabled. If the PLC is not notified that the item was not released, then it can cause the arm to slam the item into another item when attempting to pick it up. There was no unsafe behavior with notifications enabled.

Another common issue due to lack of notifications was a safe yet divergent behavior (**d**). Because the PLC's view of device state did not match the actual state, it drove the plant indefinitely into a safe, but undesired state. In practice this would require an operator to guide the process back into its proper sequence. In several cases, the plant required several scan cycles to converge (**c**) to a desired state. For example, the Approximate discipline underwent the maximum allowed movement within the quarter hemisphere after the violation of property e. The PLC then guided the arm back to its proper destination. Given that these undesired behaviors was caused by a lack of notifications, we conclude that communication between C^2 and PLCs is crucial for the enforcement of physical device policies.

7.4.3 Security

To better understand the security of C^2's architecture we examine it for the three Anderson reference monitor properties [108]. As will be seen, we defend our claim that C^2 mitigates all control channel attacks against devices, and only requires trust in process engineers and physical sensors.

Complete Mediation. Complete mediation refers to a monitor's ability to check every potentially sensitive operation. Due to its placement on the control lines leaving a PLC, C^2 achieves complete mediation of all device-bound control signals. A malicious control signal may be due to operator actions, crafted sensor input, or a compromised PLC. In either case, the operation is denied. For the sake of checking *sensor conflicts*, C^2 is only vulnerable to the direct compromise of the sensors themselves. This is however, not an issue of mediation, but one of sensor integrity.

Tamperproofness. Adversaries may attempt to compromise C^2 through the TAP, which must implement the PLC's protocol stack for control room communications. However, the protocol between the TAP and C^2 monitor is extremely simple to implement, leaving C^2 protected in the event of a TAP compromise. Privilege separation [109] of the control network protocol stack from the rest of the TAP logic could prevent the suppression of notifications by a compromised TAP. Under such circumstances, C^2 would perform correctly under attack by any remote adversary. If physical tampering is a concern, a hardware root of trust may be used [110].

Verifiability. An enforcement monitor should be simple enough to be thoroughly tested, and preferably formally verified. While the use of procedural languages and autogenerated parser code

Table 7.3. C^2 policy check times on an embedded Raspberry Pi computer.

System	Time (ms)	95% C.I.	% Overhead
Motor	0.168	[0.163, 0.173]	3.368
Saw	0.141	[0.137, 0.144]	2.815
Sorting	0.126	[0.123, 0.129]	2.514
Neut.	0.164	[0.159, 0.169]	3.282
Assembly	0.114	[0.108, 0.120]	2.278
Traffic	0.182	[0.162, 0.202]	3.637

makes formal verification of our implementation difficult, C^2 correctly identified the violations demonstrated in the previous section, and did not exhibit any false positive denials. While our prototype demonstrates correct behavior on all test cases, an implementation in a functional language like ML or Haskell would make full formal verification possible.

7.4.4 Performance

7.4.4.1 C^2 Performance

Most commodity PLCs support scan cycle times as low as a few milliseconds. Any additional run time monitoring must not impose substantial overhead. To examine the overhead introduced into a system by C^2, we record the time necessary to check individual operations for each of our six test systems. As some control environments may require a low power or fully solid state solution, we perform these tests on a Raspberry Pi embedded computer with a 700 MHz processor.

The results are shown in Table 7.3. As can be seen, most checks require between 0.1 and 0.2 ms. The percentage overhead for a 5 ms scan cycle is shown in the rightmost column. While these overheads are sufficiently low for many applications, we note that they may be completely avoided by a tighter coupling of C^2 and the PLC. The time to execute the scan cycle is dominated by buffering of sensor inputs and control signal outputs. If the policy checks were performed as each control output was calculated, but before buffering of output signals, the overheads shown in Table 7.3 would be completely removed from the scan cycle time.

7.4.4.2 Instrumented PLC Code Performance

In addition to the performance overheads introduced by policy checks, it is possible that PLC code instrumented for notifications may increase scan cycle time. In the common case, when no operations are denied, the check for new notifications requires only two instructions. If however, an operation is denied, the notification handling routine will be executed. For the 11 instructions shown in Section 7.3.2, this would constitute less than a 1% overhead to the scan cycle time.

Chapter 8

Conclusions

This dissertation has explored the use of physical behavioral specifications in both attacking and defending sequential control systems. This was done for a practical reason. Modern control systems integrate physical machinery and processes with large networks of exposed and vulnerable computers. Obtaining 100% assurance about the security of those computers and networks simply is not practical. However, it is also not necessary. Given that the most critical assets in modern control systems are the physical processes themselves, we can settle on guaranteeing that they will not misbehave, regardless of the security state of the surrounding computer systems. This contested space model of security is emerging as a promising foundation towards guaranteeing the safety and efficiency of control systems with large, vulnerable attack surfaces. It was the goal of this thesis to demonstrate that physical specification-based enforcement is a practical and effective realization of this foundation.

As the center point for this work, we have focused on Programmable Logic Controllers (PLCs) for their unique role in control systems. This choice may seem somewhat arbitrary as first, as malicious actions against a control system can originate anywhere throughout the system, including with malicious human operators. However, the PLC is uniquely positioned to be the only computational device in the system capable of directly driving actuators to physical devices. Thus, if the PLC is secured, safe behavior can be guaranteed. (Assuming the PLC will not then execute malicious commands on behalf of malicious actors upstream.) Similarly, if the PLC is compromised, the attacker has free reign over the system, and any upstream security measures become futile.

The approaches outlined above all center around the idea that impact on the most critical assets should be considered first in any security effort. The work on attack trees gave direction to the often ad-hoc processes of penetration testing by starting with the outcomes of attacks. This ensures that the discovered vulnerabilities can be linked to more semantically meaningful end results, and their impacts assessed. The work on automated attacks against PLCs using SABOT allowed the adversary to reason about the physical behavior of the system, without having to know how the PLC actually drives that physical behavior in code. Based only on behavior, a system-specific attack payload was instantiated. The Trusted Safety Verifier (TSV) attempted to keep PLCs free of bad code by statically verifying that all PLC-bound code satisfied a set of engineer supplied safety properties. On the other hand the controller controller (C^2), monitored the PLC's behavior to check for disallowed device usage. In all cases, the sufficiency of surrounding firewalls, access control policies, virus scanners, and other perimeter defenses was orthogonal to the security of the proposed mechanisms.

The specification-based security approach has a number of other advantages towards ensuring safe control system behavior. It represents a significant reduction in the *Trusted Computing Base* (TCB) of the control system. In modern Supervisory Control and Data Acquisition Systems, the TCB includes not only the PLC and connected computers, but potentially hundreds of other computers, including those in corporate networks [10]. Often the compromise of any one of these components can allow an adversary to eventually reach a critical physical component. By focusing on physical behaviors, we can reduce the TCB to the PLC and some other minimal set of components. In the case of TSV, the verifier itself, e.g., a Raspberry Pi module and the PLC must be trusted. In the case of C2, even the PLC itself may be completely compromised, as C2 monitors all commands after the PLC has transmitted them, and receives all sensor measurements *before* they reach the PLC. Another advantage of the specification-based approach is the ability to better leverage engineers expertise to construct the safety specifications. Normally, the construction of security policies is an ad-hoc and error prone process. Given the degree of specialization of engineers in the plants that they build, there is good reason to believe they will write more comprehensive and succinct policies than traditional security policies, such as the 40,000 line SELinux policy [111].

The work on defenses presented above has been done under the assumption that reasoning about physical behavior is a more tractable task than ensuring perfect perimeter security for large networked control systems. However, this is not to trivialize the approach in question. Specification-based defenses represent one of the most fundamental problems in engineering, coming up with, and enforcing or checking, a specification that details *exactly* the desired behavior. We now explore three future directions for work in specification-based attacks and defenses. First, we examine how SABOT-like attacks can be made more accurate given fuzzier adversarial specifications. Second, we consider how the TSV mechanism can be made partially dynamic to allow for run time checking of highly complex systems while still allowing for the advanced detection of potential safety property violations. Finally, we consider how safety policies themselves may be automatically generated. Key to this idea is differentiating the usefulness of white and blacklists in specifying physical behaviors. We conclude that a mixed white and blacklist technique like C^2's is a firm starting ground for research in policy generation. Additionally, we consider the equally important problem of autogenerating *response policies.* A response policy dictates what should be done with the physical system in the event an attempted property violation is detected.

8.1 More Robust Specification-based Attacks

Several alternative approaches to SABOT's incremental mapping were considered. Abstract State Machines (ASMs), which define state machines in a high level language [112], can be used to generate test cases for verifying that an implementation conforms to a specification [113]. One can imagine that the specification may be written as one or more ASMs. Test cases can then be generated from the sketch under a mapping, and if the control logic passes all tests, then the mapping is correct. Of the alternatives considered, ASMs seem the most promising, as they allow for a similar incremental mapping approach to SABOT's.

Two additional approaches, behavioral equivalence checking (such as bisimulation) and structural equivalence checking (such as subgraph isomorphism) were also considered. The downside for SABOT of using behavioral equivalence checking is that it requires all names to be mapped at once, guaranteeing an expensive best case running time. Structural equivalence checking is also limited in isolation because the adversary does not know implementation specifics, but used in tandem, it could help SABOT to reduce run times, e.g. by filtering out calls to the model checker for properties containing structurally independent control variables.

Along with the checking of structural properties, two basic techniques may be useful to increase SABOT's accuracy and impact. First, an adversary may provide multiple specifications to handle different contingencies, in cases where false negatives are likely. E.g., two specifications may be provided describing a plant with and without an emergency shutdown switch. Second, SABOT may leverage device configuration data available in a control system. High-end PLCs support DeviceNet and PROFIBUS [76] communications, which can broadcast device-specific metadata. SABOT can use this extra information to eliminate names from $V_{\mathcal{M}}$ before searching for a mapping, thus reducing running time and incorrect mappings.

8.2 Hybrid Static and Dynamic Enforcement

The above two chapters on TSV and C^2 took two separate approaches to behavioral specification enforcement. TSV's static approach is only run once at the time when a control logic is uploaded to a PLC. C^2's enforcement, on the other hand, occurred concurrently with the PLC. This raises a question: *Is one preferable over the other?* We know that for any given system, the set of statically enforceable policies is a subset of the set of dynamically enforceable policies [29]. Thus C^2 can enforce any policy that TSV can. Then of what use is TSV? In an information processing system, the ability to detect a property violation one instruction before it might occur is sufficient. However, in a cyber physical system, there may be an advantage to knowing that a violation will occur *long before it is ever possible.* This has shown to be advantageous by the simplex architecture for secure control systems [114, 100], which can redirect the system to a safe state when it is bordering on going bad.. This is because many physical systems have momentum. Additionally, in the case of a dynamic enforcement monitor, the notification that bad code has been loaded onto a PLC comes much later, whereas with a dynamic enforcer, the code is never

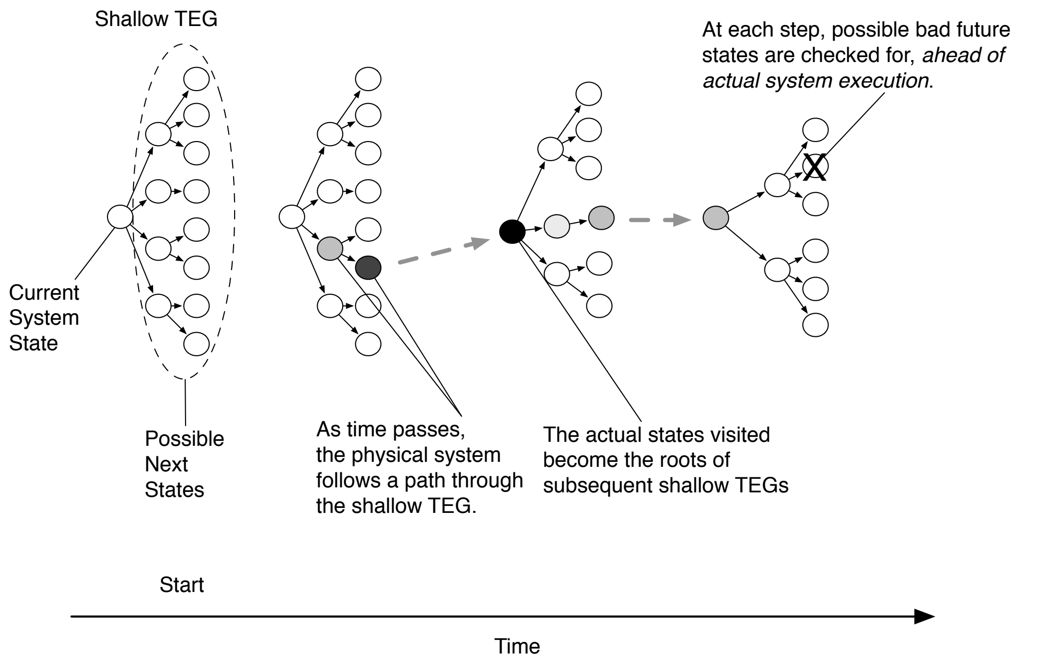

Shallow TEG
At each step, possible bad future states are checked for, *ahead of actual system execution*.
X
Current System State
Possible Next States
As time passes, the physical system follows a path through the shallow TEG.
The actual states visited become the roots of subsequent shallow TEGs
Start
Time

Figure 8.1. Execution of a combined static and dynamic enforcement monitor using shallow TEGs.

loaded in the first place. Thus, the intrusion, or potential mistake, is discovered much earlier.

Besides not being able to check some properties that can be checked by dynamic checking, static verification also has performance limitations. Complex safety properties or very long PLC control logics will require impractical amounts of time to check, i.e., hours for a single design. Given the benefits of dynamic enforcement being low overhead, and the benefits of static enforcement being advanced warning about a problem, we would like to combine them to obtain the benefits. Such a *mixed enforcement* monitor should detect when invalid operations are looming in the possible future, while being performant enough to keep up with the real-time requirements of the control system. One design that seems satisfactory to this approach uses the TEG from TSV. Recall that the TEG is the temporal execution graph, which is a tree structure showing all possible future states for some n future state transitions. Once constructed, a TEG, especially a shallow one, can be efficiently checked against a set of specifications. We have already made strides towards realizing this approach, described as follows.

To obtain the property of finding potential bad states long before the system actually reaches them, we use a *shallow TEG*. This is a TEG representing execution of some small constant number of steps into the future, e.g., 10 steps. We can achieve a hybrid static and dynamic enforcement monitor by constantly reconstructing the shallow TEG, based on the actual known state of the physical system. This is illustrated in Figure 8.1. The idea is that the shallow TEG is always representing future states, and thus allowing detection of future bad states before they are ever reached. At each iteration, two separate operations happen. First, the new current state of the system is located within the shallow TEG, and this branch is removed. The rest of the

TEG is then discarded. The branch representing the current execution is then extended by at least one additional step into the future, and the new TEG is checked for property violations.

There are a number of interesting and novel questions that arise in the design of a hybrid static and dynamic enforcement monitor for physical behavioral specifications. First, how deep should the shallow TEG be? This is a tradeoff between how advanced a warning one gets before a violation may occur, and the computational efficiency requirements of the enforcement monitor. Another interesting problem regards how opportunistic the monitor might be in executing as far ahead as possible. For example, if a path is followed through the shallow TEG that has a low branching degree, it may suddenly become computationally feasible to increase the TEG depth for some number of steps, thus guaranteeing more advanced warning in case of a detected violation. A final question, which is more of policy than of mechanism, is how close should the system be allowed to approach to a bad state in a shallow TEG? For example, consider that a shallow TEG is five steps deep. If a potential violation is detected at some branch at depth five, should the system be halted immediately? It may be the case that the system will never actually reach that branch. Similarly, waiting until the system is about to execute the branch is also problematic. It would be better to gather additional evidence suggesting the system is on a likely trajectory to the neighborhood containing that branch.

8.3 Automating the Specification Writing Process

In the two specification-based defense mechanisms presented above, TSV, and C2, we saw that the specifications are manually created. This can be a problem in the case of highly complex systems, where mistakes in the specification writing process may lead to either: (*i*) the preventing of legitimate behavior or (*ii*) the failure to block malicious behaviors. It is worth mentioning, at this point, that the problem of manual specification writing for cyberphsycal systems *is more tractable* than that for traditional information processing computer systems. As described in the previous chapter on C2, security policies for traditional information processing systems must consider platforms that contain millions of files, tens or hundreds of millions of lines of code, and hundreds of individual machines. Nonetheless, we cannot take the ability to write good behavioral specifications for granted. While manual effort can go a long way, it can also be expensive and time consuming. Thus, in this section, we consider methods for automating all or parts of the specification writing process.

The first step is to more carefully define exactly what a physical or behavioral specification is. One way to do this is in terms of typical security policy language. The most basic question we can as is: *Would we like a whitelist policy or a blacklist policy?* A whitelist policy is one in which a behavior must be in the whitelist to be allowed. In a blacklist policy, any behavior is allowed, as long as it is not on the blacklist. To give some examples, UNIX access control policies are whitelist policies, as any access attempt not specified as allowed is denied. Windows access control entities, on the other hand, are more like firewall rules, in that they contain both explicit allow and explicit deny statements. Can this same idea apply to behavioral specifications?

First, we must define the equivalent of black and whitelists for behavioral specifications. For this, we will introduce the idea of an operation *holding* under a specification model. For an operation to hold, a checker must return true, stating that it conforms to the spec. Given this definition, we can now define a behavioral specification as being a blacklist when operations are denied only when they hold. Similarly, a behavioral specification is a whitelist when operations are denied only when they *do not* hold. We can now safely say that TSV uses a blacklist policy. TSV, for example, is a whitelist policy. Similarly, SABOT uses a whitelist policy, where only expected behaviors are considered to identify victim devices. C^2 on the other hand, is a mix. It starts with a whitelist policy in the form of the device policies. Each device policy consists of a state machine that encodes the total allowed behavior of the device. The conflicts on the other hand, form a blacklist to the set of all behaviors created by the product of all device state machines.

Consider that a whitelist is the difference between the universe of all possible operations and the blacklist. That is, the blacklist and whitelist are mutually exclusive and exhaustive. Thus, the smaller one is, the larger the other is. Quite arguably, the universe of all possible operations in a given control system is significantly larger than the number of operations that will ever actually be executed in that system. One might expect that this means the whitelist should be significantly smaller than the blacklist. However, the set of malicious operations may be much smaller compared to the universe of all possible operations, thus we cannot make such a claim. We also note that it is impossible to infer potential malicious behavior by watching the correct execution of a control system. Thus, any policy generation relying on empirical recordings will by its very nature use whitelists.

Blacklists have their own advantages. In the case of C^2, there are certain device conflicts that are obvious, for example, those preventing two physical objects from attempting to occupy the same space at the same time. The use of conflicts for blacklisting makes such properties easy to encode. Note that this method of using blaclisting as pruning only works given a set of operations to prune. In C^2's case, this was the product of all device policies. Here, we find some insight for the partially automated generation of device policies. Automatic methods depending on empirical observations (such as network protocol analyzers [115]) of a system can effectively form a whitelist. However, creation of C^2-like blacklists require manual effort. Fortunately, the process of pruning the whitelist according to the manually generated blacklist can itself also be automated.

There is some prior work in the direction of generating a specification for supervisory control systems based on a high level description of desired behavior. One method of doing this is to autogenerate a controller that only allows a system to perform transitions described in a high-level language. [43]. Since this initial work, there have been demonstrations showing that such controllers can be generated for simple systems. [116]. In the general case, the problem of generating a controller from a high-level description likely has no polynomial solution [44]. Additionally, some attempts have been made at defining policies for robot behavior [117]. However, little is known about policy autogeneration.

8.3.1 Response Policies

Along with the problem of generating the definition of allowed or disallowed behavior, we also have to ask *What should be done in the event of an attempted property violation?* This question may sound familiar as it was addressed in the C^2 work. The solution in that case was to introduce *deny disciplines*. Recall that there were four deny disciplines, Drop, Notify, Retry, and Approximate. These were general approaches, not aimed at any particular system. However, an automatic policy generator may additionally be able to generate these response policies according to the severity of the problem. For example, an engineer may supply a meta policy containing several *shutdown states* that represent safe resting positions for the system in the event an attempted property violation occurs. The automated policy generator could then find sequences of operations to transfer the plant from each bad state to the allowed policy violation.

Another potential approach is to simply *nudge* the plan around the safety violation. Similar to the notion of the Approximate deny discipline, this would drive the plant into the nearest allowable state. Ideally, because closed loop control systems are to be self-regulating, the system may then recover from the nudge and return to normal operation. In the worst case, the system may continuously diverge due to a malicious actor. In this case, repeated nudging could guarantee that the system enters a benign but sub-optimal state each time.

Bibliography

[1] Roberts, P. F. (2008), "Zotob, PnP Worms Slam 13 DaimlerChrysler Plants," http://www.eweek.com.

[2] News, B. (2008), "Computer Viruses Make it to Orbit," http://news.bbc.co.uk.

[3] Krebs, B. (2008), "Cyber Incident Blamed for Nuclear Power Plant Shutdown," http://www.washingtonpost.com.
URL http://www.washingtonpost.com/wp-dyn/content/article/2008/06/05/AR2008060501958.html

[4] Grad, S., "Engineers who hacked into L.A. traffic signal computer, jamming streets, sentenced," http://latimesblogs.latimes.com.

[5] Leall, N. (2009), "Lessons from an Insider Attack on SCADA Systems," http://blogs.cisco.com/security/lessons_from_an_insider_attack_on_scada_systems/.

[6] Zetter, K. (2010), "Clues Suggest Stuxnet Virus Was Built for Subtle Nuclear Sabotage," http://www.wired.com/threatlevel/2010/11/stuxnet-clues/.

[7] Leyden, J. (2008), "Polish Teen Derails Tram after Hacking Train Network," http://www.theregister.co.uk/2008/01/11/tram_hack/.

[8] Meserve, J. (2007), "Sources: Staged Cyber Attack Reveals Vulnerability in Power Grid," http://articles.cnn.com/2007-09-26/us/power.at.risk_1_generator-cyber-attack-electric-infrastructure?_s=PM:US.

[9] Pollet, J. (2010) "Electricity for Free? The Dirty Underbelly of SCADA and Smart Meters," in *Proceedings of Black Hat USA 2010*.

[10] Byres, E. and J. Lowe (2003) "The Myths and Facts behind Cyber Security Risks for Industrial Control Systems," in *ISA Process Control Conference*.

[11] Piètre-Cambacédès, L., M. Trischler, and G. N. Ericsson (2011) "Cybersecurity Myths on Power Control Systems: 21 Misconceptions and False Beliefs," *IEEE Transactions on Power Delivery*.

[12] McLaughlin, S., D. Podkuiko, and P. McDaniel (2009) "Energy Theft in the Advanced Metering Infrastructure," in *Proceedings of the 4th International Workshop on Critical Infrastructure Security (CRITIS)*.

[13] McLaughlin, S., D. Podkuiko, S. Miadzvezhanka, A. Delozier, and P. McDaniel (2010) "Multi-vendor Penetration Testing in the Advanced Metering Infrastructure," in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*.

[14] Liu, Y., P. Ning, and M. K. Reiter (2009) "False Data Injection Attacks against State Estimation in Electric Power Grids," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*.

[15] Mo, Y. and B. Sinopoli (2010) "False Data Injection Attacks in Control Systems," in *Proceedings of the First Workshop on Secure Control Systems (SCS)*.

[16] Zetter, K. (2011), "Attack Code for SCADA Vulnerabilities Released Online," `http://www.wired.com/threatlevel/2011/03/scada-vulnerabilities/`.

[17] Peterson, D. G. (2012), "Project Basecamp at S4," `http://www.digitalbond.com/2012/01/19/project-basecamp-at-s4/`.

[18] Computer Emergency Response Team (2011), "ADVANTECH/BROADWIN WEBACCESS RPC VULNERABILITY," ICS-CERT Advisory 11-094-02.

[19] ——— (2011), "SIEMENS SIMATIC HMI AUTHENTICATION VULNERABILITIES," ICS-CERT Advisory 11-356-01.

[20] Beresford, D. (2011) "Exploiting Siemens Simatic S7 PLCs," in *Black Hat USA*.

[21] Newman, T., T. Rad, and J. Strauchs (2011), "SCADA & PLC Vulnerabilities in Correctional Facilities," White Paper.

[22] McLaughlin, S. (2013) "Securing Control Systems from the Inside: A Case for Mediating Physical Behaviors," *IEEE Security & Privacy*.

[23] Ericson, C. A., II (1999) "Fault Tree Analysis — A History," in *Proceedings of the 17th International System Saftey Conference*.

[24] Schneier, B. (1999) "Attack Trees: Modeling Security Threats," *Dr. Dobb's Journal*.

[25] Byres, E. J., M. Franz, and D. Miller (2004) "The Use of Attack Trees in Assessing Vulnerabilities in SCADA Systems," in *Proceedings of the International Infrastructure Survivability Workshop (IISW)*.

[26] Falliere, N., L. O. Murchu, and E. Chien (2010), "W32.Stuxnet Dossier," `http://www.symantec.com/connect/blogs/w32stuxnet-dossier`.

[27] Triangle Research International, "Connecting Super PLCs to the Internet," `http://www.tri-plc.com/internetconnect.htm`.

[28] Control Technology Corp., "Blue Fusion: Model 5200 Controller," `http://www.ctc-control.com/products/5200/5200.asp`.

[29] Hamlen, K. W., G. Morrisett, and F. B. Schneider (2006) "Computability Classes for Enforcement Mechanisms," *ACM Trans. Program. Lang. Syst.*, **28**, pp. 175–205.

[30] Sarwate, A. (2011) "SCADA Security: Why is it so Hard?" in *BSides Iowa*.

[31] Cárdenas, A. A., S. Amin, and S. Sastry (2008) "Research Challenges for the Security of Control Systems," in *Proceedings of the 3rd conference on Hot topics in security*.

[32] American National Standards Institute (1981), "ANSIX3.92-198 Data Encryption Algorithm," .

[33] Mayor, D., K. K. Mookhey, J. Cervini, and F. Roslan (2007) *Metasploit Tookit: for Penetration Testing, Exploit Devevlopment, and Vulnerability Research*, Syngress.

[34] Langill, J. (2011), "White Phosphorus Exploit Pack Ver 1.11 Released for Immunity Canvas," `http://scadahacker.blogspot.com/2011/04/white-phosphorus-exploit-pack-ver-111.html`.

[35] Constantin, L. (2012), "Researchers Expose Flaws in Popular Industrial Control Systems," `http://www.pcworld.com`.

[36] Groote, J., S. van Vlijmen, and J. Koorn (1995) "The Safety Guaranteeing System at Station Hoorn-Kersenboogerd," in *Computer Assurance, 1995. Proceedings of the Tenth Annual Conference on Systems Integrity, Software Safety and Process Security'*, pp. 57–68.

[37] Hartonas-Garmhausen, V., S. Campos, A. Cimatti, E. Clarke, and F. Giunchiglia (1998) "Verification of a Safety-critical Railway Interlocking System with Real-time Constraints," in *Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing.*

[38] Ferreira, N. G. and P. S. M. Silva (2004) "Automatic Verification of Safety Rules for a Subway Control Software," in *Proceedings of the Brazilian Symposium on Formal Methods (SBMF).*

[39] Sachdev, M., P. Dhakal, and T. Sidhu (2000) "A Computer-Aided Technique for Generating Substation Interlocking Schemes," *Power Delivery, IEEE Transactions on*, **15**(2), pp. 538 –544.

[40] ——— (2000) "Design Tool Generates Substation Interlock Schemes," *Computer Applications in Power, IEEE*, **13**(2), pp. 37 –42.

[41] Park, T. and P. I. Barton (2000) "Formal Verification of Sequence Controllers," *Computers & Chemical Engineering.*

[42] Huuck, R. (2005) "Semantics and Analysis of Instruction List Programs," *Electr. Notes Theor. Comput. Sci.*

[43] Ramadge, P. and W. Wonham (1984) "Supervisory Control of a Class of Discrete Event Processes," in *Analysis and Optimization of Systems* (A. Bensoussan and J. Lions, eds.), vol. 63 of *Lecture Notes in Control and Information Sciences*, Springer Berlin / Heidelberg, pp. 475–498, 10.1007/BFb0006306.
URL `http://dx.doi.org/10.1007/BFb0006306`

[44] Gohari, P. and W. M. Wonham (2000) "On the Complexity of Supervisory Control Design in the RW Framework," *IEEE Transactions on Systems, Man, and Cybernetics.*

[45] Brothman, A., R. D. Reiser, N. L. Kahn, F. S. Ritenhouse, and R. A. Wells (1965) "Automatic Remote Reading of Residential Meters," *IEEE Transactions on Communication Technology*, **13**(2), pp. 219 – 232.

[46] Rosenfeld, A. H., D. A. Bulleit, and R. A. Peddie (1986) "Smart Meters and Spot Pricing: Experiments and Potential," *IEEE Technology and Society Magazine.*

[47] Goldberg, M. (2010) "Measure Twice, Cut Once," *IEEE Power and Energy Magazine*, pp. 46 – 54.

[48] King, C. S. (2001) *The Economics of Real-Time and Time-of-Use Pricing For Residential Consumers*, *Tech. rep.*, American Energy Institute.

[49] Kelley, R. and R. D. Pate (2008), "Mesh Networks and Outage Management," White Paper.

[50] American National Standards Institute (2006), "C12.18 Protocol Specification for ANSI Type 2 Optical Port," .

[51] McDaniel, P. and S. McLaughlin (2009) "Security and Privacy Challenges in the Smart Grid," *IEEE Security & Privacy Magazine*.

[52] Lewson, N., "Smart meter crypto flaw worse than thought," `http://rdist.root.org/2010/01/11/smart-meter-crypto-flaw-worse-than-thought`.

[53] Fehrenbacher, K., "Smart Meter Worm Could Spread Like A Virus," `http://earth2tech.com/2009/07/31/smart-meter-worm-could-spread-like-a-virus/`.

[54] Hull, D. (2010), "PG&E details technical problems with SmartMeters," `http://www.siliconvalley.com/news/ci_14963541`.

[55] Enck, W., P. Traynor, P. Mcdaniel, and T. L. Porta (2005) "Exploiting Open Functionality in SMS-capable Cellular Networks," in *Proceedings of the 12th ACM Conference on Computer and Communication Security (CCS)*, ACM Press, pp. 393–404.

[56] Traynor, P., M. Lin, M. Ongtang, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta (2009) "On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core," in *Proceedings of the 16th ACM Cnference on Computer and Communications Security (CCS)*, ACM, New York, NY, USA, pp. 223–234.

[57] Hart, G. W. (1992) "Nonintrusive Appliance Load Monitoring," *Proceedings of the IEEE*.

[58] ——— (1989) "Residential Energy Monitoring and Computerized Surveillance via Utility Power Flows," *IEEE Technology and Society Magazine*.

[59] Lisovich, M. A., D. K. Mulligan, and S. B. Wicker (2010) "Inferring Personal Information from Demand-Response Systems," *IEEE Security and Privacy*, **8**(1), pp. 11–20.

[60] Kinney, R., P. Crucitti, R. Albert, and V. Latora (2005) "Modeling cascading failures in the North American power grid," *The European Physical Journal B - Condensed Matter and Complex Systems*, **46**(1), pp. 101–107.
URL `http://dx.doi.org/10.1140/epjb/e2005-00237-9`

[61] International Electrotechnical Commission, "International Standard IEC 61131 Part 3: Programming Languages," .

[62] Huth, M. and M. Ryan (2004) *Logic in Computer Science*, Cambridge University Press.

[63] Moon, I., G. J. Powers, J. R. Burch, and E. M. Clarke (1992) "Automatic Verification of Sequential Control Systems Using Temporal Logic," *AIChE Journal*, **38**, pp. 67–75.
URL `http://dx.doi.org/10.1002/aic.690380107`

[64] Cimatti, A., E. Clarke, F. Giunchiglia, and M. Roveri (1999) "NuSMV: A New Symbolic Model Verifier," in *Computer Aided Verification*, Springer Berlin / Heidelberg.

[65] Cavada, R., A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev (2010), "NuSMV 2.5 User Manual," `http://nusmv.fbk.eu/`.

[66] Hoglund, G. and G. McGraw (2004) *Exploiting Software: How to Break Code*, Addison Wesley.

[67] Gegick, M. and L. Williams (2005) "Matching attack patterns to security vulnerabilities in software-intensive system designs," in *SESS '05: Proceedings of the 2005 workshop on Software engineering for secure systems—building trustworthy applications*, ACM, New York, NY, USA, pp. 1–7.

[68] Vesely, W., F. Goldberg, N. Roberts, and D. Haasl (1981) *Fault Tree Handbook*, U.S. Nuclear Regulator Commission.

[69] Eilam, E. (2005) *Reversing: Secrets of Reverse Engineering*, Wiley.

[70] Takanen, A., J. DeMott, and C. Miller (2008) *Fuzzing for Software Security Testing and Quality Assurance*, Artech House Publishers.

[71] "B.T. Aluminum Tamper Seal," `http://www.brooksutility.com/catalog/product-detail.asp?ID=302`.

[72] The Asterisk Project, "Asterisk Open Source PBX," `http://www.asterisk.org`.

[73] "Nmap Reference Guide," `http://nmap.org/book/man.html`.

[74] Infigo.hr, "Infigo FTPStress Fuzzer," `http://www.infigo.hr/en/in_focus/tools`.

[75] Canet, G., S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen (2000) "Towards the Automatic Verification of PLC Programs Written in Instruction List," in *IEEE International Conference on Systems, Man, and Cybernetics*, vol. 4, pp. 2449–2454.

[76] PROFIBUS (2010), "IMS Research Estimates Top Position for PROFINET," `http://www.profibus.com/news-press/detail-view/article/ims-research-estimates-top-position-for-profinet/`.

[77] Ferrari, A., G. Magnani, D. Grasso, and A. Fantechi (2011) "Model Checking Interlocking Control Tables," in *FORMS/FORMAT 2010* (E. Schnieder and G. Tarnai, eds.), pp. 107–115.
URL `http://dx.doi.org/10.1007/978-3-642-14261-1_11`

[78] James, P. and M. Roggenbach (2009) "SAT-based Model Checking of Train Control Systems," in *Proceedings of the CALCO Young Researchers Workshop*.

[79] Yang, L. and C. XianFeng (2009) "Design of Traffic Lights Controlling System Based on PLC and Configuration Technology," in *International Conference on Multimedia Information Networking and Security*.

[80] Yardley, T. (2008) "SCADA: Issues, Vulnerabilities, and Future Directions," *;login*, **34**(6), pp. 14–20.

[81] Erickson, K. T. and J. L. Hedrick (1999) *Plantwide Process Control*, Wiley Inter-Science.

[82] Falcione, A. and B. Krogh (1992) "Design Recovery for Relay Ladder Logic," in *First IEEE Conference on Control Applications*.

[83] Schwartz, M. D., J. Mulder, J. Trent, and W. D. Atkins (2010) *Control System Devices: Architectures and Supply Channels Overview*, *Tech. Rep. SAND2010-5183*, Sandia National Laboratories.

[84] Slay, J. and M. Miller (2007) "Lessons Learned from the Maroochy Water Breach," in *International Conference on Critical Infrastructure Protection*.

[85] Wilhoit, K. (2013), "Who's Really Attacking Your ICS Equipment," Trend Micro.

[86] "SHODAN," `http://www.shodanhq.net`.

[87] National Energy Regulatory Comission (2006), "NERC CIP 002 1 - Critical Cyber Asset Identification," .

[88] Bobba, R. B., K. M. Rogers, Q. Wang, H. Khurana, K. Nahrstedt, and T. J. Overbye (2010) "Detecting False Data Injection Attacks on DC State Estimation," in *Proceedings of the First Workshop on Secure Control Systems (SCS)*.

[89] Sandberg, H., A. Teixeira, and K. H. Johansson (2010) "On Security Indices for State Estimators in Power Networks," in *Proceedings of the First Workshop on Secure Control Systems (SCS)*.

[90] Song, D., D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, , and P. Saxena (2008) "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security*.

[91] McLaughlin, S., D. Pohly, P. McDaniel, and S. Zonouz (2014) "A Trusted Safety Verifier for Process Controller Code," in *Proc. ISOC Network and Distributed Systems Security Symposium (NDSS)*.

[92] De Moura, L. and N. Bjørner (2008) "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 337–340.

[93] Ferrante, O., L. Benvenuti, L. Mangeruca, C. Sofronis, and A. Ferrari (2012) "Parallel NuSMV: a NuSMV Extension for the Verification of Complex Embedded Systems," in *Computer Safety, Reliability, and Security*, Springer, pp. 409–416.

[94] Dwyer, M. B., G. S. Avrunin, and J. C. Corbett (1999) "Patterns in Property Specifications for Finite-State Verification," in *Proceedings of the 21st international conference on Software engineering (ICSE)*, ACM, pp. 411–420.

[95] Biha, S. O. (2011) "A Formal Semantics of PLC Programs in Coq," in *Proceedings of the 35th IEEE Annual Computer Software and Applications Conference*.

[96] McLaughlin, S. and P. McDaniel (2012) "SABOT: specification-based payload generation for programmable logic controllers," in *Proceedings of the 2012 ACM conference on Computer and communications security*, New York, NY, USA, pp. 439–449.

[97] Stouffer, K., J. Falco, and K. Scarfone (2008) "Guide to Industrial Control Systems (ICS) Security," *NIST Special Publication*, **800**, p. 82.

[98] U.S. Department of Energy Office of Electricity Delivery and Energy Reliability (2005), "A Summary of Control System Security Standards Activities in the Energy Sector," .

[99] Weiss, J. (2009) "Are the NERC CIPS making the grid less reliable," *Control Global*.

[100] Mohan, S., S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo (2012), "S3A: Secure System Simplex Architecture for Enhanced Security of Cyber-Physical Systems," `http://arxiv.org`.

[101] Cheung, S., B. Dutertre, M. Fong, U. Lindqvist, K. Skinner, and A. Valdes (2007) "Using Model-based Intrusion Detection for SCADA Networks," in *Proceedings of the SCADA Security Scientific Symposium*.

[102] Goble, W. M. (2010) *Control Systems Safety Evaluation and Reliability*, International Society of Automation.

[103] Stouffer, K., J. Falco, and K. Kent (2006), "Guide to Supervisory Control and Data Acquisition (SCADA) and Industrial Control Systems Security," National Institute of Standards and Technology (NIST).

[104] Éireann P. Levertt (2011) *Quantitatively Assessing and Visualising Industrial System Attack Surfaces*, Master's thesis, University of Cambridge.

[105] McLaughlin, S. and P. McDaniel (2012) "SABOT: Specification-based Payload Generation for Programmable Logic Controllers," in *19th ACM Conference on Computer and Communications Security (CCS)*.

[106] Parr, T. and K. Fisher (2011) "LL(*): the Foundation of the ANTLR Parser Generator," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*.

[107] Siemens AG, "Anweisungsliste (AWL) für S7-300/400," `http://automation.siemens.com`.

[108] Anderson, J. P. (1972) *Computer Security Technology Planning Study, Tech. Rep. TR-73-51*, Air Force Electronic Systems Devision.

[109] Provos, N., M. Friedl, and P. Honeyman (2003) "Preventing Privilege Escalation," in *12th USENIX Security Symposium*.

[110] LeMay, M. and C. A. Gunter (2009) "Cumulative Attestation Kernels for Embedded Systems," in *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)*.

[111] Karger, P. A. and R. R. Schell (2002) "Thirty Years Later: Lessons from the Multics Security Evaluation," in *Annual Computer Security Applications Conference (ACSAC)*.

[112] Börger, E. (2010) "The Abstract State Machines Method for High-Level System Design and Analysis," *Formal Methods: State of the Art and New Directions*.

[113] Grieskamp, W., L. Nachmanson, N. Tillmann, and M. Veanes (2003) "Test Case Generation from AsmL Specifications," in *ASM*.

[114] Bak, S., K. Manamcheri, S. Mitra, and M. Caccamo (2011) "Sandboxing Controllers for Cyber-Physical Systems," in *ICCPS*.

[115] Wang, Y., Z. Zhang, D. Yao, B. Qu, and L. Guo (2011) "Inferring Protocol State Machine from Network Traces: A Probabilistic Approach," in *Applied Cryptography and Network Security* (J. Lopez and G. Tsudik, eds.), vol. 6715 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 1–18.
URL `http://dx.doi.org/10.1007/978-3-642-21554-4_1`

[116] Liu, J. and H. Darabi (2002) "Ladder Logic Implementation of Ramadge-Wonham Supervisory Controller," in *Proceedings of the Sixth International Workshop on Discrete Event Systems*.

[117] Finnicum, M. and S. T. King (2011) "Building Secure Robot Applications," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Security.*

Vita

Stephen Elliot McLaughlin

Education:
Computer Science and Engineering, Pennsylvania State University:

- PhD (2008 - 2014)
- MS (2008 - 2010)
- BS (2003 - 2007)

Selected Publications:

- Stephen McLaughlin, Dmitry Podkuiko, and Patrick McDaniel. **Energy Theft in the Advanced Metering Infrastructure**. *4th International Workshop on Critical Information Infrastructure Security (CRITIS 2009)*, Bonn, Germany. September, 2009.
- Stephen McLaughlin, Dmitry Podkuiko, Sergei Miadzvezhanka, Adam Delozier, and Patrick McDaniel. **Multi-vendor Penetration Testing in the Advanced Metering Infrastructure**. *26th Annual Computer Security Applications Conference (ACSAC 2010)*, Austin, TX, USA. December 2010.
- Stephen McLaughlin, **On Dynamic Malware Payloads Aimed at Programmable Logic Controllers**. *6th USENIX Workshop on Hot Topics in Security*, San Francisco, CA. August, 2011.
- Stephen McLaughlin and Patrick McDaniel. **SABOT: Specification-based attacks on Sequential Control Systems**. *18th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, USA. October 2012.
- Stephen McLaughlin **Stateful Policy Enforcement for Control System Device Usage**. *29th Annual Computer Security Applications Conference (ACSAC)*, New Orleans, LA, USA. December 2013.
- Stephen McLaughlin, Devin Pohly, Patrick McDaniel, and Saman Zonouz, **A Trusted Safety Verifier for Process Controller Code**. *ISOC Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, USA. February 2014.

Awards:

- Outstanding Research Assistant Award in CSE, 2013
- Diefenderfer Graduate Fellowship in the College of Engineering 2012-2014
- ACM CCS Student Travel Grant, 2011 and 2012
- Student Scholarship to the TCIPG Summer School on Cyber Security for Smart Energy Systems, 2011
- ACSAC Student Conferenceship, 2010
- Travel Grant to the DIMACS Workshop on Algorithmic Decision Theory for the Smart Grid, 2010
- HotSec Student Travel Grant, 2010
- USENIX Security Symposium Travel Grant, 2008 and 2009
- Harry G. Miller Fellowship in the College of Engineering for the spring 2007 semester
- R. P. Drenning Memorial Scholarship in the College of Engineering for the 2007-08 academic year
- John F. Kray Sr. Memorial Scholarship in the College of Engineering for the 2006-07 academic year