Context-based Anomaly Detection in Critical Infrastructures

A Study in Distributed Systems

Thomas Richard McEvoy

Thesis submitted to the University of London
for the degree of Doctor of Philosophy

Royal Holloway
University of London

2013

Context-based Anomaly Detection in Critical Infrastructures

Department of Mathematics
Royal Holloway, University of London

Most cats, when they are out want to be in, and visa versa, and often simultaneously.

(Louis J. Camuti)

To my wife, Lesley Anne McEvoy, for her dedication and hard work.

Declaration of Authorship

I, Thomas Richard McEvoy, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Signed:

(Thomas Richard McEvoy)

Date:

Preface

This thesis examines the problem of detecting malicious software in industrial control systems. This is a growing problem since such systems are increasingly exposed to attack as modernization heightens the degree of system interconnectivity. At the same time, such systems are ill-equipped with suitable security mechanisms to allow them to fend off these attacks.

The approach adopted assumes that the attacker will make no egregious protocol errors which would reveal the fact of penetration before that phase of the attack completes. Instead, the detection problem is one of uncovering an attack during the exploitation phase by drawing on a rich set of potential data relationships and using these, if possible, to locate the source of the attack and manage its outcome. The future direction of this work should permit us to detect integrity attacks in a timely fashion and use the results to bring control systems back under operator supervision, at least, for a subset of such systems.

We consider this research will be valuable to individuals working to defend control systems – and distributed systems in general – against malicious software attacks.

Abstract

The modernization of critical infrastructure exposes a large attack surface in a set of systems, key to the sustainability of civilization, at a time when targeted malicious attacks are growing in sophistication, particularly with regard to stealth techniques, which are particularly difficult to uncover in distributed systems due to multiple possible orderings of state.

We argue that by making use of a set of known relationships (which we label a *context*) between states in disparate parts of a distributed system and the provision of suitable concurrent (or near-concurrent) observation and comparison mechanisms, we can provide the means to detect such anomalies and locate their source as a precursor to managing outcomes.

As a necessary prerequisite to our research, we establish an adversary capability model which allows us to make explicit statements about the feasible actions and subsequent impacts of adversary and demonstrate the validity of any detective methods.

We focus primarily on integrity attacks. The first technique we present is a security protocol, using traceback techniques, which allows us to locate processes which manipulate message content between an operator and a control unit. The second technique allows us to model algebraically possible sequences in host system states which may be indicative of malicious activity and detect these using a multi-threaded observation mechanism. The third technique provides a process engineering model of a basic non-linear process in a biochemical plant (pasteurization in a brewery) which shows how the provision of, even minimal, additional sensor information, outside of standard telemetry requirements, can be used to determine a failure in supervisory control due to malicious action. This last technique represents an improvement over previous approaches which focused on linear or linearized systems.

All three techniques pave the way for more sophisticated approaches for real-time detection and management of attacks.

Thanks go to my supervisor, Stephen Wolthusen, for his encouragement and counsel, and to my examiners, Keith Mayes and Emiliano Cassalichio, for their comments and questions. I am also grateful to the anonymous reviewers of the papers submitted for publication as part of the work for this dissertation for their comments, advice and corrections and to Diageo PLC for access to data on their beer pasteurization process. Finally, I would like to thank my wife and family for their patience and support over the past seven years.

Contents

List of Figures

List of Tables

List of Theorems

List of Definitions

List of Algorithms

Chapter 1

Introduction

If you hold a cat by the tail you learn things you cannot learn any other way.

Mark Twain

1.1 Preamble

The modernization of industrial control systems has left them exposed to exploitation by the increasingly sophisticated malicious software threats. We show how we can address these issues through novel detective methods which make use of the *context* created by data relationships between disparate parts of a distributed system. We introduce a proof method based on the instantiation of a formal adversary capability model for malicious software using the π-calculus.

1.2 Threats to Distributed Systems in Critical Infrastructures

In industrial control systems (ICS) , isolated computing systems, using proprietary operating systems and network protocols have been displaced over recent decades by Internet-facing, commercial-off-the-shelf systems (COTS), incorporating open network protocols. ICS are key to the to the preservation of critical infrastructures (CI), such as energy production, transport, waste disposal and food production which sustain human civilization. These alterations create security exposures in such systems for which they are intrinsically ill-prepared to cope [67, 17, 65, 43].

The modernization of ICS has rendered the devices, which we use to support our existence, exposed to the threat of software-borne malicious attack – for example, [41, 11] – which have the potential to arbitrarily alter system behavior, threatening our environment . Faster, high bandwidth networks with tighter interconnections also result in the potential for malicious software to propagate faster than signature updates or patch updates – especially in ICS, where the real-time demands and lengthy nature of production runs (whose duration can be measured in months, or even years) create services which cannot economically bear unplanned disruptions caused by security hot fixes and have difficulty adapting to patching strategies more suitable to enterprise security architectures for business systems [21]. Meanwhile the sophistication of delivery and payload mechanisms in malicious software advances continually in rivalry to the development of effective defenses[106]. In particular, for our research, we consider the capability of malicious software to hide its presence and actions by undermining the integrity of state reporting and state reporting mechanisms [117, 29]. Adversaries rapidly deploy stealth techniques shortly after injection, leading to, at best, momentary anomalies, which may be undetectable to human operators, or to normal system mechanisms. Such techniques not only conceal malicious acts, but may also allow malicious agents to present themselves as legitimate actors in the system[41].

Additional factors which advantage the attacker are the intrinsic lack of global time and state information and delays in processing and communication on distributed networks which lead to multiple possible legitimate linearizations in state – a feature which, to some extent, is also true of multiprocessor hosts as a result of how memory writes are handled – see Appendix B and [46] – and the capability, not only to deceive operators and system defenses, but also to undermine the latter by direct or indirect attack [117, 93].

We further acknowledge malicious software detection is an undecidable problem in terms of determining the integrity of software [28]. Any approach to detection is necessarily additive in nature rather than representing a complete solution – although we may choose to judge the efficiency and effectiveness of techniques by the generality of their application – see section 5.7.3.

Finally, in ICS , we are necessarily constrained in our solutions by the computational limitations which exist in such environments.

1.3 Research Questions

Our research question is, therefore, can we develop successful detection methods in relation to integrity attacks in the face of identified constraints?

- Increasing sophistication of stealth methods
- Direct and indirect attacks on security and state reporting mechanisms
- Intrinsic characteristics of distributed environments in relation to the ordering of states
- Environmental constraints on computational performance and capacity

In doing so, we must take consider the difficulty of modeling an adversary which can act arbitrarily to alter the behavior of those environments. Indeed, we should account for the possibility of multiple adversaries and the interventions of defending agencies. Ideally, we should go beyond the mere detection of the presence of malicious activity to being able pinpoint its location and facilitate the management of attack outcomes.

1.4 Research Approach

We define a formal threat model , based on the Dolev-Yao model [74, 38], to represent the use of malicious software by knowledgeable and experienced adversaries, whose existence we assume. Our particular approach to modeling adversary behavior is based on [74], but with clear differences in emphasis and outcome as our subject is the consideration of malicious behavior in systems rather than the security of key exchange protocols.

We assume the adversary may, using malicious messages, subvert a subset of processes to act maliciously (or else inject such a process) in a system, but not undermine the integrity of the whole system. As a result, subverted processes may selectively manipulate messages in accordance with adversary goals. We make the nature of such manipulations explicit, allowing us to create feasible attack scenarios. Such processes may also have the ability, through messaging (whether overt or covert), to co-ordinate actions between different parts of the system in achieving overall goals. They could also compete, or possibly, co-operate with processes belonging to other malicious agents. Obviously, the impacts of such actions should be considered, along with the effects of operator interventions and egregious system events in determining adversary success or failure.

We assume that the adversary is perfectly stealthy until he acts. So we focus on providing early warning of a damaging attack in progress rather than detecting initial penetration and subversion – though these could be modeled by our approach.

We use two variants of the applied π-calculus to instantiate our threat model, selecting the appropriate variant depending on application. The family of process algebras and calculi form a well-known and well-defined tool set [121, 7, 34] for these purposes. The π-calculus has the particular advantage of taking account of the intrinsic characteristics of distributed systems, while also allowing the re-specification of systems "on the fly".

Turning to detective methods, we adopt an approach which we label, though not originally, *context-based anomaly detection*. In this approach, we provide a stateful comparison of related data sets in disparate parts of the distributed environment, using a suitable concurrent observation mechanism. The information from such processes is collated and compared, not necessarily centrally, to uncover anomalies based on the expected behavior of the system. In our research, we created three such models.

The first model uses copies of messages passing through communication channels to create a model of its own expected behavior, i.e., the integrity of communication between source and destination. Comparing a data packet early in the communication stream with itself at some later point provides an obvious opportunity to discover anomalies as the result of malicious packet manipulation. At the same time, we can make use of encrypted routing information in packets to locate the source of the manipulation within two routing hops. We designed and formally proved (using our π-calculus) a security protocol, based on IP Traceback techniques [66], for carrying out this operation. We also show how it could be used not only to locate subverted processes, but also to manage systems under attack. This latter example demonstrates, in passing, the ability to model co-ordination between attacking agents, using our techniques.

The second approach makes use of a stochastic, causal model of system action instantiated using a process algebra variant similar to CCS (the calculus of communicating systems[88]) and an associated order-isomorphic variant of process graphs. We use this approach to create a negative model of system behavior which captures potentially malicious behavior semantically by creating a set of linearizations which are legitimate and subsequently considering all non-feasible linearizations to be potentially malicious. In addition, we can augment this model using known semantic patterns, which we label *characteristic vectors* to increase our ability to detect attacks. We create an observation mechanism for host-based anomaly detection which allowed us to use multiple observer threads, distributed between CPUs, to concurrently observe transitions in state at different levels of abstraction in the system and in disparate, but related, kernel structures. The nature of such transitions can be compared with the expected behavior expressed (negatively) by the model. We demonstrated this approach using a "proof of concept" based on rootkit detection using a Linux Kernel Module (LKM) and using sample rootkit techniques to devise and test some basic observational possibilities.

The third approach makes use of classic process control engineering modeling techniques to demonstrate how additional sensors could be included in a plant (on an "out of band" (OOB) channel) and used to validate outputs from the sensors used for the control of the plant. Again, the presence of anomalies enables us to detect potentially malicious behavior. This approach is not unique, but, in this case, was applied to a non-linear chemical process, which is an improvement over previous techniques which only considered linear or linearized systems.

In each case, we seek to demonstrate that our techniques are resilient in the face, not only of known intrinsic difficulties with establishing state in distributed systems, but also of direct and indirect attempts by the adversary to undermine our efforts.

Overall, the three techniques which we describe, taken together, form a method for end-to-end observation and monitoring of network and host integrity in industrial control systems (ICS). The threat model we devise and the associated algebraic modeling techniques provide a means of formally proving such methods for specific attack scenarios.

Our approach is limited at this stage in its development by a lack of suitable model checking tools (which represents another kind of research challenge). In addition, we focused primarily on the development of initial models and further testing and simulation to relate our results to real-world worlds, including addressing limitations on the performance and capability of industrial control systems, is required. At this point, such limitations are only dealt with theoretically.

Future development will focus on enhancing these basic techniques using timing information and more sophisticated approaches to state estimation. Our final goal is to demonstrate that we can detect integrity attacks, despite adversary sophistication in stealth techniques, in sufficient time to provide a managed intervention for a significant subset or category of industrial control systems. Each of the techniques individually also warrants additional research.

1.5 Research Contributions

Our work makes the following original contributions:

- A formal adversary capability model, adapted from Dolev-Yao, suitable for dealing with the action of malicious processes/agents with a suitable π-calculus instantiation

- A traceback technique for detecting the source of integrity attacks in networks
- A causal model for creating linearizations of host behavior and an associated observation mechanism for detecting kernel-level rootkits
- A "proof of concept" approach to identify aberrations in non-linear plants under control, using basic systems engineering techniques

1.6 Original Papers

Our research is based on several original papers which we summarize here. In [77], we define a formal adversary capability model which would be capable of representing the action of malicious software in systems. We extended this model to include the ability to reason about groups of processes co-operating to achieve malicious goals in [76]. We used variants of the π-calculus to express our theory. The approach is primarily geared to proving security protocols and we provide examples drawn from [76, 78] but can also be used for research in risk analysis and attack modeling [79].

In [80], [83] and [85] we set out our approach to modeling expected behavior in networked and multiprocessor systems. We also model and demonstrate the use of multi-threaded observer process to monitoring such environments, using the relatively simple example of a multiprocessor host to illustrate our approach [85].

Finally, we made use of process control engineering disciplines to create a mathematical model of how a specific non-linear biochemical process should physically behave given certain reported observations from the SCADA system and comparing these, using independent secondary measurements, with actual behavior. We used a basic process control model to demonstrate the concept [84, 81]. We further suggest that the two approaches of causal and data models can be conjoined [86], providing the basis for future work in using process algebra with timing and sophisticated state estimation techniques to prove the success of detective approaches for, at least, a subset of such systems.

1.7 Outline of Dissertation

Chapter 2 describes notational conventions and nomenclature.

Chapter 3 provides a literature review, including some background reading on distributed systems and the mathematical prerequisites for understanding our research.

Chapter 4 sets out in detail our research problems and our approach. We provide an overview of the difficulties of modeling adversaries and distributed systems and taking into account the actions of operators and of environmental factors. We discuss the difficulties of detecting such an adversary and state our assumptions. We describe context-based anomaly detection and its application in our research.

Chapter 5 presents the adversary capability model . We discuss the conceptual models relating to distributed systems and to malicious adversaries making use of software attacks. We describe our approach to modeling adversaries in such environments. We provide two formal instantiations of the threat model, using the π-calculus variants which we developed for the purposes of this research. Additionally, we provide a technique, using our approach, which can be used to calculate impact analysis in such systems.

Chapter 6 is a direct follow on from chapter 5 and provides two examples of how our formal modeling techniques may be used while giving an account of a suitable security protocol, based on IP Traceback techniques, for anomaly detection and locating potentially malicious processes. This chapter provides, therefore, both a demonstration of how we can model processes and agents in systems and also our first detective technique based on context-based reasoning.

In chapter 7, we focus on the question of host integrity and presents a process algebra and a process graph variant, order-isomorphic to the algebra, which may be used for specifying probabilistically the set of possible traces of a host system. We provide a "proof of concept" observation mechanism for observing such traces on a multiprocessor Linux host to demonstrate the validity of the approach. The code may be provided in electronic format on request.

In chapter 8, we show how we can utilize additional sensor readings in a nonlinear control system which permit us to detect aberrations in its behavior which may indicate potential system subversion. We provide a simulation of the process control system to illustrate our approach.

Chapter 9 sets out our conclusions, discussing both the contributions and current limitations of our approach, and suggests how each of the individual topics covered in the previous chapters may be followed up as well as discussing the possibility of combining these approaches to develop real-time detection and response capabilities, using sophisticated state estimation techniques for anomaly detection in distributed environments, and proving these methods using a combination of a process algebra with timing and process control techniques.

1.8 Summary

In this chapter, we have outlined our research questions and the approach we have taken to them. We have also listed our original contributions and the papers which form the basis of this dissertation.

Chapter 2

Notation and Definitions

One Mathematical Cat, Please!

Dr. Carol J.V. Fisher Burns

2.1 Preamble

This chapter outlines the mathematical conventions followed in the text and defines key terminology. A glossary and appendices supplement this chapter with further detail.

2.2 Set and Algebraic Operations

We use conventional notation for set representation and operations. Sets are generally assumed to be finite and naïve set theory is sufficient as a theoretical foundation. Capital letters are used to represent sets $A, B, \ldots$. We use $\mathcal{A}$ to refer to a set with sets as its members where members are distinguished by subscript notation, hence $\mathcal{A} = \cup_i A_i$. Membership is denoted by $a \in A$ or $A_i \in \mathcal{A}$. Let A be a set, then 2^A is the power set of A.

Arithmetic and algebraic conventions are followed. Variable names $x, y, z, \ldots$ or $U, T, R, \ldots$ may be upper- or lower-case in line with conventions for representing engineering calculations and subscripted, if required. The normal operations $\{+, -, ./\}$ are used with their expected meaning. Where it is clear ab is substituted for $a.b$ to indicate a product.

Dot notation (Newtonian) is used to represent differential products. Let x be formula then $\dot{x}$ represents first-order differentiation over x and $\ddot{x}$ represents second order differentiation over x.

2.3 Distributed Systems

A formal introduction to distributed systems, along with the examples used in our research is provided in appendix B. Such systems consist of *processes* and *channels*. Processes consist of non-deterministic choices of sequences of functions. The most basic functions are communication functions - for sending and receiving messages. Other functions may be introduced for transforming the content of messages. Channels are not treated separately from processes in this thesis. They are represented by corresponding functions for sending and receive messages between distinct processes. Messages may be assumed to represent data items or vectors of data items.

In general, we use capital letters $P, Q, R\dots$ to represent individual processes which may be subscripted using the index set $P_1, P_2, \dots, P_N$, where N is the number of processes. P may also represent a set of processes, where subscripts are going to be employed [1]. The meaning is clarified by the context.

Messages are treated as variables which are represented by lower case letters $u, v, w, \dots$ which again may be subscripted. A message may stand for a single data item such as the name of a process or channel or a data value or for a complex data structure such as a packet. A vector of messages is represented by using $\tilde{v}$. The usage should again be made clear by the context.

We use subscripted variables s_i and e_i to refer to states and events in distributed systems and correspondingly refer to the set of states S_i for a process P_i or set of events E_i. A set S is the set of all states of all processes P (likewise E,though we treat only with states in our research).

In general, states are treated as partially ordered sets or posets. We also define and create de-composed partially ordered sets or deposets which are partial orders de-composed as lattice structures with each state having an infimum and supremum (*inf* and *sup*). Various partial orders may be used in relation to states– a total order or chain, "happened before", potentially caused and caused[2].

Order relations may be represented by the notation $(S; \leq)$ or $(S, \rightsquigarrow)$. We say, under a given order, that a state s precedes a state t, written $s \preceq t$, or vice versa $t \preceq s$ else we write $s \| t$ meaning that s and t are incomparable. We may also distinguish between $s \preceq t$ in a local process, written $s \rightarrow t$, and $s \preceq t$ in communication between two , written $s \rightsquigarrow t$.

2.4 Process Algebra and Calculi

We provide a more lengthy introduction to process algebra and calculi in appendix A. In general, we prefer the term "process algebra" to describe both algebras and calculi as these theories of processes represent the same family of mathematics [5] where distinctions between different algebra depending on choices over functions, concurrency and binding of operators. The algebraic representation of processes in process algebra bears a strong relation to the use of notation in dealing with distributed systems.

In general, processes and sub-processes are represented using capital letters P, Q, R and may be subscripted. The term sub-process is used loosely here. A process may consist of a single sum of non-deterministic sequences of functions or a set of concurrent processes. But a non-deterministic sum in a process may also be regarded as a subprocess. Processes consist of functions which may be regarded as the names of states which are constant 0-ary functions or n-ary functions over names (where names are messages as defined in section 2.3).

The operators we use to express order relations between states are dot $a.b$ to indicate sequence, which may be written ab where there is no ambiguity, $p + q$ to indicate non-deterministic choice, which may be exclusive for two sequences p and q and $p|q$ to indicate that states p and q are concurrent. We also use $\bar{a}\dots|a\dots$ to

[1]We do not use $\mathcal{P}$ here as P is not a set of sets.

[2]The last is generally not used because the cost of computation is too high.

indicate sending and receiving of messages along a channel a, i.e., a communication function.

In the π-calculus, we also use the period $x(s).y(z)$ to indicate a sequence of events. We expand the set of operators to distinguish between exclusive choice $p \oplus q$, which indicates either p or q but not both, and contingent choice $p + q$, which means either p, or skip, followed by either q, or skip, dependent on the validity of the action. We also add silent functions which are indicated by Greek letters $\alpha, \beta, \ldots'$ or symbols, e.g., $\uparrow$, and defined for each modeling application. We used brackets $\langle \ldots \rangle_c$ for names which are being sent and $(\ldots)_c$ for receiving names. Characteristics c which may be associated with names are shown as a subscript external to the brackets as they may also relate to a characteristic of the communication function such as routing which are *free* until a name is received and only bound to the characteristics of the name received. The process definition operator for the π-calculus is $::=$ and for individual process states during a proof by $:=$. Just $=$ is used for the process algebra defined in chapter 7. Action relations or transitions between states are represented by $\xrightarrow{a}$ where a is the event which causes the transition. In a proof reduction, each succeeding set of equations representing state is preceded by $\rightarrow$.

We define n-ary functions in the π-calculus algebraically and functions *result*, written $\supset$, in a vector of names, which may be empty. The $\supset$ operator symbol was selected to avoid confusion with the proof reduction symbol $\rightarrow$, which would be caused either by using $\rightarrow$ or else $\mapsto$.

In the process algebra, proofs relating to bisimilarity are not required in our research. Our aim is to create a set of traces which represent the linearizations of a defined set of concurrent processes using term re-writing to create a sum of alternate possible sequences and their probabilities. We use $\Rightarrow$ to show a concurrent process implies a non-deterministic sequential trace. We formally prove π-calculus expressions using the techniques of reduction and actions relations. These are explained in appendix A and we introduce these sections by PROOF, though no theorem may precede them.

2.5 Terminology

We have already discussed the terminology of distributed systems and process algebra in sections 2.3 and 2.4 respectively. Further information may be found in appendices A and B. The two example distributed systems we use in our research are also covered in appendix B along with associated terminology. A glossary of all terms is provided at the rear of this work, but we cover the basic usage of terms in this section.

Critical infrastructure systems here refers to any system where a computer is used to control physical phenomenon in order to ensure not just our prosperity but also our survival as a civilization (using this term in the proper sense of "living in cities"). Without industrial systems ensuring energy, communication, transportation, waste disposal, water purification and so forth on a large scale, modern society could not function. As a corollary, attacks on such systems can disrupt and, in some cases, seriously threaten our well-being. There are various kinds of systems with distinct control infrastructures.

In our research, we focus in particular on industrial control systems (ICS) which are often, slightly misleadingly, referred to as SCADA (supervisory control and data acquisition) systems. The term "SCADA" is somewhat misleading because a SCADA system represents only a part of a control network. A complete control network will also contain a distributed control system (DCS). A SCADA system communicates with a DCS, also known as a field system, on a best effort basis, both to update telemetry and also to issue instructions to, for example, alter system settings by opening or closing valves or solenoids in actuators . The DCS executes instructions on a real-time basis and provides updated readings, either on a planned schedule, on update or on request, back to the SCADA system. The distinctions are, perhaps, less important today than they were in the past and sometimes we simply refer to process control networks (PCN) to emphasize the system under control. In the same way, remote terminal units (RTU) and process logic controllers (PLC) may be referred to as process control devices (PCD) to conflate what is now a less meaningful distinction in the devices' operating philosophy. We may also, more simply, talk about control units.

A final note on the use of the word "process" is that clearly computer *processes* and *process* control represent two distinct semantic meanings, the first referring to ICT systems and the second to kinetic processes. To make this distinction clear, we frequently use the term "plant" or "plant under control". *Plant* is also used as a term to mean a set of differential equations representing the (kinetic) process under control. The distinction is not all that important for our purposes and the reader may simply regard the plant as the control unit plus the kinetic process which it controls.

2.6 Summary

This chapter summarizes the mathematical conventions followed and basic conventions with regards to terminology. Where appropriate, the reader is referred to relevant appendices, which contain additional detail.

Chapter 3

Literature Review

Isn't there always a cat napping
on whatever you're reading?

Anonymous

3.1 Preamble

This chapter reviews the relevant literature and compares and contrasts approaches. The review has been divided by subject. We discuss our modeling approach and related work on the use of process algebras, software agency, risk analysis, network attack/defense modeling and formal threat models. We also outline the recent history of malicious software development and its detection. We examine, in general, approaches to intrusion and anomaly detection in relation to both SCADA systems and host-based intrusion detection strategies and the use of IP Traceback techniques for tracing the location of malicious activity. We list background reading material covering the prerequisites for reading this thesis.

3.2 Background Material

The material in this thesis on the nature of distributed systems was taken from [46]. The discussion of multiprocessor systems was based on [16] and is summarized in appendix B. The following books provide a thorough grounding in industrial control systems and related security issues - [67, 17, 65]. A summary, based on these sources, is provided in appendix B. The process engineering techniques were derived from [13].

3.3 Process Algebras and Calculi

Automated testing is one basis for specifying systems and proving the validity of protocols or the safety and liveness of system behavior [12]. However, a process algebra allows us to create an abstract specification which may also be verified using equational reasoning or using model checking or a combination of both [7]. Our algebraic formulation of the protocol uses an applied π calculus [34] which was developed from CCS (calculus of communicating systems) by Robert Milner [88], along with Davide Sangiorgi and David Walker.

The development of process algebra and calculi during the 70s and 80s provided a theory for the development of concurrent and distributed systems in the same period from early beginnings in the 60s. Previously, only Petri nets provided a theory of concurrency or message passing and program semantics were cemented into a input/output model which did not take account of non-deterministic aspects of concurrent and distributed programming. Our use and variation on these models represents a long line of continuing evolution [5].

One variation to the π-calculus was to make use of the concept of *silent actions* to represent explicitly actions which are unobserved, but which may, in fact, be vital for an operator to observe in order to take appropriate action w.r.t. the state of system. This is clearly distinguished from the normal use of a silent action to abstract from an interaction between processes which need not be observed. In addition, we also extended the use of the silent action to functions which could act over names (e.g., loss, delay) or processes (e.g., subversion). This extension was particularly useful in developing a means of calculating the impact of agent actions on systems in terms of their security characteristics.

The other variation to the π-calculus was transforming processes or sub-processes into goals which could be grouped as families to create agents. This alteration allows us to reason at a higher level about the potential outcomes (in terms of success and failure) of the actions of agents. At the same time, we can transform this higher order calculus into our usual π-calculus variant to check explicitly conditions for success or failure of goals within the system.

The distinction between $+$ and $\oplus$ operators in our π-calculus variants, representing contingent and exclusive choices, which is not present in the original, was primarily introduced as a convenience for dealing with re-ordering of summations under priority, but also allowed us to deal neatly with algebraic subclauses which could be feasible or unfeasible under different conditions. It also provides greater flexibility in specifying conditional branching structures.

Our algebraic specification for a causal interpretation of distributed systems was based on a variant of CCS [88][1], with graph-theoretical elements clearly drawn from work on finite state automata and process graphs. Our primary motivation in developing this approach was to abstract away from communication considerations when dealing with disparate parts of a distributed or multi-threaded concurrent system and focus on causal links.

3.4 Threat Models

We base our algebraic approach to the formal modeling of adversary capabilities on [74]. However, we altered assumptions about communication and access capabilities in the network. From the beginning we were not concerned with the secure exchange of key material in systems, but with the feasible behavior of adversaries in systems. Moreover, we broke sharply with normal assumptions regarding such threat models by stating that our work focused on what we defined as *closed distributed systems*, not open networks. Importantly, our model is not monolithic - meaning that malicious processes are not automatically the recipient or communicator of all messages in the network [23][10]. In other words, we do not assume a single adversary, but multiple potential adversaries or, at least, malicious processes which are necessarily distributed in action and affected by the same issues

[1]Or basic process algebra (BPA), depending on the reader's taste.

as other agents in such an environment – a lack of knowledge of global state and time, process failure or delay and communication loss or delay.

We expanded our model to include the concept of agency. Software agents may act on autonomous decisions based on their perception of the environment [129], possibly including learning behaviors. In recent years, malicious code has exhibited increasingly agent-like behavior. This trend is likely to continue [87, 98]. Software agency also permits the attacker to launch more sophisticated attacks, including co-ordinated attacks. There are several advantages to the attacker in implementing this class of attacks [20, 19]. Dealing with such attacks also requires operators to make dynamic interventions in the face of changing adversary behavior as the result of agent cooperation. This requirement is underlined by the intrinsic nature of critical infrastructure systems [17, 65] which may require to continue operations, even in a compromised state. Finally, the scale and complexity of SCADA systems makes dealing with a co-ordinated attack unfeasible by human operators. The use of agents to respond autonomically in relation to a wide range of security tasks provides a distinct advantage in countering agent-based attacks and could be considered a long term goal of our research [118].

Our approach also lends itself to applications in risk analysis and attack/defense analysis. Attacks and countermeasures have been modeled using attack/defense trees. These techniques impose a logical order of alternating attack and defense moves which counter each other. Game-theoretical approaches can be shown to be equivalent in outcome [75, 63, 15]. However, such approaches do not necessarily capture the (logical temporal) ordering of events or order dependencies between sub-goals [75]. In our approach, it is possible to capture these orderings.

Another approach uses attack graphs to model and calculate exposures to network attacks. However, in critical infrastructure networks, such techniques would be limited due to the requirement, in many cases, to use vulnerability scans to calculate attack reachability and limitations in scale and complexity for many such approaches [55, 72]. Such approaches also appear to assume that the subversion of a critical host is a requirement for attack success, whereas we argue that in distributed control system the subversion of any host or network node may – due to the transitive effects of loss of data integrity, availability or confidentiality – turn out to be critical.

In contrast, the use of attack coordination graphs addresses the issue of discovering coordinated attacks and allows the generation of novel attacks, but again appears to ignore the possibilities offered by the transitive effects of attacks on information flows in the system [19].

Our approach is based on the formal adversary capability model and the associated applied π-calculus [34, 77] used to define the capabilities' action algebraically. However, we extend the approach using a new class of silent functions which allow us to capture manipulated, diverted or denied traffic flows and determine the impact on the system by considering the impact on application functionality in support of the business processes. This is most similar to the use of Petri nets to model penetration attacks [130, 32] in the sense that we can model attacker access and communication within the network and determine subversion paths. At the same time, however, we have the advantage of being able to consider direct data manipulation and subsequent impacts on the business processes the system supports.

3.5 Developments in Malicious Software Techniques

It is difficult to discuss the development of malicious software techniques without immediately discussing intrusion detection methods and vice versa. The development of each drives the advancement of the other. However, this section primarily focuses on the development of stealth techniques and the threat to critical infrastructure systems which results. Subsequently, we will discuss the methods used to detect the presence of malicious activity on host systems and in networks.

3.5.1 Background

Szor [117] provides a definitive and relevant account of the design and implementation of malicious software in the form of viruses, trojans, logic bombs and worms and so forth. Various attack vectors are addressed – such as boot sector viruses, overwriting viruses, appending and prepending viruses, parasitic strategies, obfuscation, metamorphic and polymorphic techniques, in-memory techniques and self-protection strategies – which constitute a fairly complete picture of how malicious software has developed from early days. All of these techniques continue to be of interest, not simply because history repeats itself – for example, boot sector viruses migrating to CDs. DVDs and USBs – but also because even attacks with novel features are likely to incorporate previous tactics [29].

3.5.2 Goals

The goals of malicious software writers vary from gaining kudos, winning IRC chat wars, competing for resources such as botnets through fraud or other criminal gain, to supporting political or state-sponsored goals [117]. In all cases, they represent a threat to the confidentiality, integrity and availability of systems.

A notable trend in modern malicious software development is a move towards targeted, or semi-targeted, attacks which may be state-sponsored in origin. These attacks are known as APT (advanced persistent threats) and make use of social engineering techniques to achieve their goals, an example being [69] – with relevant ICS-related examples being [41] and [11]. Using either email or media, victims are persuaded to take actions which load a dropper program for the malicious payload onto their devices. Once the main payload has been downloaded, it can act arbitrarily with regard not only to the host system it is on, but also, in general, in relation to the network of which that host system is part, since many systems focus primarily on perimeter defenses and ignore the enemy within. The "social engineering" angle could be said not just to cover the human angle, but also target trust mechanisms in systems, for example, using forged certificates to pass malicious software programs as legitimate.

Such attacks can aim to steal information [69]. More serious and more pertinent to our analysis, Stuxnet [41] represented the first highly public attempt to cause damage to an ICS by manipulating control parameters and it has been followed by other similar attempts such as Shamoon [44]. The key danger to such systems lies in the loss of availability and loss of integrity due to attacker manipulation of hosts and the subsequent manipulation of communication between operators and control units which hides the attack until too late [43].

3.5.3 Stealth Techniques

A key goal of attackers – and most relevant to our research – is the desire to hide themselves from defenders. This effort can be divided into two phases –

- Concealment during initial penetration
- Hiding the fact or nature of the attack
- Maintaining a stealthy presence on systems

Original techniques for detecting malicious software during penetration primarily relied on what is known as *misuse* detection. These techniques make use of characteristic signatures representing short sequences of code or data strings which could be used to identify a malicious program. These methods are still employed today by commercial antivirus software (along with some of the others we will describe). Such techniques are bypassed by malicious software writers making use of software obfuscation techniques, similar to those employed in digital rights protection [101]. Malicious software writers frequently encrypt their code during transit and decrypt and execute it on successful insertion. Signature based techniques can still work against decrypted code, so, in addition, *polymorphic* viruses introduce obfuscation into the decryption algorithm – for example, introducing multiple NOPs or changing the order but not the effect of instructions. Malicious software writers may also alter the complete structure of viruses (*metamorphism*) and also obfuscate the virus simply by adding further functionality [28, 90, 107].

Furthermore, during an ongoing attack, malicious software retains a true view of the system state and presents defenders with a falsified picture of its state. Various categories of attack behavior enable this. An attacker can hijack the control flow of a program so that legitimate code is not executed, but only a malicious version of that code. By control interception, the adversary may alter the control flow graph (CFG) of code so that it acts maliciously. The adversary may also create software which does not directly hook attack code but "taps" into its usage, for example, manipulating the *execve* command in Linux systems. Furthermore, it is possible for the attacker to contaminate kernel data (direct kernel object manipulation or DKOM), for example, to reduce the effectiveness of random number generation or to manipulate data structures producing inconsistent views of kernel states such a process or module hiding or manipulating cache structures in security software modules [9, 93, 100].

Detecting these contradictions, after concealment techniques have been applied, represents a severe problem in commonly used computational models (see, e.g.,[91, 24]) as any illegitimate states are subsequently indistinguishable from legitimate ones. The level of difficulty is increased because multiprocessing operating systems are characterized by non-determinism – that is, computation on such systems must be regarded as a set of partial orders where memory management may force one of a set of potential linearizations on the outcome.

Either during or after an initial attack, malicious software writers and hackers may also seek to establish a more permanent presence on a system by using rootkits – so-called because the attacker needs to have escalated his privileges to superuser (or root access) level to be able to deploy them fully. Like other malicious software techniques, such methods have evolved over time as detection methods have improved. Rootkits first appeared during the early 1990s. The sophistication of such concealment techniques continues to grow and its evolution could be characterized as an attempt to work "below the radar" in the literal sense of burying malicious payloads at lower and lower levels of abstraction in operating systems. Various kinds of rootkit exist. Applications at user level can conceal such code. Software libraries used during compilation time may be affected. More recently, attackers have focused on creating various means of hiding in kernels making use of loadable kernel modules (in Linux) and dynamic linked libraries in Windows [52, 56, 29]. These attacks make similar, but ongoing use, of the concealment techniques we addressed above. They may also introduce backdoors into systems, allowing continued ease of access, continue to attack the system (for example, using key loggers or password crackers) and may also be used to attack other systems. The presence of such software is characterized by the layering of techniques to create an "offense in depth" capability which renders it difficult to be certain such code has been successfully removed, particularly since it works actively to remove evidence of its existence and to undermine detection mechanisms [29].

Novel techniques also continue to be developed [119]. Researchers in this area have also demonstrated that virtual machines and hardware components or operating characteristics can also be used as a means of concealing the presence of malicious code. In [50], the writer demonstrates how the ACPI BIOS can be altered to download malicious software during the boot sequence with minimal alteration of kernel structures. Rutkowska [105] shows that it may be even be possible to defeat hardware-based monitoring and forensics techniques such as [8]. Virtual machines become another way of hiding malicious code, either operating as a separate guest on a hypervisor or making using of virtual machine manipulation techniques to alter state undetectably in [106]. The operating characteristics of hardware on certain processors may also be manipulated to create hiding places for such software by manipulating memory spaces [33].

Finally, attack concealment applies not only to how malicious software obscures the presence of its artifacts on systems, but also how it can attempt to conceal its behavior from system operators by manipulating signals. Relevant to our research are attacks, or potential attacks, which specifically target ICS [120, 116, 23]. ICS are publicly known to have been subject to attacks incorporating the use of concealment techniques with the capability to manipulate process signals on control units [26], or communication routes [120] on a long term basis. Such attacks may prevent an operator from acting to a relevant alteration in system's state, or, alternatively, cause an operator to act inappropriately in response to a falsified report of the systems state. The work by Svendsen et al. [116] is of particular interest since it demonstrates that a naïve use of statistical techniques can yield a misleading result because attackers would be capable of "hiding in the noise", something we also demonstrate in our research in chapter 8. This requirement is underlined by the introduction of sophisticated control processes which rely on multivariate controls and hence require more complex forms of supervision [110] and by the alteration in turnover rates in the labor market which means that experienced operators who might be capable of sensing a change in system state due to their expertise in a given system are likely to operate such systems and detect faults intuitively [43].

Clearly such attacks provide a strong motivation for research into defense techniques due to the potential damage which could be caused, particularly in relation to industrial control systems (ICS) [65].This can be split into the analysis of the integrity of communication on the one hand and the integrity of process control units as hosts, on the other, including the integrity of signals communicated by sensors.

3.6 Intrusion Detection

In general, intrusion detection systems may be classified as host-based or network-based [42]. They may use signature-based identification or heuristics [68]. They may depend on statistical analysis for anomaly detection [59], or specify acceptable system behaviors, thus identifying unwanted behaviors [62]. There has been an emphasis on switching analysis from examining usage anomalies to process anomaly detection - see [51] for an early example. Our work carries similarities to this approach, but we distinguish it by focusing on the computational outcomes of processes rather than their operation and thus we abstract away from architectural considerations.

Our approach is most closely related to techniques in semantics- and specification-based detection - for example, [1], but draws on the current context of systems behavior, including non-computational aspects, to overcome the limitations currently imposed by considering program behavior in isolation [28]. We call this context-based anomaly detection after [113].

3.6.1 Network-based Intrusion Detection on ICS

ICS form a vital part of the critical infrastructure and are attractive targets for adversaries. Such systems are increasingly vulnerable to cyber attack due to modernization and exposure to untrusted networks [124, 92, 65, 22] leading to increased interest in intrusion detection [104]. As for conventional network systems, existing work divides between signature-based approaches, generally at perimeter defenses, and anomaly-based approaches which address the insider threat and direct attacks against control processes [43, 132, 71].

Our approach to detection is based on discovering anomalies in data content in the network which are traceable to potentially malicious hosts or processes. Two techniques are presented. The first is based on IP Traceback methods, normally used for detecting the source of distributed denial of service attacks, and the second on using process engineering techniques to uncover sensor anomalies. Unlike much intrusion detection research which targets large-scale manifestations of malicious activity such as worm infections [60], our work considers low-level covert data manipulation – which over the lifetime of a system may be ultimately more damaging to systems performance [73, 40].

3.6.1.1 Traceback Techniques to Uncover Integrity Attacks

Although the study of IP Traceback mechanisms was originally motivated by the existence of denial-of-service attacks, as noted in [36, 109] there are other potential uses for these techniques such as the analysis of legitimate traffic in a network, congestion control, robust routing algorithms and dynamic network reconfiguration. Nonetheless we have not encountered in the literature examples of the use of Traceback techniques for other purposes. Our example of using Traceback mechanism to determine traffic integrity and to trace threats to traffic integrity appears to be relatively unique.

We originally proposed the use of Traceback techniques in [86]. But we complicated our approach by the requirement to compare sensor readings and the content of instructions concurrently. The approach we set out here, based on [76], divides the problem of considering network integrity from the issue of sensor integrity, and consequently makes fewer assumptions. If we can determine where node subversion has taken place, the use of state estimation techniques [114], can be confined (except possibly for initial detection of anomalies) to consideration of the integrity of control systems, hence lowering the overall computational burden on our approach.

In our work, we adopt the assumptions established in [109] for DOS attacks regardless of a specific target domain:

1. Packets may be lost or reordered
2. Attackers send numerous packets
3. The route between attacker and victim is fairly stable
4. Network nodes are both CPU and memory limited
5. Network nodes are not widely compromised.
6. An attacker may generate any packet
7. Multiple attackers may conspire
8. Attackers may be aware they are being traced
9. A compromised router can overwrite any upstream information.

Clearly, these assumptions, with some change of emphasis, are as relevant with regard to packet manipulation, dealing with loss of message integrity in a SCADA environment, as they are to tracing the source of (possibly distributed) denial of service attacks. We further augment these assumptions using a formal adversary capability model we have developed for SCADA systems which postulates an adversary which can subvert processes to work as "agents" on its behalf – see chapter 5.

There are a variety of traceback techniques. A useful, if brief, survey can be found in [66] which summarizes currently researched techniques and the payoffs in terms of storage and complexity. Such techniques are easily adapted to tracing manipulated packets in distributed networks such as SCADA systems [128].

Each technique normally represents a payoff between router storage, packet length and form and message complexity. Protocols may either use "pipeline" techniques where packet routing information is stored as partial message copies at routers and forwarded later to reduce throughput, or else linked lists inside packets containing routing information. Such lists clearly grow rapidly in size. These payoffs may be reduced using hybrid techniques and probabilistic packet marking [96, 3]. iTrace is a variant ICMP based traceback [115, 57] where a copy of the packet is and sent to both the originator and destination addresses (in the context of detecting nodes causing denial of server) – and is related to our technique which also uses a form of message passing. Another interesting approach is an algebraic alternative using a finite field $GF(p)$ and solving for a polynomial of degree d with at most $d+1$ hops [36]. Another similar technique requires solving a linear equation over a field F_q [108], but allowing a number of router nodes to be stored, bringing results similar to those we achieve in terms of reduced complexity. A paper which demonstrates the advantage of gathering information about sets of nodes, i.e., paths, rather than individual nodes, is [61]. These results are also comparable with [39] which similarly uses node authentication information to reduce the number of packets required to trace subverted nodes during DOS attacks. Better results are obtainable by deterministic marking schemes – for example, [131].

The advantage of our approach is that, since both the origin and destination of packets are known, it enables us to efficiently pinpoint agent processes using a two-directional process of elimination . It also identifies packet injection as well as manipulation. Compared with probabilistic packet marking where the expectation of the number of packets required to trace an attack is given by a hypergeometric probability distribution and is in the order $O(n \log n)$ [109], we are able to use knowledge of network routes and operations (defined algebraically) to rapidly eliminate valid nodes with notable economies. Although it is possible to apply our approach deterministically, we selected against this approach because the extra load on communication could undermine some performance sensitive industrial control systems.

The packet encryption scheme we use is not original and is based closely on the one proposed in [115]. This approach is resilient to attack since insufficient time exists for guessing keys while both the packet identity and its contents are hidden from the attacker [133], hence packets cannot be directly manipulated and may only be arbitrarily delayed, dropped, re-routed or vandalized. However, these forms of attack only delay rather than disrupt discovery – section 6.2.6.

The traceback protocol developed was an example of what we call a context-based anomaly detection technique – making use of the identity relation between message packets in a distributed system. In common, with the other techniques we develop, it requires multiple, concurrent observers. In this cases, all feasible process nodes in the system may be be implicated, albeit probabilistically, in the observation of messages as they pass through the system. The development of a traceback protocol for checking the integrity of communication channels yielded some nice results in terms of how the additional information garnered could be employed to locate the adversary or to retain control of a system under attack. The work is incomplete in the sense that no particular existing protocol was identified as a candidate for the approach or tested in a simulation in contrast to normal testing procedures such as examples in [12]. However, our aim at this stage was to establish the principle formally that such techniques could be used, opening up the way for future research into practical applications.

3.6.1.2 Using Sensor Measurements to Uncover Integrity Attacks

Recent research on the security of SCADA and DCS systems has focused on anomaly detection at the protocol level, since traffic on networks interconnecting SCADA or DCS components should be well-characterized and hence particularly amenable to such techniques, e.g., Coutinho et al. [30]. The predictable nature of SCADA protocol and usage is perceived to be an advantage in detecting system anomalies [27, 132, 120], but the limitations of this approach in the face of knowledgeable attackers capable of manipulating computational states or utilizing signal "noise" to obfuscate attacks, based on legitimate commands, making non-anomalous use of ICS protocols – have been clearly recognized [132, 120, 116, 23] – that is, in the presence of channel compromise, adversaries may use protocols correctly and present both syntactically and semantically correct messages, resulting in a failure to signal anomalies by conventional detection techniques. We regard this as particularly true of non-linear systems [77].

Approaches using physical state estimation techniques have also been researched [113], but these have been limited to linear systems. But many industrial systems, including biological and chemical processes, however, exhibit non-linear behavior or require non-linear control laws. This result in limited accuracy or less well-defined conventional models [43]. Real-time detection techniques are an important area for research in this context [25].

The use of additional sensors [14, 30] which make use of differing points of view for anomaly detection [120] has been proposed. Taking this approach, we show that success can be achieved even with a small number of such sensors. We use functional models to map systems [99] and identify suitable *redundant* characteristics for evaluating process behavior. We combine readings of these values with a simulation of process operation to detect signal manipulation [13]. This approach obviates the need for linear approximations found even in explicit control models as used, e.g., by Lin *et al.* [70]. This approach has strong parallels to fault detection strategies in SCADA environments [123], but in contrast to fault detection approaches, sensors cannot be assumed to be – in general – reliable. It also requires us to have reliable models of system's behavior which we can base on well known process engineering techniques [13] and, in future, advanced state estimation techniques such as non-linear Kalman filters [114].

3.6.2 Host-based Malicious Software Detection

As we have seen in section 3.5, intruders on computer systems may make use of a variety of concealment techniques on hosts in both user space and kernel space to hide their presence and to disguise alterations to distributed operations in networks. It is known that malware detection is undecidable[28] which explains the co-evolution of malicious software techniques and detection methods [101]. It is also clear, therefore, that there is no "silver bullet" in terms of detecting malicious software and techniques are necessarily additive.

No one detection mechanism will be capable of detecting all potential attack vectors, although we will argue in this thesis that attackers are forced to reveal themselves, especially during the early stages of attack, by alterations to the state of security mechanisms and business logic.

In addition, a primary concern has become to protect the detection mechanism itself from attacks which would seriously compromise its ability to perform, using both obfuscation techniques and by isolating it from direct tampering, while, at the same time, ensuring that the "semantic gap" between the knowledge of host state by the monitoring system and the real-time state of the host does not become too wide [135, 58].

On host systems, co-processors for general security applications have been proposed in [134] and for intrusion detection in [135] and [89]. The advantages of such approaches are that they render the ID mechanism in general immune to tampering and they enable the use of various ID techniques such as sampling memory areas, process flow and memory shadowing. Their disadvantages are that they are unable to interpose themselves between the malicious process and the operating system and because they operate asynchronously, they suffer limitations in their ability to reconstruct kernel states [135].

More recently, VMM-based (virtual machine manager) approaches have proven popular such as proposed by [45] and [125]. VMM permits techniques such as memory shadowing as discussed by Riley [103] and tracking instructions and calls as exemplified in [37] and [54]. Techniques also include seeking to reconstruct state based on a VMM "snake's eye view" of operating system actions [58] and tracking changes to control flow induced by malware [100]. Advantages include once again isolation from attack (although some data-based attacks may theoretically be possible) and, in this case, VMM approaches are capable of interposing themselves in the way of malignant alterations to the kernel either through preventing such alterations directly or by redirecting kernel space invocations to a known good copy of the kernel, thus passively foiling the success of any alterations. Disadvantages relate to the additional overheads placed on processing tasks and the cost of reconstructing a picture of kernel activity from observations of hardware state and the subsequent loss of granularity.

Multiprocessor approaches to intrusion detection have been described in [127], applying the co-processor approach to a multiprocessor environment monitoring execution through dividing up security and productions tasks between processors. A multithreading-based approach to monitoring system reliability is shown in [95], similar to that in [94], proposing the use of speculative execution and parallelization of security checks on multiple concurrent processing units. These approaches lose the advantage of complete isolation from malicious software, though they retain some aspects of it. They gain greater granularity in their view of the kernel, although this has to be weighed against costs in performance overheads. They are also able to interpose themselves between malicious software and its target, but to do so must block processing at certain key stages. Other approaches include sampled execution of kernel modules before loading to determine malignancy [126] similar to "sandboxing" processes used by anti-virus software [117], this traces the control flow of the module and uses naïve Bayesian inference to determine whether or not the module is malicious.

An alternative to both co-processors and VMM is the use of PCI cards and software engineering tools such as Daikon for both automated inference of data invariants and detection of violations . This approach can be undermined by malicious manipulation of hardware and does suffer, as do other off-operating system methods, from asynchronous readings of state [105][8].

A key handicap in specification-based approaches to intrusion and anomaly detection is that the static analysis of systems is limited by the constraint that it is problematical to generate and test all possible inputs to a system [101]. Work has been carried out on overcoming this limitation [8].

Another approach which uses kernel resident software uses pro-active memory guarding techniques to prevent rootkits from loading [18, 102].

Our approach to host-based detection in chapter 7 is based on semantic analyses and makes use of multiprocessor technology, but in a way which is intrinsic the environment rather than as a security "add-on". We do not rely on obfuscation, concealment or isolation to prevent attacker tampering. Rather we utilize the intrinsic advantages of operating in a concurrent, non-deterministic environment to render our mechanism unfeasible of prediction and control and benefit from having a potentially large number of simultaneous observers monitoring both attacks and the defensive mechanism itself. We deal with asynchronous OS states – which we argue are an intrinsic feature of modern OS – by recognizing the need to have a strong model of the environment to deal with such states and associated dynamically altering structures [80, 93, 122, 49]. Our proof-of-concept implementation benefits from having direct access to kernel states in contrast to both co-processor and VMM approaches, allowing us greater granularity of analysis, yet without the performance overheads associated with utilizing speculative threads. In addition, our model includes determining from the outset where attacks will be focused, thus reducing the overall burden of kernel integrity monitoring, which we underline by utilizing lightweight threads and a message passing architecture which does not place undue burdens on operating system performance in current systems. A current limitation is the inability to intervene in process flows.

We combine this concurrent observation mechanism with a semantic model of system behavior which uses the process algebra to specify cause and effect between, possibly disparate, parts of the operating system [83]. This allows us to define violations of rules regarding data structure consistency [9, 122, 8] which can subsequently be detected by our concurrent, multi-threaded observer mechanism. However, with the difference, that instead of inferring invariants from data structures, we propose to use the relationship between data structures to infer expected behavior. For example, in a Linux system, if all_tasks alters, then we can also reasonably expect run_list to alter. We base some features of our approach – such as the use of monitors and vector clocks on [112], but our approach may be distinguished by the creation of a language for specifying behavior across distributed systems as opposed to a single multi-threaded application and the proposed techniques calculating potentially malicious behavior and the creation of an external observation mechanism rather than one which is intrinsic to the application observed (and hence less vulnerable to subversion). Our approach is implemented as a kernel driver and focuses on semantically significant events similar to [97], but, unlike this approach, does not use multiple kernel drivers, but a single loadable kernel module with multiple observer threads. These methods are also self-guarding since they allow the observation mechanism to, at least, quasi-autonomously, observe its own action. This allows us take the more risky approach of directly engaging in intrusion detection in the kernel rather than isolating ourselves from it, hence closing the “semantic gap”, but protecting ourselves through the numerical superiority our observation mechanism and its ability to self observe. The following papers [122, 9, 8] provide further examples of attacks which this approach may be used to detect. We argue our approach is also applicable to distributed network systems - though the practical aspects remain a problem for future work [48].

3.7 Summary

In this chapter, we have provided a review of related literature and clearly delineated where we have made an original contribution to information security research. Our research could be characterized as applying previously discovered techniques to novel domains. We use a formal adversary capability model, a technique normally applied to the analysis of cryptographic protocols, to characterize the behavior of malicious software in networks. We make use of traceback technique, normally used to trace the source of denial of service attacks, to trace the source of an integrity attack. We exploit process graphs and process algebra to support modeling causality in systems between disparate, but related structures and process, rather than considering direct process interaction. Finally, we apply the use of additional sensors and a knowledge of control laws to show how we can predict non-linear system behavior in ICS for the purposes of anomaly detection – in contrast to previous approaches which focused only on linear systems and tended to rely on already extant signals.

Chapter 4

Problem and Approach

Honest as a cat when the meat's out of reach.

Old English saying

4.1 Preamble

We provide a formal statement of the research issues. Our aim is detecting and, ultimately, responding to the action of malicious software in ICS, even though the latter goal is outside of the scope of this current research effort. To do this, we need to be able to effectively model the action of malicious processes in distributed systems and to reason over its outcomes and the outcomes of any detective measures and subsequent interventions or the applicaton of preventative techniques. Subsequently, we can use an instantiation of this model to prove any detective countermeasures. We outline the three techniques we have selected.

4.2 Research Issues

We consider the following research issues: first, the ability of the adversary to conceal his presence and actions in systems, both at host and at network level; second, the arbitrary nature of adversary actions and systems; third, the issues we face in demonstrating the success and survivability of any detective techniques in the face of adversary knowledge or direct adversary attack, and, fourth (throughout), the additional constraints imposed on us by the nature of distributed systems and by the particular examples of distributed systems, industrial control systems, we have selected for our research.

In chapters 1 and 3, we describe the increasing sophistication of malicious software in disguising its presence on systems and undermining detective efforts. – including the ability to conceal the presence and action of malicious software on systems. For the purposes of our research, we assume an adversary which makes no protocol errors and has the ability to conceal system states from the operator. The difficulty of detection is increased in distributed systems and multiprocessor systems (which imitate their characteristics [46]), since a system run may result in multiple valid linearizations – see appendix B – while, at the same time, syntactic clues are obfuscated by techniques such as polymorphism and metamorphism and, equivalently, in kinetic systems, where the adversary may take advantage of "noise" to hide their presence in the signal [116]. Hence we require novel forms of detection which do not rely on unforced errors in adversary behavior, but can be used to interpret system states, so as to uncover his presence.

This leads onto the second issue. An adversary's goals are, at least, initially unknowable and hence their actions unpredictable. How can we predict the effects of his behavior and know how and where to apply any observation and monitoring methods? Malicious agents may be potentially found in any part of a distributed system. The effects of their attack will likewise be distributed across the system and may be transitive or recursive in nature. Examples of possible goals are: stealing system resources, using systems as communication relays, undermining process logic, stealing information or carrying out denial of service attacks. Even within these categories, there are numerous variations. These factors alone make predicting the possible range of anomalies difficult. We must consider not just what the adversary can do, but what the adversary must do. The ability to set up multiple possible scenarios with some degree of ease also contributes to understanding potential adversary actions.

It is not just the adversary which is unpredictable, but also the environment and its defenders. An adversary may not always succeed in subverting processes. Their goals, as the result of artifacts in the environment, may not always be successful. Operator actions and system defenses may prevent, or aid, the adversary in achieving his aims. Additionally, the system may be attacked simultaneously by multiple foes who can compete, or cooperate, to gain system resources. Any model of incursion and its effects should allow us to consider the actions of all agents in the system.

We need also consider the resilience of any detective techniques to attack. Since a primary goal of the adversary is to resist detection, an obvious means of achievement is to undermine or disable detective mechanisms. We assume that the adversary has a complete knowledge of such mechanisms and hence any method must be resistant to adversary subversion despite the knowledge the adversary has of the system and its defenses.

The final set of constraints under which we must operate are obvious, in the sense that they are natural to any computational problem – that is, any approach we propose should not exceed the performance or capacity of the systems for which it is proposed and should be of manageable computational complexity. However, it is worth noting that in ICS, these constraints are emphasized by the low computational capacity of the equipment, which is dedicated in nature and is intended to have decadal lifetimes, which means that we may have to use legacy computational devices which are not generally able to bear the implementation of modern security mechanisms that place high performance demands on the system, particularly in a real-time environment. This points to using techniques which have low complexity to implement.

To summarize, we are required to solve, at least partially, the following problems:

- How to model distributed systems under arbitrary interventions by multiple agents
- Develop formal or experimental techniques, at least, to "proof of concept" for anomaly detection in distributed systems

- Demonstrate that these techniques are resistant to adversary attack
- Show the techniques created are low in computational complexity
- Prove, formally or by experiment, that our methods take account of dealing with multiple linearizations of state.

4.3 Research Approach

We begin by setting out a threat model for the action of malicious software in systems. This adversary is characterized by the possession of data manipulation capabilities typical of Dolev-Yao type models – such as message replay, falsification and so forth – but differs from that approach by limiting adversary communication to a small subset of processes in the system rather than making the adversary the recipient and sender of all messages in the system. Hence the adversary is required to subvert additional processes and endow them with suitable capabilities in order to increase his control over the system. We can interpret these actions at the level of both individual processes and groups of processes (or goals) belonging to malicious agents which collaborate with each other.

We formally instantiate this model using two variants of the π-calculus . This modeling technique allows us to naturally consider the interaction of processes, whether malicious or friendly, in a distributed environment. It also allows us to freely mutate processes as they are subverted by the adversary. Adversary capabilities are considered to be functions in the system which affect how messages are treated. Hence we can set up various feasible scenarios and play out the resulting behavior of the system, including any potential interventions by operators, or friendly software agents which can also be represented functionally. This approach becomes the basis for developing any security protocols or methods we might use as it allows us to formally prove the outcome of our techniques.

For the purposes of our research, since we recognize the adversary may choose from a wide range of possible attacks, we choose to focus our attention on a smaller subset of integrity attacks. This is driven by the greater importance of integrity over confidentiality in critical infrastructure systems and, since we are considering stealth methods, attacks on availability, such as large scale denial of service attempts, do not qualify by their obviousness. We select three examples of integrity attacks where stealth techniques must be applied:

- Manipulation of messages in the network between the operator and the control unit
- Manipulation of state reporting in operating systems (on a putative host such as an RTU)
- Concealment of alterations in sensor signals at a control unit

The logic of our choice is that the adversary in addition to manipulating messages to control units must also conceal the results of his actions from operators. There are two places in which this could be done – first, at a point in network communications, second, at the control unit itself. The control unit will also have a host operating system and associated state reporting mechanisms that the adversary must undermine to conceal his presence on the system.

Intuitively, our approach attack detection is based on the fact that integrity attacks alter data and business logic in the system, but that related data, which may not have been subverted (since we assume the adversary may not subvert all processes in a system), exists in other parts of the environment at the same time and, additionally, updates to such data are subject to the vagaries of process delay and message loss, providing a potential gap in time for detection of anomalies to take place. By placing a suitable set of concurrent observer processes in the system, we can observe these data artifacts and determine if the relations which exist between them are consistent or anomalous. Where anomalous, this information can be used in turn to detect the loci of attack.

We refer to our overall approach as *context-based anomaly detection* – though we do not claim to be original in our phrasing [113]. We regard this concept – in modeling terms – to be closely related to semantics-based intrusion detection , specification-based intrusion detection and model-based intrusion detection [101]. Indeed, in many cases, the methods will overlap. The difference lies in the fact that the systems we deal with are distributed and represent a mixture of computational and non-computational data sources.

In algebra, a context is a hole $[\cdot]$ in a system where if we substitute one process, say P, for another, say Q, and the behavior of the system does not change, we say that P is congruent to Q and write $P \cong Q$. In our approach, we expand this concept to include systems which can send messages – W, V – where two processes $P(W)$ and $Q(V)$ may be of different kinds (for example, kinetic versus computational) and the messages W, V they process of different natures, yet nonetheless we can establish a map between the expected behavior of P over W and the expected behavior of Q over V, at least, for the messages they have received to date. We have the choice of comparing $P(W)$ and $Q(V)$, if they are system processes (preferable), or else $P(W)$ may be a model (or anti-model) of $Q(V)$. Normally, we consider such messages in terms of system transitions (which we denote by using δ). Hence we talk of δ-context and δ-congruence. This notion has similarities to Galois connections, since the behavior of $P(W)$ may be an abstraction of the behavior of $Q(V)$ and the behavior is considered to be order-preserving with respect to cause and effect, and so we use similar notation.

Formally, we define the concept of δ-congruence which is the relation which underpins any contextual relationship between disparate parts of a distributed system.

Definition 4.1 (δ-Congruence)
Let P, Q be two distinct sets of processes. Let W and V be messages sent to P and Q respectively where there is a map $\phi : W \mapsto V$ and a map $\varphi : V \mapsto W$. We write $P(W) \rightleftharpoons_\delta Q(V)$ and say P is δ-congruent with Q where there exists a map $\alpha : P(W) \mapsto Q(V)$ and a map $\beta : Q(V) \mapsto P(W)$ which expresses the expected behavior of P and Q over messages. The δ notation transitions in state.

The behavior of P and Q is determined both by the process functionality and by the messages they receive. The reason we refer to this as a *congruence* is the maps are order-preserving with regard to causality, though the messages may not be ordered. The map may be total or partial.

We provide some sample cases.

Case (1) We may consider some restriction of the respective message sets W_r, V_s when considering subsets of processes associated with P, Q where $r \sim s$.

Case (2) If there is a defined delay of j units such that $V_{k+j} = W_k$ we can assert $P_k(W) \leftrightharpoons_\delta Q(V)_{k+j}$.

Case (3) If $V \simeq W$, for example, in a kinetic system, there exist minor differences due to signal disturbances, then we can hold that $P(W) \leftrightharpoons_\delta Q(V)$ for a distance function ϕ such that $\phi(W, V) = \varepsilon \leqslant D$ where D is a threshold value and ε is an n-ary vector returned by ϕ.

Case (4) We can assert that case 3 also exists due to an approximate delay rather than a specified delay as in case 2, thus $P_k(W) \leftrightharpoons_\delta Q(W)_{\Omega(k+j)}$ where Ω represents a probability distribution over the time at which $V \sim W$. This model applies where there are minor differences in message delivery timing in the system.

Case (5) We may also devise a negative case for any of the above instances where $P(W) \leftrightharpoons_\delta \neg(\neg Q(V))$. This negative relation means that given the messages W which P receives, Q should not receive the messages V.

Using δ-congruence for context-based anomaly detection requires us to identify suitable relationships and model these. The model may be trivial or complex to derive. We also need a suitable concurrent observation mechanism to monitor the behavior of $P(W)$ and $Q(V)$ in relation to each other, taking account of potential delays in processing or communication to ensure we preserve the contextual relationship. We have developed three techniques during our research which we outline.

Traceback Protocol We make use of the applied π-calculus to define and prove a traceback protocol to be used during network communication to locate subverted processes. This made use of the trivial fact that a message sent between two known endpoints should retain integrity during transmission. Monitoring for a loss of integrity along the chain of communication allows us to identify and locate potentially malicious processes and, also, to instruct the operator to ignore falsified messages.

Multi-threaded Observation Our second method focuses on the issue of malicious software detection in operating systems in multiprocessing hosts. We show how we can build a model, using a process algebra variant, of cause and effect between disparate data structures and processes in the system. We also illustrate how we can create a suitable observer mechanism with an associated message-passing architecture for detecting state transitions. Inconsistent readings indicate the presence of potentially malicious activity.

Sensor Augmentation Finally, we show how additional sensors placed on a plant could be used to detect inconsistencies in the control relations based on a knowledge of the relationships between sensor readings. We use a simple example of a heat exchanger used for flash pasteurization for these purposes. This use of sensor is not novel, but we extend its use to the simulation of non-linear systems and show how a small number of sensors can achieve the task – a key point where, as in ICS, cost savings are a priority.

Mathematically, the proposed models represent different cases of δ-congruence . In terms of attack resilience, in each case, we can demonstrate that either the protocols are secure – for example, by the deployment of out of band channels or the use of suitable encryption techniques – or else that the technique makes use of both numerical advantage , in terms of the number of observing processes, or stochastic techniques which render the adversary incapable of knowing when they are being observed. In the case of the host observation technique, we additionally can use the mechanism to, quasi-autonomously, observe its own action. These advantages, particularly taken in combination, render our techniques, if not immune, at least, highly resilient to attacker subversion.

4.4 Summary

In this chapter, we have set out the key problems we face and our approach to them. We consider the following research issues: first, the ability of the adversary to conceal his presence and actions in systems, both at host and at network level; second, the arbitrary nature of adversary actions; third, the issues we face in demonstrating the success and survivability of any detective techniques, fourth (throughout), the additional constraints imposed on us by the nature of distributed systems and by the particular examples of distributed systems we have selected for our research.

We create a formal model of adversary action in systems to enable us to prove any detective (and, ultimately, defensive) countermeasures. We identify relationships in our environment between disparate datasets and use these relationships to detect anomalies. Our approach is based on the assumption that the adversary cannot subvert all parts of the system simultaneously and that process and communication delays allow a gap for detecting inconsistent data transitions which break the “context” of the system.

Chapter 5

Attack/Defense Analysis in Distributed Systems

I have seen the enemy and she is my cat.

Humorously corrupted quotation

5.1 Preamble

We set out a conceptual threat model which we instantiate using the π-calculus. This allows us to demonstrate the interaction between multiple adversaries and operators in a distributed system environment, including the impact of various interventions. A primary characteristic of such systems is that they mutate over time. We propose extensions to the π-calculus which enable us to capture this complexity.

5.2 Threat Model

We begin by defining our threat model conceptually and set out the assumptions underlying its construction. A conceptual construction frees other researchers and ourselves to consider other forms of instantiation. Our threat model represents an adversary with the capability to inject malicious software into systems and manipulate process behavior. In fact, we define two models,the *malicious process model* and, its extension, the *malicious agency model* . The fundamental units of the models are respectively *processes* and *goals* – which are very similar in construction, but ultimately differ in capability. The latter allow us to reason over agency in systems.

5.2.1 Malicious Process Model

The malicious process model forms the basis for the malicious agency model. We begin with the malicious process model. We start by defining a process:

Definition 5.1 (Process)
A process P is an entity which can:

1. *Send messages*
2. *Receive messages*

3. *Act conditionally on messages (i.e.,* make decisions*)*

A process in our definition is identical with a process as defined in a distributed system – see appendix B. Clearly, a malicious process is one which does these activities to nefarious ends. Acting conditionally on messages includes any unconditional actions (where the condition to act is set to $\top$). We define the set of malicious processes inductively.

Definition 5.2 (Set of Malicious Processes)
Let Ω be the initial *malicious principal. Let P, Q be any system processes. Let $>$ be the relation* subverts. *Then*

1. *Ω is a malicious process.*
2. *If $\Omega > P$ then P is a malicious process.*
3. *If $P > Q$ then Q is a malicious process.*

We label the set of malicious processes Mal. There is no requirement to assume that processes in Mal originate from the same adversary.

5.2.2 Malicious Agency Model

The *malicious agency model* is an extension of the malicious process model which allows us to capture higher order relations between processes (which we redefine as *goals*), regarding them as acting in a co-ordinated fashion where they belong to the same agent, while goals belonging to different agents may act either cooperatively or competitively. Hence we can reason over alliances of agents. As a result, the relationship "subvert" $>$ is replaced by a relationship "recruit" $\triangleright$ which not only alters the intrinsic nature of a goal (e.g., from friendly to malicious), but also alters its relationship with agents in the system (i.e., transferring it from operator to malicious adversary control). We start by defining the nature of agency in our theory.

We define agency informally as the ability to garner information about environment by receiving messages, to make autonomous decisions based on the information in those messages and to influence the environment through taking action by sending messages. An agent is, therefore, any entity capable of these three feats[129]. More formally, we regard *agents* as subsets of goals in a *goal domain*, where goals are a kind of process.

Definition 5.3 (Goal)
Let G be a goal then G:

1. *Belongs to an agent*
2. *May be active or inactive*
3. *Receives messages from goals*
4. *Send messages to goals*

5. *Acts conditionally on messages (i.e., makes decisions)*

6. *May invoke other goals belonging to the same agent*

else let G be the null goal G_0 which terminates itself.

Sending and receiving messages is defined as for processes. Acting conditionally is similarly defined where the truth or falsehood of a predicate may be used to select subsequent goal action. The relationship "invoke" allows one goal to activate another goal. Goals, unlike processes, may be initially active or inactive. Sending a message to an inactive goal has a null effect in contrast to sending a message to a process which has not previously acted. We include an null goal G_0 for a goal which allows other goals to terminate their action by invoking it. The domain of goals and the domain of agents is related as shown in the following definitions.

Definition 5.4 (Goal Domain)
The goal domain is the set of all goals which form a distributed system S the universe of discourse.

Definition 5.5 (Agent)
Let A be an agent then A is a subset of goals in a goal domain which may invoke each other. Goal invocation is a partial ordering over the set of goals belonging to an agent.

It should be clear that a single goal may be an agent in its own right. We also define the concepts of *agent domain* and *agent subdomain*.

Definition 5.6 (Agent Domain)
An agent domain is the set of all agents. Hence the agent domain is a mapping ϕ of the universe of discourse from goals to agents.

Definition 5.7 (Agent Subdomain)
An agent subdomain is a subset of agents, selected depending on application.

For example, a malicious subdomain could consist of a set of agents which are co-ordinating an attack on a system.

A primary application of using agent subdomains is the distinction of malicious and non-malicious agents.

Definition 5.8 (Malicious Agent)
A malicious agent is an agent to which malicious goals are assigned and which must belong to a malicious agent subdomain.

We sharply distinguish malicious and non-malicious agents. If a goal is recruited by a malicious agent, it is also a malicious goal or agent (depending on the application) belonging to the same agent subdomain as its recruiting agent. No non-malicious agent may contain a malicious goal or belong to a malicious subdomain. It may be helpful to regard a particular agent as being the primary agent in a malicious subdomain, responsible for co-ordinating the actions of other agents belonging to that subdomain.

Malicious goals are capable of sending malicious messages to other goals (belonging to agents in other agent subdomains) and recruiting them to work on behalf

ACC	Reachability or access
COM	Overt Communication
SUB,REP,MOB	Create and send malicious messages
SP-PAR	Spawn Additional Processes
MN	Message Read
MV	Message Diversion
MD	Message Delay
MP	Message Drop
MR	Message Replay
MM	Message Manipulation
MI	Message Injection
CC	Covert Communication
D-MAT	Decision Making
LB	Learning Behavior
COOP	Cooperate
P	Indistinguishability Assumption

Table 5.1: Adversary Capabilities

of the malicious agent. Recruiting an agent also acts to separate the goal from its original agent subdomain, create it as a distinct agent, and transfer it to the agent subdomain of the agent whose malicious goal recruited it. We note that there may be more than one malicious agent in the system and there is no requirement for malicious agents to belong to the same agent subdomain. In other words, it is possible for more than one malicious agent to exist in the agent domain and be responsible for launching distinctive attacks. It is equally possible for malicious agents to belong to the same subdomain and coordinate resources and actions in an attack. Moreover agents belonging to distinct malicious agent subdomains may cooperate. Finally, because a single goal may be an agent and by recruitment replicate itself and subsequently terminate its previous instantiation, we consider it is possible for agents to be mobile in the agent domain. The malicious agent model can map the set of malicious processes Mal to the set of malicious agent subdomains Mal_S, converting processes to goals.

5.2.3 Towards A Formal Adversary Capability Model

Having defined malicious processes, goals and agents along with their subdomains and their universes of discourse, we proceed to examine the capabilities of a single malicious process or goal in more detail. These capabilities arise naturally from the ability of the process (or goal) to participate in the action of a system by sending and receiving messages and by acting (conditionally) on messages. We use a set of labels – table 5.1 – to describe the capabilities which may be attributed to malicious processes or goals. We show that these capabilities are derived by considering a variant of the Dolev-Yao model for open distributed systems, but making the assumption that the distributed systems we deal with are *closed* networks – a concept which we define. We justify our assumptions in terms of our target environments.

We provide an informal description of these capabilities. These capabilities should be understood to be semantically distinct and we seek to make this distinction clear in our description of them. By considering memory which is accessed, authorized or unauthorized, on a shared basis to be a communication channel (or buffer, in the case of storage) the model can also be applied to host-based systems. In the latter case, it is usually easier to refer to "data" rather than "messages". So, for example, a process may inject false data rather than false messages.

ACC represents the bounds set on the communication with a subset of processes (or goals) in the network, directly or indirectly. These bounds are set by the knowledge of process addresses available to a malicious process.

COM is the capability of sending and receiving messages with a subset of processes in the network. COM should not be mistaken for ACC. It represents the channels directly available to a process for sending messages, not the set of channels known to a process.

SUB, also REP or MOB is the capability of a malicious process to subvert a legitimate system process. SUB is exercised by sending a malicious message m from a malicious process to another process P which causes P to be overwritten "$>$" or "$\triangleright$" as a malicious process P' (goal). SUB is dependent on ACC and COM. There is nothing to prevent one malicious process being overwritten by another malicious process. REP refers to the ability of agents to replicate themselves by subversion and MOB indicates that replication makes agents mobile in distributed systems by changing ACC and COM relations. These additional labels are occasionally useful for distinguishing some types of malicious action.

SP-PAR means that an overwritten process may spawn other processes which also act, possibly cooperatively (COOP) as malicious agents. The original process may, in fact, be left untouched, but malicious processes are spawned in parallel with it.

MN refers to the capability of malicious processes to intercept messages not intended for them. This capability is limited by COM. The malicious process may only receive messages directed to or via the channels controlled by it. This contrasts with other formal adversary capability models where the malicious principal is assumed to receive all messages in the system and send all messages in the system.

MV is the capability to divert messages to unintended destinations.

MD is the capability to delay messages from reaching their destination.

MP is the capability to prevent messages (by dropping them) from reaching their destination.

MR is the capability to replay messages which have been previously received or intercepted.

MM is the capability to manipulate the content or format of messages.

MI is the capability to inject false messages.

CC is the capability for malicious processes to communicate covertly amongst themselves.

D-MAT is the capability of malicious processes to make decisions regarding messages. This is simply a repetition of the definition of a process or a goal to act conditionally on messages.

LB is the capability of malicious processes to alter their behavior in response to their environment. It does not necessarily imply a learning system in the common sense, nor is it necessarily an example of decision making. For example, malicious processes which have their capabilities upgraded by another malicious process or the adversary may be regarded as exhibiting learning behavior.

COOP is the capability for malicious to cooperate with other malicious processes, not necessarily associated with the same adversary process. [1]

P is the assumption that a malicious process may appear to be a legitimate process. More precisely P states that an agent will not make any egregious process errors.

Our threat model is recognizably a variant of Dolev-Yao [38], but we make some distinct assumptions which lead to the conclusion that an adversary must subvert processes to act on its behalf.

The assumptions underpinning the Dolev-Yao model are [74, 38]:

- The environment consists of a large network of computers, devices and resources (which we treat abstractly as a set of processes)
- The network is open, so that principals can join and starting send and receiving messages without authorization
- Malicious principals can:
  - Eavesdrop on messages
  - Alter messages
  - Forge messages
  - Duplicate messages
  - Re-route messages
  - Delete messages
  - Inject messages

As a result, a malicious principal can:

- Obtain any message passing through the network

[1]This label has not appeared previously in published work, but it is a natural implication of that work to include it here.

- Act as a legitimate user
- Can initiate a conversation with any other user
- Can receive messages from any principal
- Can send messages to any principal by impersonating any other principal

This implies that in the Dolev-Yao model any message sent to the network can be considered to have been sent to the adversary and any message sent by the network can be considered to have been sent by the adversary. In other words, the network can be considered to be the adversary. In addition, other restrictions on computing power in the Dolev-Yao model are elided as they pertain to the security of key exchange protocols which are outside the scope of this research. They may be re-introduced if necessary.

These capabilities are a recognizable subset of those which we attribute to a malicious process. Following the reasoning in [74], we have also included decision-making capabilities and, based on [98], learning behaviors as well. The *indistinguishability assumption* P follows from our remarks on the evolution of stealth and obfuscation techniques in malicious software development. Finally, we have added the ability of processes (goals) to subvert (recruit) other processes (goals) to complete our model.

But it does not follow that our model is a superset of the Dolev-Yao model. The restrictions we introduce on sending and receiving messages differentiates our model from the common Dolev-Yao model and, indeed, other monolithic threat models [10]. We do not assume the adversary is the network. Furthermore, we do not assume the adversary may subvert any process. Subversion attempts are regarded as non-deterministic in outcome and, in particular, we assume that the adversary may only act on a strict subset of processes, usually much smaller than the number of processes in the system.

We formalize these notions by defining the distributed systems we deal with as *closed*. In the following discussion, the word "process" may be substituted by "goal".

Definition 5.9 (Closed Network)
Let S be a distributed environment as defined in appendix section B.3.1. If

- *Each process $P_i \in S$ communicates with a strict subset $Q \subset P$ of other processes*
- *Let P, Q be any processes in S, P and Q may only communicate if they share a channel*
- *Every process must be authorized to interact with other processes in S.*

We describe S as a closed network.

Let Ω be the adversary and external to S. It is easy to see that even if Ω can persuade processes in S to interact with him that interaction is limited by COM, that is the number of processes in S with which he shares channels, and by ACC, the number of processes he knows which he can address directly or transitively by COM. We consider a strict subset of processes $W \subset P$ which may communicate with systems which are outside of S, directly or indirectly.

Definition 5.10 (Initial Attack Surface)
Let S be a closed network consisting of a set of processes P as before. Let $W \subset P$ be the set of processes which may communicate with other systems than S, either directly or indirectly (as defined by ACC). We call W the initial attack surface *of S.*

Hence, if the adversary is located in a process which external to S, its access to S is limited to the processes in W. Two possibilities exist:

- The adversary may engage in an apparently "legitimate" conversation with a process in W and attempt to persuade this process to provide further information about other processes in the network, to pass messages or obey instructions
- The adversary may attempt to subvert a process in W and use it to launch further attacks against the other processes in S

The first option is limited. We assume any process $W_i \in W$ will only be able to carry out legitimate operations. For example, it will not manipulate messages directly, though it may send messages to the adversary and, possibly, accept falsified messages in return. However, it can only be in receipt of messages which can be legitimately sent to a process W_i or processes which it has informed the adversary about. To increase control of the network, it is useful to subvert additional processes to increase the range of messages which may be actively manipulated. So it becomes more fruitful for the adversary to subvert a process $\Omega > W_i$ where W_i is a process in W, the initial attack surface, creating a secondary attack surface, exposing further processes for subversion until an attack goal may be achieved.

Definition 5.11 (Secondary Attack Surface)
Let S be a system consisting of a set of P processes as before. Let $>$ be "subvert" as before. Let $W_i \in W$ be a process in the initial attack surface and let $V_i \in V$ and be any processes where $V \not\subseteq W$. Then we can have $W_i > V_i$. We say that V forms a secondary attack surface.

Any process once subverted may have the capability to subvert other processes, so the attack surface increases as mediated by ACC and COM.

To maintain the constraint on the adversary that only a strict subset of processes may be subverted, we treat subversion as non-deterministic with regards to outcome (for any single attack). Hence, let P, Q be any processes then if $P \not> Q$, no further subversion attempt can be made on Q without justification, such as a change in subversion method, or the appearance of a fresh attacker.

It should be clear that there are limitations on the action of malicious processes in our model, including the adversary (or initial) processes, which are considered to be neither omnipotent, nor omnipresent in the system, nor omniscient with regard to messages. The presence and impact of malicious processes on a network is constricted by ACC and COM – that is, the standard assumption, using for examining secure key exchange protocols, that a malicious process receives all messages regardless of the intended recipient is held to be false in our model. A malicious process may only receive messages intended for it. Likewise it may only send processes by channels it owns and to processes it knows. This limits the size of the attack surfaces, both initial and secondary. The capability to subvert another process by SUB is also limited non-deterministically. The capabilities for message manipulation, including decisions over what to do with messages, are exactly those set out in the Dolev-Yao model[38, 74]. Other aspects of Dolev-Yao such as the inability to guess nonces in a sufficiently large random space are not generally required for our analysis, but can be introduced if desired.

Our assumptions can be justified in terms of our target environments. A multiprocessor host may, of course, be considered as a single (heavyweight) process which may be subverted and which communicates with a limited set of other such hosts or network devices in a network. However, internally, it may also be modeled as set of processes which communicate via memory. However, memory access must be authorized and memory is segmented accordingly, closing the internal "network". An ICS system, on the other hand, is clearly a large network, consisting of multiple processes. But access to this network is intended to be constrained to legitimate users and there are a limited number of processes (i.e. hosts or network devices) which can interact with other systems external to the network. Hence it should be clear that the malicious process model and, by extension, the malicious agency model, may be applied to both environments.

Finally, an obvious question is how do malicious processes inherit capabilities from the malicious process which subverted them. This is not straightforward. It is certainly not the case that if P and Q are processes and P is a malicious process, then if $P > Q$ that Q can only inherit the capabilities of P. For example, Q may have greater access ACC to other processes than P. Q may be able to intercept messages by MN while P may only communicate COM with Q. We also have to consider that if malicious processes can communicate among themselves, then such communication can be transitive. So if P, Q, R are malicious processes such that $P > Q > R$ then it is possible for P to send a message by Q which increases the capabilities of R above those of Q. Hence we should consider that capabilities are derived both from the process of creating a malicious process, upgrading a malicious process and the degree of access and communication which a malicious process enjoys in a network of such processes in relation to the overall system.

5.2.4 Operator Capabilities

As a final part of our conceptual model, we also need to understand operator capabilities in a distributed system and what constraints exist on operator knowledge of the system. In fact, we can turn our adversary threat model on its head and consider that adversaries are simply malicious operators and that the capabilities which they possess are identical to the capabilities which operators have. For example, operators and operator processes are limited by ACC and COM. Operators can "subvert" processes – though we normally call this upgrading or installation. Operators and non-malicious processes can likewise manipulate messages, again for non-nefarious ends, make decisions and exhibit learning behavior and, in a network of networks, co-operate with other operators and processes outside of their subdomain. Hence we can also define a non-malicious set of principals Sys which corresponds to Mal and, in treating operators as agents with goals, we can replace Sys by Sys_G which the agent subdomain for non-malicious principals.

The operator may also have some advantages over malicious agents in terms of their knowledge of global time and state. Intrinsically distributed systems deny an operator a knowledge of global time and state and limit knowledge of local states, e.g., with regard to deadlock or other conditions. However, [46] also reveals that there exist a number of possible approaches to resolving these issues using various classes of distributed algorithm. For example, there exist algorithms which enable us to solve questions with regard to the timing and ordering, even the causality, of events.

We provide two examples of classes of algorithms for establishing time and state (which may be skipped by the knowledgeable reader) in appendix B which we take advantage of in our research. These show that operators may make special efforts to gain knowledge of global state and time to provide themselves with the ability to control and intervene in computations as required. However, these algorithms are not suitable by themselves for solving the problems we present as they assume that, while processes may be subject to error, they are not subject to deliberate tampering or attempts to conceal state. Nonetheless, where appropriate to solving problems to do with detecting adversary presence in distributed systems, we assume or explicitly make use of the existence of such algorithms.

5.3 Applied Process Calculi

We introduce an applied π-calculus and a higher order Goal Transform $G\{\pi\}$-calculus to allow us to model the adversary (and other principals) in distributed environments. The basic π-calculus and associated techniques is described in appendix A. In this section, we set out the capabilities of the applied π-calculus and the goal transform calculus, and discuss the other extensions which appear during the thesis. This calculus is used to define formally the adversary capabilities we have already discussed. It is also used to define the normal behavior of systems and operator actions. In particular, we extend the π-calculus using *silent* functions which give us the capability to express concretely concepts such as data loss or security impacts which otherwise appear as inactions in the system which would normally be shown as a skip or null value and to allow us to calculate the impact of adversary actions on systems which would otherwise be difficult to track.

5.3.1 Capabilities, Summations and Processes

The prefixes which are its capabilities (for action) are:

$$\pi ::= \bar{x}\langle y\rangle_c | x(z)_c | f(z)_c \supset u_d | \tau | [\mathcal{L}]\pi$$

The meaning of the capabilities is given in table 5.2. The set of names is expanded to include the names of data variables and constants as well as channels. We also make a point of including a subscript, which represents the set of *characteristics* associated with a name. In effect, we treat names as data packets and characteristics describe attributes of those data packets such as data values, message priority, probability that a message will be sent without loss and so forth. Characteristics may only be *potentially* rather than directly observable - that is, they may represent behavioral artifacts that cannot – without special efforts – be known by an operator, such as message corruption. This is a key feature of our approach where we seek to demonstrate the limits of what can be observed in the face of adversary

$\bar{x}\langle y\rangle_c$	Send a name, with its characteristics
$x(z)_c$	Receive a name, with its characteristics
$f(z)_c \supset u_d$	A function over a name and its characteristics
τ	Silent functions
$[\mathcal{L}]\pi$	Conditional exercise of any capability, except mismatch over channel names

Table 5.2: π-Calculus Summands and Processes

action to prevent such observation and how we may overcome these limitations, at least partially.

We introduce an n-ary function over names which results in a new name being created, or a current name (or set of names) being updated. Functions may be defined by some suitable encoding. The addition of such functions enables us to express adversary capabilities and to construct security protocols.

The use of *silent* actions or functions requires some comment. This takes on a special meaning in our calculus. It is not an abstraction from the action of a process as is conventional in process algebraic systems, rather it represents an action which a process may take which may not be observable by the operator unless special efforts are made. In some cases, they may not be observable at all. For example, message loss λ may only be observed by the operator, or by other processes, if there is a mechanism (assumed, or explicit) for detecting this condition. However, decision-making τ by the adversary can never be observed by other processes. Other examples of silent actions are ω for "subversion option", ϕ for "process failure" and so forth. Such silent actions can also be treated as a class of n-ary functions for certain applications.

$\mathcal{L}$ is defined to be any first-order logic with ordering and equivalence. There is a restriction over the conditional capability. As with the π-calculus, where names represent channels rather than messages, the mismatch prefix cannot include a $\neq$ sign to prevent processes losing capability for action. The formulation $y \neq y$ where y is the name of a channel is an example of this violation. The summations and processes of the applied π-calculus are:

$$\begin{aligned} P &::= M|(P|P')|\nu z \quad P|!P \\ M &::= \mathbf{0}|\pi.P|M+M'|M \oplus M \end{aligned}$$

The meaning of the processes and summations of the applied π-calculus are given in table 5.3.

The primary difference in the processes and summations is the introduction of a *contingent choice* operator, in which both sides of the operator may be selected (normally, in sequence of appearance) and acted on. This adds the following action relations:

$\mathbf{0}$	Null action: a process which does nothing
M	A sum over capabilities
$M \oplus M'$	Alternate composition, exclusive - or
$M + M'$	Contingent choice; a sequence of potential actions
$\pi.P$	A process with a single capability; the process cannot proceed until it executes
$P + P'$	An exclusive choice between P and P'
$P\|P'$	P and P' proceed independently; but may interact by shared names
$\nu z\ P$	The scope of the name z is restricted to P
$!P$	Infinite composition of $P\|P\|P$, allowing process replication

Table 5.3: π-Calculus Summands and Processes

ASUM-L	$\dfrac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'+Q}$	$[1]P$
XSUM-L	$\dfrac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$	$[0]P$

This contingent choice operator may, therefore, be regarded as a way of encoding a sequence of "IF . . . THEN . . . " statements. However, it is primarily introduced to make it easier to deal with message prioritization on send or receive and to deal with invalid code snippets created by re-writing processes or the nullification of goals (see section 5.3.4). The $\oplus$ operator binds more tightly than the $+$ operator.

Where message priorities are used, sequences formed using this operator may be re-ordered in accordance with priority, which partly explains its inclusion – see section 5.3.2. It also becomes useful where processes are modified on the fly or goals invoked or nullified, so a feasible action in one version of a process may become unfeasible when it is re-written, for example, to exclude the channel name by which the action is carried out. Hence, the condition $[1]$ placed on the process P states that P may take its next action if and only if the conditions for P to act exist. These conditions may be explicit (such as a boolean expression in front of the sum P) or implicit, such as P containing valid actions like sending a name along an existing channel. As soon as P ceases to be valid, for example, by terminating, Q may act, provided it also is a valid sum.

We note that $\tilde{x}$ may be used to indicate an n-tuple of names $x_1, x_2, \ldots, x_n$ and we use $:=$ as our defining operator for processes, e.g, $P := (\nu\tilde{z}) \quad Q$. Conventionally, a sum M can be represented using $\sum$. We sometimes use the symbol $\coprod$ to index an array of concurrent processes.

5.3.2 Applied π-calculus with Priorities

A variant of the applied π-calculus, used in [84], characterizes messages with priorities. This was used to discuss the effect of different communication strategies between messages in real-time and best effort systems on external hacking attempts.

Message priority is easiest explained by some examples of its use. For example, equation 5.1 states that z should be sent along channel x after u is sent along channel y.

$$P := \bar{x}\langle z\rangle_2.\mathbf{0} + \bar{y}\langle u\rangle_1.\mathbf{0} \tag{5.1}$$

This is regarded as being different from equation 5.2:

$$P' := \bar{x}\langle z\rangle_2.\bar{y}\langle u\rangle_1.\mathbf{0} \tag{5.2}$$

which states that z should be sent with priority 2 and u with priority 1, but maintains a strict order of communication by ".".

To make this clear, we re-order messages over $+$, e.g., equation 5.3 is structurally congruent with equation 5.1.

$$P := \bar{y}\langle u\rangle_1.\mathbf{0} + \bar{x}\langle z\rangle_2.\mathbf{0} \tag{5.3}$$

We also give $|$ a special meaning in this variant so that messages sent concurrently between any two processes are sent in priority order. In equation 5.4, the process with the highest receive priority will obtain the message.

$$Q := \bar{y}\langle u\rangle_1.\mathbf{0}|y(z)_2.\mathbf{0}|y(z)_1.\mathbf{0} \tag{5.4}$$

It should be noted that where the two processes are not in communication with each other, the messages may be sent in arbitrary orders – see equation 5.5 where the transmission of messages will be arbitrarily ordered.

$$R := \bar{x}\langle u\rangle_1.\mathbf{0}|\bar{y}(z)_2.\mathbf{0}|y(z).\mathbf{0}|x(s).\mathbf{0} \tag{5.5}$$

Priorities may be attached to both *send* and *receive* actions, although we normally select only one of these options. Messages of equal priority are sent in non-deterministic order.

5.3.3 Applied π-calculus with Routing

Attaching a routing address to a name is another feature which was introduced in [84]. In essence, every channel in a system is assigned a unique name. Uniqueness (of channel names) must be retained under substitution, but substitution does not require that identity be preserved – that is, if the name of a channel is changed, any subsequent message routed to that name will either arrive with a new channel bearing that name, or fail (if the name no longer exists). Unique channel names can be assigned using numbered subscripts, or by labeling channels with process names. Routing is performed by checking the routing address associated with a name and conditionally forwarding it[2].

[2]This represents an alteration from the original proposed semantics for routing used in [84] , but is more convenient to use.

For example, equation 5.6

$$P := \bar{x}_Q\langle a\rangle_{x_R}.\mathbf{0}|x(z)_Q.\mathbf{0} + (\bar{x}_S\langle z\rangle_{[S]}.\mathbf{0} \oplus \bar{x}_R\langle z\rangle_{[R]}.\mathbf{0}) \tag{5.6}$$

routes a to channel x_R. The routing is conditional and this is shown by using the square brackets "[ ]" in the subscript, but could also be expressed using dotted notation, e.g., $[z.r = R]\bar{x}_R\langle z\rangle_R.\mathbf{0}$ where r is the channel address. The latter syntax would be useful in the context of creating a scripting language equivalent to the calculus.

5.3.4 Goal Transform Calculus

The goal transform or $G\{\pi\}$ Calculus is a variant of the applied π-calculus which makes the calculation of the interaction of agents and goals rather than processes central to the abstraction. An *agent* is defined as a set of *goals*. The capabilities and action relations of the calculus subsequently allow goals to act sequentially or concurrently, goals to invoke other goals conditional on success or failure, goals to send messages to goals by channels and choices to be made between goals or whether goals may be validly exercised. This calculus was introduced in the context of analyzing co-ordinated attacks and multi-agent defenses and is intended to be applied to variants of multi-agent attack/defense scenarios in distributed environments.

We describe the $G\{\pi\}$-calculus.If G is a goal then

$$G ::= \mathbf{0}|\pi.G|\nu z\, G|G.G|G + G|G \oplus G|(G|G')|!G|[\mathcal{L}]G \tag{5.7}$$

where the possible actions α of G are defined in Table 5.4.

Term	Semantic
$\mathbf{0}$	Null action
$\pi.G$	Exercise a π-calculus capability
$\nu z\, G$	Declare a new goal and its (restricted) names
$G.G$	Execute goals sequentially
$G + G'$	Execute feasible goals in order
$G \oplus G'$	Execute exclusive goals
$G\|G'$	Execute two goals concurrently
$!G$	Replicate goal action
$[\mathcal{L}]G$	Execute a goal, based on a first order logic condition

Table 5.4: $G\{\pi\}$-Calculus Syntax

where $\mathcal{L}$ is a first order logic with equivalence and ordered relations and π is a capability of the π-calculus which we have defined in 5.3. Goal actions are, therefore, the capabilities of the applied π- calculus – see section 5.3.1. The action relations of sums and goals remain the same as for sums and processes with the exception of goal invocation.

By convention, $G_\mathbf{0}$ is the *null* goal which terminates itself without invoking other goals. Hence

$$\begin{aligned} ||G.G_{\mathbf{0}}||_A &\rightarrow ||\mathbf{0}||_A \qquad (5.8) \\ ||G_{\mathbf{0}}.G||_A &\rightarrow ||\mathbf{0}||_A \qquad (5.9) \end{aligned}$$

With respect to action relations, it should be noted that a goal does not terminate unless it invokes the null goal G_0, but may invoke a countably infinite number of other goals.

An agent is defined as a set H of goals $\|H\|_{Agentname}$. As well as names which are restricted to the scope of goals, names may be declared within the scope of an agent using the conventional syntax $(\nu z)\,\|H\|_{Agentname}$. Since not all goals will be activated when an agent initially activates, we may use the syntax $\bullet G$ to indicate which goals are initially, or currently, active. The goals for each agent are defined using applied π-calculus statements and any associated constants, variables and functions are defined (formally or informally).

As with the π-calculus, proof is by reduction or the use of action relations. A proof reduction is indicated using the notation in equation 5.10.

$$||Goal.Subgoal.Action||_{Agent} \rightarrow ||Goal.Subgoal.NextAction||_{Agent} \qquad (5.10)$$

where $NextAction$ is any capability or a goal invocation. This reduction technique is identical to that of the π-calculus [34], except that goal labels are used to limit the consideration of the reduction to the active ($\bullet$) goals of each agent. It should be noted that where a goal is not active, it may for convenience be elided from the listing of any proof. It is re-inserted when, or rather if, it is invoked.

5.4 Instantiating a System Model

We illustrate how a simple SCADA system can be specified using the π-calculus. We assume that a SCADA environment consists of a set of processes P – partitioned hierarchically into supervisor processes S, one or more generations of communication processes M and control loops consisting of controllers C, sensors R and actuators A. The set of names $N = \{W, X, Y, Z\}$ are names of channels which are indexed to show assignments. We use the notation $\nu\ X$ to indicate $\nu\ x_1, x_2, \ldots, x_n$. U is the set of names for commands and data values.

$$\begin{aligned} P \;:=\; \nu \quad & WXYZU(!S_1|\ldots|!S_i|!M_{1,1}|\ldots|!M_{i,j}|!C_{1,1,1}|\ldots| \\ & !C_{i,j,k}|!A_{1,1,1}|\ldots|!A_{i,j,k}|!R_{1,1,1}|\ldots|!R_{i,j,k}) \qquad (5.11) \end{aligned}$$

We map channel names to processes as follows:

$$S \mapsto W, X \qquad M \mapsto W, X, Y \qquad C \mapsto W, Y, Z \qquad A, R \mapsto Z$$

In general, channels X, Y, Z process SCADA data and commands in U and W is used for network supervision and interprocess requests such as routing. All names are free (i.e. unique) in the scope of the system, but we assume that each process only has access to a limited set of named channels and that name assignments are stable. We assume some redundancy in processes and channels.

Message addressing may be achieved by routing. In addition, where a system requires processes to meet both hard and soft real time process targets we can represent this, we can use message priorities. Lost or delayed messages can be represented using silent functions. The internal function of individual processes can be represented by the communication and other functions which we specify in our calculus. We provide a couple of simple examples.

For example, an operator which is an entity is defined by:

$$Operator := \bar{x}u.\mathbf{0} \oplus x(u).\mathbf{0} \oplus \tau.\mathbf{0} | !Operator \tag{5.12}$$

An operator can send messages, receive messages and make decisions by τ about messages. In line with sections 5.2.4 and 5.5.1, an operator is regarded as having equivalent, if non-malicious, capabilities to a malicious principal and is defined identically.

A controller may be specified likewise:

$$\begin{aligned} Controller \quad := \quad \nu \quad u, p, k, e \quad & !((z(e)_1.\bar{z}\langle e\rangle.Control(p,k,e,i) \supset \bar{z}\langle u\rangle_1.\mathbf{0} \\ & +y(\tilde{s})_2.UpdateControl(p,k,\tilde{s}) \supset (p', k').\mathbf{0}) \oplus \omega) \end{aligned} \tag{5.13}$$

This simple example shows a control function for a plant which responds to an error signal e and produces a control signal u. In addition, the control parameters can be updated by a message $\tilde{s}$ from the operator. Note, the use of priorities to indicate real and best effort traffic communication:

Functions may be specified informally as here or may be more formally defined using pseudo-code. Obviously, it is possible to build more complex examples of ICS components using this approach. We can also instantiate other kinds of standalone, distributed and mobile systems using the same approach.

5.5 Instantiating the Adversary Capability Model

In this section, we provide a definition of a malicious agent and provide examples of how we can instantiate individual capabilities in the model for subverted processes, using the applied π-calculus defined in section 5.3. In section 5.6, we show how these capabilities may be used to build an attack. Our examples are based on the malicious process model, but equally apply to the malicious agency model.

5.5.1 Defining the Adversary

An adversary is defined in the same way as an operator – see section 5.4. Indeed, one view of an adversary is that they are simply a malicious operator. Hence, for the channels available by ACC and COM (section 5.5.2.1), an adversary may send messages to the system, receive messages from the system and by τ make decisions about messages. This definition matches that used in [74], but for the π-calculus rather than for CCS. Subsequently, we show how our approach departs from that proposed in [74].

Let S be a distributed system. Let Ω be an initial malicious process (the adversary process). Using the π-calculus notation, the adversary is defined by equation 5.14

$$\Omega := \nu(\tilde{m}) \quad \bar{x}\langle m\rangle_r \oplus x(z) \oplus \tau|!\Omega \tag{5.14}$$

where m is a malicious message, x is a channel, r is a routing address and z is a placeholder for any messages received by Ω. τ is a silent function representing the decision-making capability of Ω. By τ, the adversary Ω may exercise any other capability – manipulating messages, replaying message and so forth – in the model. In fact, we assume that Ω can have any number of channels $\sum_i x_i$ which it shares in common with processes $P_i \in S$ forming the initial attack surface.

5.5.2 Instantiating Individual Adversary Capabilities

The adversary definition acts as a starting place. However, it is clearly not useful for either a source adversary process nor any subverted process to have its malicious actions defined by τ as by τ it can exercise any capability over messages. Instead, we want to instantiate attack scenarios by attributing various individual capabilities, which we encode as interacting functions,to such processes. We show how this may be done by feasible encodings of such capabilities using the π-calculus.

5.5.2.1 Access and Communication – ACC and COM

Communication, COM, follows straightforwardly from the functions *send* $\bar{x}\langle a\rangle$ and *receive* $x(z)$ of the π-calculus. For any process, these define the channels by which a process may communicate. Access, ACC, is the ability of a malicious agent to send any message to any other process, not just those defined by the channels available. It depends on the messages a process has received or what a process already knows about the existence of other processes in a system. In other words, ACC defines the attack surface available to a malicious process. Let P, Q, R be processes such that

$$\begin{aligned} P &:= \bar{x}\langle a\rangle.\mathbf{0}|!P & (5.15)\\ Q &:= x(z).\mathbf{0} + \bar{y}\langle z\rangle.\mathbf{0}|!Q & (5.16)\\ R &:= y(z).\mathbf{0}|!R & (5.17)\end{aligned}$$

then $COM(P) = \{x_Q\}$ while $ACC(P) = \{x_Q, y_R\}$. So while P may only communicate with Q directly, it may send any message to R as this is implicit in Q. Routing may also be used to code such knowledge explicitly.

5.5.2.2 Subversion – SUB

Defining an adversary as before – equation 5.14 – Let us also assume that P shares a channel x with Ω and hence can send or receive messages with respect to Ω. We define P in equation 5.18.

$$P := \nu(\tilde{a})!(\quad x(z).P1 + \omega) \tag{5.18}$$

where $\tilde{a}$ is the set of names, possibly empty, in the scope of P, $x(z)$ is the action of receiving a message, possibly malicious, and $P1$ is the continuation of the process. ω is a silent function which indicates that P is non-deterministically vulnerable to subversion by a malicious process.

Equations 5.14 and 5.18 state that a malicious process can send a message to any process $P \in S$, possibly leading to the subversion of that process. We define the effect of m on P in equation 5.19.

$$\omega(m, P) \supset P' \tag{5.19}$$

where P' is a malicious process $P' \in Mal$.

The key difference between subversion under the malicious process model and subversion in the malicious agency model is that subversion is equivalent to recruiting processes from one agency subdomain to another. For example, an agent with two goals X, Y such that $||X|Y||_{good}$ is separated into two agents $||X||_{good}, ||Y'||_{bad}$. The "bad" agent is recruited to a malicious subdomain.

5.5.2.3 Spawning Concurrent Malicious Process – SP_PAR

By τ, a malicious agent may consist of additional concurrent processes to the original process P which is subverted, i.e., $\omega(m, P) \supset \coprod_i P_i^{'}$.

5.5.2.4 Message Interception(MN) and Message Diversion(MV)

By definition of COM – see section 5.5.2.1 – any malicious process P will intercept messages sent to it. Message diversion, MV, is based on message interception, MN. Any message received by a malicious process can be diverted to another process. For example, in equations $P|Q|R$ be processes.

$$P := \bar{x}\langle a\rangle.\mathbf{0} + \omega|!P \tag{5.20}$$
$$Q := x(z).\mathbf{0} + \omega|!Q \tag{5.21}$$
$$R := y(z).\mathbf{0}|!R \tag{5.22}$$

P is a process which normally sends messages a to Q and is re-written $P \overset{\omega}{\rightarrow} P'$ where P' is defined in equation 5.23 to send messages to R.

$$P' := \bar{y}\langle a\rangle.\mathbf{0}|!P' \tag{5.23}$$

5.5.2.5 Message Delay and Message Replay – MD, MR

Using appropriate variables in the scope of the process,*store* and *receive* functions, it is possible to delay and replay messages. We include these functions, informally defined, in P' in equation 5.24.

$$P' := \nu(\tilde{v}^0, t) \quad !(x(s).\mathbf{0}+store(\tilde{v}, s) \supset \tilde{v}^{n+1}.\mathbf{0}+\ldots\oplus retrieve(\tilde{v}) \supset \tilde{v}^{n-1}, s'.\mathbf{0}+\bar{y}\langle s'\rangle.\mathbf{0}) \tag{5.24}$$

We can also set conditions on the release of messages - see section 5.5.2.10.

5.5.2.6 Message Drop – MP

Let P be a process which we define in equation 5.25

$$P := x(z).\mathbf{0} + \ldots + \bar{y}\langle z\rangle.\mathbf{0} + \omega|!P \tag{5.25}$$

we can reduce P by ω to P' as shown in 5.26. P' will selectively filter messages received by x. We could equally redefine P' to filter messages sent by y. Let ϕ be the filter condition.

$$P' := x(z).\mathbf{0} + [\phi]drop(z) \supset \emptyset.\mathbf{0} + \ldots + \bar{y}\langle z\rangle.\mathbf{0}|!P' \tag{5.26}$$

5.5.2.7 Message Manipulation – MM

Let P be a process which we partially define in equation 5.27.

$$P := x(z).\mathbf{0} + \ldots + \bar{y}\langle z\rangle.\mathbf{0} + \omega|!P \tag{5.27}$$

We can reduce $P \overset{\omega}{\to} P'$ so that P' manipulates messages, possibly selectively. ϕ is the, possibly empty, filter condition.

$$P' := x(z).\mathbf{0} + [\phi]alter(z) \supset z'.\mathbf{0} + \ldots + \bar{y}\langle z'\rangle.\mathbf{0}|!P' \tag{5.28}$$

5.5.2.8 Message Injection – MI

Let P be a process, partially defined in equation 5.29, as before. P may be re-written to inject messages. shown in equation 5.30.

$$P := x(z).\mathbf{0} + \ldots + \bar{y}\langle z\rangle.\mathbf{0} + \omega|!P \tag{5.29}$$

$$P :=!\nu(a) \quad x(z).\mathbf{0} \oplus [\phi]inject(a) \supset a.\mathbf{0} + \ldots + [\phi]\bar{y}\langle a\rangle.\mathbf{0} + [\not\phi]\bar{y}\langle z\rangle.\mathbf{0} \tag{5.30}$$

5.5.2.9 Covert Communication – CC

There are various means of covert communication such as encrypting messages, using dead space in protocols, timing information and so forth. Hence, in the model, covert communication means that two malicious processes can communicate using a set of channels which are not available to the operators or legitimate processes in the system. Since a malicious process represents an arbitrarily re-written process, we could do this directly by simply sharing channel names outside the scope of the set of channels known to other system principals. This would, however, assume all channels were pre-assigned. A more interesting approach is to use the π-calculus technique for *extrusion of scope*. Extrusion of scope allows one process to share a name, previously only known to it, with another process and use that name to communicate.

Let $S := \nu(\tilde{c}) \quad P|Q$ be processes where P has a set of channel names unknown to Q and a channel x which is shared with Q. Let Q be subverted so that it contains the capability to receive and communicate by channel names sent from P – shown in equation 5.31,

$$Q' := Q + x(s).\mathbf{0} + \bar{s}a.\mathbf{0} \oplus s(z).\mathbf{0}|!Q' \tag{5.31}$$

then Q' can receive a channel name by s and use it to communicate message a covertly to P. We could extend this functionality to enable channel name s to be stored until a further substitution is made. This creates a much more flexible approach to modelling covert channels.

5.5.2.10 Decision Making – D-MAT

We have already given examples above of using a logical condition to filter the application of functions to messages. Clearly, this can be extended to allow complex autonomous decision-making behavior in malicious processes.

5.5.2.11 Learning Behavior – LB

Since we accept that malicious processes may overwrite other processes, it follows that a malicious process may also revise another malicious process. This include the ability for a process to upgrade itself. For example, an agent could try an exploitation method and, on failing, interrogate a database of other exploitation methods to select a novel approach [98].

5.5.2.12 Co-operation – COOP

It is implicit in the model that malicious processes created by a single adversary will act in a coordinated fashion to achieve their goals. It is also possible for malicious processes from different adversaries to co-operate if their decision making capability allows it. However, to model this explicitly requires a higher order of semantics. To do this, we have subsequently developed a goal-transform calculus for which the applied π-calculus is a sub-calculus to enable us to reason explicitly about interactions between cooperating *agents* in a distributed system, where agents are defined as a set of goals (which are a special kind of process). We show an example of a coordinated attack in section 5.6.2 in this chapter and continue with it in chapter 6.

5.5.2.13 Indistinguishability – P

Definition 5.12 (Indistinguishability Assumption – P)
Let P be a process and let P' be the process which results from the subversion of P. We assume that P' can act identically to P. We call this the indistinguishability assumption.

This assumption states two things. First, we assume that any subverted agent can continue to act like its former legitimate version, in addition to selectively exercising its new malicious capabilities. Second, we assume any subverted agent will make no egregious protocol errors in interacting with other principals in the network.

The first assumption may be considered reasonable for the range of persistent, covert attacks. The second assumption, however, renders the adversary stronger than most current experience w.r.t the detection of malicious activity in networks where the presence of protocol errors is a key factor in such detection [27].

5.6 Attack Construction Based on the Adversary Model

Each individual capability by itself is unlikely, except in unusual circumstances, to form a complete attack. Capabilities will normally have to be grouped constructively to form an attack with a significant impact in any system of scale. A simple example of this principle is a "man in the middle" (MITM) attack. Another more complex example is a co-ordinated attack which can be built using the goal transform calculus.

5.6.1 Man in the Middle Attack

In this attack, there are four processes. Two processes, say P and Q are supposed to be communicating directly with each other. Through process subversion and the re-writing of channels, a third process R is able to eavesdrop on the conversation and send the information to the adversary process Ω.

We define P and Q in equations 5.32 and 5.33. P sends a message a to Q, while Q sends a message u to P.

$$P \quad := \quad \nu a \quad \bar{x}\langle a\rangle.x(z).\mathbf{0} + w(s).\mathbf{0} + \omega|!P \tag{5.32}$$

$$Q \quad := \quad \nu u \quad x(s).\bar{x}\langle u\rangle.\mathbf{0} + w(t).\mathbf{0} + \omega|!Q \tag{5.33}$$

Let Ω be the adversary and assume that Ω shares the externally facing channel w with P and Q. By SUB and ACC, Ω sends a malicious message m to Q which rewrites Q and spawns a malicious process R by SP-PAR so that $Q \overset{COM,ACC,SUB,SP-PAR}{\rightarrow} Q'|R$ – see equations 5.34 and 5.35.

$$Q' \quad := \quad \nu u \quad y(s).\bar{y}\langle u\rangle.\mathbf{0} + w(t).\mathbf{0}|!Q \tag{5.34}$$

$$R \quad := \quad (x(s).\mathbf{0} \oplus \bar{y}\langle s\rangle.\mathbf{0}) \oplus (y(s).\mathbf{0} + \bar{x}\langle s\rangle.\mathbf{0}) + \bar{w}\langle s\rangle.\mathbf{0} \tag{5.35}$$

The resulting subsystem $P|Q'|R|\Omega$ diverts copies of messages between P and Q to Ω by R which we show in proof reduction 1 where messages a and u are intercepted by R and sent to the adversary process (not shown) by channel w.

Proof

$$
\begin{aligned}
&\nu a \quad \bar{x}\langle a\rangle.x(z).\mathbf{0}|\nu u \quad y(s).\bar{y}\langle u\rangle.\mathbf{0}|\\
&(x(s).\mathbf{0} \oplus \bar{y}\langle s\rangle.\mathbf{0}) \oplus (y(s).\mathbf{0} + \bar{x}\langle s\rangle.\mathbf{0}) + \bar{w}\langle s\rangle.\mathbf{0}|!Q'|!P|!R'\\
&\rightarrow COM, P\\
&\nu a \quad x(z).\mathbf{0}|\nu u \quad y(s).\bar{y}\langle u\rangle.\mathbf{0}|\\
&\bar{y}\langle a\rangle.\mathbf{0} + \bar{w}\langle a\rangle.\mathbf{0}|!Q'|!P|!R\\
&\rightarrow MN\\
&\nu a \quad x(z).\mathbf{0}|\nu u \quad \bar{y}\langle u\rangle.\mathbf{0}|\bar{w}\langle a\rangle.\mathbf{0}|!Q'|!P|!R\\
&\rightarrow MV, MR\\
&\nu a \quad x(z).\mathbf{0}|\nu u \quad \bar{y}\langle u\rangle.\mathbf{0}|\\
&(x(s).\mathbf{0} \oplus \bar{y}\langle s\rangle.\mathbf{0}) \oplus (y(s).\mathbf{0} + \bar{x}\langle s\rangle.\mathbf{0}) + \bar{w}\langle s\rangle.\mathbf{0}|!Q'|!P|!R\\
&\rightarrow COM, P\\
&\nu a \quad x(z).\mathbf{0}|Q'|\\
&\bar{x}\langle u\rangle.\mathbf{0} + \bar{w}\langle u\rangle.\mathbf{0}|!Q'|!P|!R\\
&\rightarrow MN, MV, MR\\
&P|\bar{w}\langle u\rangle.\mathbf{0}|Q'|!P|!Q'|!R
\end{aligned}
$$

■

5.6.2 Coordinated Attack

In keeping with the malicious process model in the malicious agency model, goals belonging to one agent may subvert, or rather recruit, goals belonging to another agent, causing them to switch agent sub-domains and behave in favor of the malicious agent whose recruit they are. Since a single goal may be an agent, we refer to this as the *malicious agency model*. In practice, we normally deal with larger alliances of agents working together to subvert, or defend, a system.

To demonstrate how this works in practice, we model a co-ordinated attack. The approach is quite flexible, allowing us to plan the attack at agent sub-domain level before applying a *transform* to translate the attack into its π-calculus form. The latter form enables us as before to reason over channels and goals (processes) explicitly.

We provide an example of a co-ordinated attack using the goal transform calculus. We use both the goal calculus for planning and providing outline proofs of the attack's validity and go on to show how it can be translated into the π-calculus syntax. The example is based on [76].

We define the agent domain D in equation 5.36 where D represents an ICS, for example, the control system in a chemical plant:

$$D := ||\langle Sys\rangle||\langle Mal\rangle||\langle Observers\rangle|| \tag{5.36}$$

The domain consists of three sub-domains: Sys which may be considered to be the operator, any network hosts or nodes and the control units; Mal which is the set of malicious agents and $Observers$ which represent some specialist agents tasked with observing network traffic. For the moment, we can ignore the presence of the $Observers$ domain, whose role will become clear in chapter 6.

We define the agent sub-domains in equations 5.37 and 5.38.

$$Mal \quad := \quad ||\coprod_{j=1}^{3}\Omega_j|| \tag{5.37}$$

$$Sys \quad := \quad ||Operator|\coprod_i X_i|\coprod_k C_k| \tag{5.38}$$

where $\coprod_{j=1}^{3}\Omega_j$ is a set of three (initial) malicious agents, the $Operator$ agent represents the operator, $\coprod_i X_i$ is a set of network nodes (whose representation is simplified by treating them as peers) and $\coprod_{k=1}^{3}C_k$ are three controllers which are responsible for the settings on three kinetically related valves $V1, V2, V3$ in a control system for a chemical process.

The three malicious agents are assumed to exist in a network external to the main Sys sub-domain. They attack the control system by subverting goals belonging to Sys in order to issue commands to set one of the values $V3$ to a steady state, to shut down one of the valves $V1$ and to open the remaining valve $V3$ to its maximum extent. This process, if achieved in the right order, will result in a serious health and safety threat arising in the chemical plant. We assume the valves normally operate dynamically and the attack depends on maintaining the states for a prolonged period of time. So each agent will seek to conceal the nature of the attack from the operator by transmitting false information about the state of the system. These actions will maintain the impression that the valves are continuing to alter state over the course of the attack.

The first malicious agent Ω_1 is defined in equations 5.39 and 5.40.

$$\|\Omega_1\| := \| \cdot P|X\| \tag{5.39}$$

where

$$\|P\|_{\Omega_i} := \sum_{i=12}^{4}(P_i + ([\phi]I \oplus [\neg\phi]J + [\sim \phi]G_0) \tag{5.40}$$

ϕ is used as a placeholder for a predicate which here indicates the success or failure of the goal. For the moment, X is defined informally to be an agent which will wait for a signal and subsequently close $V1$, while concealing the result from the operator. X will replace and X_i in Sys as the result of a series of attempted subversions (subgoal) P_i. Success or failure of the overall attempt will be signalled to the third malicious agent Ω_3 by subgoals I and J respectively and, on either success or failure, the goal will terminate itself by invoking the null goal G_0.

The second agent is similarly defined except that in its case the job of goal X is to open a valve. For simplicity, we assume that the agents will subvert separate goals in Sys.

$$\|Q\|_{\Omega_2} := \sum_{i=4}^{12}(Q_i + ([\phi]I \oplus [\neg\phi]J) + [\sim \phi]G_0) \tag{5.41}$$

The third agent Ω_3 waits W for the other two agents to report whether or not they have succeeding in subverting a node X_i . The goal of X in the third agent

Term Rewriting System

$P + G \to P.G \oplus P \oplus G \oplus \mathbf{0}$
$P|G \to PG \oplus GP$
$[\phi]G \to G \oplus \mathbf{0}$ $!G \to G.G.G\ldots$

is to set the valve to a steady state and, on achieving this, to signal the other two valves to activate.

$$\begin{aligned} \| \bullet W|R|X\|_{\Omega_3} &:= (W + ([\varphi]R \oplus [\neg\varphi]G_0) \\ &+ [\sim \varphi]G_0|!W|\sum_{i=4}^{12}(R_i \\ &+ [\phi]I \oplus [\neg\phi]J) + [\sim \phi]G_0)) \end{aligned} \tag{5.42}$$

We can use the following term re-writing system – table 5.6.2 – to allow us to show the valid set of outcomes for this attack without having to transform the goal calculus to the π-calculus.

Hence, we can re-write $\|P\|_{\Omega_1}$ – see equation 5.43.

$$\|P\|_{\Omega_1} := P_4.I.G_0 \oplus P_4.P_5.I.G_0 \oplus \ldots \oplus P_4 \ldots P_{12}.(I \oplus J)G_0 \tag{5.43}$$

The transitions of $\|Q\|_{\Omega_2}$ are similar. $\|W\|_{\Omega_3}$ either succeeds or fails, equation 5.44,

$$W \to W \ldots W.R \oplus W \ldots W.G_0 \tag{5.44}$$

$\|R\|_{\Omega_3}$ functions similarly to $\|P\|_{\Omega_1}$ and $\|Q\|_{\Omega_2}$.

Taking this approach enables us to demonstrate in outline that the attack launched by the sub-domain Mal is valid, but does not provide any detail on the channels and capabilities exercised during the attack phase we have illustrated, or provide information on the success or failure conditions. We use a transform of a goal to the π-calculus to provide this additional detail. We can only accept the validity of a proof after the transform using reduction techniques or labeled transitions as appropriate.

For example, we can transform $||P||_{\Omega_1}$ as shown in equation 5.45.

$$\nu(m) \quad ||P||_{\Omega_i} := \sum_{i=12}^{4} (\bar{x}_9\langle m\rangle_{x_i}.\mathbf{0} + ([\phi]\bar{y}_3\langle s\rangle.\mathbf{0} \oplus [\neg\phi]\bar{y}_3\langle f\rangle.\mathbf{0} + [\sim \phi]G_0) \tag{5.45}$$

where m is a malicious message, each x_i is a channel shared with processes X_i in Sys. y_3 is the channel shared with Ω_3 and s and f indicate *success* and *failure* flags respectively. The malicious messages are routed via x_9 which is the only channel exposed to the agent, forming the initial attack surface W. To complete the example, we model the system which is under attack and take the attack through to the point where it is fully implemented in chapter 6.

Of course, we could further extend this example by considering other adversaries also attacking the system in parallel and the potential for clashes or for cooperation between such adversaries and their malicious cohorts. This is a key advantage of the algebraic approach as it allows us to easily introduce multiple concurrent alterations to processes and consider the effects of different subversion orders. We may also be able to extend these results by considering timing or probability within such models as well.

5.7 Incorporating Impact Analysis in the Model

An obvious requirement of an attack to be successful, from the point of view of the attacker, is that it must have the desired impact on the system. But impacts on the system are unpredictable. First, a poorly planned or executed attack may not succeed in achieving the goal of the attacker. Second, security countermeasures on the system or interventions by operators may prevent the full impact of the attack from occurring. Third, the actions of other adversaries may interfere in the attack. Attackers who subvert computer systems are known, for example, to install countermeasures against further subversion by others [29]. Fourth, attackers as much as operators are subject to the difficulties of establishing control over distributed systems due to lack of knowledge of global time or state and process or communication delays or process failures. Finally, distributed systems normally encompass large, complex and frequently geographically extensive infrastructures which may have multiple exposures in any part of the system to entry and subversion by attackers. Access and subversion attempts by attackers may also be limited by zoning restrictions internal to such systems. Any of these factors may be present concurrently in a distributed system which makes the impact of an attack difficult to model. In turn, understanding the impact of feasible attacks is a necessary precursor to network attack and defense analysis and to business risk analysis (which is used to prioritize security investment in systems).

5.7.1 Impact Source and Sink Functions

One of the applications of our approach is to introduce into our models silent functions – see section 5.3 – that allow us to calculate impact reachability and to introduce concurrently into our analysis of such systems any and all of the factors which may affect the outcome of an attack. The analysis is possibilistic in nature. It demonstrates that an attack has an impact and characterizes the impact in terms of confidentiality, integrity and availability – treating these characteristics as binary in nature. It demonstrates how the impact is realized in the system, including immediate, transitive and recursive (including reflexive) impacts. In π-calculus terms, the impact is analyzed in terms of functions over names, that is the transformation of names to names. Excluding the creation of new names by replication (which can be subsumed in the analysis of transitive effects), this limits the analysis to a finite set of functions. The approach also allows us to demonstrate that distinct attacks may be equivalent in impact and to define the effectiveness and efficiency of security countermeasures or defensive interventions.

Let $K = \{C(onfidentiality), I(ntegrity), A(vailability\}$ be the set of security characteristics. Let $\uparrow_\kappa (\bar{n})_P$ be a *source* function where $\kappa \subseteq K$ and $\bar{n}$ is a vector over names in a process P. When introduced to a process in associated with a *subversion* function, the source functions mark the names over which they are invoked with the security characteristics which are lost for that name, written $\bar{n}_\kappa, \bar{r}$ where $\bar{r}$ is a vector which captures routing information.

Let $\downarrow_\kappa (\bar{m})_Q$ be a *sink* function where $\bar{m}$ is a vector over names and Q is a process. The introduction of a sink function marks where a process has a function which makes use of a name which has been previously subverted. Any names generated by the process will also be considered to be subverted, meaning that impacts can be calculated transitively by inserting a new source function. These names are called *subverted names*. A subversion function is any function introduced or altered by the attacker so as to manipulate names or the behavior of names in the system. Recursive (including reflexive) effects may be captured by paying attention to routing information in association with subverted names. Finally, we may also, by inserting dummy processes, capture conjoint impacts where impacts in disparate parts of the system create an overall impact which is greater than the sum of its parts.

Impact sinks are defined as follows. Let S be a system:

Definition 5.13 (Loss of Confidentiality)
Loss of confidentiality arises where any subverted process W receives a name it is not authorized to receive. In particular, if it is an adversary process outside the bound of the system.

Definition 5.14 (Loss of Integrity)
Loss of integrity arises where any process $P \in S$ receives a subverted name whose content has been altered by the adversary.

Definition 5.15 (Loss of Availability)
Loss of availability arises where a process consistently *fails to receive a name which it would receive in normal processing. Loss of availability may be direct as the result of name dropping or process failure, or indirect due to channel starvation.*

Various cases arise from our calculation which we outline here:

Case 1 – A source may be a sink without any further steps after subversion, that is, attack initiation has an immediate impact within a process. In this outcome, the only impacts are loss of availability and loss of integrity as the process by definition in inside the boundary of the system, so loss of confidentiality may not be considered. All the action is also within the process and there are no transitive computational effects.

Case 2 – A single source may lead to a single sink. The loss of all security characteristics may be considered. The impact is a single impact on another process.

Case 3 – A single source may lead to multiple sinks. Multiple impacts occur possibly for the loss of all security characteristics.

Case 4 – Multiple sources may lead to a single sink. This outcome gives rise to being able to consider attack variants which are equivalent w.r.t. impact or multiple attacks from independent sources which cooperate to produce a single impact.

Case 5 – Multiple sources may lead to a multiple sinks. This outcome is an obvious extension of the fourth case.

Case 6 – A sink may transitively become an impact source (not necessarily with the same characteristics) for further sink

Case 7 – A source may act recursively becoming a sink for itself after a number of intervening steps

Case 8 – A source may act transitively resulting in a further sink acting recursively

Case 9 – An attack may result, under different orderings, in different impacts. This outcome reveals a situation where an attack starting with apparently identical conditions can result in multiple possible impacts, depending on event ordering given a projection over events. Part of the strength of our approach is it enables structural reasoning over these possible outcomes by considering all orderings of events, rather than imposing an artificial order on attack and defense maneuvers or outcomes.

Let P, Q, R be processes as defined in equations 5.46, 5.47 and 5.48 such that P sends a name a to Q which in turn transmits it to R and returns it to P. Two transformations occur over the name in P and R.

$$P := \bar{x}\langle a\rangle.\mathbf{0} + s(u).(f(u) \supset a).\mathbf{0}|!P \tag{5.46}$$
$$Q := x(z).\bar{y}\langle z\rangle.\mathbf{0} + \omega|!Q \tag{5.47}$$
$$R := y(t).(f(t)\ supsetv).\bar{s}\langle v\rangle.\mathbf{0}|!R \tag{5.48}$$

We assume an attacker inserts a malicious function which subverts Q, shown in equation 5.49

$$Q' := x(z).MI(z) \supset z'. \uparrow_I (z)_{\{Q\}}.\bar{y}\langle z_{I,\{Q\}}\rangle.\mathbf{0}|!Q' \tag{5.49}$$

We can summarize the subsequent transitions using the action relations shown in the set of transitions in the proof reduction. P sends a to Q by channel x. Subsequently, the name is manipulated by the malicious function and passed to R.

PROOF

$$\begin{aligned}
&P \xrightarrow{\bar{x}a} P1\\
&P1 := s(u).(f(u) \supset a).\mathbf{0}|!P\\
&Q' \xrightarrow{xa} Q1'\\
&Q1' := MI(a) \supset a'. \uparrow_I (a)_{\{Q\}}.\bar{y}\langle a\rangle_{I,\{Q\}}.\mathbf{0}|!Q'\\
&Q1' \xrightarrow{MI(a)} Q2',\\
&a \xrightarrow{\uparrow_I} a_{I,\{Q\}}\\
&Q2' \xrightarrow{\bar{y}a_{I\{Q\}}} Q3'\\
&Q3' := \bar{y}\langle a\rangle_{I,\{Q\}}.\mathbf{0}|!Q'\\
&R \xrightarrow{ya_{I\{Q,R\}}} R1\\
&R1 :=\downarrow_I (a)_{\{Q,R\}}.(f(a)_{I,\{Q,R\}} \supset v). \uparrow_I (v)_{\{Q,R\}}.s\langle v\rangle_{I,\{Q,R\}}.\mathbf{0}|!R
\end{aligned}$$

■

This set of actions results in $R1$ being redefined to include a sink function with a transitive source function in relation to the name v which depends on a. Further transitions result in a recursive impact on P. We remind ourselves of the state of P in equation 5.50.

$$P1 := s(u).(f(u) \supset a).\mathbf{0}|!P \tag{5.50}$$

P receives the subverted name $v_{I,\{Q,R\}}$ by the action relations $R1 \xrightarrow{\bar{s}v_{I,\{Q,R\}}} R2$, $P1 \xrightarrow{sv_{I,\{Q,R\}}} P2$ from $R1$ and is subsequently re-defined with a sink and source function respectively.

$$P := \bar{x}\langle a\rangle_{I,\{P,Q,R\}}.\mathbf{0} + s(u).(\downarrow (u)_{I,\{P,Q,R\}}.(f(u)_{I,\{P,Q,R\}}) \supset a_{I,\{P,Q,R\}}).\mathbf{0}|!P \tag{5.51}$$

This example shows us how source and sink functions can be identified in the system by considering action relations amongst processes and re-defining processes accordingly. It should be noted that any source or sink function once inserted is considered to be stable until an explicit action is taken to remove its effect. In some cases, for example, where a process receives the same name from a number of different sources, this may require distinguishing between subverted names received from malicious sources and names which have not been affected. Algebraic notation supports distinguishing cases using routing information and the function $+$ which excludes actions which may not occur due to the falsification of any conditions associated with those actions. Indeed, it is recommended practice to insert source and sink functions using the $+$ notation to permit conditional exercise of the functions and it is only avoided here for simplicity.

5.7.2 Search Space

An important consideration in analyzing attacks and defenses is the size of the search space. Here we show that the search space is of order $O(n)$ and is constrained by the number of *fresh names* (i.e., variables) in the system.

We assume the number of processes in a system is finite. Let L be the set of fresh names (data variables) in a system. Fresh names may be initiated, sent and received and acted on by functions. We say that names are *transformed* by functions, allowing us to distinguish between the same name where the value associated with the name has been updated.

Definition 5.16 (Image Finiteness)
Let Act be the set of action relations over inputs and outputs of a process. For any process P let there be finitely many processes Q for which $P \xrightarrow{\alpha}$ where $\alpha \in Act$. This is property is called image finiteness.

Definition 5.17 (ω Finiteness)
Let F be the set of functions in a process which are not input or output functions. We assume F is finite (by construction). These are functions which transform names to names. Let $Act_\omega = Act \cup F$. For any process P let there be finitely many processes Q for which $P \xrightarrow{\alpha}$ where $\alpha \in Act_\omega$. We call this property ω-finiteness.

We need the notion of ω finiteness as, in our theory, functions may be substituted for functions and functions may be inserted arbitrarily into processes.

Lemma 5.1 (Image Finiteness)
All processes are image-finite.

PROOF Any process may be shown to be image-finite up to structural congruence [34]. The proof shows that there are finitely many names per process which can be subject to input and output actions.

Corollary 5.2 (ω-finiteness)
Since Act_ω is finite and all processes are image finite, it follows that all processes are ω-finite.

PROOF Obvious. ■

Hence our search space is finite since there are finitely many names per process which may be subject to input and output actions and functional transformations.

Theorem 5.3 (Number of Impacts in a System)
The maximum number of impacts in a system is equal to 8 times the number of names in the system and hence is order $O(n)$, including any transitive, recursive and reflexive impacts.

PROOF Ω finiteness tells us that there are a finite number of actions over names per process up to structural congruence. Ignoring communication function, the remaining functions are transformations of names to names. Let $N = |L|$ be the number of fresh names in the system. We can easily see that the maximum number of transformations of names to names is N, including the possibility of the transformation $\emptyset \to l \in L$ on initiation.

Let L be the set of names. Let 2^L be the power set of L. Let H be a Hesse diagram which represents 2^L. Let each set in H represent a possible grouping of names in a process (i.e., the names forming the input to a process, or the names generated or transformed by a function in the process). We trace the set of transformations of names to names by navigating the edges of H and eliminating any edges which do not represent a dependency of names on names - that is we only consider sets of names which are included in a function in the system. We include the initiation of names (which is considered a transformation of the $\emptyset$ to a name $l \in L$. Let G be the set of sets of names on the graph formed by eliminating edges from H.

For example, let $L = \{a, b, c\}$. Let P be a process with a function f such that $f(a, b) \supset c$. Let f be the only function in the system which is not an input or an output, but a transformation of names to names. It is easy to see that there exist transformations $\emptyset \rightarrow \{a\}$ and $\emptyset \rightarrow \{b\}$. There is no transformation $\{a\} \rightarrow \{a, b\}$ or $\{b\} \rightarrow \{a, b\}$ as these sets result from inputs or outputs to processes. There is a transformation $\{a\} \rightarrow \{a, b, c\}$ and correspondingly for $\{b\}$. Finally, c cannot exist except in the presence of both a and b. The corresponding graph is shown in figure 5.1.

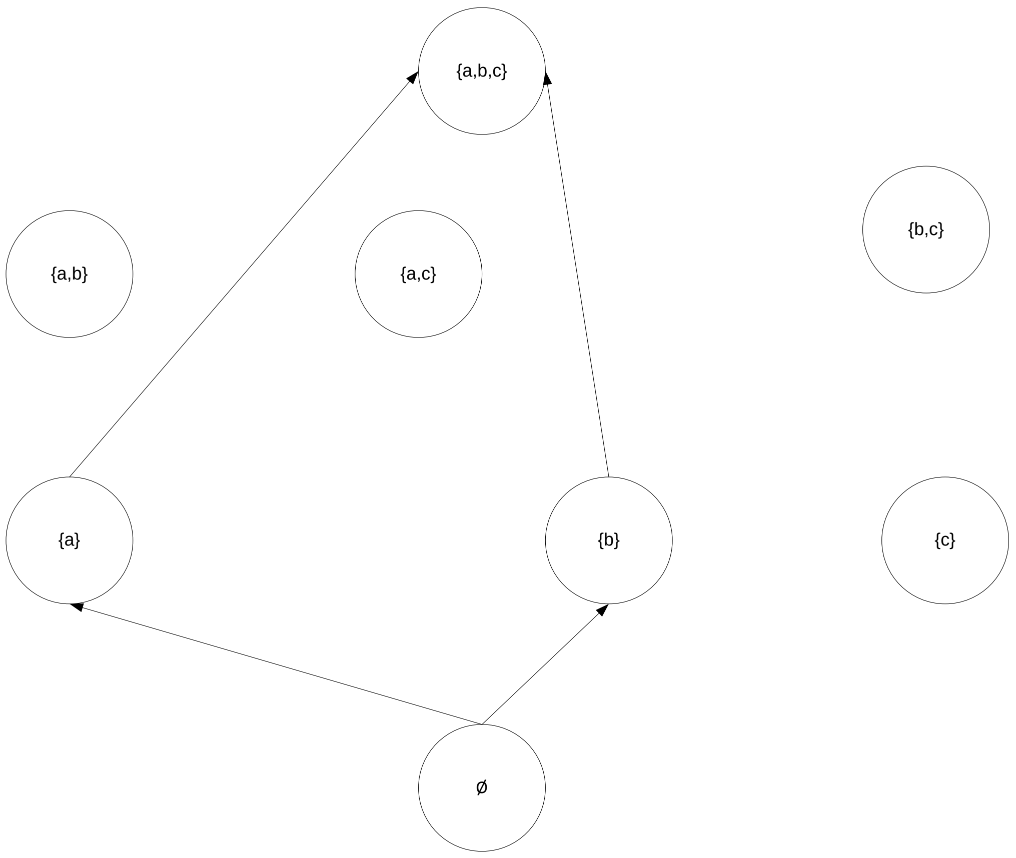


Figure 5.1: Transformation Dependencies

This approach allows us to easily see that the maximum number of possible transformations in system which, including the null transformations (on initiation) from $\emptyset \rightarrow l$ where $l \in L$ is the same as the number of fresh names $|L|$.

Let K be the set of security characteristics as before. We assume the characteristics are binary in nature. Let 2^K be the power set of K. Let $N = G \times 2^K$ be the cross product which represents all possible groups of names in L and all possible associated impacts. It is easy to see that the total number of impacts to be considered in any system is $|L| \times |2^K| = 8|L|$ which is of order $O(n)$ as required.

. ■

This proof suggests a technique for limiting the search for impact sources and sinks to considering only processes which contain functions which transform names to names and making appropriate use of routing information. However, this is only initially useful as the effectiveness of the attack must be considered under all linearizations (or runs) of the system which could be done by model checking the system specification (or, more laboriously, as a manual proof) . In particular, once we start to consider security countermeasures and defensive interventions, the search must be extended to all possible runs of the system. But our proof demonstrates that the search space is finite for any initial impact analysis regarding an attack scenario.

This approach – searching all potential orderings of events – which is one of the advantages of an algebraic method, ensures that we take into account events extraneous to the processes under attack which may nonetheless affect the outcome of such attacks and includes effects such as process failure and communication losses and delays as well as operator actions, which may or may not be related to the actions of the attacker.

5.7.3 Attack and Countermeasure Equivalence

By the pigeon hole principle, assuming the number of potential attacks is larger than the number of potential impacts, theorem 5.3 implies that there may be attacks which are impact equivalent. In this section, we define what it means for attacks to be equivalent to each other. We also define equivalence with regard to countermeasures and the notions of effectiveness and efficiency of countermeasures. Countermeasures are considered to mean both defenses already implemented in a system and any dynamic interventions by system operators or automated agents in defense of the system.

Definition 5.18 (Attack Equivalence)
Let I, J be two distinct attacks. Two attacks I and J are held to be equivalent $I \sim J$ if the maximum impact of I is the same as the maximum impact of J, written $im(I) \approx im(J)$ for a given system S.

Informally, equivalence says that both I and J affect the loss of the same security characteristics in K for the same set of names $\tilde{n} \in L$. This is qualified under the requirement that equivalence only applies to the maximum impact detected when considering all linearizations of the system. It may be that under certain linearizations attacks are less effective or indeed more effective than intended by the attacker or on the initial review of their potential impact sinks. Seeking to establish equivalence between attacks with different impact sources as a result of different event orderings is considered too fine a distinction. In other words, we only consider worst case scenarios.

We also define *defensive interventions* which are countermeasures, which may be static or dynamic and what it means for countermeasures to be effective, efficient and equivalent.

Definition 5.19 (Defensive Intervention)
An defensive intervention *is a programmed action or countermeasure by the operator which may be applied prior to or during an attack may result in an* impact reduction*.*

Definition 5.20 (Countermeasure Effectiveness)
An intervention is considered effective *if it reduces to zero, the impact of an attack I. Partial effectiveness is present where the the impact of the attack is reduced, but not to zero.*

Definition 5.21 (Countermeasure Efficiency)
An intervention is considered to be efficient if for an equivalence class $[I]$ of attacks, it is equally effective.

Definition 5.22 (Countermeasure Equivalence)
Two interventions E and F are equivalent $E \sim F$, if for the same attack I on the same system $S(P)$, they are equally effective.

Definition 5.23 (Countermeasure Congruence)
Two interventions E and F are congruent $E \cong F$, if for the same attack class $[I]$ on the same system $S(P)$, they are equally effective.

Definition 5.24 (Countermeasure Set Equivalence)
Two sets of countermeasures $\mathcal{E}$ and $\mathcal{F}$ are equivalent $\mathcal{E} \sim \mathcal{F}$, if for the same attack I on the same system $S(P)$, they are equally effective.

Definition 5.25 (Countermeasure Set Congruence)
Two sets of countermeasures $\mathcal{E}$ and $\mathcal{F}$ are congruent $\mathcal{E} \cong \mathcal{F}$, if for the same attack class $[I]$ on the same system $S(P)$, they are equally effective.

We note that unlike attacks, where only worst case scenarios are considered, we may consider countermeasures to be equivalent (or congruent) based on partial outcomes with respect to the reduction of impacts. This approach respects traditional approaches to security where attacks are normally analyzed on a worst case basis, while it is not required that countermeasures be wholly effective for them to be deployed. Nevertheless, our definitions are intended to encourage the design of efficient and effective countermeasures and to encourage choice between equivalent countermeasures and sets of countermeasures. The latter, in particular, are defined to encourage thinking in terms of "defense in depth".

5.8 Summary

In this chapter, we have defined a conceptual threat model which allows us to attribute malicious capabilities to processes or goals belonging to agents. These are comparable to the capabilities assumed in the Dolev-Yao model but differ since they may only be exercised locally, constraining the adversary to the same level of knowledge of the system as operators or other processes. The model is formally instantiated in our research, using two variants of the π-calculus, depending on our application, as we believe this is the most appropriate technique, based on its past usage, for capturing the complexities of a dynamic, interactive and mutating environment. We illustrate the use of the model by, first, providing suitable encodings of each of the capabilities we ascribe to the adversary and, second, showing how these capabilities may be built into realistic attack scenarios. Applications include exploring possible adversary behavior for the purposes of attack and defense and risk analysis and demonstrating the safety or liveness of security protocols and techniques for dealing with malicious software incursions. Basic examples are provided in this chapter and, in chapter 6, as part of the discussion on the use of traceback protocols , we provide two extended examples of this approach. We discuss this work in relation to other contributions in chapter 9

Chapter 6

Traceback Protocols for Defending Communications Integrity

The best material model for a cat is another cat, or preferably the same cat.

Norbert Wiener

6.1 Preamble

In chapter 5, we set out our approach to algebraic modeling of adversaries in systems. This chapter may be regarded as an extended example of that approach in action. We also use this chapter to address the issue of maintaining the integrity of network communications in ICS between the operator and the controller. We introduce a security protocol based on the use of IP Traceback methods, normally used for detecting the source of denial of service attacks, to detect integrity attacks and locate the subverted processes implicated. We describe the protocol in section 6.2. We show how it may be used to detect an adversary who manipulates messages (man-in-the-middle attack) in an ICS network and, in section 6.3, we demonstrate how it can be used to filter falsified messages. We prove the safety of the protocol. We also show how the approach continues to function under dynamic network alteration where the adversary becomes mobile in the network.

6.2 A Traceback Protocol for Adversary Detection

One means of undermining an ICS system is to manipulate network communication between the operator at an HMI and the control unit. This can be used to manipulate the controller while hiding the results from the operator . Alternatively, it may be used to convince the operator to carry out unnecessary, potentially dangerous, actions. This approach may also be combined with the direct manipulation of the control unit , perhaps to prevent side effects showing up elsewhere in a system as symptoms of manipulating a plant. Hence we see a potential association between such techniques and a coordinated attack against an ICS. Our purpose is to locate a subverted process which is responsible for undermining the integrity of network communications.

We limit our consideration to integrity attacks which result from the subversion of processes in the communication and control network [120]. We also only consider attacks which are subject to manipulation of the data package and assume that attacks on timing or ordering of messages are taken care of by other protocols such as message synchronization or timestamps (though our protocol may support their application).

6.2.1 Assumptions

We identify the subverted process with a network node and, as a simplifying assumption, treat all nodes as peers in the system. We also assume, momentarily, that we have some other means of detecting anomalies in the system – for example, by employing the techniques in chapter 8 – which allows us to initiate the protocol. So our goal, at this point, is not initial detection, but to discover the subverted node or, at least, a small set of candidates for investigation. As a further simplification, we focus on a single process along a single communication channel. The technique will be extrapolated later to cover multiple processes and routes. It should be noted that our technique is based on IP Traceback protocols, but we do not limit ourselves to IP communication as a category. The protocol is defined algebraically and could be applied to other SCADA protocols. We simply wish to establish the principle that such techniques may be used to establish message integrity, opening up a new area for original research.

We adopt the assumptions established in [109] for messages sent by the adversary:

1. Packets may be lost or reordered,
2. Attackers will send numerous packets (i.e., manipulation is constant, not arbitrary)
3. The route between attacker and victim is fairly stable,
4. Network nodes are both CPU and memory limited, and
5. Network nodes are not widely compromised.
6. An attacker may generate any packet,
7. Multiple attackers may conspire,
8. Attackers may be aware they are being traced,
9. A compromised router can overwrite any upstream information in the direction of communication

Clearly, these assumptions are valid in terms of our threat model as set out in chapter 5. Therefore, if we detect an anomaly by other means, we may assume that there exists a subverted process which either falsely inject messages MI or manipulates them in transit MM, whether they are instructions from the operator or else sensor readings from control units. The protocol may be applied to both. We simplify the argument, at least initially, by considering only the latter. But we also avoid any assumption that the subverted processes used to manipulate signals from the controller to the operator also manipulate commands from the operator to the controller. Finally, we also assume that the adversary is "protocol perfect" in their interaction with other processes which is the indistinguishabilty assumption P from chapter 5, i.e., that subversion is only detectable by using anomalies in data values and not by other means.

6.2.2 Outline of Protocol Action

On detecting an anomaly in the physical behavior of the system, we initiate the protocol by sending a suitably encoded message[1] to activate all nodes between the operator and the implicated control unit(s). We assume the control unit provides each message it sends with a unique identifier (for example, based on packet characteristics). On initiation of the protocol, each node will, in addition, attach its identity, suitably hashed, to messages as they pass through the network, thus marking the route the message took.

Each node between the controller and the operator, subsequent to the initiation of the protocol, will probabilistically make and forward a copy of any flagged messages, which it encrypts using a rotating key algorithm which we describe in section 6.2.5. Subsequently, the keys will be released to the operator and may be used to decrypt the message packets. Confirmation of anomalous messages is trivial using simple comparison. However, the routing information associated both with the message and the message copy enables us to reason about the location of subverted processes in the network. In essence, a message which has not been subverted has transitioned across a set of non-malicious process nodes, while a message which has been manipulated or falsely injected has passed through or originated in a subverted node. By comparing routes we can through a process of elimination identify to within two process nodes the malicious process.

We exclude the endpoint control units from the initial scope of consideration. Where the control unit has been subverted, no conflicts in messages would appear in the network data stream. However, subsequent to applying our technique, subverted control units may be detected negatively – by considering the nature of the physical anomalies report and the lack of message conflicts on channels relevant to these control processes, leading to the conclusion that particular control units have been subverted rather the communication channels which map to them. Where the operator is acting maliciously, our approach is invalidated.

The advantage of our approach is that it enables us to efficiently pinpoint subverted processes. Compared with probabilistic packet marking where the expectation of the number of packets required to trace an attack is given by a hypergeometric probability distribution and is in the order $O(n \log n)$ [109], we are able to use knowledge of network routes, source and origin (here defined algebraically, though not necessarily so), to rapidly eliminate valid nodes with notable economies, reducing the messaging complexity to $O(n)$.

6.2.3 Algebraic Specification

We provide a formal description of process action and define it using the π-calculus. We begin by describing the action of the protocol, assuming that the adversary is attacking the system, rather than seeking to undermine the protocol directly. Under

[1]for example, requesting setting a flag in all consequent packets sent from the control unit

this assumption, we seek to show that the protocol will allow us to determine to within two nodes, which node is involved in the manipulation of data traffic.

Let V be any node which is not an operator , nor a endpoint control unit . Initially, for all packets, we set a flag $f = 0$ and packet observation does not occur. We assume an anomaly in physical system responses causes the operator to initiate the protocol for relevant processes $V_1, V_2, \ldots$ by setting $f = 1$ on message packets and the controller corresponds. Packets include information about origin, identity and routing. On receiving flagged packets, whether inputs or outputs, nodes determine, based on some uniform probability p, to copy identified packets. The $Observe()$ function – equation 6.2 – copies the packet and encrypts its contents, the packet identity. Then it sends the copied packet, which we label *observed*, to the operator.

Finally, after a period of time, each node $V_i, V_{i+1}, \ldots$ publishes a previous key (in sequential order) used for encrypting packets to the operator and generates a new key in such a way that subsequent (and previous, unrevealed) keys cannot be predicted – section 6.2.5. This action is controlled by the function $NewKey()$ – see equation 6.3 – and it should be noted that the names used by this function for keys and time are restricted to the scope of the function. Hence these values can only be known to the adversary in nodes which he has already subverted.

On receiving copied packets whose identities match, the operator can compare the data contents of copied packets with the original, noting its origin and routing. Hence where packet contents differ due to adversary manipulation, it can be determined from several packets which routes may have been subverted by the adversary, simply by observing the hashed routing information of observed packets and whether or not they have been manipulated.

A process of elimination over each route leads to an *approximate* determination of which nodes have been subverted on which routes – see section 6.2.4 – which is sufficient to reduce the scope of investigation to a couple of processes, say V and U.

Formally, we define the action of V in equation 6.1

$$V := \nu(a, \tilde{k}, t, p) \quad (B.K + C)|!V \tag{6.1}$$

where a is the node identity, $\tilde{k}$ are a set of keys (see section 6.2.5) and t is a logical clock based on message passing events. B is the summation which receives and observes packets and is defined in equation 6.2. K is the summation which creates keys and releases them after a period of time to the operator and is defined in equation 6.3. C is the summation (equation 6.4) which forwards packets, marking them (not shown) with the node identity.

$$B := \sum x_i(u)_{(r,f,d,\alpha)}.Mark(r) \supset u_{(r',f,d,\alpha)}.Observe(p, V, f, t, u, k_i) \supset \bar{x}\langle w \rangle_{\langle r,0,d,\alpha \rangle}.\mathbf{0} \tag{6.2}$$

where x_i is a set of channels, p is the observation probability, u is a placeholder for names (i.e. packets) to be received, r is the final address to which the packet is routed, d is the packet identity and α is vector of node identities created during the onward transmission of the packet by S. V is the node identity and f is the current flag value. x is a reserved channel (one of x_j – see equation 6.4) used to send observed packets to the operator and w represents an observed and encrypted copy. k_i is the current key. The $Observe$ function is defined by algorithm 6.1 though this is really just a feasible coding and other more efficient algorithms may exist. It determines if the observation flag is set to 1 and consequently determines based on the marking probability whether or not to copy and encrypt the packet and forward it to the operator:

Algorithm 6.1: Observe

```
1
  Parameters: $Prob, NodeIdentity, ObservationMode, Time, Packet, CurrentKey$
  Var: $HashedPacket, HashedAddress, HashedData$

  If $ObservationMode \neq 0$
   If $Prob \leqslant Rand()$
     $Increment(Time)$;
     $HashedAddress := Hash(NodeIdentity, CurrentKey)$;
     $HashedData := Hash(Packet, CurrentKey)$;
     $HashedPacket := Concatenate(HashedAddress, HashedData)$;
     Return(HashedPacket)
   Endif
  Endif
```

The t or $Time$ variable is a simple integer value which is incremented every time a packet is copied.

In equation 6.3, t is the logical clock as before, k is the fresh key to the operator which is generated by the function $NewKey$ defined (again trivially) in algorithm 1. However, $k_{t-\delta}$ is a previous key used for encryption being forward to the operator. f is the observation mode flag. x is the reserved channel for communicating to the operator. r is the routing address of the packet as before. Note that we set the flag to 0 for packets which need not be observed.

$$K := NewKey(t, k, \tilde{k}, f) \supset (\bar{x}\langle k\rangle_{\langle r,0,t-\delta\rangle}.\mathbf{0}) \tag{6.3}$$

The C summation (equation 6.4) contains the set of channels to which the node V sends packets x_j, a placeholder u for the packet to be forwarded. The characteristics are defined as for previous equations.

$$S := \sum \bar{x}_j\langle u\rangle_{\langle r,f,d,\alpha'\rangle}.\mathbf{0} \tag{6.4}$$

6.2.4 Protocol Complexity

We now consider the average number of observation packets required for the detection of a subverted process. In fact, it is not possible to fully fix a subverted process, but we may reduce our search to a small number, possibly two.

Defining the network topology as a graph, let R be a single route (or path) which is part of a larger subgraph of routes $\mathcal{R}$ between an operator and a control process and consider the detection of a single subverted process. We initially limit our consideration to the manipulation of a single output signal. The equivalent technique used to check input commands originating with the operator is omitted for simplicity. First, we define the *observation range* of R:

Definition 6.1 (Observation Range)
On R, we label the process directly adjacent to the operator S and the control unit T respectively. We call the set of processes from S to T, but not including T and S, the observation range *of R and we write $Obs(R)$.*

Let $<$ be the relation "follows" in (a total) order of communication. We require observation packets to be sent by the processes in $Obs(R)$. If we determine on a new S' or a new T' then we have a new observation range for R which is $Obs(R')$. For convenience, we label the processes in $Obs(R)$ which are not S or T as a set V, possibly empty, with members $\{V_1, V_2, \ldots, V_{n-1}\}$ in order of communication, e.g., $V_2 < V_1$. We also note that if the number of processes from S to T inclusive is n then the number of processes in $Obs(R)$ is $n-2$.

If we can already see mismatches between observation packets and original packets, we assume that, at least, process S is producing a manipulated packet. Hence we use $\mathcal{S}$ to designate the set of processes which communicate invalid packets. Likewise, we assume that T (for the moment) is producing valid packets and use $\mathcal{T}$ to designate the set of nodes which communicate valid packets. To find out how many processes are in $\mathcal{S}$ and how many are in $\mathcal{T}$ we need to collect information about every process in R.

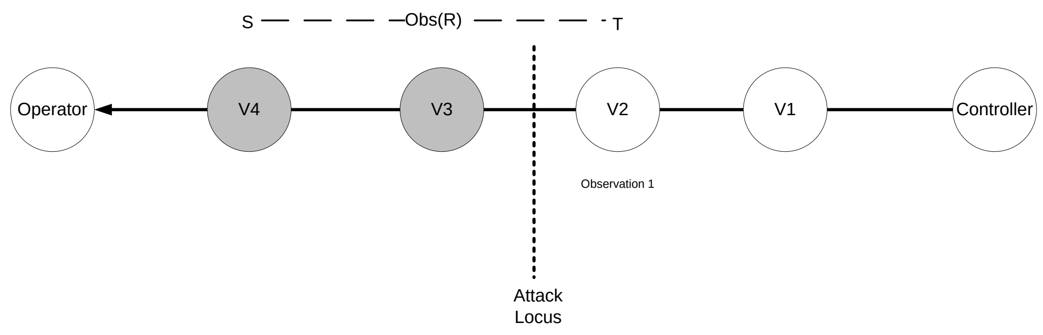


Figure 6.1: Packet Collection in $Obs(R)$

If we observe a valid packet from a previously unobserved node V_i then we can designate $V_i \in \mathcal{T}$ and also all nodes $V_{i-1}, V_{i-2}, \ldots$ which sent the packet to V_i. We can determine which nodes are in $\mathcal{S}$ using similar reasoning regarding invalid packets. Hence we can designate the process V_i to be the new endpoint for the search (either the new S' or a new T'). This means that the next valid observation operation will take place in the new *observation range* $Obs(R')$. This situation is shown, after an observation has already been taken, in figure 6.1. The action of observation is strictly monotonic since we ignore observations from any nodes already designated part of $\mathcal{S}$ or $\mathcal{T}$.

Once we have determined membership of $\mathcal{S}$ and $\mathcal{T}$, we consider that any two adjacent nodes in $\mathcal{S}$ and $\mathcal{T}$ are potentially subverted. This depends on whether the subverted process manipulates the data from the control process before or after it is observed. If it is subverted before it is observed, then the first process in $\mathcal{S}$ is the subverted process. If it is subverted after it is observed, then the last process in $\mathcal{T}$ is the subverted process. Obviously, this can only be determined by forensics analysis.

The protocol's complexity is related to the card collector's problem which is encountered in IP traceback using probabilistic packet marking for detecting the source of DOS attacks [96], but with strong efficiencies, assuming a uniform observation probability. These arise because each packet observation inside $Obs(R)$ may allow the elimination of a random number of packets in $Obs(R)$.

Let X_i be a random variable which is the number of observations required to observe a packet in $Obs(R)_i$ where $i = 1, 2, 3, \ldots$ is the number of previous observations inside each successive $Obs(R)$. On each observation, the number of processes in $Obs(R)$ shrinks by a random amount d_i. Since the probability of a process making an observation is uniform, it follows that the (independent) probability of making an observation inside $Obs(R)_i$ is $\frac{n-\sum_i d_i}{n}$. (The number of actual packets observed will be a ratio $\frac{1}{p}$ of the observation probability). Let Y_j be the number of observations required to acquire total knowledge (w.r.t membership of $\mathcal{S}$ and $\mathcal{T}$) of the processes in $R_i \in \mathcal{R}$ then the expectation of Y_j which is $E(Y_j)$ is calculated (based on a geometric probability distribution function) as shown in equation 6.5 since the observation range shrinks randomly by a distance of d_i nodes for which a determination has been made on each observation and $d_0 = 2$ and the sum $\sum_j d_j = n - 1$ where $j \geq 0$ (because we exclude S and T from $Obs(R)$).

$$E(Y_j) = \sum_i E(X_i) = \frac{n}{n-d_0} + \frac{n}{n-(d_0+d_1)} + \frac{n}{n-(d_0+d_1+d_2)} + \ldots + \frac{n}{n-(\sum_j d_j)} \tag{6.5}$$

The sum $\sum_j' d_j' = n - 3$ where is $j' > 0$ (i.e., excluding the constant $d_0 = 2$) implies that we have positive integer solutions $d_1 + d_2 + \ldots + d_r = n - 3$ for each $r \in \{1..n-3\}$. Hence there are $Q = \sum_{r=1}^{n-3} \binom{n-4}{r-1}$ possible sums each of which is equally likely to occur. Let Y be a random variable which indicates the average total number of packets required to take complete observation of the set of nodes R, then we have

$$E(Y) := \sum_{j=1}^{Q} \frac{1}{Q} \left[E(Y_j)\right] \tag{6.6}$$

We should also consider the interesting situation where a single process is implicated for the adjacent nodes S and T (at the end of our procedure) along more than one one route R_i. Implication in more than one such relationship for a single process increases the likelihood that the implicated process is also the subverted one. This situation is illustrated in figure 6.2 where detection along dual routes gives a clear indication of the subverted node.

Considering detection for all routes R_i in a subgraph $\mathcal{R}$ connecting an operator with a single control process, we see that further efficiencies may be gained where

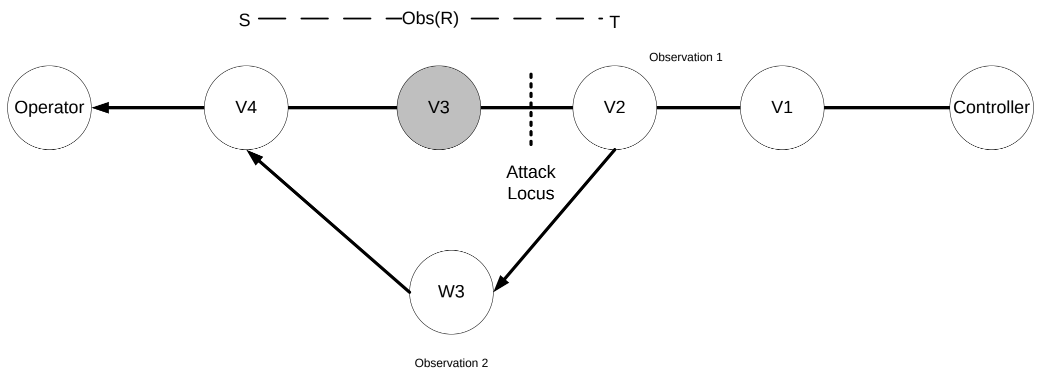


Figure 6.2: Detection Efficiencies in the the Subgraph $\mathcal{R}$

routes are not edge-disjoint since node validity may be determined along several routes simultaneously. Hence the maximum expectation $\sum_{k=1}^{m} [E(Y)_{max}]_k$ needed for observation occurs when each route $R_i \in \mathcal{R}$ is edge-disjoint. Hence we reach the surprising conclusion that increasing complexity in the subgraph $\mathcal{R}$ need not necessarily increase the protocol's complexity required by our search but that this depends on the degree of edge-disjointedness. This finding has implications for the design of ICS systems since it implies that while having multiple routes is obviously useful for resilience, lowering the degree of edge-disjointedness has advantages for detection using traceback techniques.

6.2.5 Time-Sequenced Key Value Release

We describe the key generation and release procedures . In essence, A network node V generates or is supplied (not shown here) an initial sequence of keys $\tilde{k}$ and subsequently generates a fresh key based on a nonce using a randomly selected key from its key chain, using a suitable one-way function. The fresh key becomes part of its key chain, randomly replacing another key, and is used for packet observation from that point. Depending on some time period, or set of discrete events (e.g., observing n packets), the network node generates another fresh key by hashing a randomly selected key from its key chain with the current key and replacing one of the keys, again at random. It subsequently publishes the previously used key to the operator. Subsequent keys are released in the order in which they are used.

This scheme is similar to the one proposed in [115]. This approach is resilient to attack since insufficient time exists for guessing keys while both the packet identity and its contents are hidden from the attacker [133], hence packets cannot be directly manipulated and may only be arbitrarily delayed, dropped, re-routed or vandalized. However, these forms of attack only delay rather than disrupt discovery – section 6.2.6. In some cases, they may accelerate it by providing further packet-based anomalies, hence detracting from agent capability to conceal their presence using protocol forgery. Algorithm 1 provides a feasible encoding.

Algorithm 6.2: New Key

```
1
        Parameters: $Time, CurrKey, KeyChain[0 \ldots n], ObservationMode$
        Const: $Limit, PrivateKey, OperatorIP$
        Var: $PreviousKey$
        Static: $Index$
        If $Time > Limit \&\& ObservationMode = 1$
         $PreviousKey := Keychain[Index];$
         $CurrKey := Hash(PreviousKey, Rand());$
         $Index := IntegerRand(n);$
         $Replace(KeyChain[Index], CurrKey);$
         $Sign(PreviousKey,\ PrivateKey);$
         $Return\ PreviousKey$
        Return
```

6.2.6 Protocol Safety

The key property of the protocol is that it enables us to reduce the task of identifying a specific malicious attack to a search of neigboring nodes. We demonstrate this property for the protocol. We also prove the safety of the protocol in the face of direct action by the adversary. We show that any form of message manipulation does not change the result.

The safety property of the protocol may be expressed by the recursive equations 6.7 and 6.8 which say that x_k represents a stopping state where either S' and T' are pairwise adjacent written $(S'|T')$, or k significantly exceeds $E(Y)$ losing the adversary the characteristic of indistinguishability (P). The nature of an anomaly (where it exists) allows us to identify candidate adjacent nodes, based on reasoning over the behavior of the protocol. In other words, we can deduce a total knowledge of the state of nodes. We refer the reader to the threat model shown in table 5.1 in chapter 5.

$$x_0 = \mathcal{R} \tag{6.7}$$

In the state x_0, all we know is that there is a subverted node somewhere between S and T

$$x_t = Observe(x_{t-1}), \neg((S'|T') \wedge t >> E(Y)) \tag{6.8}$$

In this state, we know that either S' or T' is the subverted node, or else that we have exceeded the expectation for the number of observation rounds and that there exists candidate modes for S' and T' which are pairwise adjacent.

Proposition 6.1 (Search Validity)
There is a x_k such that $\neg(k >> E(Y))$ and (it can be shown) $\exists(i,j) : \{V_i \in \mathcal{S}, V_j \in \mathcal{T} | S' \mapsto V_i, T' \mapsto V_j, (S'|T')\}$

The proof is supplied using the techniques from the extended π-calculus. There are several cases and a complete proof reduction would be burdensome without the aid of a model checker. However, we can outline such a proof by considering the action relations and an impact analysis on the names in the model.

PROOF Initially, we establish that action relations in the model conserve the direction of communication. Hence let V_i, V_j be pairwise adjacent nodes, we need to show for each i and j and message m that there is an action relation $V_i \overset{\bar{x}_j\langle m\rangle}{\rightarrow} V_i'$ and an action relation $V_j \overset{x_j(u)}{\rightarrow} V_j'$ and, w.l.o.g, provide a partial ordering for over x_i and x_j which shows that if $x_i \leq x_j$ then x_j follows x_i in the direction of communication. This is established by the routing map which the system operator owns or may be shown by induction over processes since all processes are regarded as networked peers. Without loss of generality, we assume that a node V_j is subverted in the following discussion and that $V_i \leq V_j \leq V_k$ in the direction of communication. We should also show that the encoding of $Mark()$,$Observe()$ and $NewKey()$ are feasible, which is trivial. Each case may be shown to apply both to a single route R and to a set of routes $\mathcal{R}$ by considering each $R_i \in \mathcal{R}$ separately, although, as discussed in section 6.2.4, there are potential economies in considering the subgraph $\mathcal{R}$.

Case 1 – Normal Operation Assuming a manipulation attack, we initiate the protocol and mark S and T. If $(S|T)$ are pairwise adjacent then we are done. Otherwise, for a node V_j we can show that for each message m received there is an source function $\uparrow_I (m)$ and a corresponding sink function with the operator $\downarrow_I (m)$ follows from the conservation of communicating action relations. The action relations $V_j \overset{\bar{x}_k\langle m_I\rangle}{\rightarrow} V_j'$ and $V_k \overset{x_k(m)_I}{\rightarrow} V_k'$ forward a message with the impact of loss of integrity while the corresponding action relations following the exercise of the $observe()$, function $V_i \overset{\bar{x}_j\langle w\rangle}{\rightarrow} V_i'$ and corresponding action relations in V_j and V_k, send a message w to the operator such that if we compare each m_I and its corresponding observational copy w, a mismatch is trivial to detect. Since the action relations preserve the order of communication $S < T$, we can safely mark all V_k such that $V_k < V_j$ as part of S in the case of a mismatch and otherwise mark all V_i such that $V_j > V_i$ part of T. We correspondingly mark the new position of S' or T' in x_{t+1} and $|V|$ is reduced accordingly. If $|V| = 0$ then we are done and $(S'|T')$ are again pairwise adjacent, else we continue as before.

Case 2 – Message Drop – As before, we assume the order of communication is preserved. Consider that there exists a $V_j' \in V$ which is a subverted node that drops (MP) all observed messages. This means that there is no action relation $V_j \overset{\bar{x}_k\langle w\rangle}{\rightarrow} V_j'$. Or that such a relation only exists for the process itself (an attempted deception!). An impact analysis shows that V_j is a source $\uparrow_A (w)$ for all messages $w \in W$ where W is observed prior to being routed to V_j. This shows that no V_{i-n} where n is a non-negative integer, which precedes it in order of communication can be marked as part of $\mathcal{T}$. However, when $k >> E(Y)$, it is clear that all nodes which precede the node V_i' in the direction of communication remain unobserved (no copies exist of these nodes on this route). Hence we can mark V_j part of $\mathcal{S}$ and the adjacent node V_i part of $\mathcal{T}$.

Case 3 – Message Diversion – Assume a subverted node V_j diverts packets (MV) to valid nodes on other routes after manipulating them. We also assume for the sake of argument that the subversion function can freely manipulate routing information in unencrypted messages. To maintain consistency in spoofing packet contents, it cannot forward them to its own legitimate successor nodes, otherwise the malformed packet may be observed. So, as in case 2, the corresponding action relations $V_j \overset{\bar{x}_k\langle m\rangle}{\rightarrow} V_j'$ and $V_j \overset{\bar{x}_k\langle w\rangle}{\rightarrow} V_j'$ do not exist. As well as recreating the situation for case 2, this also results in no node V_{i+l} where l is a non-negative number being marked as part of $\mathcal{S}$ producing an additional anomaly which is the mirror image of case 2. Alternatively, any re-routing of messages may break the operator's understanding of how system communication is routed, violating the indistinguishability assumption P.

Case 4 – Message Manipulation/Injection – This case does not require us to consider action relations. Spoofing or manipulating observed packets is prevented by the encryption technique. Packet vandalism is possible, but does not prevent successor nodes being observed forming the set $\mathcal{S}$. In fact, packet vandalism reveals the subverted process by breaking the indistinguishability assumption P.

Case 5 – Message Replay – This case does not require us to consider action relations. Even in the absence of a freshness protocol (which we have assumed since packets are marked based on the characteristics), the expiration on keys and packet identity will foil attempts at message replay. Since the subverted node has to replay messages from all nodes which precede it and possibly itself, we can place all nodes associated with replayed packets in $\mathcal{T}$ and proceed as before.

Case 6 - Message Delay – Considering the other capabilities in the formal model, delaying messages (MD) will slow the action of the protocol, but not halt it. A similar statement may be made about any denial of service attack. Such techniques may also be countered by increasing the probability p with which observations take place. A similar payoff between message injection by the adversary and selection of the probability value was explored in [96].

Case 7 - Replication – Obviously, the adversary may also replicate itself to other processes by REP. These increase the number of nodes to detect. However, replication does not prevent detection of subverted nodes. The other capabilities in the model are not relevant as attacks to undermine the protocol's action.

Case 8 - Preventing Initiation – it may be argued that the adversary can prevent process initiation by resetting message flags so that observation does not take place by message manipulation (MM). Again, this removes from the system any action relations forwarding w to the operator, creating the availability impact as before. This form of attack will not succeed because a failure of a chain of nodes to return copied packets on exceeding $E(Y)$ would reveal the initial node in the chain as the attacker, similar to case 3, upholding the security characteristic claimed for the property. Moreover, and, trivially, received messages where the observation mode flag is set to 0 instantly shows the presence of an adversary in the system, again, breaking the indistinguishabilty assumption P.

Hence, based on feasible attacks in the adversary capability model and considering the action relations and impact analysis (effects on names) of such attacks, we can show that the protocol is resilient in the face of an process subversion and message manipulation.

6.2.7 Space and Storage Requirements

An important advantage of probabilistic marking protocols is the constant space requirement for packet header size and that storage requirements on routers were not overburdened [109]. These advantages are retained for this protocol for recording packet identity , but with the overhead of marking routes and creating a new packet on each observation. However, the protocol is only activated when required, minimizing the associated cost. Further economies could be gained in space requirements by utilizing a knowledge of the network topology [115].

Online storage requirements, aside from space required for storing hash value key chains (see section 6.2.5), are assumed to be met by the operational control center capacity and not to affect network node storage. Storage requirements are anyway minimal set by the observation probability where X is the number of packets per route (or routing subgraph) required before an observation takes place and $E(X) = \frac{1}{p}$. Once an observation has taken place, unobserved packets can be discarded and observed packets archived for forensic purposes.

6.3 Co-ordinated Attacks, Anomaly Detection and Communication Integrity

We extend the work of previous sections by presenting an extended example, based on the co-ordinated attack the start of which we illustrated in chapter 5 and using the traceback protocol set out in section 6.2 which shows how we might maintain observability and controllability of a network under attack. This example also demonstrates how we can reason over the roles of agents and alliances of agents in attack and defense and is an example of an application of the goal transform calculus.

The attack represented is based on [47]. Three key valves in a control system when set to the values of *open*, *closed* and a *steady state* for a short period of time (measured in minutes) result in a critical health and safety failure at the plant. The original attack was conceived for an attacker acting externally on the system and it could be shown that the attack arbitrarily failed due to network physics, such jitter, traffic delay or message loss which are characteristic of a distributed system and make external control of such systems difficult.

Here we re-conceive the attack as a malicious software attack where the initial attacking processes which are *agents* in our conceptual model – chapter 5 – insert three hostile agents in processes in the ICS system. These agents act internally and are not affected by network physics. These agents set the valves to the appropriate values, while manipulating messages to confuse observers over the state of the system (i.e., a naïve observer would perceive the state of the valves as altering dynamically). The obfuscation strategy would be short lived and unsuccessful, if its intent was to conceal the fact of manipulation long term, but it is only meant to create confusion until the attack succeeds. The attack is necessarily co-ordinated to insure events occur in the correct order.

We subsequently show how three autonomous agents can act on behalf of the operator to calculate the true state of the system despite these obfuscations by utilizing the protocol we have already described – see section 6.2. We use the example as a further proof of the protocol's effectiveness by calculating linearizations of the attack and the detection using the goal transform calculus which we have proposed for reasoning about agent interactions. Obviously, a full defense would involve creating intervening software agents who could act autonomically to repair the system. However, this extension lies outside the scope of our current research. For simplicity, we also assume the observer agents are inviolate. An extended version of the protocol would introduce additional means of comparing observational readings (see chapters 7 and 8) which would underpin the validity of any observer views.

6.3.1 Co-ordinated Attack

We begin by formally defining the full set of agents involved in the co-ordinated attack - the three *Insertion* agents, the three *Attack* agents and the *System* agent which represents the ICS with which they interact. In the following, if the names of variables are not specifically restricted by the agent declaration, then they are assumed to be universal in scope. In this application of the protocol, it is implemented from the initiation of the system.

The system agent consists of the 15 network nodes which are a set of goals $\sum_{i=1}^{15} X_i$ (which can represent various objects such as servers, routers, switches and so forth) and 3 controllers – see equation 6.9. We define each channel $X_i \in X$ by by defining a *Send* goal on a per node basis. We take note at this stage that every node participates in the $Mark$ and $Observe$ goals which respectively mark messages with node address information and probabilistically take copies of messages for comparison with the originals, using appropriately defined functions. These are forwarded both to the operator and to a set of observer goals, whose action we will define subsequently.

$$System ::= \quad || \bullet \coprod_{i=1}^{15} X_i | \bullet \coprod_{j=1}^{3} C_j || \tag{6.9}$$

Each goal X_i may be defined in terms of equation 6.10

$$\begin{aligned} \coprod_{i=1}^{15} X_i \quad &:= \nu\, kc\, (x_i(z).\mathbf{0} \\ &+([(z \in C_i \vee z \in Op](Mark + [Rand() \leqslant p)]Observe) \\ &+Send_i \oplus \omega|!X_i) \end{aligned} \tag{6.10}$$

where $Mark := Mk(x_i, k, z) \supset z'$ and Mk is a function which adds a hash of the the channel identity to the current routing information. $Observe := Bv(z, k) \supset \nu \quad c$ is a function which probabilistically copies and forwards messages. $Rand()$ is a function which generates a random number in the range $(0 \ldots 1)$. p is the observation probability. ω is the subversion option. z is any message. c is a copy of any message. C_i is a set of messages from controller C_i. Op is a set of messages from the operator. So each network node X_i receives messages and probabilistically observes those sent by the operator or controller $Send_i$ is a goal which appropriately routes message for each X_i. A full listing is provided in equations 6.11 to 6.28.

$$\begin{aligned} Send_1 \ &:= \delta + \bar{x}_{Op}\langle z\rangle_{[Op]}.\bar{x}_{B_1}\langle z\rangle.\bar{x}_{B_2}\langle z\rangle\bar{x}_{B_3}\langle z\rangle.\mathbf{0} \\ &\oplus\bar{x}_4\langle z\rangle_{[C]}.\mathbf{0} \oplus \bar{x}_5\langle z\rangle_{[C]}.\mathbf{0} \oplus \ \bar{x}_9\langle z\rangle_{[X]}.\mathbf{0} \\ &+\bar{x}_{Op}\langle c\rangle.\bar{x}_{B_1}\langle c\rangle.\bar{x}_{B_2}\langle c\rangle\bar{x}_{B_3}\langle c\rangle.\mathbf{0} \end{aligned} \tag{6.11}$$

$$\begin{aligned} Send_2 \ &:= \delta + \bar{x}_{Op}\langle z\rangle._{[Op]}.\bar{x}_{B_1}\langle z\rangle.\bar{x}_{B_2}\langle z\rangle\bar{x}_{B_3}\langle z\rangle.\mathbf{0} \\ &\oplus\bar{x}_5\langle z\rangle_{[C]}.\mathbf{0} \oplus \bar{x}_6\langle z\rangle_{[C]}.\mathbf{0} \oplus \ \bar{x}_{10}\langle z\rangle_{[X]}.\mathbf{0} \\ &+\bar{x}_{Op}\langle c\rangle\bar{x}_{B_1}\langle c\rangle.\bar{x}_{B_2}\langle c\rangle.\bar{x}_{B_3}\langle c\rangle.\mathbf{0} \end{aligned} \tag{6.12}$$

$$\begin{aligned} Send_3 \ &:= \delta + \bar{x}_{Op}\langle z\rangle_{[Op]}.\bar{x}_{B_1}\langle z\rangle.\bar{x}_{B_2}\langle z\rangle\bar{x}_{B_3}\langle z\rangle.\mathbf{0} \\ &\oplus\bar{x}_6\langle z\rangle_{[C]}.\mathbf{0} \oplus \bar{x}_7\langle z\rangle_{[C]}.\mathbf{0} \oplus \ \bar{x}_9\langle z\rangle_{[X]}.\mathbf{0} \\ &+\bar{x}_{Op}\langle c\rangle\bar{x}_{B_1}\langle c\rangle.\bar{x}_{B_2}\langle c\rangle\bar{x}_{B_3}\langle c\rangle.\mathbf{0} \end{aligned} \tag{6.13}$$

$$\begin{aligned} Send_4 \ &:= \delta + \ \bar{x}_1\langle z\rangle_{[Op]}.\mathbf{0} \oplus \bar{x}_9\langle z\rangle_{[C]}.\mathbf{0} \\ &\oplus\bar{x}_{10}\langle z\rangle_{[X]}.\mathbf{0} + \bar{x}_1\langle c\rangle_{Op}.\mathbf{0} \end{aligned} \tag{6.14}$$

(6.15)

$$\begin{aligned} Send_5 \ &:= \delta + \bar{x}_1\langle z\rangle_{[Op]}.\mathbf{0} \oplus \bar{x}_2\langle z\rangle_{[Op]}.\mathbf{0} \oplus \\ &\bar{x}_8\langle z\rangle_{[C_1,C_2]}.\mathbf{0} \oplus \bar{x}_{10}\langle z\rangle_{[C]}.\mathbf{0} \oplus \bar{x}_9\langle z\rangle_{[X]}.\mathbf{0} \\ &+\bar{x}_1\langle c\rangle_{Op}.\mathbf{0} \end{aligned} \tag{6.16}$$

$$\begin{aligned} Send_6 \ &:= \delta + \bar{x}_3\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_2\langle z\rangle_{[Op]}.\mathbf{0} \oplus \bar{x}_3\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus\bar{x}_9\langle z\rangle_{[C_1,C_2]}.\mathbf{0} \oplus \bar{x}_{11}\langle z\rangle_{[C]}.\mathbf{0} \oplus \bar{x}_{10}\langle z\rangle_{[X]}.\mathbf{0} \end{aligned} \tag{6.17}$$

$$\begin{aligned} Send_7 \ &:= \delta + \bar{x}_3\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_3\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus\bar{x}_{10}\langle z\rangle_{[C]}.\mathbf{0} \oplus \bar{x}_9\langle z\rangle_{[X]}.\mathbf{0} \end{aligned} \tag{6.18}$$

$$\begin{aligned} Send_8 \ &:= \delta + \bar{x}_5\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_5\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus\bar{x}_{13}\langle z\rangle_{[C_1,C_2]}.\mathbf{0} \ \oplus \bar{x}_{10}\langle z\rangle_{[X]}.\mathbf{0} \end{aligned} \tag{6.19}$$

(6.20)

$$\begin{aligned} Send_9 \quad &:= \delta + \bar{x}_4\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_4\langle z\rangle_{[Op]}.\mathbf{0} \oplus \bar{x}_6\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus \bar{x}_{12}\langle z\rangle_{[C_1]}.\mathbf{0} \oplus \bar{x}_{14}\langle z\rangle_{[C_2,C_3]}.\mathbf{0} \\ &\oplus \sum_{i=1}^{15} \bar{x}_i\langle z\rangle_{[X_i]}.\mathbf{0} + \bar{x}_{Ad}\langle u\rangle_{[Ad]}.\mathbf{0} \end{aligned} \tag{6.21}$$

$$\begin{aligned} Send_{10} \quad &:= \delta + \bar{x}_5\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_5\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus \bar{x}_7\langle z\rangle_{[Op]}.\mathbf{0} \oplus \bar{x}_{11}\langle z\rangle_{[C_1,C_2]}.\mathbf{0} \\ &\oplus \bar{x}_{15}\langle z\rangle_{[C_3]}.\mathbf{0} \oplus \sum_{i=1}^{15} \bar{x}_i\langle z\rangle_{[X_i]}.\mathbf{0} \\ &+ \bar{x}_{Ad}\langle u\rangle_{[Ad]}.\mathbf{0} \end{aligned} \tag{6.22}$$

$$\begin{aligned} Send_{11} \quad &:= \delta + \bar{x}_6\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_6\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus \bar{x}_{14}\langle z\rangle_{[C_2,C_3]}.\mathbf{0} \oplus \ \bar{x}_9\langle z\rangle_{[X]}.\mathbf{0} \end{aligned} \tag{6.23}$$

$$\begin{aligned} Send_{12} \quad &:= \delta + \bar{x}_9\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_9\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus \bar{x}_{C_1}\langle z\rangle_{[C_1]}.\mathbf{0} \oplus \bar{x}_{10}\langle z\rangle_{[X]}.\mathbf{0} \end{aligned} \tag{6.24}$$

$$ \tag{6.25}$$

$$\begin{aligned} Send_{13} \quad &:= \delta + \bar{x}_8\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_8\langle z\rangle_{[Op]}.\mathbf{0} \oplus \bar{x}_{10}\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus \bar{x}_{C_1}\langle z\rangle_{[C_1]}.\mathbf{0} \oplus \bar{x}_{C_2}\langle z\rangle_{[C_2]}.\mathbf{0} \\ &\oplus \bar{x}_9\langle z\rangle_{[X]}.\mathbf{0} \end{aligned} \tag{6.26}$$

$$\begin{aligned} Send_{14} \quad &:= \delta + \bar{x}_{11}\langle c\rangle_{Op}.\mathbf{0} + \bar{x}_9\langle z\rangle_{[Op]}.\mathbf{0} \oplus \bar{x}_{11}\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus \bar{x}_{C_2}\langle z\rangle_{[C_2]}.\mathbf{0} \oplus \bar{x}_{C_3}\langle z\rangle_{[C_3]}.\mathbf{0} \\ &\oplus \bar{x}_{10}\langle z\rangle_{[X]}.\mathbf{0} \end{aligned} \tag{6.27}$$

$$\begin{aligned} Send_{15} \quad &:= \delta + \bar{x}_{10}\langle c\rangle_{[Op]}.\mathbf{0} + \ \bar{x}_{10}\langle z\rangle_{[Op]}.\mathbf{0} \\ &\oplus \bar{x}_{C_3}\langle z\rangle_{[C_3]}.\mathbf{0} \oplus \bar{x}_9\langle z\rangle_{[X]}.\mathbf{0} \end{aligned} \tag{6.28}$$

The controller goals are defined generically in equation 6.29.

$$C_i := (\nu y)\, x_{C_i}(z) + Control + Send|!C_i \tag{6.29}$$

where the sub-goal are defined as follows:

$$C_i.Control := C(\tilde{u}, z) \supset y', C_i VALVE' \tag{6.30}$$

$$C_1.Send := \bar{x}_{12}\langle y\rangle_{Op}.\mathbf{0} \oplus \bar{x}_{13}\langle y\rangle_{Op}.\mathbf{0} \tag{6.31}$$

$$C_2.Send := \bar{x}_{13}\langle y\rangle_{Op}.\mathbf{0} \oplus \bar{x}_{14}\langle y\rangle_{Op}.\mathbf{0} \tag{6.32}$$

$$C_3.Send := \bar{x}_{14}\langle y\rangle_{Op}.\mathbf{0} \oplus \bar{x}_{15}\langle y\rangle_{Op}.\mathbf{0} \tag{6.33}$$

For all C_i, $\tilde{u}$ is a vector of names holding the gain and setpoint parameters used by the control function. y is a vector holding the report state of a controller. C is the control function used by a controller to alter valve settings. z is any message.$C_i.VALVE$ are state variables for each control valve

The *Launch* agents have three tasks. The first two agents send malicious software to overwrite network nodes recruiting them for the adversary. The third agent waits until the first two are successful before launching its own attack.

We define the first agent $L1$

$$L1 ::= \nu \quad s, i, m, g \quad || \bullet \mathit{Stage1}|| \tag{6.34}$$

where

$$\mathit{Stage1} := s(\bot).\sum_{i=4}^{12}(\bar{x}_9\langle g_2\rangle_{X_i}.x_{S1}(s).\mathbf{0} + [s]G_0 + [i \geqslant 12]\mathit{Fail}) \tag{6.35}$$

and the second agent similarly by:

$$\begin{aligned} L2 \quad &::= \nu \quad s, j, m, g \quad || \bullet \mathit{Stage2}|| & (6.36)\\ \mathit{Stage2} \quad &:= s(\bot).\sum_{j=12}^{4}(\bar{x}_{10}\langle g_3\rangle_{Xj}.x_{S2}(s) \\ &\quad +[s].G_0 + [j \leqslant 4]\mathit{Fail}) & (6.37)\end{aligned}$$

The third agent represents a more complex goal progression in equations 6.38 to 6.42:

$$\begin{aligned} L3 \quad &::= \nu \quad a, k(0), l, n, s, m, g \quad || \bullet \mathit{InitialSuccess}|\mathit{Stage3}| \\ &\quad \mathit{DetectSuccess}|\mathit{UpdateAttack}|| & (6.38)\\ \mathit{InitialSuccess} \quad &:= x_{Ad}(u).\mathbf{0} + [u \in \mathit{Ack}]\mathit{UpdateAttack} + [a]\mathit{Stage3} \\ &\quad +[k \geqslant 2]G_0|!\mathit{InitialSuccess} & (6.39)\\ \mathit{Stage3} \quad &:= s(\bot).\sum_{l=4}^{12}(\bar{x}_9\langle g_1\rangle_{X_l}.s(\top).\mathbf{0} \\ &\quad +[s].\mathit{DetectSuccess} + [l \geqslant 12]G_0)) & (6.40)\\ \mathit{DetectSuccess} \quad &:= x_{Ad}(w).\mathbf{0} + [w \in X_l \wedge w]G_0|!\mathit{DetectSuccess} & (6.41)\\ \mathit{UpdateAttack} \quad &:= \mathit{Update}(u) \supset a'.(k++).\mathbf{0} & (6.42)\end{aligned}$$

For each agent:

$$\begin{aligned} \mathit{Fail} \quad &:= (\bar{x}_{Ad}\langle s\rangle.\mathbf{0}).G_0 & (6.43)\\ g_i \quad &:= (\bar{x}_{X_*}\langle m_i\rangle_{X_*} \oplus \delta).\mathbf{0} \oplus \phi & (6.44)\end{aligned}$$

and $\tilde{m}$ is the set of subverted messages sent by each g_i which are goals that temporarily rewrite X_9 and X_{10} to re-route the subverted messages to their final destinations. s is a boolean variable indicating success or failure of a subversion attempt.

i, j, k, l are integers. u, w are any message from the system. a is a boolean predicate which is defined as $(InPlace(Agent2, Agent3))$. $Update$ is a function which updates the boolean predicate. Ack is the set of message acknowledgments indicating a successful goal achievement during the attack. We use $k++$ as a shortened from of $Increment(k) \supset k'$ purely for convenience. δ indicates a potential delay in messaging and ϕ a failure condition.

We show some initial proof reductions over the action of the *System* agent. We need to show that each network node X_i responds appropriately to messages. We show this for node X_1 in equation set 7. The action of each node is chiefly differentiated by the $Send_i$ goal. The X_1, similar to X_2 and X_3, passes messages from the operator and routes them to controllers and also passes messages from the controllers to the operators, but it also passes messages to observer agents. Any messages intended for other network nodes are routed by X_9. The node also marks all messages between the operator and controllers and copies messages for comparison by the observers with the original.

PROOF

$||X_1.x_1.(z)||_{System}$

$\rightarrow$ Message received

$||X_1.Mark||_{System} \quad (z \in C_i \vee z \in Op)$

$\rightarrow$ $*$Mark the message with the node information

$||X_1.Observe|| \quad (Rand() \leqslant p)*$

$\rightarrow$ Copy it, depending on a random outcome

$||X_1.Send.\bar{x}_4\langle z\rangle_{[C]}||_{System}$

$\rightarrow$ Forward the message to various destinations

$||X_1.Send.\bar{x}_5\langle z\rangle_{[C]}||_{System}$

or

$\rightarrow ||X_1.Send.\bar{x}_9\langle z\rangle_{[X]}||_{System}$

or

$||X_1.Send.\bar{x}_{Op}\langle z\rangle_{[Op]}\bar{x}_{B_1}\langle z\rangle.\bar{x}_{B_2}\langle z\rangle.\bar{x}_{B_3}\langle z\rangle||_{System}$

$\rightarrow$ Send message to the observers

$||X_1.Send.\bar{x}_{B_1}\langle z\rangle.\bar{x}_{B_2}\langle z\rangle.\bar{x}_{B_3}\langle z\rangle||_{System}$

$\rightarrow$

$||X_1.Send.\bar{x}_{B_2}\langle z\rangle.\bar{x}_{B_3}\langle z\rangle||_{System}$

$\rightarrow$

$||X_1.Send.\bar{x}_{B_3}\langle z\rangle||_{System}$

$\rightarrow **$

or

$\rightarrow$

$||X_1.\bar{x}_{Op}\langle c\rangle.\bar{x}_{B_1}\langle c\rangle.\bar{x}_{B_2}\langle c\rangle\bar{x}_{B_3}\langle c\rangle||_{System} \quad \exists(c)$

$** \rightarrow$

$||X_1||_{System}$

$\rightarrow$ Or message delayed

$X_1.\delta$

We can now prove the action of the launch agents in more detail. It should be noted is that by explicitly naming channels we introduce potential constraints on adversary action – for example, if only one system node formed the initial attack surface, rather than two as in our example. This shows why the goal transform calculus requires a transform to the applied π-calculus during the course of any proof reduction.

Proof

$||Stage1.s(\bot)||_{L1}$

$||Stage2.s(\bot)||_{L2}$

$\rightarrow$ Initial state

$||Stage1.\bar{x}_9\langle g_2\rangle_{X_i}||_{L1} \quad (i = 4)$

$||Stage2.\bar{x}_{10}\langle g_3\rangle_{X_j}||_{L2} \quad (j = 12)$

$||X_9.x_9(z)||_{System}$

$||X_{10}.x_{10}(z)||_{System}$

$\rightarrow$ Send messages systematically until a successful subversion occurs

$||Stage1.x_{S1}(s)||_{L1}$

$||Stage2.x_{S2}(s)||_{L2}$

$||X_9.(\bar{x}_i\langle m_2\rangle_{X_i}.\bar{x}_{S1}\langle\top\rangle \oplus \phi.\bar{x}_{S1}\langle\bot\rangle_{S1})||_{Agent0}$

$||X_{10}.((\bar{x}_j\langle m_2\rangle_{X_j}.\bar{x}_{S2}\langle\top\rangle \oplus \phi.\bar{x}_{S2}\langle\bot\rangle_{S2})||_{Agent00}$

$||X_i.x_i(z)||_{System}$

$|X_j.x_j(z)||_{System}$

$\rightarrow$ Case 1 $*$ Both attempts succeed

$||Stage1.x_{S1}(s)||_{L1}$

$||Stage2.x_{S2}(s)||_{L2}$

$||X_9.\bar{x}_{S1}\langle\top\rangle_{S1}||_{Agent0}$

$|X_{10}.\bar{x}_{S2}\langle\top\rangle_{S2}||_{Agent00}$

$||X_i{'}|G||_{Agent2}$

$|X_j{'}|G||_{Agent3}$

$\rightarrow$ Case 2.1 X_i succeeds, while X_j fails and is re - attempted

$||Stage1.G_0||_{L1}$

$||X_i{'}|G||_{Agent2}$

$||Stage2.\bar{x}_{10}\langle g_3\rangle_{X_j}||_{L2} \quad (j = 11)$

$||X_{10}.x_{10}(z)||_{Agent00}$

$\rightarrow$ Case 2.2 X_j succeeds, while X_i fails and is re - attempted

$\rightarrow$ Case 3 Both attempts fail and are re - tried

$||Stage2.\bar{x}_9\langle g_3\rangle_{X_i}||_{L1} \quad (i = 5)$

$||Stage2.\bar{x}_{10}\langle g_3\rangle_{X_j}||_{L2} \quad (j = 11)$

$\rightarrow$ $(*)$ Both attempts succeed

Up to this point we show the success conditions. We complete the proof by considering the failure conditions.

$\rightarrow$ ($**$) One subversion fails
$||Stage2.\bar{x}_{10}\langle g_3\rangle_{X_j}||_{L2} \quad (j = 4)$
$||X_{10}.x_{10}(z)||_{System}$
$\rightarrow$
$||Stage2.x_{S2}(s)||_{L2}$
$||X_{10}.((\bar{x}_i\langle m_2\rangle_{X_i}.\bar{x}_{S2}\langle\top\rangle \oplus \phi.\bar{x}_{S2}\langle\bot\rangle_{S2})||_{Agent00}$
$\rightarrow$
$||Stage2.x_{S2}(s)||_{L2}$
$||X_{10}.\bar{x}_{S2}\langle\bot\rangle_{S2}||_{Agent00}$
$\rightarrow$
$||Stage2.Fail||$
$||X_{10}||_{System}$
$\rightarrow$
$||Stage2.Fail.s(\bot).\bar{x}_{Ad}\langle s\rangle.G_0||_{L2}$
...
$\rightarrow$ ($* * *$) Both attempts fail and the attack aborts

Signals to $L1$ from $L2, L3$ are negative while the acknowledgment signals from the agents $Agent2$, $Agent3$ are positive. We also need to show that no matter what order and what signals arrive that $L1$ responds correctly either by aborting or initiating the final phase of the attack. This leads to a requirement to show that the third launch agent can abort or proceed for several different cases of fail on launch or a successful outcome (from the point of view of the malicious agent).

Proof

$||InitialSuccess.x_{Ad}(u)||_{L3} \quad (k = 0)$
$\rightarrow$ (1) Both attacks succeed
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k + +)\ ||_{L3}$
$\ldots \rightarrow \quad (u = m_2, a = m_2 \wedge \neg m_3, k = 1)$
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k + +)\ ||_{L3}$
$\ldots \rightarrow \quad (u = m_3, a = m_2 \wedge m_3, k = 2)$
$||Stage3||_{L3}$
or $\rightarrow$ (2) $L3$ sends a negative flag though $Agent2$ succeeds
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k + +)\ ||_{L3}$
$\ldots \rightarrow \quad (u = m_2, a = m_2 \wedge \neg m_3, k = 1)$
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k + +)\ ||_{L3}$
$\ldots \rightarrow \quad (u = \neg g_3, a = m_2 \wedge \neg m_3, k = 2)$
$||G_0||_{L3}$
$\rightarrow$ (3) $L2$ and $L3$ send negative flags
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k + +)\ ||_{L3}$
$\ldots \rightarrow \quad (u = \neg g_2, a = \neg m_2 \wedge \neg m_3, k = 1)$
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k + +)\ ||_{L3}$
$\ldots \rightarrow \quad (u = \neg g_3, a = \neg m_2 \wedge \neg m_3, k = 2)$
$||G_0||_{L3}$
$\rightarrow$ (4)$Agent3$ succeeds and $L2$ fails
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k + +)\ ||_{L3}$
$\ldots \rightarrow \quad (u = \neg g_2, a = \neg m_2 \wedge \neg m_3, k = 1)$
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k + +)\ ||_{L3}$
$\ldots \rightarrow \quad (u = m_3, a = \neg m_2 \wedge m_3, k = 2)$
$||G_0||_{L3}$
$\rightarrow$ $(*)$ We can swap the order of arrival to get the same result

We now formally define the *Attack* agents which are the malicious software payloads inserted by the launch agents. It should be noted that while we effectively treat this as a single attack, in fact, it could also represent an co-operative attack from multiple different malicious sources, each of whom contributes one part of the attack. Since each agent also waits for specific events to initiate the next phase of the attack, this also means that the attack could be launched over a period of time. Only the final insertion of launch agent $L3$'s payload completes the attack initiation. The action of each agent is defined together with the subverted code which mutates the system goals to send a command to the controller and obfuscate messages to the operator reporting control state.

There are three agents. Agents 2 and 3 are launched first by launch agents $L2$ and $L3$ and wait for Agent 1 which is launched by $L3$ only when agents 2 and 3 are in place. The attack is completed when Agent 1 is installed and sets controller

C_2 to a *steady* value and signals Agent 2 who sets controller C_1 to *Open*, on success, signalling Agent 3 who sets controller C_3 to *Closed*, completing the attack and sending a *Success* flag back to $L3$. It should be noted that each of the agents overwrites a network node. The attack order is provable at goal level without resorting to transforming the π-calculus, assuming there is no defensive intervention at this point and the attack is only detected once it has commenced.

$Agent1$ is defined by the following equations:

$$\begin{aligned}
Agent1 &::= \| \bullet {X_k}' | \bullet SetSteady | \bullet Conceal \| && (6.45)\\
Conceal &:= (x_c(u).O(u) \supset u'.\bar{x}_c\langle u' \rangle_{Op} | !Conceal) && (6.46)\\
SendToC &:= \bar{x}_c\langle z' \rangle.x_c(z') && (6.47)\\
SetSteady &:= set(u, Steady) \supset u'.\bar{x}_s\langle u' \rangle_{C2}.FlagSteady && (6.48)\\
FlagSteady &:= x_s(u) + [u = Steady]\bar{x}_s\langle s \rangle_{X_5}.G_0 | !Flagsteady && (6.49)
\end{aligned}$$

along with its subverted version of the system agent,

$$\begin{aligned}
X_k' := \;& \nu\, \mathrm{x}_s x_c kc\, (\mathrm{x}_k(z)\\
&+[z \in C_i, z \in Op](Mark(x_i, z, k) + [Rand() \leqslant p)]Observe)\\
&+[z \in C_2](SendToC + \bar{x}_s\langle z \rangle) + x_s(z)\\
&+(Send \oplus \delta)).\mathbf{0}|!X_k) && (6.50)
\end{aligned}$$

where C_2 is defined as the set of messages which have the property that they originate from controller C_2, z is a message, s flags attack success to $Agent3$, $Steady$ is the desired value set by the attacker and O is a function which conceals the true signal from the operator and observers.

$Agent2$ is defined similarly except that it sets the valve to open, waits for this goal to succeed and subsequently signal the third

$$\begin{aligned}
Agent2 &::= \| \bullet {X_i}' | \bullet Ack2 | \bullet Conceal \| && (6.51)\\
Conceal &:= (x_c(s).O(s) \supset s'.\bar{x}_c\langle s' \rangle_{Op} | !Conceal) && (6.52)\\
SendToC &:= \bar{x}_c\langle z' \rangle.x_c(z') && (6.53)\\
Ack2 &:= \nu \quad v(\top) \quad (\bar{x}_s\langle v \rangle_{Ad}.Waitfor1) && (6.54)\\
Waitfor1 &:= (x_s(s) + [s = Steady]OpenValve | !Waitfor1) && (6.55)\\
OpenValve &:= \nu \quad w\ (set(w, Open) \supset w'.\bar{x}_s\langle w \rangle_{C_1}.FlagOpen && (6.56)\\
FlagOpen &:= \nu \quad w\ (x_s(s) + [s = Open]\ \bar{x}_s\langle u \rangle_{X_7}.G_0 | !FlagOpen) && (6.57)
\end{aligned}$$

where w is a real number, v is an acknowledgment that $Agent2$ is in place, u is a signal to $Agent3$ that $Agent2$ is successful and the system replacement goal is defined by:

$$\begin{aligned} X_i{}' \quad &:= \nu \quad x_s x_c k c \, (\mathrm{x}_i(z) \\ &+[z \in C_i, z \in Op] Mark(x_i, z, k) \\ &+[z \in C_2](\bar{x}_s\langle z\rangle + SendToC(z)) + x_s(z) \\ &+[(z' \in C_i \vee z \in Op) \wedge (Rand() \leqslant p)] Observe(z, k) \\ &+(Send \oplus \delta)).\mathbf{0}|!X_i) \qquad (6.58) \end{aligned}$$

where C_1, C_2 are sets of messages originating from those respective controllers and c is a copy of a message.

Finally, we define $Agent3$ which are, on receiving success messages from agents 1 and 2, launches the final part of the attack, setting the third valve to closed to create the critical health and safety condition which is the end goal of the attack. It should be noted that although we assume delays will disorder messaging, they will not normally be significant enough to prevent the attack from succeeding.

$$\begin{aligned} Agent3 \quad &::= || \bullet X_j{}'| \bullet Ack3| \bullet Conceal|| && (6.59) \\ Conceal \quad &:= x_c(u).O(u) \supset u'.x_c\langle u'\rangle_{Op}.0|!Conceal && (6.60) \\ SendToC \quad &:= \bar{x}_c\langle z'\rangle.x_c(z') && (6.61) \\ Ack3 \quad &:= \nu \quad v(\top) \quad (\bar{x}_s\langle v\rangle_{Ad}.Waitfor2) && (6.62) \\ Waitfor2 \quad &:= x_s(u) + [u \wedge FromAgent2(u)] CloseValve|!Waitfor2 && (6.63) \\ CloseValve \quad &:= \nu \quad w \quad (set(w, Closed) \supset w'.\bar{x}_s w'_{Op}.FlagSuccess) && (6.64) \\ FlagSuccess \quad &:= \nu \quad \mathrm{s} \quad x_s(u) + \\ &([u = Closed]\bar{x}_s\langle s\rangle_{Ad}.G_0)|!FlagSuccess) && (6.65) \end{aligned}$$

We also define the subverted system goal as before.

$$\begin{aligned} X_j{}' \quad &:= \nu \quad \mathrm{x}_c x_s k c \quad (\mathrm{x}_7(z) \\ &+[z \in C_i, z \in Op](Mark + [Rand() \leqslant p)] Observe) \\ &+[\mathrm{z} \in C_3](\bar{x}_s\langle z\rangle + SendToC(z)) + x_s(z) \\ &+[z \in Agent2]\bar{x}_s\langle z\rangle \\ &+(Send \oplus \delta)).\mathbf{0}|!X_j) \qquad (6.66) \end{aligned}$$

where C_2, C_3 are sets of messages, u, w are real variables, v an acknowledgment to the final launch agent that $Agent3$ is in place, s is a boolean variable set to $TRUE$

The proof of the attack at goal level follows.

PROOF

$$||X_k{}'|SetSteady|Conceal||_{Agent1}$$
$$||X_i|Waitfor1|Conceal||_{Agent2}$$
$$||X_j|Waitfor2|Conceal||_{Agent3}$$
$$\rightarrow$$
$$||X_k{}'|SetSteady.set(u, Steady) \supset u'||_{Agent1}$$
$$||X_i|Waitfor1||_{Agent2}$$
$$||X_j|Waitfor2||_{Agent3}$$
$$\rightarrow$$
$$||X_k{}'|SetSteady.\bar{x}_s\langle u'\rangle_{C2}||_{Agent1}$$
$$||X_i|Waitfor1||_{Agent2}$$
$$||X_j|Waitfor2||_{Agent3}$$
$$\rightarrow$$
$$||X_k{}'.Send|FlagSteady||_{Agent1}$$
$$||X_i|Waitfor1||_{Agent2}$$
$$||X_j|Waitfor2||_{Agent3}$$
$$\ldots \rightarrow$$
$$||X_k{}'|FlagSteady.x_s(u)||_{Agent1}$$
$$||X_i|Waitfor1||_{Agent2}$$
$$||X_j|Waitfor2||_{Agent3}$$
$$\rightarrow$$
$$||X_k{}'|FlagSteady.[u = Steady]\bar{x}_s\langle s\rangle_{X_5}.G_0||_{Agent1}$$
$$||X_i|Waitfor1||_{Agent2}$$
$$||X_j|Waitfor2||_{Agent3}$$
$$\rightarrow$$
$$||X_k{}'.Send(u)|G_0||_{Agent1}$$
$$||X_i|Waitfor1.x_s(s)||_{Agent2}$$
$$||X_j|Waitfor2||_{Agent3}$$
$$\ldots \rightarrow$$
$$||X_i|OpenValve||_{Agent2}$$
$$||X_j|Waitfor2||_{Agent3}$$
$$\rightarrow$$
$$||X_j|FlagSuccess||_{Agent3}$$
$$\rightarrow$$
$$||DetectSuccess||_{L3}$$

■

6.3.2 Attack Detection

We define the *Observer* agents and the associated set of state variables.We detect the attack using a distributed observation algorithm, made up from the actions of the network nodes in the $System$ agent in marking and copying messages and a set of $Observer$ agents who use this information to decide which routes to the controllers may be trusted and make a state determination over the control valves only over trusted routes. The $Observers$ also send an alert to the $Operator$ if the state is critical. The $Observer$ agents will store messages, compare copied messages to stored messages and use a graph to mark untrusted nodes where there is a discrepancy between the copied and stored message data. Subsequently, when determining state, any messages sent along routes with untrusted nodes are ignored.

The reduction of the observer agent falls into three parts -

1. Explaining the variant of distributed algorithm
2. Showing that the observers will block paths appropriately for 2 cases
   a) Malicious manipulation before observation
   b) Malicious manipulation after observation
3. Showing that a mobile agent will not cause the algorithm to function incorrectly

The operator employs observer agents to make a state determination over trusted routes, alerting on critical conditions. Each network node (including adversary agent nodes) through which a message passes will mark the route followed with its address. Each node may also probabilistically forward a message copy to *observer* agents for comparison.

Again, the capability to evaluate the environment and invoke goals provides a key additional capability in our thinking about security protocols. We show how the observer algorithm uses messages and copies to determine a trusted set of paths. State determination is restricted to considering messages which arrive by trusted paths. An observer is defined in equation 6.67 through equation 6.72. We show the initial reduction of the $Observer$ in equations 11 and 12.

$$\nu \quad \tilde{k}\tilde{c}, STORE, STATE, CRITICAL, C_iTREE \quad || \bullet \mathit{Observe}|| \tag{6.67}$$

$$\begin{aligned}
Observe &:= x_{Ob_j}(z) + [z \in M]UpdateState + [z \in C]UpdatePath && (6.68)\\
UpdateState &:= \nu \quad p \quad (Store(z, STORE) \supset STORE' \\
&+\sum_i [z \in C_i \wedge \neg(Marked(z.path)]Store(z, STATE) \supset STATE' \\
&+([p \leqslant rand()]EvaluateState \oplus Observe) && (6.69)\\
EvaluateState &:= \quad (Evaluate(STATE, \tilde{c}) \supset CRITICAL' \\
&+[CRITICAL]Alert) + Observe && (6.70)\\
UpdatePath &:= \quad Compare(u, z, \tilde{k}, STORE) \supset w. \\
&\sum_i [\neg w]MarkPath(u, z, C_iTREE) \supset C_iTREE' \\
&+\sum_i [w \wedge Marked(z.path)]UnMarkPath(u, z, C_iTREE) \supset C_iTREE' \\
&+Observe && (6.71)\\
Alert &:= \nu \quad f(\perp) \quad (\bar{x}_{Op}\langle f\rangle_{Op}) && (6.72)
\end{aligned}$$

The first case is a copied message used to determine route trustworthiness. The observer receives a message which it evaluates to be a copy and invokes the goals $UpdatePath$ which compares the message with the original. If no discrepancy is found, it moves to the next message. If a discrepancy is found then it notes the route and marks the forward neighboring node in order of communication as untrusted. It can also remove marks – for example, taking account of interventions which might return a node to a trusted state. We can represent marked messages using a graph, defined algebraically for the purposes of the proof using $\vee$ to represent a choice of routes and $\wedge$ to represent a sequence of communication nodes on a single route. For example, equation 6.73 shows that X_8 is no longer a trusted node by placing a bar over the node.

Proof

$|X_i.Send|| \quad (i = 1, 2, 3)$

$||Observe.x_{ob_j}(z)||_{Observer_j} \quad (z \in C)$

$\rightarrow$

$||UpdatePath.Compare(u, z, \tilde{k}, STORE) \supset w||_{Observer_j}$

$\rightarrow$ No discrepancy, message on unmarked path

$||Observe||_{Observer_j}$

$\rightarrow$ No discrepancy, message on previously marked path

$||UpdatePath.UnMarkPath(u, z, C_iTREE) \supset C_iTREE'||_{Observer_j}$

$\{w = TRUE \wedge Marked(z.path)\}$

or $\rightarrow$ Discrepancy found

$||UpdatePath.MarkPath(u, z, C_iTREE) \supset C_iTREE'||_{Observer_j} \quad (w \neq TRUE)$

The proof that the path marking algorithm will respond correctly depending on whether the message is marked before or after being manipulated follows. If the message is copied before it is manipulated, then the malicious agent node will appear to deliver trustworthy messages, but any subsequent node it sends the message to will appear to be untrustworthy. Hence we always mark the next node up in order of communication to the node transmitting the copy. Alternatively, any previous node will appear to deliver a trustworthy copy and we will bar the agent node. So either any node the agent node sends to will be marked as untrustworthy or the agent node will be marked as untrustworthy. In either case, any message traveling by the agent node will not be trusted for state determination.

$$\begin{aligned} C_1TREE \quad &= Op \vee (X_1 \vee (X_4 \wedge X_9 \wedge X_{12} \wedge C_1) \\ &\vee (X_5 \vee (\bar{X}_8 \vee X_{10}) \wedge X_{13} \wedge C_1)) \\ &\vee (X_2 \vee (X_5 \vee (\bar{X}_8 \vee X_{10}) \wedge X_{13} \wedge C_1) \\ &\vee (X_6 \wedge X_9 \wedge X_{12} \wedge C_1)) \\ &\vee (X_3 \vee (X_6 \wedge X_9 \wedge X_{12} \wedge C_1) \\ &\vee (X_7 \wedge X_{10} \wedge X_{13} \wedge C_1))) \end{aligned} \tag{6.73}$$

PROOF

$|X_i.Send||\quad (i = 1, 2, 3)$

$||Observe.x_{ob_j}(z)||_{Observer_j}\quad (z \in M)$

$\rightarrow$

$||UpdateState.Store(z, STORE) \supset STORE'||_{Observer_j}$

$\rightarrow$ Only if message is trusted

$||Store(z, STATE)||_{Observer_j}$

$\rightarrow \quad (p > rand())$

$||Observe||_{Observer_j}$

$\rightarrow\ (p \leqslant rand())$

$||EvaluateState.Evaluate(STATE, \tilde{c}) \supset CRITICAL'||_{Observer_j}$

$\rightarrow$ Not in a critical state

$||Observe||_{Observer_j}$

$\rightarrow$ State is critical

The second case is an normal message and, depending on probability, a snapshot of state. The observer receives a message which is not a copy. It stores the message in $STORE$ which is use to log all messages. But only if the message arrives on a trusted does it store the message in $STATE$ which is the set of messages used to make a determination over the state of the system. Finally, if the state is detected to be critical (as in our attack) the operator is signaled by the $Alert$ goal.

We include the pseudo-code of the functions for completeness. These are feasible encodings rather than necessarily the most efficient algorithms.

Algorithm 6.3: Compare

```
1
              $Compare(, u, z, \tilde{k}, STORE)$

                $Decrypt(z, \tilde{k})$
                $FetchOriginalMessage(u, z)$
                $CompareMessages(u, z) \supset w$ // Flag discrepancies
                $Return \quad w$
```

Algorithm 6.4: Path Marking

```
1

  $MarkPath(u, z, C_iTREE)$

  $Var$
    $path, node, i$

    $path = z.path - u.path$
    $i = z.message.origin$
    $node = z.path.lastnode$

   // Search the C$_iTREE$ and mark each route with the untrusted node
  // Don't complete this process if an already marked node is encountered en route

    $MarkNode(node, C_iTREE)$
    $Return \quad C_iTREE$
```

Algorithm 6.5: Unmark Paths

```
1
    $UnMarkPath(u, z, C_iTREE)$

    $Var$
      $path, node, i$

      $path = z.path - u.path$
      $path = path - path.lastnode$

    // We shorten the path by one node to avoid removing marks

      $i = z.message.origin$

    // Remove all marks up to the end of the path

      $UnMark(path, C_iTREE)$
      $Return \quad C_iTREE$
```

The final aspect to examine is how this approach allows us to consider behavioral aspects, that is, proofs over dynamic alterations to the system. Here we use the simplified example of an agent node transmitting itself to another node and deleting itself on the first node. The example is a little contrived, but intended to demonstrate that the observer algorithm will function in a dynamic situation.

$$\begin{aligned} C_1TREE = \quad & Op \vee (X_1 \vee (X_4 \wedge X_9 \wedge X_{12} \wedge C_1) \\ & \vee(X_5 \vee (\bar{X}_8 \vee X_{10}) \wedge X_{13} \wedge C_1)) \\ & \vee(X_2 \vee (X_5 \vee (\bar{X}_8 \vee X_{10}) \wedge X_{13} \wedge C_1) \\ & \vee(X_6 \wedge X_9 \wedge X_{12} \wedge C_1)) \\ & \vee(X_3 \vee (X_6 \wedge X_9 \wedge X_{12} \wedge C_1) \\ & \vee(X_7 \wedge X_{10} \wedge X_{13} \wedge C_1))) \qquad (6.74) \\ \rightarrow & \\ C_1TREE = \quad & Op \vee (X_1 \vee (X_4 \wedge X_9 \wedge X_{12} \wedge C_1) \\ & \vee(X_5 \vee (X_8 \vee X_{10}) \wedge X_{13} \wedge C_1)) \\ & \vee(X_2 \vee (X_5 \vee (X_8 \vee X_{10}) \wedge X_{13} \wedge C_1) \\ & \vee(\bar{X}_6 \wedge X_9 \wedge X_{12} \wedge C_1)) \\ & \vee(X_3 \vee (\bar{X}_6 \wedge X_9 \wedge X_{12} \wedge C_1) \\ & \vee(X_7 \wedge X_{10} \wedge X_{13} \wedge C_1))) \qquad (6.75) \end{aligned}$$

Consider the graph transformation defined algorithmically in the equations 6.74 and 6.75. Various cases may arise, depending on ordering. The value of our approach is to consider these various cases and the number of discrete steps which may occur to establish them and the resulting (probabilistic) effect on state determination. The same results cannot be obtained purely from calculating the expectation of the number of discrete steps to establish the various cases.

PROOF

$$\begin{aligned}
&\coprod_i ||Observer||_{Ob_i}\\
&||C_1.\bar{x}_{13}(y)x_{Op}|X_{13}.x_{13}(z)||_{System}\\
&\rightarrow\\
&||X_{10}.Mark.\bar{x}\langle y\rangle_{Op}|X_{10}.x_{10}(z)|C_1.\bar{x}_{12}\langle z\rangle_{Op}|X_{12}.x_{12}(z)||_{System}\\
&\rightarrow\\
&||X_{10}.Mark.\bar{x}_8\langle y\rangle_{Op}|X_{12}.Mark.\bar{x}_9\langle y\rangle_{Op}|X_9.Mark.x_9(z)|C_1.\bar{x}_{13}\langle y\rangle_{Op}|X_{13}.x_{13}(z)||_{System}\\
&||X_8{}'.x_8(z)||_{Agent2}\\
&\rightarrow\\
&||X_8{}'.Mark.\bar{x}_5\langle y\rangle_{Op}||_{Agent2}\\
&||X_{13}.Mark.\bar{x}_{10}\langle y\rangle_{Op}|X_9.Mark.Observe(y)|C_1.\bar{x}_{12}\langle z\rangle_{Op}|X_{12}.x_{12}(z)|X_{10}.x_{10}(z)|X_5.x_5(z)||_{System}\\
&\rightarrow\\
&||X_{10}.Mark.\bar{x}_7\langle y\rangle_{Op}|X_5.Mark.\bar{x}_2\langle y\rangle_{Op}|X_{12}.Mark.\bar{x}_9\langle y\rangle_{Op}|X_9.Mark.\bar{x}_6\langle y\rangle_{Op}|C_1.\bar{x}_{13}\langle y\rangle||_{System}\\
&\rightarrow \text{ Case (1) } X_8 \text{ and } X_6 \text{ are marked}\\
&\coprod_i ||UpdatePath.MarkPath(c, y, C_1TREE)||_{Ob_i}\\
&\rightarrow \text{ Case (2) Neither node is marked}\\
&\rightarrow \text{ Case (3) } X_6 \text{ is marked but not } X_8
\end{aligned} \tag{6.76}$$

■

6.4 Summary

We have used this chapter as an extended example of the methods and techniques developed in chapter 5, showing how we can incorporate both the adversary model and impact analysis into the π-calculus to provide us with a demonstrate of the validity of a security protocol. The protocol is itself of interest as an example of the application of traceback techniques to a network where both the source and origin of traffic are already known and the network is well-defined. This brings considerable advantages in terms of reducing protocol complexity to the operator and software agents working on his behalf. We illustrate how the protocol may be used not just to locate adversary processes but also to enable continuing accurate state determination over a system which has been subverted.

Chapter 7

Expected Behavior and Multithreaded Observation Mechanisms In Multiprocessor Systems

The mathematical probability of a common cat doing exactly as it pleases is the one scientific absolute in the world.

Lynn M. Osband.

7.1 Preamble

Clearly, it would be desirable to avoid the recruitment of network nodes or control units by malicious adversaries in the first place. In this chapter, we set out an approach to the early detection of the subversion of network hosts by malicious software which takes advantage of the nature of multiprocessor operating systems which are now ubiquitous in computer systems. In such systems, memory management introduces similar dis-orderings of process and communication to those present in distributed systems at network level. Multiprocessor scheduling permits observers associated with different CPUs to observe the action of other processes in the system concurrently with that action and allows us to detect probabilistically alterations in state of key processes, for example, security subsystems. These stateful observations may then be compared with a model of expected behavior, which we specify using a variant of CCS (along with an associated approach using state automata). We begin by defining the how the model is specified, since it captures novel features in relation to understanding causation in such systems, and go on to provide a description of the observation mechanism and a "proof of concept" implementation of the method.

7.2 Expected Behavior Model

In this section, we set out our approach to modeling the expected behavior of a distributed computation. This model abstracts from the detail of implementation, so, for example, it is irrelevant whether we are treating, as for our "proof of concept", a concurrent system as a distributed system or dealing with a network system. The approach extends to both.

As we have stated in appendix B, a distributed system consists of set P of N processes, $P = \{P_1, P_2, \ldots, P_N\}$. Each process P_i , where $1 \leq i \leq N$ passes through a finite sequence m of local states $(s_1, s_2, \ldots, s_3)$ where $m \geq 1$.

Let S_i be the sequence of local states in P_i. Then S can be defined as

$$S = \coprod_i S_i$$

A distributed computation trace can be modeled as a tuple $(S_1, S_2, \ldots, S_N, \leadsto)$, where $\leadsto$ indicates a state in S_i logically preceding a state in S_j $(1 \leq i \leq j \leq N)$. This structure forms a decomposed partially ordered set (deposet).

We now apply this approach to a subset of elements in S. Let $X \subseteq S$ be a selected set of data elements in a distributed computation. For our purposes, the elements of X are selected on the basis that they behave consistently (i.e. are invariant) under normal operating conditions, but behave arbitrarily under abnormal conditions such as the incursion of malicious software [1]. We term the set X, the set of *semantic chokepoints* within the application. The implication is, not simply that these elements might behave differently under subversion, but that they *must* do so – although, we can accept high probability cases as well.

Each element $x \in X$ passes through a sequence of states S_x during a run of a distributed computation. Let

$$S_X = \coprod_{x \in X} S_x$$

It should be clear that S_X also forms a deposet. It is not necessary that and element $s_x \in S_X$ be a complete ordering in relation to S. For example, it could be a function which is repeatedly invoked and operates over a shorter period of time, which can be designated a *service period*[53].

We may model the expected behavior of P in terms of the expected behavior of X. X is an abstract representation of P. That is, since each data element $x \in X$ is associated with a process $P_i \in P$, there exists a function ϕ which such that $\phi : S_i \mapsto S_x$. This mapping is surjective since for each state in S_x there exists, at least, one state transition of S_i (i.e., itself). ϕ is, therefore, a mapping between the global states of S and the states S_X. Hence the state of S_X may be used to indicate the possible states of S_i. Clearly, the model of behavior derived will be less finely grained than if we had direct access to the actual states of P, but we have the advantage of reduced complexity. Moreover, the model proposed allows us to consider behavior more directly relevant to the purposes of anomaly detection. Thus, if we record the states of S_X we may feasibly compare the resulting set of global sequences (or parts thereof) with an equivalent set of predicted sequences and potentially use our findings to uncover the presence of malicious software.

We also declare a probability distribution function over the states of $\phi : P_X \mapsto P$. This approach is initially useful for determining if behavior matches expectation. It also enables us to abstract further from the action of P. For example, we might deal with states which relate to the program environment rather than the data states of P [101].

[1]Clearly, the selection criteria should be altered for different applications.

7.2.1 Modelling Approach

We show how the behavior of P_X may be modeled both as a graph and as a process algebraic statement. The two representations are isomorphic. Although, for reasons of economy, the algebraic approach is preferred for specifying complex distributed systems. The graph-theoretical approach is useful for illustrating action at particular points.

7.2.1.1 Expected Behavior Diagram

We may specify the legal states and permitted transitions of an $x \in X$, denoted $x_i \to x_j$, using a modified state diagram, which we call an *expected behavior graph*. The primary modifications are to eschew the use of loops and to label transitions with probability values [2]. For convenience, we may choose to represent replication by an asterisk.We provide an example in figure 7.2.1.1.

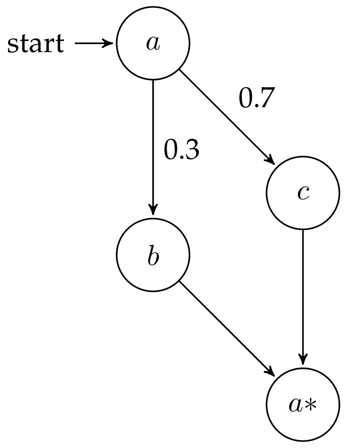


Figure 7.1: Expected Behavior Graph Showing Legal States and Transitions of x

This graph is useful as it enables us not only to distinguish between legal and illegal behavior, but also to identify low probability behavior which may be indicative of anomalous conditions. In effect, the specification serves not only as a description of the probability space Ω of the behavior of x, but also as a "fuzzy" partial order for the relation "preferred successor state" which expresses what behaviors are more likely that others to belong to a set of normal behaviors for the computation. This approach lends itself to a statistical analysis of the computational traces, although our primary intent is to achieve an early warning of malicious behavior by more granular analysis of causality. Nonetheless, there exist occasions where the former technique is more useful than the latter, such as the unlikely high repetition of error messages for a given function [9].

Clearly, it is infeasible for the graph to express a complete set of global sequences (or even one), which might occur during an extended run of the computation. Rather, borrowing from order theory, we designate the initial condition as the *sup* and the terminal state as the *inf*. We assume the *sup* is unique, but, clearly, in some cases, an *inf* may not exist and we deal with set of minimal states. To deal

[2]Similar to the concept of "likely" and "unlikely" in Linux kernel programs used to deal with error conditions.

with replication, we may also term any state from which process behavior replicates a *sub-maximal* and its immediate predecessors *sub-minimals*. A set of global sequences may be modeled by chaining these bounded sequences, gluing a subminimal and a sub-maximal element together. In effect, the sup and inf define a *service period*.

We may extend this approach to specify the behavior of more than one data element concurrently, showing each data element as distinct components of the same expected behavior graph. This approach is well-defined where causal relations exist between the states of different elements. We formally define three such relations – *conditional dependency* , *strong causal dependency* and *weak causal dependency*. These relations may be used to specify where the relation "causes" ($\rightarrow_c$) occurs, with some probability, in our model, allowing us to induce a partial order over the predicted states of S_X, and hence deduce a set of corresponding global sequences, or rather parts thereof, which is consistent with the "happened before" relation [46]. This differs from graphical representations of a deposet in distributed systems – see B – because we claim, from state analysis, a more complete knowledge of cause and effect in the program. Therefore, the relation is more granular than the "potentially causes" ($\rightarrow_p$) relation – but correspondingly adds complexity to our analysis. So we balance the increase complexity of understanding causality at a granular level against the reduction in state space provided by only considering the set X.

Where a data element $x \in X$ may only achieve a given state, say x_j, concurrent with a state of another element $y \in X$,say y_k, x_i is said to be conditionally dependent on y_k.

Definition 7.1 (Conditional Dependency)
Let $x, y \in X$ be data elements. Let $x_i, x_j \in S_x$ be states of x and $y_k \in S_y$ be a state of y respectively. If x_i may not transition to x_j unless y is in y_k, we say that x_j is conditionally dependent *on y_k.*

We show this relation as a directed edge, which is dashed, on our expected behavior graph from y_k to x_j. y_k is called the *permitting node* of x_j. x_j is called the *dependent node* of y_k. A joint dependency may also exist, shown by linking co-permitting nodes with an undirected dashed edge. This is called a *permitting component*. Clearly, a permitting node is also a permitting component.

Two other forms of dependency may also exist where a transition in one component mandates a transition in the other. We graph this by showing a graph with both the "master" and "slave" components and directing an edge from the "master" component to the "slave" component. We use colors to distinguish nodes.

Definition 7.2 (Strong Causal Dependency)
Let $x_i, x_j \in S_x$ and $y_k \in S_y$ be states of $x, y \in X$. If the transition to y_k forces x_i to transition to x_j, and the transition to x_j only occurs due to y_k, we say that y_k strongly causes x_j, and we call this a strong causal dependency.

The definition for *weak causal dependency* is similar, but removes the condition of uniqueness.

Definition 7.3 (Weak Causal Dependency)
Let $x_i, x_j \in S_x$ and $y_k \in S_y$ be states of $x, y \in X$. If the transition to y_k forces x_i to transition to x_j, and the transition to x_j may also occur as a result of other conditions, we

say that y_k weakly causes x_j, and we call this a weak causal dependency*. This definition extends to unique nodes whose forcing action has a probability $p < 1$ – that is other factors in causation are unlisted.*

Figure 7.2.1.1 provides an example representation.

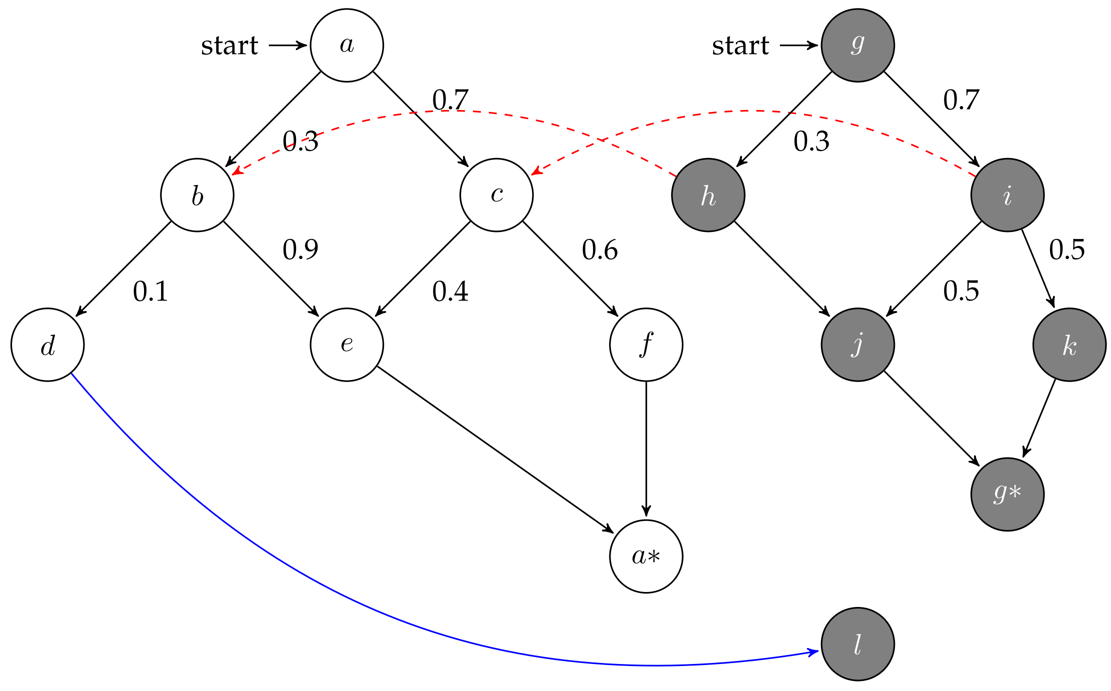


Figure 7.2: Expected Behavior Graph with Dependencies

We are capable of inducing a linearization over a predicted set of states which when combined will allow us to model the possible global sequences of S_X. We denote this the *predictor set* R_X. This will consist of a set of tuples $(R_x, R_y, ..., \rightsquigarrow_p)$ – where $\rightsquigarrow_c$ indicates the relation (remotely) "causes", which is consistent with the "happened before" relation [46]. This relation is defined stochastically in this case.

For example. Let $(\ldots)$ indicate an ordered, bounded n-tuple of states. Let $[\ldots]$ indicate a permutation of states. Let $\langle(\ldots), (\omega_i)\rangle$ be an ordered pair which indicates the probability of an ordered tuple. Let $\perp$ indicate process termination. Let $*$ indicate a replication. Let $X = \{red, blue\}$ be two distinct data elements with causal relations between their respective states.

From figure 7.2.1.1, using $+$ to signify *or*, the set of expected orders of red and $blue$ [3] are –

$$\begin{aligned} red &= \{\langle(a, b, d, \perp), (0.03)\rangle, \\ &+ \langle(a, b, e, a*), (0.27)\rangle, \\ &+ \langle(a, c, e, a*), (0.28)\rangle, \\ &+ \langle(a, c, f, a*), (0.42)\rangle\} \end{aligned}$$

[3]shown as white and gray respectively

$$\begin{aligned} blue &= \{\langle(g,h,j,g*),(0.3)\rangle, \\ &+ \langle(g,i,j,g*),(0.35)\rangle, \\ &+ \langle(g,i,k,g*),(0.35)\rangle\} \end{aligned}$$

while the set of partial orders for X, based on the causal relations which exist between these sets of sequences, consists of

$$\begin{aligned} R_X &= \{X_1, X_2, \ldots, X_6\} \\ &= \{\langle([ag],h,b,d,l,\perp),(0.03)\rangle, \\ &+ \langle([ag],h,b,[ej],*),(0.27)\rangle, \\ &+ \langle([ag],i,c,[ej],*),(0.14)\rangle, \\ &+ \langle([ag],i,c,[ek],*),(0.14)\rangle, \\ &+ \langle([ag],i,c,[fj],*),(0.21)\rangle, \\ &+ \langle([ag],i,c,[fk],*),(0.21)\rangle\}. \end{aligned}$$

Taking the element of R_{X_2} in the order shown as an example, using the interleaving assumption, the set of linearized sequences is given by equation 7.1.

$$R_{X_1} = \{(a,g,h,b,e,j,*)+(g,a,h,b,e,j,*)+(a,g,h,b,j,e,*)+(g,a,h,b,j,e,*)_{0.675}\} \tag{7.1}$$

each of which may occur with $p = 0.675$. The boundary states are the sub-maximals a and g and the sub-minimals d and l, including the inf and sup respectively. The information provided is sufficient to model a global sequence.

7.2.2 Stochastic Casual Tracing Algebra

The approach shown in section 7.2.1.1 is consistent with the representation of systems as process diagrams. Accordingly, it also lends itself to more economical representation using a process algebra – as we have anticipated in the creation of the predictor set in section 7.2.1.1. We provide an axiomization of this algebra, which is variant of CCS, in this section which allows us to define the predictor set without the necessity of drawing a graph. We may subsequently use the model specified by our algebra to reason about the expected states of processes. In fact, we may go further and define a set of unacceptable linearizations as a language over finite words,i.e, state vectors whose occurrence is detectable as a reject condition by *monitors* – whose existence we assume – which are formally or informally defined processes [112]. In some cases, these words may be up to an identifiable semantic "fingerprint" or "characteristic vector" for layered detection mechanisms. In addition, using probabilities, we may give a quantitative as well as a quantitative verdict w.r.t the "safety" of OS states [4]. This is key because we not only identify invalid states, but also may detect where a set of superficially legitimate linearizations of state conceal an improbable set of system operations exampled by

[4]Informally, "safety" means that bad things do not happen during a program and "liveness' implies that good things do happen.

$x_p + y_q = y_q + x_p$	A1
$(x_p + y_q) + z_r = x_p + (y_q + z_r)$	A2
$(x_p + y_q)z_r = x_p z_r + y_q z_r$	A3
$z_r(x_p + y_q) = z_r x_p + z_r y_q$	A4
$(x_p y_q)z_r = x_p(y_q z_r)$	A5
$x_p y_q = xy_{pq}$	P1
$x_p + y_q = (x + y)_{p+q}$	P2
$(\neg x)_p = x_{(1-p)} + (\neg x)_p$	P3
$x_p \vert y_q = (xy + yx)_{pq}$	PC1
$\bar{x}_p \vert [x]y_q = (xy)_{(pq)}$	PC2
$x_p + \bar{y}_q \vert \neg z_s + [y]z_r = (x\neg z + \neg z x)_{(ps)'}$ $+ (xz + zx)_{(p(1-s))'} + yz_{(qr)'} + y\neg z_{(q(1-r))'}$	PC3

Table 7.1: Axiomization of $SCTA$

[9] which result from malicious software obfuscating its presence using apparently legitimate system responses. This approach is related to semantics-based detection of malware [101], but allows the approach to be generalized to cover unexpected behavior, which may not be immediately identifiable as an obfuscated malicious software attack, but may be indicative of one (or, at least, of some systemic error which should cause equal concern). This allows us not only to directly detect malicious behavior , but also uncover side effects of such behavior.

7.2.2.1 Equational Specification

We begin immediately with the equational specification of our algebra $SCTA$ (Stochastic Causality Tracing Algebra): $SCTA = (\sum_{SCTA}, E_{SCTA})$. $\sum_{SCTA}$ has three binary operators, $+, \cdot, |$ and three unary operators, $\neg, \bar{\cdot}$ and $[\cdot]$. It also has a set of constants A denoted by $a_p, b_q, c_r, \ldots$, where p, q, r indicate variable probabilities. The set of constants is denoted by A. A is not sized and may be considered a parameter of our theory.

E_{SCTA} consists of the equations shown in Table 7.1. We use the following notational conventions:

1. The operator $\cdot$ may be omitted. Thus, xy means $x \cdot y$.
2. Many brackets are omitted for legibility, and we follow operator precedence conventions; in particular, $\cdot$ binds stronger than $+$, and $+$ binds stronger than $|$.

The equations contain variables $x_p, y_q, z_r, \ldots$ which are assumed to be universally quantified. If $\mathcal{M}$ is a model for $SCTA$ then the elements of its domain are called *stochastic states*. So the variables stand for states in some arbitrary model of $SCTA$.

$\cdot$ is product or **alternative composition**; $x_p \cdot y_q$ means that state x occurs with probability p then state y occurs with probability q. In general, we may omit these probabilities if they are equal to 1.

$+$ is sum or **alternative composition**; $x_p + y_q$ means that either state x occurs with probability p or state y occurs with probability q.

$|$ is **concurrent composition**; using the *interleaving assumption* [46] either x or y may occur first. Note that, in probability terms, either ordering is regarded as the same event by $A3$

$\neg$ is **negativity** and is used to indicate the non-occurrence of a term, i.e., either x_p occurred or $\neg x_{(1-p)}$ occurred. Negative expressions may be avoided by a label substitution σ or using a "skip" operator, i.e., $A := A \cup \epsilon$ for null events.

$\bar{\cdot}$ **cause** indicates that the causal information in a state has been made available for processing.

$[\cdot]$ **effect** indicates that causal information about another state may be processed and determines whether the term right adjacent to it may occur.

The use of $'$ with a probability value indicates the resulting process equation may require to be normalized w.r.t probability using a Bayesian re-formulation.

We provide the specification $SCTA$ with the following intuitive meaning. The sets of constants with their variable probabilities are the probable states of selected data elements under invariance:

- A1 (the **commutativity of** $+$) says that a choice between x_p and y_q is the same as a choice between y_q and x_p;
- A2 (the **associativity of** $+$) says that a choice between x_p and choosing between y_q and z_r is the same as a choice of z_r and choosing between x_p and y_q;
- A3 (the **right distributivity of** $\cdot$ **over** $+$) says that a choice between x_p and y_q , followed by z_r is the same as a choice between x_p followed by z_r and y_q followed by z_r;
- A4 (the **left distributivity of** $\cdot$ **over** $+$) says that z_r followed by a choice between x_p and y_q is the same as a choice between z_r followed by x_p and z_r followed by y_q;
- A5 (the **associativity of** $\cdot$) should be evident;
- P1 the **stochastic product** of sequentially composed states;
- P2 (the **sum of conjoint events**) says that the probability of the alternate composition of two states is the sum of the probabilities of the individual states
- P3 (the **failure condition**) says that the probability of a state not occurring is the complement of the probability of it occurring and implies that *basic terms* should sum to unity;
- PC1 (the **stochastic interleaving assumption**) says that any concurrent causally independent states may be modelled in any sequential order and this is regarded as the same logical event for stochastic purposes;
- PC2 (**causal stochastic dependency of** $\bar{\cdot}$ **and** $[\cdot]$) states that any states related by causal dependency must occur in the order of causality;

- PC3 (**complementarity of the stochastic dependency of $\bar{\cdot}$ and** $[\cdot]$ says that any causally dependent states occur with complementary probability to non-causally related states and implies that causal effects override local probabilities.

There are three aspects to note in the above definitions.

First, the full associativity of $\cdot$ over $+$ is in contrast to most process algebras where timing considerations permit a choice of alternatives expressed by only permitting right distributivity. Here we are not interested in the timing of choices, but only in trace semantics, so full associativity is required.

Second, observe that stochastic causality principle means that probability values do not obey $P3$ in the presence of $\bar{\cdot}$ and $[\cdot]$, so terms containing these operators are not *basic terms*. For example, equation 7.2

$$a_{0.5} + \bar{b}_{0.5}|(e_{0.3} + \epsilon_{0.7}) + [b]f_1 \tag{7.2}$$

implies that if b occurs then so must f, while if a occurs (or more precisely b does not) f may not occur.

Third, binding rules are: . binds more closely than $+$ and $+$ binds more closely than $|$. Thus, the binding is similar to the π-calculus.

Basic terms are defined inductively:

1. Every stochastic state a_p is a basic term
2. If t_ω is a basic term and a_p is an stochastic state then $a_p \cdot t_\omega$ is a basic term
3. If t_ω and s_α are basic terms then $s_\alpha + t_\omega$ is a basic term

By substituting $\rightarrow$ for $=$ in Table 7.1 for all equations except $A1$ and $A2$, we create a *Term Rewriting System* for our $SCTA$. Traditionally, process algebras define operational semantics. We are not interested in these aspects but rather in *trace semantics*. Hence, after rewriting, our basic terms represent the set of all possible traces (or linearizations) of the computation. Using a TRS is an easier means of calculating linearizations than the approach presented in section 7.2.1.1.

We extend our algebra to include replication by making use of of labels and recursive equations in $SCTA$ following normal conventions e.g $X = (xy_p + zy_q)X$ and $X_1 := xy_pX$ and including axioms for recursion. But recursion is not well-defined for single, non-causal states. We may also use superscripts or projection functions to define finite recursion – though this is not done here.

In general, we can equate causal effects to the results of a communication carrying information from one process to another process, not necessarily directly. More specifically, we require a formal definition of the *causal primitives* which we previously defined in section 7.2.1.1.

- *strong causality* where a *dependent state* **must** occur as the result of a unique set of *precursor states*;
- *weak causality* where a dependent state **must** occur as the result of several alternative sets of precursor states;

- *conditional dependence* where a dependent state is permitted to occur as the result of alternative sets of precursor states, which need not be uniquely defined

Formally, using our algebra, we define these causal primitives as conditional equations, which can be derived from the set of equations in Table 7.1, in particular, $PC2$ and $PC3$.

Strong causality (SC) is given by equation 7.3, while weak causality (WC) is given by equation 7.4 with conditional dependence (CD) being defined in contrapositive form in equation 7.5

$$\bar{x}_p|y_q + [x]z_1 \Rightarrow (xz)_p(SC) \tag{7.3}$$

$$\bar{x}_p|\bar{y}_q|u_r + [x]z_1 + [y]z_1 \Rightarrow u'_r + (xz)_{p'} + (yz)_{q'}(WC) \tag{7.4}$$

$$\neg\bar{x}_{(1)}|y_q + [x]z_r \Rightarrow ((\neg x)y + y(\neg x))_q(CD) \tag{7.5}$$

where $r < 1$. That is, the absence of the *precursor state* x strongly causes $\neg z$. Strong causality and conditional dependency are thus logically contrapositive. It follows that terms containing conditional dependencies must sum to unity in the absence of other causal relations.

Note that multiple causes (conjoint weak causality) may be shown by a vector of states using the notation $[\overrightarrow{v}]x$ where the states are assumed to arrive in any order. For example, in equation 7.6, let $\tilde{v} = \{x, y\}$.

$$\bar{x}_p|\bar{y}_q|u_r + [\tilde{v}]z_1 \Rightarrow u'_r + (\tilde{v}z)_{pq'}(CWC) \tag{7.6}$$

Similarly, a single cause with multiple effects can be shown by a superscript over the predecessor state $\bar{a}^n$ where n is the number of concurrent effects, in whatever order, which may occur.

7.2.3 Anomaly Detection Using Linearizations of Process States

We have shown how a distributed computation may be modeled as a partially order set of states $(S, \rightarrow_c)$ – section 7.2 – where $\rightarrow_c$ is the relation "causally happened before" or simply "causes" [112]. We have extended this approach to considering a subset of states S_X whose sequences are considered to be security relevant. S_X is also a partial order. This partial orders may be linearized as a set of total orders which may be calculated by our algebra or using an expected behavior graph as convenient.

For the purposes of anomaly detection, we label the sequences $S_x(i)$. An element of the sequence will be $S_x(i, \epsilon)$ and the order in which elements appear is denoted by an ordered set $S_x(i, C)$ where C is described as a *permutating vector clock* of $S_x(i)$. C may be calculated by indexing the order of each possible sequence $S_x(i, j)$ of a process using an equation described in our algebra. Each individual sequence is denoted $S_x(i, j, C)$. The clock value of an individual element of such a sequence is denoted by $S_x(i, j, \epsilon, c)$ where c is the index order of ϵ. This allows us to distinguish sequences which share the same states, but where the transitions between states occur in a different order.

A linearization in our model is a *variform mapping* φ of the sequences of $S_x(i)$ and $S_x(j)$, where $i \neq j$, and φ is the set of all linearizations of $S_x(i)$ and $S_x(j)$ which are causally consistent – that is, which obey the axioms of our algebra. We also define a set of *formless mappings* ϕ which are the set of all possible linearizations of the sequences of $S_x(i)$ and $S_x(j)$ where $j \neq i$, i.e., where all combinations of ordered sequences are considered regardless of causality.

We may therefore define a set of inconsistent linearizations B_X in our model as the set $\phi - \varphi$. We extend the set B_X in practice to include linearizations which contain any set of conditions which defy expectation under some threshold function (which may indicate the use of apparently legitimate states to conceal subversion) and any unique sequence of states which are known to be indicative of subversive behavior which are defined in B_X as an illegitimate word or characteristic vector in a model language [101]. For identifying the characteristic vectors in B_X, we assume the existence of a set of monitors which are devices or programs which may be formally defined as an automaton which accepts some language over finite words [112]. The set B_X, therefore, forms a negative context for the anomaly detection. Given a set $P(W)$ of program states and $\neg[Q(V)]$ of model states, the anomaly intrusion context is defined by $P(W) \leftrightharpoons_\delta \neg(\neg[Q(v)])$.

There are four cases to consider. Cases 1 through 4 may be detected by considering a set of concurrent observations T_X (captured by some as yet undefined mechanism) and matching words in T_X with the characteristic state vectors of B_X using the set of monitors. The complexity of the computation is limited by the length m of the longest characteristic vector in B_X. We can also consider techniques which increase search efficiency, e.g., including high probability characteristic vectors indicative of malicious activity at the top of the search order. We may also give consideration to partial matches using some probability threshold. Case 5 asks us to consider the consistency of logical clock values (or physical timestamps where available) with observed causal relations to eliminate false negatives. A simple calculation over the clock values of precursor and dependent states provides the conditions for an alert.

Case 1: Alien State Case 1 is trivial. The set of linearizations B_X only contains states which are consistent with S_X or characteristic vectors associated with attacks. An "alien state" will not exist in B_X unless it forms part of a characteristic vector. In either case, it reveals a potential attack.

Case 2: Inconsistent Linearizations An inconsistent linearization is a sequence, belonging to a single process, which is a member $f \in \phi$ defined as the set of formless mappings which is not part of $v \in \varphi$ the set of causally consistent mappings and is detected by its presence in B_X.

Case 3: Inconsistent Causality In Case 3, we make the assumption that causal relations will be strongly synchronized. By this, we intend that the "pigeonhole principle" applies and the number of effects must be less than or equal to the number of causes. Moreover, in considering cause and effect (and contrapositively, conditional dependency and effect), every preceding cause must be satisfied in vector clock order (see Case 5).

We may observe two sub-cases:

1. A cause with no effect where the subsequent transitions in the dependent sequence are not compatible with the precursor state;
2. An effect with no cause where the transitions in the preceding sequence are not compatible with the observed effects.

Case 4: Known Subversive Behavior If we observe a unique set of states which form a characteristic or feature vector associated with known subversive behavior, we may raise an alert. We may also set a threshold value for probable candidates which partially match such a vector in B_X.

Case 5: Clock Violation A vector clock inconsistency where the last time stamp in round of observations in which the dependent state is observed precedes the earliest logical timestamp for the observation of the precursor state, i.e., the order of cause and effect have been reversed without meeting the condition for the satisfaction of causes. A physical timestamp will also provide the same condition. Let p and q be precursor and dependent states respectively and c is a clock value. If $p(c) > q(c)$ then an alert is given. We may record a set R of causal relations pRq to enable us to track such transactions.

7.3 Concurrent Observation Mechanism

Up until now we have defined two related means of specifying the behavior of a subset of key processes and derived from this a means of calculating a set of characteristic vectors B_X which would be indicative of potential malicious activity in a given distributed system. We have assumed the existence of an observation mechanism which is capable of collecting concurrent with the run of a system observations of state for each of the key processes and hence allows us to compare a set of observations T_X with B_X to create anomaly detection events. We have also assumed the existence of software agents as monitors which can detect semantically significant traces.

In this section, we define a candidate mechanism for obtaining these observations on a multiprocessor host system. We provide a formal description of the mechanism's operation, its architecture and communication strategy. We also provide a probability model for observational success and conclude with an outline of its resilience to attack. We go on to provide a "proof of concept" for the model's effectiveness using a multiprocessor host system. We use the π-calculus in its goal transform variant to provide a formal demonstration of the mechanism's operation.

7.3.1 Model and Architecture

On a multiprocessor system , we define a set of observers $\mathcal{O}$. Each observer $O \in \mathcal{O}$ is an instance $O_{u,v,w}(m)$ where u is the observed (e.g. kernel) feature, v is the observation role, w is the instance of this function and $m \in M$ is a vector representing the measurement function and other role/task parameters, e.g., number of instances to assign to this task. Each thread is assigned to a CPU. Ensuring that ensembles of observers have different assignments allows them to concurrently observe state. The outcome in terms of the ordering of observation depend on the coding synchronization primitives and hardware memory handling features. Each observer $O \in \mathcal{O}$ uses algorithm 7.1 in its basic operation.

Intuitively, the observer thread takes a measurement and communicates its findings to all other **concurrent observers** assigned to equivalent observation roles as itself, noting that such observers are to be distributed to as many independent processing units as are available. Each observer also receives observations for every corresponding observer and, after comparison, logs any inconsistencies in observation. Finally, it waits a random period up to a specified time limit before recurring. The algorithm is set out in 7.1.

Algorithm 7.1: $Observe(measure(), NTasks, Period, OwnId)$*

```
1
Parameters: $measure(), ntasks, period, ownid$
Var: $u$
  $u := 0$
  $while(continue())$
    $u = measure()$
    $...$
    $send(u, ownid)$
    $for \quad i = 0 \dots ntasks$
      $receive(w, jobid)$
      $if(u \neq w)$
      $...$
      $log(u, w, ownid, jobid)$
      $endif$
    $endfor$
  $wait(period - (period \backslash rand())$
  $endwhile$
```

It should be noted that **each observer is stateless**. This enables the system to scale well compared to potential future demands in multiprocessing systems [4] .

For large scale multiprocessing hosts – of the kind this approach is targeted at, the utilization of synchronization primitives in shared memory for interprocess communication is unfeasible. Instead, we substitute a communication model analogous to a best effort asynchronous system which we simulate in shared memory. The reader will observe that this is, in effect, a distributed system.

Let $\mathcal{M}$ be the observer mechanism which we use to implement our observer processes. Hence $\mathcal{M}$ consists of a set of N processes $\{O_1, O_2, \ldots, O_N\}$ which are our observer threads and a set of unidirectional channels C. Each channel $c \in C$ connects two processes. We can view the topology of the resulting system as a directed graph in which the vertices represent the processes and the edges represent the channels. A bi-directional channel may be represented as two unidirectional edges. We divide $\mathcal{M}$ into partitions of observers (ensembles) which are linked by bi-directional channels. Let $\overrightarrow{G} = V(E)$ be a graph. Let V be an ensemble of observers. Let E be the set of channels of V. Let $\overrightarrow{G}$ be a complete graph $\overrightarrow{K}$ connected by bi-directional channels. The implementation of this messaging architecture enables the use of well-known algorithms for "weakly" synchronizing process action, keeping logical time and taking global snapshots in state [46]. These may be realized by additional coding to the basic algorithm 7.1 as required.

During observation, any inconsistency in measured states results in a log being created. Let O_i and O_j be two communicating observers whose states differ during a single round of observation, then $V_{o_i,o_j} = \{(o_i, o_j), (o_j, o_i)\}$ is a vector which results from the comparison of states. In fact, V_{o_i,o_j} consists of two ordered pairs of observations from each observer O_i and O_j of their observed and received values.

Let V_O and W_O be the set of all such vectors which results from two distinct rounds of observation. If we know the initial state, say T_0, of the operating system characteristic under observation, then V_O and U_O should provide us with sufficient knowledge to determine a set of partial orders $(T', \rightsquigarrow)$ resulting from T_0 which are the set of reported transitions in state, eliminating any repeated 2-tuples.

7.3.1.1 Observation Strategy

We may formally prove the validity of this approach using the π-calculus with regards to its application to multiprocessor host devices. Assume two agents $\|P\|Q\|$ which both invoke processes which report state. Let R_i and R_j be subprocesses which return the state of a variable x, which we assume is significant in the security of the system. Let P message R_i and Q message R_j and assume, w.l.o.g, that $i < j$.

In an operating system, R_0 will invoke a function with a query message q to R_{i+1}, which is a subgoal, which in turn will invoke other subgoals until a goal R_n is reached which returns the current value as a response r which provides the value of a state variable x. Each invocation represents a call to a lower level of abstraction in the system. Q acts likewise, but may access a lower level of abstraction directly. Multiple processes $Q_1, Q_2, \ldots$ may exist..

$$\begin{aligned} P &::= R_0|R_1|\ldots\|R_n \\ Q &::= R_j|R_{j+1}|\ldots|R_n \\ R_i &::= \bar{x}\langle q\rangle_{R_{i+1}} + x(r)_{R_{i+1}} \\ R_n &:= f(q) \supset \bar{x}\langle r\rangle_{R_{n-1}}\langle x\rangle \end{aligned} \tag{7.7}$$

It is easy to see that P and Q are competing for the same resource – knowledge the state of $x = r$ for any x defined – and that the result is non-determined, depending on memory handling as to which subgoal retrieves the last written value of x from memory first. If we further subvert one of the subprocess in the system to misreport x having altered its value,R_k for any of R_1 to R_n. It should be clear that if $k > j$ then an alteration in x might at best be momentarily observable depending on how memory reads and writes are ordered. However, it $i < k \leq j$ then the inconsistency of x as reported by $P|Q$ is readily spotted. In addition, both observers will note any legitimate alteration in x.

Two possible results follow from this approach. First, we can easily observe inconsistent reporting of state – in a method comparable to chapter 6 – including probabilistically any momentary inconsistencies. Second, we can also otherwise track the state of x and other such variables and determine from our expected behavior model if these are consistent with each other.

7.3.1.2 Observational Probabilities

For the lossless asynchronous messaging architecture we describe in relation to a multiprocessor host, the probability distribution of observing a single transition is given. Let L be the length of an observation round. Assume a transition event occurs at some point during L. Observer threads may take a measurement at any time during L where the linearization of transition and observation is non-deterministic, owing to both randomness in scheduling processing and memory access sequencing. Let p be the probability of observing before the transition event T. Let q be the probability of observing after the transition event T. Then the probability of detecting $P(T)$ is the same as the probability of at least one observation event occurring before T and at least one observation event occurring after T. Let n be the number of observer threads. It follows that $P(T) = 1 - (p^n + q^n)$ – that is, T is not observed if all state measurements take place either before or after T.

Given a transition between a legitimate and an illegitimate state, which will subsequently be reversed to give the appearance of legitimacy, what is the probability of detecting the malicious state? In fact, by letting q be the probability of an observer taking a measurement during an illegitimate transition in state and p be the probability of observing any legitimate state, we see that the same binomial distribution applies. It follows that the probability of observing any transition in state will in general increase as the number of concurrent observers. For large scale multiprocessing systems, this probability may be rendered arbitrarily high.

However, *such arbitrary increases in the probability of success are unlikely to be achievable on small scale systems* as constraints are naturally introduced for such systems where process scheduling overheads rapidly come to dominate. This observation technique is, therefore, primarily designed for observations of state in a host system with larger numbers of independent processing units (hyper-threading, dies,cores). On smaller platforms, optimum detection rates will be a trade-off between the frequency of observation rounds and the efficiency of the observational functions, suggesting that observations for a given frequency will rise to a maximum and then either stabilize or even diminish. This is a limitation in the context of ICS, but could be overcome by considering the additional use of co-processors to augment the approach.

Finally, *observational probabilities should not be confused with detection probabilities*. The results of observation are cumulative over several rounds, leading up to a detection event – that is the creation of a characteristic vector in B_X as defined in section 7.2.3. Detection probabilities are therefore much higher and detection can take place even in the presence of missing observations, although confidence levels in the results will be lowered. For example, if a characteristic vector consisted of six elements with an observational probability of 50% for each element, the probability of failing to observe any two out of three events yields $p \approx 0.02$. Hence detection levels can be raised to be comparable with any modern IDS. We consider that having at least a 50% probability of observation is a requirement for any component we wish to observe. Frequently, higher probabilities can be achieved.

7.3.1.3 Resilience to Attack

We show that the multiplicity of observers and the multiplicity of communication channels results in raising the barrier to subversion of the mechanism. This primary advantage is underpinned by the ability to self-observe.

Let n be the minimum number of observers per ensemble and N be the number of processors. Let $n > N$. It follows, assuming that the subversion of each observer is non-trivial, that an adversary has insufficient computational capability to simultaneously undermine the observers belonging to any ensemble in the mechanism, i.e., assuming the adversary commandeers N processors, the condition that N is strictly less than n prevents their concurrent subversion.

Let O be an observer ensemble. Let $o_1, o_2, \ldots, o_n \in O$ be observers. Each observer o_i communicates its findings with every other observer o_j. Let $|O| = n$. There exist $n(n-1)$ ordered pairs of data readings. Any alteration of an already observed value during a round will instantly reveal subversive activity. The attacker must successfully subvert $n(n-1)$ individual channels to control the outcome for a single ensemble. Clearly, we present the attacker with a combinatorial barrier to subversion. The ability of the mechanism to self observe arises because it is possible to assign a set of independent observers to functions in relation to this task. This underpins the advantages bestowed by self validation by providing further integrity checks.

The multiplicity, independence and distinctness of the observers clearly provides a degree of resilience for the observer mechanism, although performance and messaging overheads need to be taken into account in setting the number of observers. In addition, if poor communications are an issue then some adjustments would need to be made to the synchronizer algorithm to take account of undue latency or communications failure, for example, a controlled time out on message waiting periods.

Considering the possibility of direct attack on the mechanism [117], the major advantage of our approach is that the independence of the observer processes enables us to instantiate a subset of these processes to observe data elements associated with the observation mechanism itself. This enables periodic tests of the integrity of the mechanism. Combined with the possession of a multiplicity of independent observers, this acts as a deterrent to attack by requiring that the capability to simultaneously subvert a set of processes which may not even share the same logical or physical platform, or be functionally equivalent – even if observing the same data element.

A partial subversion of the mechanism is likely to fail because even if a subset of observers are subverted, or have their communications spoofed, this will appear to as a series of continual transitions in state whose permutations are unlikely to be equivalent to the expected behaviors of our selected data elements. If instead we consider an attack *en masse* where the attacker seeks to take advantage of a single vulnerability across an ensemble (or more than one) of observers, we find that this possibility is closed to them. Even within a single ensemble because the same data is observed from different points of view (representing either different APIs or different logical or physical platforms) requiring the use of distinct functionality, the likelihood that a single exploitable vulnerability will result in the mass compromise of the observer processes is low. The use of different platforms also implies that coding or systems operation details may be distinct, further adding to the attacker's difficulties.

Thus, the probabilistic nature of our mechanism acts as a deterrent to attackers since they may not easily predict where and when any actions of theirs may be observed, nor is it computationally feasible for them to seek to control multiple observers which work at different points in process time and space and which possess distinct functionality.

In other words, the constraints which we face in detecting malicious activity are redoubled for the attacker considering the subversion of a distributed computation which is guarded by our mechanism, even with full knowledge of the mechanism. Effectively, we have turned the attacker's apparent strength in being difficult to detect in a distributed environment "jiu jutsu"-like against him[5]

7.4 "Proof of Concept" Mechanism

We implemented a basic version of the proposed mechanism as a loadable kernel module (called KRAKEN) in Linux v2.6.21 on a VMware platform on a 2-core Intel system. We used basic concealment techniques to evaluate functionality [56]. In particular, we were interested in demonstrating: First, that it could implement concurrent observations of different APIs within the operating system. Second, that it could observe distinct, but related, kernel features concurrently. Third, that it was capable of self observation. We also wished to establish that we could arbitrarily raise the probability of observation by increasing the number of observers up to natural performance constraints (due to the limited nature of the platform). The cumulative nature of observations subsequently raises the probability of detection. In particular, the concurrent observation of distinct, but related, kernel features enables this potential.

We describe in detail the experimental work. The program[6] is is a loadable kernel module (LKM) which declares and initiates a set of observer threads, divided into various subsets which are categorized by the variables they observe. Alterations in the value of these variables represent key events in relation to operating system states which we call "semantic chokepoints". A sequence of alterations, which can be defined algebraically by a stochastic expression representing the probable trace semantics, which we call a *characteristic vector* may be used either to detect good behavior, i.e., "liveness", or else to detect potentially malicious behavior which violates "safety". The observers work concurrently with the operating system and with each other, exchanging and comparing information after

[5]If the reader prefers Chinese martial arts, then he should consider the example of Choy Li Fut, where the student is trained to consider all objects in the local environment as weapons in the fight against his opponent.

[6]The listing is available in electronic format on request.

each round of observation in a strictly monotone sequence. Any comparison which detects a non-equivalence in observations is flagged as a alteration in the value of the variable. The observation is probabilistic in nature, meaning that some alterations may be missed. But the overall sequence is deducible by reasoning over causal relations. Some additional programming artifacts such as a logical clock and a global snapshot algorithm were included to demonstrate that these features could be incorporated into the mechanism to aid post-incident forensics. We also provide listings for the two pseudo-rootkits used to test the observation module and provide in full the results of the experiments and a version of the experimental notes which have been edited to improve formatting and grammar.

7.4.1 Experiments and Results

In a cross-sectional view of operating states, we examine observations of semantically related values at different APIs and compare results. We selected the virtual file system as an example of a kernel subsystem which consists of several layers of abstraction. A classic attack is to "hook" the system call table functions which invoke kernel functionality in this area to selectively hide static file structures. A more sophisticated attack at a lower level of abstraction – demonstrated by e.g. the *adore_ng* rootkit – is to subvert the function pointers for the *readdir()* function, leaving the system table unaltered.

We created 3 observer ensembles, which used clean copies of kernel code to measure root directory size at 3 different APIs, including the system call table, *getdents64()* and *readdir()* functions. We could increase the numbers of observers in each ensemble and also vary the period of time between observation rounds.

In an initial trial, we subverted the system call table. This resulted in immediate and ongoing reporting of inconsistencies, showing that where a lower level API which retained integrity was compared with a subverted higher one, the anomaly is invariably detectable, matching our formal model of the system, shown in section 7.3.1.1. Subsequently, we selected to subvert the *readdir()* function to hide a randomly named file in the root directory[7]. The simulated attack created and then hid the file such that any subversion would cause a momentary transition in root directory size, but would not be detectable thereafter, based on our system model. 100 transitions were observed per trial. For measuring a single characteristic, we took as our hypothesis that observation rates greater than 50% would be acceptable *given likely constraints on performance* – see 7.3.1.2.

The results, listed in the following subsections, show that it is possible to achieve a satisfactory observational probability for observing transitions in values between lower and higher level APIs and detecting any inconsistencies. The first three experiments were used to demonstrate that we could detect changes for a single component in the kernel. We then show that we can detect across a set of components to create a "characteristic vector". Finally, we created another Linux kernel module which was able to destroy threads in the observation mechanism to test its ability to self observe and alert on this attack.

[7]this feature is common to several attacks.

No. Threads	%	CI Min	CI Max
3	75	66.51	83.49
5	72	63.20	80.80
7	79	71.02	86.98
9	77	68.75	85.25
n=100	Jobs = 3	Hiatus = 10	

Table 7.2: File Object Alteration – 10ms Hiatus

7.4.1.1 File Object Alteration Detection

Hypothesis: Using a multiprocessor host based multisensor program (KRAKEN), an 'concealed' alteration in a file structure (in this experiment, a directory object on a Linux system) could be detected with a probability greater than 50%.

Null Hypothesis: The detection rate would be lower than 50%.

Method: In this experiment, we generated sensor threads which used clean copies of kernel code to read the file object size characteristic at different APIs within the kernel. The sensor threads would read a value and communicate it to all other threads for comparison. Any mismatches would be reported. The observations were randomized over time periods of different lengths and the number of threads increased systematically throughout. A second program which used rootkit techniques to create and conceal the alteration was similarly written. This used the technique from the rootkit *Adore_NG* to conceal the file's existence by pointing the *readdir()* function address to malformed code which selectively read the directory and ignored the file's existence. The second module was inserted and removed from the system using a Python program which created a randomly named file for the 'rootkit' program to conceal on insertion and reveal on removal. The procedure was repeated 100 times with 5 second intervals between each activation. The multisensor program logged any detected differences in file size in the directory object (the or root directory).

Results: The hypothesis is upheld for all variations in thread production and maximum observation frequency with the exception of the lengthiest suspension of activity between observation rounds where results were (and could have been expected to be) less satisfactory. We would expect similar findings for any other file object using equivalent clean code. The results are reproduced here.

We note that the greatest increase in observational success, based on the calculated averages, by decreasing the time period over which observations occurred. However, on a constrained system such as the one we are using, we might expect performance to level out beyond certain limits in thread generation or increases in observational frequency due to performance hits and sensors literally getting in each other way (communication periods being one part of this and competition for CPU time the other between too few CPUs). This may explain the apparent drop off in observational scores for higher numbers of threads, though it may simply

No. Threads	%	CI Min	CI Max
3	65	55.65	74.35
5	72	63.20	80.80
7	65	55.65	74.35
9	67	57.78	76.22
n=100	Jobs = 3	Hiatus = 20	

Table 7.3: File Object Orientation – 20ms Hiatus

No. Threads	%	CI Min	CI Max
3	54	44.23	63.77
5	67	57.78	76.22
7	76	67.63	84.37
9	70	61.02	78.98
n=100	Jobs = 3	Hiatus = 30	

Table 7.4: File Object Orientation – 30ms Hiatus

No. Threads	%	CI Min	CI Max
3	53	43.22	62.78
5	58	48.33	67.67
7	70	61.02	78.98
9	64	54.59	73.41
n=100	Jobs = 3	Hiatus = 40	

Table 7.5: File Object Orientation – 40ms Hiatus

be a statistical freak. The detection rate is not acceptable for intrusion detection systems taken in isolation. However, the goal of the system is to provide a cumulative detective effect. The probability scores displayed are more than acceptable in this context where 50 % is taken to be the lowest acceptable probability for a given sensor task.

7.4.2 Read/Write Flag

Hypothesis: Using a multiprocessor host based multisensor program (KRAKEN), an alteration in a read/write flag for a kernel object (the system table) could be detected with a probability greater than 50%

Null Hypothesis: The detection rate would be lower than 50%

Method: A multi-sensor program was written as a Linux Kernel Module in C. It could generate sensor threads which used clean copies of kernel code to read the flag value. The sensor threads would read a value and communicate it to all

No. Threads	%	CI Min	CI Max
3	55.5%	48.61	62.39
5	56.5%	49.63	63.37
7	61%	54.24	67.76
9	64%	57.35	70.65
n=200	Jobs = 1	Hiatus = 10	

Table 7.6: Read/Write Flag Alteration – 10ms Hiatus

No. Threads	%	CI Min	CI Max
3	44.5%	37.61	51.39
5	51.5%	44.57	58.43
7	57%	50.14	63.86
9	57%	50.14	63.86
n=200	Jobs = 1	Hiatus = 20	

Table 7.7: Read/Write Flag Alteration – 20ms Hiatus

No. Threads	%	CI Min	CI Max
3	41.5%	34.67	48.33
5	51%	44.07	57.93
7	58%	51.16	64.84
9	56%	49.12	62.88
n=200	Jobs = 1	Hiatus = 30	

Table 7.8: Read/Write Flag Alteration – 30ms Hiatus

other threads for comparison. Any mismatches would be reported. The observations were randomized over time periods of different lengths and the number of threads increased systematically to improve detective performance. A second program which used rootkit techniques to create and conceal the alteration was similarly written. This used kernel code to change the flag status. The second module was inserted and removed from the system using a Python program which created a randomly named file for the 'rootkit' program to conceal on insertion and reveal on removal. The procedure was repeated 100 times with 5 second intervals between each activation. This led to 200 alterations in flag status in total.

Results: The hypothesis is upheld for all variations in thread production and maximum observation frequency except for 3 threads at 20,30 and 40m which represent the lowest observation values. The results are reproduced here.

We note that the greatest increase in observational success, based on the calculated averages, was for increasing the number of threads. We do not see the same drop leveling off in performance as for experiment on file objects but this is because fewer threads in total are being used. Performance could probably be further im-

No. Threads	%	CI Min	CI Max
3	46%	39.09	52.91
5	52.5%	45.58	59.42
7	53.5%	46.59	60.41
9	73.5%	67.38	79.62
n=200	Jobs = 1	Hiatus = 40	

Table 7.9: Read/Write Flag Alteration – 40ms Hiatus

proved until these limits are reached. Against this, a reduction in the period during which a change in flag status is detectable could reduce this probability.

The detection rate is not acceptable for intrusion detection systems taken in isolation. However, the goal of the system is to provide a cumulative detective effect. Most of the probability scores displayed are more than acceptable in this context where 50% is taken to be the lowest acceptable probability for a given sensor task.

The highest scores were recorded for the maximum number of threads with the longest observation period. These scores were significant and suggest that this represents an optimum strategy for detecting this kind of alteration. But further research would be required to confirm this hypothesis. It may simply be a result of the synchronization of timing between insertion and deletions and this time period - a sinusoidal effect purely as a result of the nature of the experiment.

7.4.3 System Table alteration

Hypothesis: Using a multiprocessor host based multisensor program (KRAKEN), an alteration in the system table could be detected with a probability greater than 50%.

Null Hypthesis: The detection rate would be lower than 50%.

Method: The multi-sensor program was written as a Linux Kernel Module in C. It could generate sensor threads which took a hash of the system table pointer addresses. Any alteration in address values would lead to a different hash value being calculated. Any mismatches would be reported. The observations were randomized over time periods of different lengths and the number of threads increased systematically to improve detective performance.A second program which used rootkit techniques to create and conceal the alteration was similarly written. This used kernel code to change the system table pointer for the *sys_read* address.The second module was inserted and removed from the system using a Python program which created a randomly named file for the 'rootkit' program to conceal on insertion and reveal on removal. The procedure was repeated 100 times with 5 second intervals between each activation. This led to 200 alterations in the system table in total. The multi-sensor program logged any detected differences in file size in the directory object (the '/' or root directory).

No. Threads	%	CI Min	CI Max
3	50%	43.07	56.93
5	58.5%	51.67	65.33
7	73%	66.85	79.15
9	67.5%	61.01	73.99
n=200	Jobs = 1	Hiatus = 10	

Table 7.10: System Table Alteration – 10ms Hiatus

No. Threads	%	CI Min	CI Max
3	40.5%	33.70	47.30
5	52%	45.08	58.92
7	68%	61.53	74.47
9	66.5%	59.96	73.04
n=200	Jobs = 1	Hiatus = 20	

Table 7.11: System Table Alteration – 20ms Hiatus

No. Threads	%	CI Min	CI Max
3	45.5%	38.60	52.40
5	52.5%	45.58	59.42
7	69%	62.59	75.41
9	70.5%	64.18	76.82
n=200	Jobs = 1	Hiatus = 30	

Table 7.12: System Table Alteration – 30ms Hiatus

Results: The results are reproduced here. The hypothesis is upheld for all variations in thread production and maximum observation frequency except for 3 threads at 20,30 and 40, representing the lowest observation values.

We note that the greatest increase in observational success was for increasing the number of threads. We do not see the same leveling in performance as for the file object alteration experimentation but this is because fewer threads in total

No. Threads	%	CI Min	CI Max
3	42.5%	35.65	49.35
5	64.5%	57.87	71.13
7	62.5%	55.79	69.21
9	69%	62.59	75.41
n=200	Jobs = 1	Hiatus = 40	

Table 7.13: System Table Alteration – 40ms Hiatus

are being used. However, the results are suggestive of two optimization points, one for increasing number of threads and the other for increasing observational frequency, suggesting that some internal constraints are present in performance terms, possibly related to the 'inefficiency' of the approach. The detection rate is not acceptable for intrusion detection systems taken in isolation. However, the goal of the system is to provide a cumulative detective effect. Most of the probability scores displayed are more than acceptable in this context where 50 % is taken to be the lowest acceptable probability for a given sensor task.

7.4.4 Detecting a Characteristic Vector

A *characteristic vector* is an ordered sequence of states which matches with a known sequence of states associated with malicious software activity. The second experiment represented an extension of the first in which we considered other related kernel features which might be implicated in a concealment attack, but not necessarily belong to the same kernel subsystem. This experiment is, in miniature, a trial in raising detection probabilities through cumulative observations.

We added additional observer ensembles to measure the integrity of the system call table used hashed values for comparison and also checked the read/write permission flag for the table (which has to be altered to change the table).

For this experiment, we simulated a more complete attack where more than one kernel substructure (i.e. the system call table, its read/write flag and the `readdir()` functionality) might be altered and subsequently the alteration reversed. Each trail consisted of 100 transitions of the LKM simulating attacker actions by first altering and then reversing the alteration to these structures. We recorded a successful *detection* if 2 out of 3 of the characteristics were observed in transition. It should be noted that additional opportunities were provided to observe the attack because the reversal of the system call table and read/write flags was obvious to the mechanism. This is a characteristic of some concealment techniques that we can happily take advantage of. We set the target for success higher at 90% as we were taking advantage of cumulative observational probabilities. The results are shown in section 7.4.4.

These results demonstrate how cumulative observations from distinct observer groups result in acceptably high levels of detection. Here it is much clearer that increasing the number of observers is the dominant strategy for increasing the probability of detection. These results show that we can concurrently measure semantically related kernel structures and underscore the high cumulative probability of detecting anomalous transitions.

Hypothesis: Using a multiprocessor host based multi-sensor program (KRAKEN), an alteration in the system table, an alteration to system table read/write flag or an alteration in file size could be detected with a probabilities greater than 90% on a 2 out of 3 combination. This characteristic vector detection has to be acceptable for IDS systems.

Null Hypothesis: The detection rate would be lower than 90%.

No. Threads	%	CI Min	CI Max
3	86	79.20	92.80
5	96	92.16	99.84
7	96	92.16	99.84
9	99	97.05	100.95
n=100	Jobs = 2	Hiatus = 10	

Table 7.14: Characteristic Vector Detection – 10ms Hiatus

No. Threads	%	CI Min	CI Max
3	85	78.00	92.00
5	92	86.68	97.32
7	98	95.26	100.74
9	98	95.26	100.74
n=100	Jobs = 2	Hiatus = 20	

Table 7.15: Characteristic Vector Detection – 20ms Hiatus

No. Threads	%	CI Min	CI Max
3	72	63.20	80.80
5	86	79.20	92.80
7	98	95.26	100.74
9	98	95.26	100.74
n=100	Jobs = 2	Hiatus = 30	

Table 7.16: Characteristic Vector Detection – 30ms Hiatus

Method: The multi-sensor program was written as a Linux Kernel Module in C. It could generate sensor threads to detect each condition. Any mismatches would be reported. The observations were randomized over time periods of different lengths and the number of threads increased systematically to improve detective performance. A second program which used rootkit techniques to create and, where appropriate, conceal the alterations was similarly written. The second module was inserted and removed from the system using a Python program which created a randomly named file for the 'rootkit' program to conceal on insertion and reveal on removal. The procedure was repeated 100 times with 5 second intervals between each activation. This led to 100 alterations in the characteristic vector. The multi-sensor program logged any detected differences.

Results: The hypothesis is upheld for medium to high end variations in observational frequency and thread generation:

We note that the greatest increase in observational success, based on the calculated averages, was for increasing the number of threads. We do not see any apparent drop offs in performance as for this experiment. This may be because observations did not rely on the success of single measurements. The results sug-

No. Threads	%	CI Min	CI Max
3	59	49.36	68.64
5	86	79.20	92.80
7	93	88.00	98.00
9	98	95.26	100.74
n=100	Jobs = 2	Hiatus = 40	

Table 7.17: Characteristic Vector Detection – 40ms Hiatus

gest that an optimization point is converging around 9 threads and between 20 and 30ms observational frequency for this particular system.

The detection rate is acceptable for intrusion detection systems, for higher end variants. However, this is a detective effort for only a single characteristic vector, not the accumulation that would be expected for a rootkit attack where several vectors might be affected. Since this effort is directed at semantic chokepoints, it has greater significance in system terms than a higher probability detection of less important characteristics.

7.4.5 Self Observation Test

In a final experiment, we evaluated the ability of the mechanism to self-observe. A LKM was created to delete observer threads in groups and replace them with inert threads. 100 observer threads were instantiated as "sacrificial victims" in addition to an ensemble of threads for self-observation. Observer threads measured a hash of the mechanism's process PIDS. Acceptable rates of observation were set as before at 50% for the measurement of a single characteristic, given potential performance constraints. Several trials were undertaken. The results are shown in section 7.4.5.

These results clearly demonstrate the feasibility of self-observation. However, we also see scheduling constraints causing observational values to peak and then fall off for increasing numbers of observer threads with a clear relation to the period allowed for observation – see section 7.3.1.2 for an explanation. This is a predictable result, which would not be encountered on larger-scale platforms (e.g. with 8 or more cores).

Hypothesis: The sensor module would be capable of gathering observations about changes to the number of processors it had when deleted in groups with greater than 50% probability.

Method: The multi-sensor program was written as a Linux Kernel Module in C. It could generate sensor threads which counted the number of threads in the process queue associated with the module.The sensor threads would read a value and communicate it to all other threads for comparison. Any mismatches would be reported. The observations were randomized over time periods of different lengths and the number of threads increased systematically during the experiment. A second program which used rootkit techniques to delete threads and replace them with dummy versions. The second module was inserted into the system and

No. Threads	%	CI Min	CI Max
3	36.67%	19.42	53.91
5	63.33%	46.09	80.58
7	63.33%	46.09	80.58
9	60.00%	42.47	77.53
n=30	Jobs = 1	Hiatus = 10	

Table 7.18: Self Observation Test – 10ms Hiatus

No. Threads	%	CI Min	CI Max
3	26.67%	10.84	42.49
5	23.33%	8.20	38.47
7	46.67%	28.81	64.52
9	73.33%	57.51	89.16
n=30	Jobs = 1	Hiatus = 20	

Table 7.19: Self Observation Test – 20ms Hiatus

No. Threads	%	CI Min	CI Max
3	30.00%	13.60	46.40
5	20.00%	5.69	34.31
7	50.00%	32.11	67.89
9	56.67%	38.93	74.40
n=30	Jobs = 1	Hiatus = 30	

Table 7.20: Self Observation Test – 30ms Hiatus

deleted threads in groups of 3 from the system 30 times. There procedure was repeated with 5 second intervals between each activation. The multi-sensor program logged any detected differences in the number of threads.

Results: The hypothesis is upheld when considering group deletions for high thread counts. The results are reproduced here.

It would be possible for an attacker to reduce this probability significantly by

No. Threads	%	CI Min	CI Max
3	23.33%	8.20	38.47
5	33.33%	16.46	50.20
7	43.33%	25.60	61.07
9	93.33%	84.41	102.26
n=30	Jobs = 1	Hiatus = 40	

Table 7.21: Self Observation Test – 40ms Hiatus

individually deleting threads with significant gaps between each deletion, but they would be left with the difficulty of explaining anomalous results for still good threads, whereas rapid deletion and replacement of a group of threads is a more viable strategy. It should be noted that each KRAKEN thread reported its own demise with 100% success, suggesting the adversary would need to alter the thread code or the behavior of the reporting mechanisms before proceeding.

This, in other words, was not the cleverest means of self observation, but it still had relatively significant success with results favoring a longer hiatus and a greater number of threads. This may suggest that rapidly repeated activity is better detected by lowering the frequency with which observation rounds are repeated in favor of increasing the number of observing threads and hence the probability of interspersing a thread observation with a malformed write to memory.

7.4.6 Performance

We addressed two performance issues. The first was examine detection capabilities under variable system loads. In Figure 7.3 we show the results of placing different loads on the CPU, virtual memory, IO and hard disk access facilities using a program called *Stress 1.0.0* which creates suitable process hogs which loop over suitable functions. The table indicates the number of hogs used and where different emphases in the loading were placed on the system. Overall, there was a drop in detection rates of around 10% but the system only appeared to suffer this initial hit on detection probabilities compared with trial results and thereafter performed consistently with minor non-significant variations.

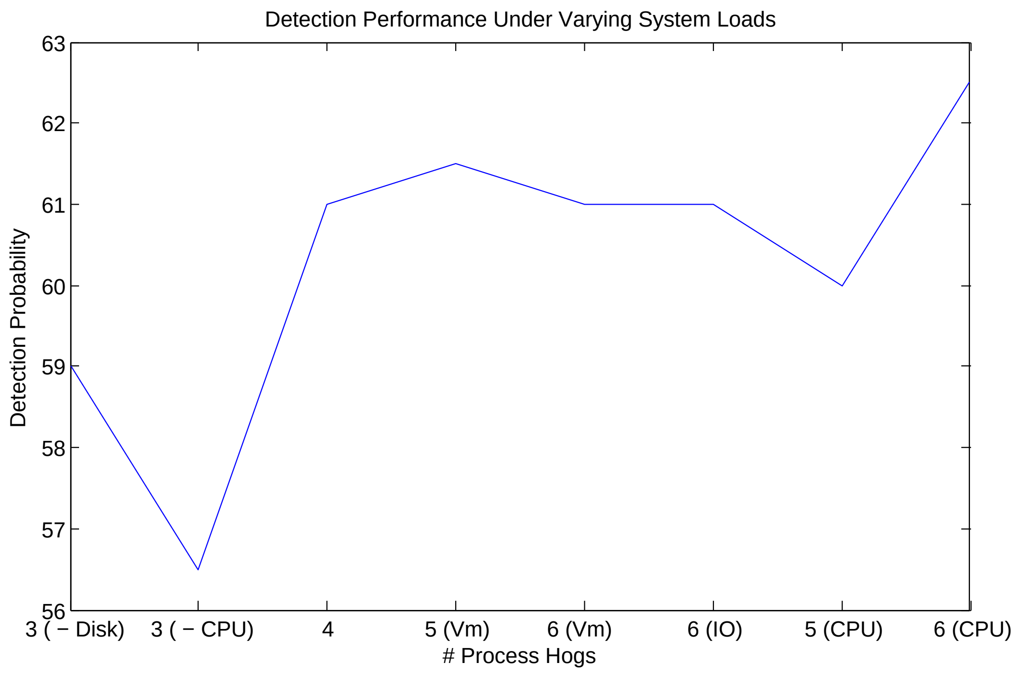


Figure 7.3: Detection Probabilities Under Stress

We also ran the module increasing the number threads from 15 to 45 and measuring the results using the *vmstat* utility. The average load on the CPU increased from 10 to 20 percent as the frequency and number of observing threads were increased. The load on i/o resources also increased gradually. There was a slight decrease in the virtual memory available. This reflects the way the program operates with i/o increasing for logging (snapshot reports) to the *messages* file and state comparisons consuming virtual memory resources. However, the generated processes frequently and voluntarily relinquish control of the CPU keeping down performance costs for CPU time as well as maintaining some advantage in scheduling by avoiding the appearance of being CPU bound. As would be expected for a kernel module, the primary CPU load is for kernel usage.

7.5 Summary

In this chapter, we provide a process algebraic and a corresponding technique, based on process diagrams, for specifying a distributed system and modeling its trace semantics. We have shown how the set of trace semantics may be converted to a set of unfeasible and malicious traces which we describe as characteristic vectors. We subsequently provide, for a multiprocessor system, a model and a "proof of concept" of an observational mechanism which can observe and report system traces to a set of monitors, whose existence we assume. We also demonstrate formally using the π-calculus our proposed mechanism. We selected examples in our "proof of concept" which show how the observation mechanism can detect significant changes in a subsystem and indeed link together alterations in state from disparate components in that subsystem. While observation is probabilistic, we argue that successful observation and detection figures may be driven arbitrarily high by the addition of multiple observer threads, although in certain environments such as ICS this would require the use of dedicated co-processors. The multiplicity of observer threads and the probabilistic nature of their action conspire to render it difficult for the adversary to subvert their action. In addition, the mechanism is capable of guarding its own integrity through self observation.

Chapter 8

Kinetic Models and Adversary Detection

You see, wire telegraph is a kind of a very, very long cat. You pull his tail in New York and his head is meowing in Los Angeles. Do you understand this? And radio operates exactly the same way: you send signals here, they receive them there. The only difference is that there is no cat.

Albert Einstein

8.1 Preamble

Kinetic interactions provide another context for determining whether or not ICS are under attack. By taking additional sensor measurements, beyond current observability and controllability requirements, and communicating them to the operator independently, we provide another layer of information about the behavior of such systems which can be used to check whether its expected behavior. This approach is not novel. However, in this case, we apply it to detecting anomalies in non-linear systems and demonstrate that only a small amount of additional sensor information is required to detect differences sufficiently significant to indicate the potential presence of malicious activity.

8.2 Example Selection

We selected a heat exchanger in association with a beer pasteurizer as a simple, but non-trivial, example. The actual process – beer pasteurization – does not, of course, form part of a critical national infrastructure[1], but represents a substitution which, we believe, is useful for researching ICS in general. Frequently, on systems *which are key to critical infrastructures*, organizations are unwilling or may even be constrained for security reasons from providing researchers with the opportunity to explore such systems directly. Hence, by substituting a "harmless" example of a similar process, it is possible to derive results which, with some adaptation, can

[1]Except to the author.

be implemented on the original systems. In this case, we achieved a double substitution since pasteurization is a key process in pharmaceutical production, particularly of vaccines, and heat exchange, as well as being ubiquitous to industrial processes, is key, in particular, to energy production and the prevention of serious breaches in health and safety on a potentially national scale.

8.3 Approach

The operation of the pasteurizer consists of a series of heat exchanges which are controlled to ensure specific targets are achieved for peak and trough flow values for the product. A set of non-linear differential equations may be used to express the behavior of the plant. An attack is defined abstractly as a manipulation of messages which affect the behavior of these relations – chapter 5 – moving them away from a desired steady state (the pasteurization rate). During an attack, key process characteristics will be concealed by attackers. Other process characteristics while not concealed may not aid analysis. The attacker can hide in the "noise" of the system which disguises signal manipulations. However, using our knowledge of the system and potential attacker behavior, we may identify suitable substitute readings for measuring key process characteristics. Having identified these, we may install additional sensors to take readings from these nodes and compare the results in real time with expected values in relation to process states to detect any anomalies.

8.4 Model

We describe the operation of a flash pasteurizer and model the identification of potential substitute measurements. We use a simulation in MATLAB/SIMULINK of the pasteurizer to develop a profile of the behavior of these readings under differing operating conditions (see section 8.7). We can subsequently use these for detecting breaches in the expected behavior of the system.

8.4.1 Pasteurization and Heat Exchange

Pasteurization is achieved by a series of heat exchanges between hot and cold fluids, where the hot side setpoint temperature is determined by flow rate, which is dependent on the rate of packaging (in this case, kegging), and the cold side temperature by packaging requirements. In flash pasteurization, the interaction of the pasteurized (hot) and unpasteurized (cold) product is used as part of the temperature cycle with target values being achieved through secondary heat exchanges using steam-heated water on the hot-side and a refrigerant (in this case, glycol) on the cold-side [82]. The basic equations for heat exchange are:

$$\dot{q} = UA(T_{in,s} - T_{out,c}) \tag{8.1}$$

$$wCp\frac{dT}{dt} = \dot{w}Cp(T_{in,c} - T_{out,c}) + \dot{q} \tag{8.2}$$

$$uCp\frac{dT_s}{dt} = \dot{u}CP(T_{in,s} - T_{out,s}) - \dot{q} \tag{8.3}$$

In equation 8.1 $\dot{q}$ is the rate of heat exchange, U is the coefficient of heat exchange for the construction material and A is the area, $T_{in,s}$ is the initial hot-side temperature (C^o), $T_{out,s}$ is the final hot-side temperature, $T_{in,c}$ is the initial product temperature and $T_{out,c}$ is the final product temperature. Equations 8.2 and 8.3 represent the respective energy balances of the cold and hot sides where $\dot{w}, \dot{u}$ represent the hot-side and cold-side flow rates respectively, w, u the liquid volume (m^3), and Cp is the specific heat capacity ($kJ/kg - K$) of the product.

Pasteurization units PU are derived from flow rate and temperature values using equation 8.4:

$$PU = \frac{w}{\dot{w}}(60)(1.393^{(T-60)}) \tag{8.4}$$

where w is the holding volume (HL), $\dot{w}$ is the flow rate (HL/HR) and T is the temperature. This is a dimensionless measure, which was derived by Dayharsh *et al.* [35]. We observe the non-linear relationship between flow rates, temperature and pasteurization values, which do not lend themselves to statistical analysis of the control process or a straightforward state estimation technique, using linearized equations (particularly where the same equipment may be used for different products).

The flow rate $\dot{w}$ is derived as follows:

$$\dot{w} = \dot{w}_{max} - \dot{w}_{min} \times \frac{(L_{actual} - L_{min})}{L_{max} - L_{min}} \tag{8.5}$$

where L is the tank level and $\dot{w}$ is the flow rate as before. The minimum and maximum tank levels for this specimen are 100 and 220 and the minimum and maximum flow rates are 250 and 500 respectively. Flow rates are clamped to their extrema when tank level values exceed their minimum or maximum values. Tank levels change almost continuously during pasteurization due to alterations in packaging (called "kegging") rates. We add the appropriate calculators to our model to simulate these set points.

The pasteurization process is dependent on the temperature and the rate (volume per second) at which the liquid, or material, to be pasteurized flows through the system. In the example system, this is achieved by heat exchange using the hot and cold liquids passing through a compartmented heat exchanger on a contraflow arrangement. A model of the plant is shown in figure 8.1.

The actual heat exchanger is shown just right and up of the center of the diagram. Unheated beer is pumped into the third chamber to the right and passed through various compartments being heated by hot beer returning the other way. The final part of the heating process is accomplished by using steam heated water to bring the beer to the desired temperature, after which it flows through a series of pipes to re-enter the heat exchanger being cooled by more beer entering the heat exchanger. The final part of cooling the beer before it is kegged is achieved by using glycol as a refrigerant.

Both the heating and refrigeration processes required to be controlled depending on the flow of beer through the plant. But the flow rate is determined by the tank level in PPB2A which in turn is determined by the kegging process downstream of the plant. In effect, the water flows randomly at different rates. Hence

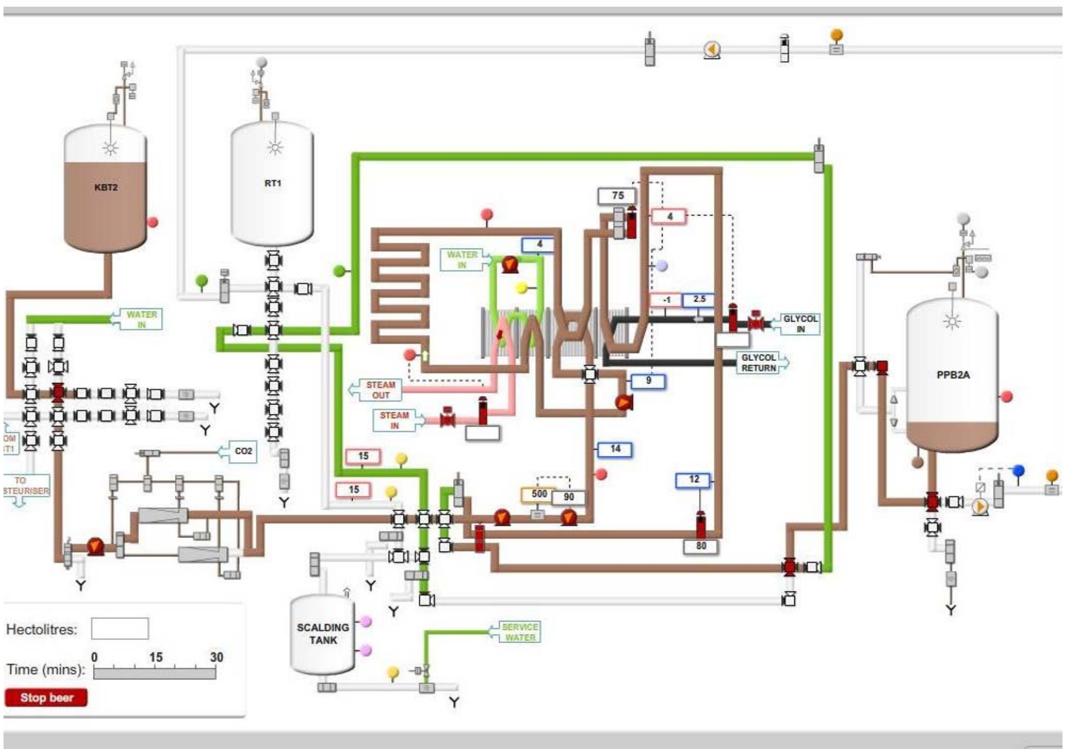


Figure 8.1: Beer Pasteurizer - Courtesy of Diageo PLC, 2009

the temperature of the water has to vary accordingly and likewise the amount of cooling supplied by the glycol.

8.5 Simulink Construction

Model construction centered on the requirement to model the heat exchange process and the “disturbances” caused by changes in tank level and hence water flow. However, we also included in the model perturbances of the heating and cooling rates and corresponding simulation of the control mechanisms (PIDs) which are used to heat or cool the product. The modelling was done using Simulink. The complete model is shown in figure 8.2 and the following sections provide an explanation of its construction.

A key to the Simulink block symbols used are provided in figure 8.3.

In brief, the *Input* block provides the input value from another block. The *Output* block corresponds. The *Sum* block is used to sum two values. The *Gain* block provides multiplication by a constant. The *Product* block provides the product of two (or more) values. The *Constant* block provides a constant value as an output. The *Random Number* provides a probabilistically generated random number which may be generated between set limits. The *Saturation* blocks sets upper (respectively lower) limits on values. The *Integrator* sums values over time. Finally, the *Scope* block provides a readout of the values sent to it. All values in Simulink are also stored in matrices which can be used by MATLAB for subsequent calculations

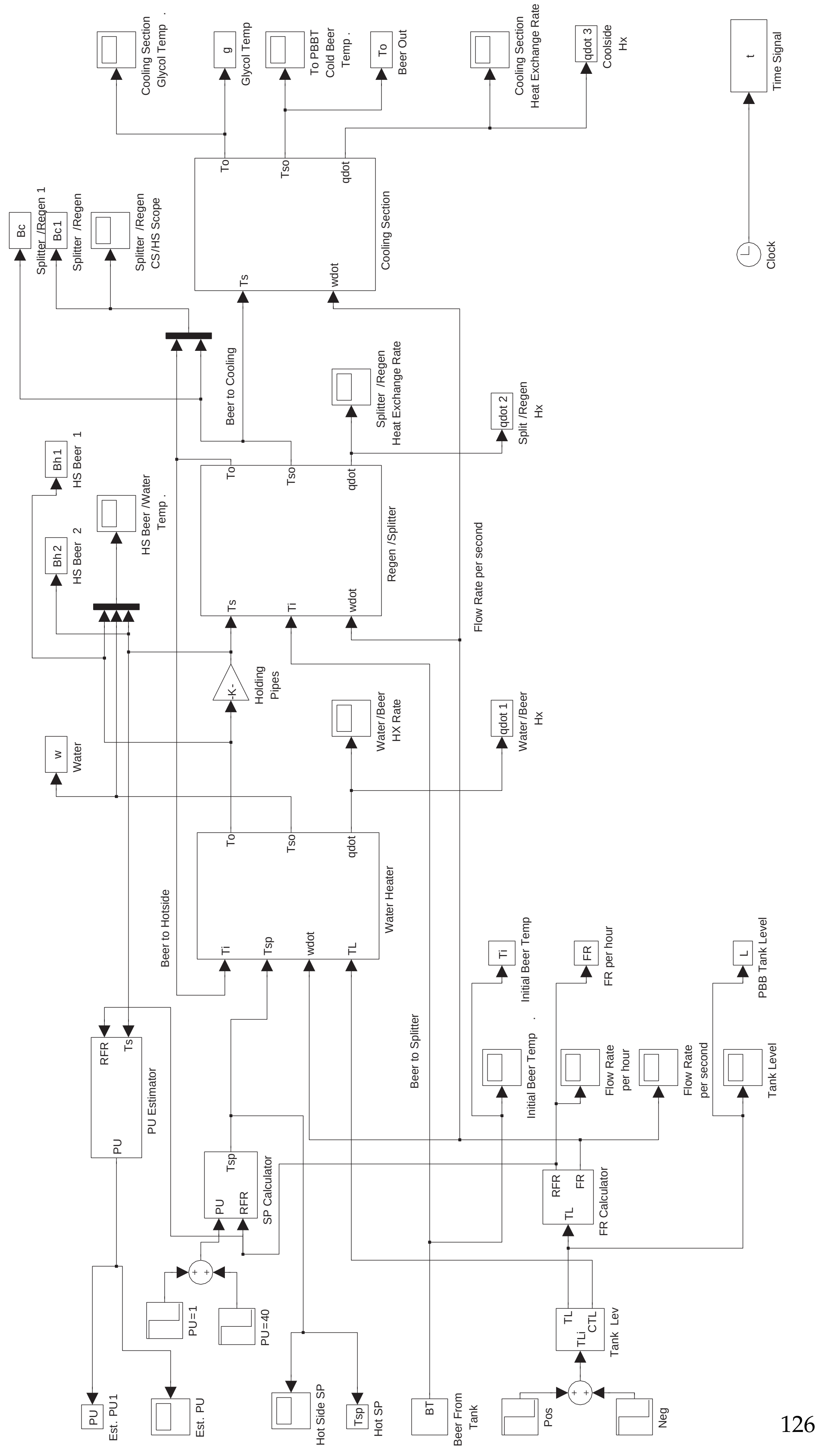


Figure 8.2: Simulink Model of a Beer Pasteurizer

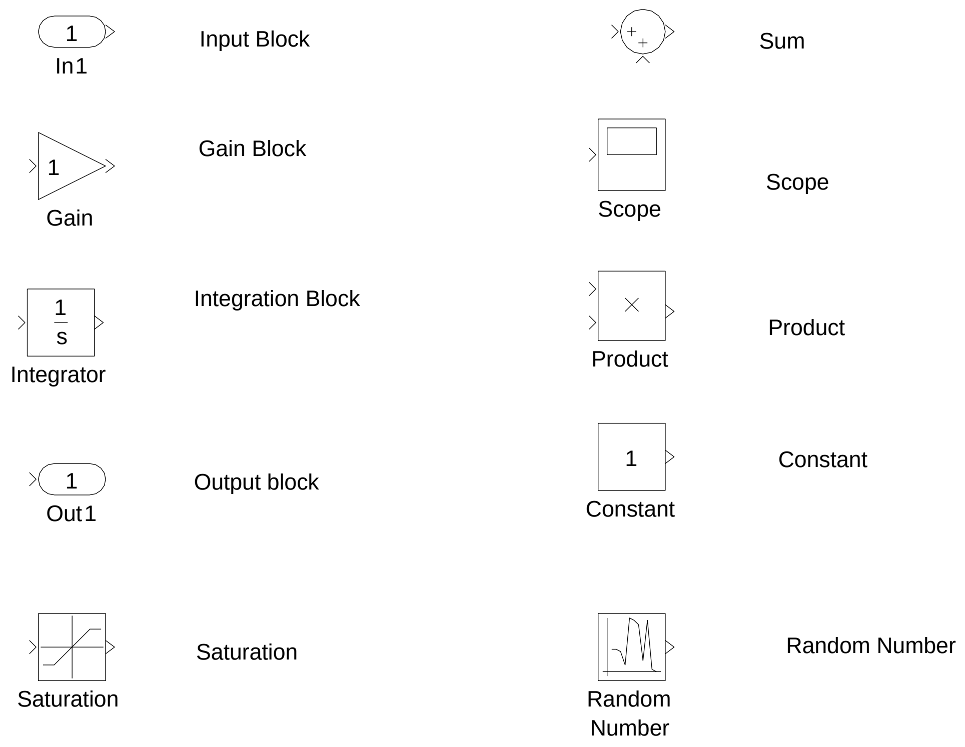


Figure 8.3: Simulink Key

or to plot graphs. The reader should refer to a MATLAB/Simulink textbook – for example [31] – for further information.

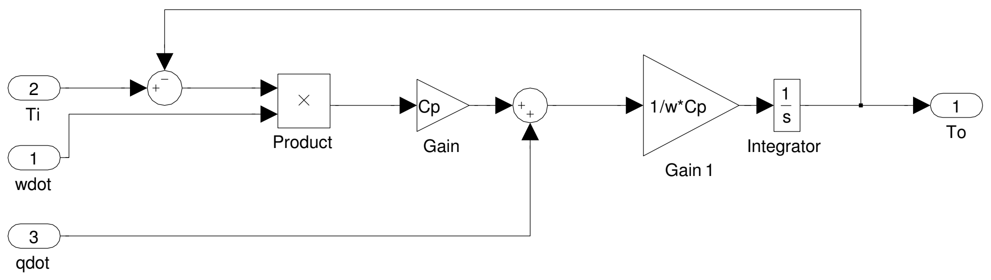


Figure 8.4: Basic Heat Exchange Calculation

To build a model of a compartmented heat exchanger, we first had to construct a basic heat exchange unit to act as a compartment. The basic heat exchange equation is represented in Simulink in figure 8.4. This represents a (partial view) of the transformation of equations 8.1, 8.2and 8.3 solving for the exit temperature $T_{out,c}$, the hot side flow rate $\dot{w}$ and the cold side flow rate $\dot{u}$ respectively. Label substitutions are provided in table 8.5.

To	$T_{out,c}$	Hotside Temperature
Ti	$T_{in,s}$	Coldside (Initial) Temperature
wdot	$\dot{w}$	Hotside Flow Rate
qdot	$\dot{q}$	Coldside Flow Rate
Cp	Cp	Specific Heat Capacity

This basic block is declared as a module in Simulink and subsequently used to build a contraflow heat exchange mechanism with totals calculated for the overall heat exchange process and heat exchange rate. The overall system is shown in figure 8.5

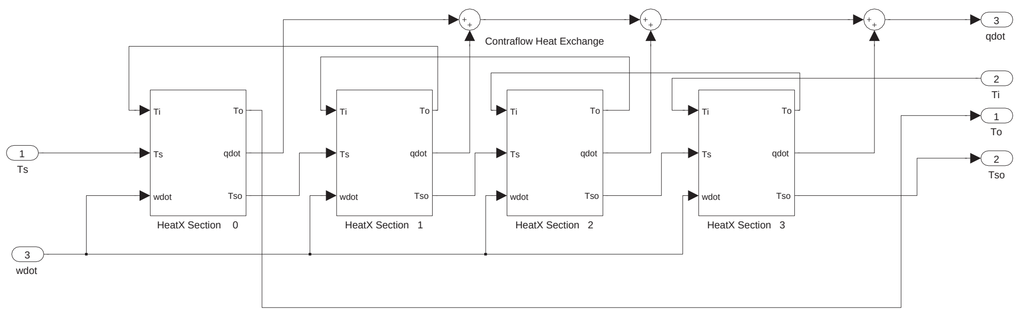


Figure 8.5: Contraflow Heat Exchanger

In each contraflow heat exchange section, Ts is the initial hot side temperature which is set by the heating control system, Ti is the initial cold side temperature, To is the temperature of the heated product and Tso is the temperature of the cooled product. We also show the flow rate $wdot$ and the heat exchange rate $qdot$ which, along with the other outputs is totalized over each section. Note that the heated output temperature To becomes the initial input temperature Ti for the next section and the cooled product output temperature Tso becomes the heated input product To for the next section, creating the contraflow.

The module breaks down into four sections as follows. First, we have dual heat exchanges each controlled by the same flow rates and heat exchange calculations. The first section is shown in figure 8.6.

The basic heat exchange calculation is repeated for both the hot and cold sides and is identical for both – see figure 8.7.

The rate of the heat exchange $qdot$ is determined using U and A as the parameters to the module – see figure 8.8.

The end sections, heating and cooling units, of the heat exchanger respectively are shown in figures 8.9 and 8.10.

In the hotside section, we simulate the action of the steam heated water by using a PID block to control the water temperature Ts but taking into account the temperature set point Tsp (which is calculated elsewhere), the beer temperature To following the contraflow heat exchange. We found that the disturbances in the tank level TL had an exaggerated effect on the model and introduced a compensating linearization and a reduced level of "disturbance" to compensate.

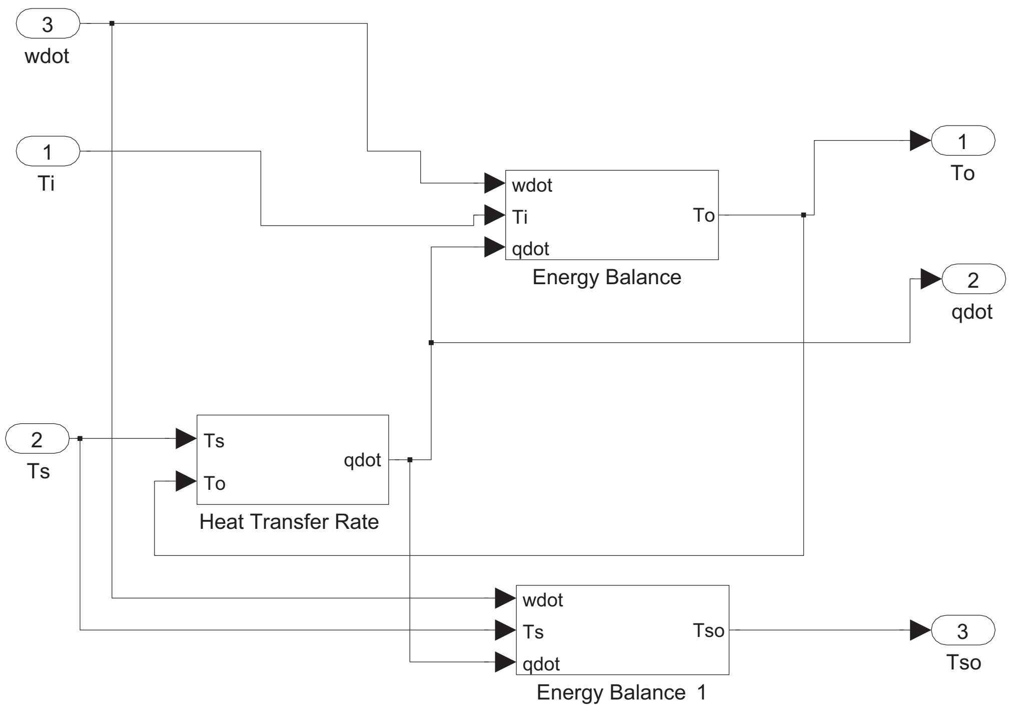


Figure 8.6: Dual Heat Exchange in Contraflow

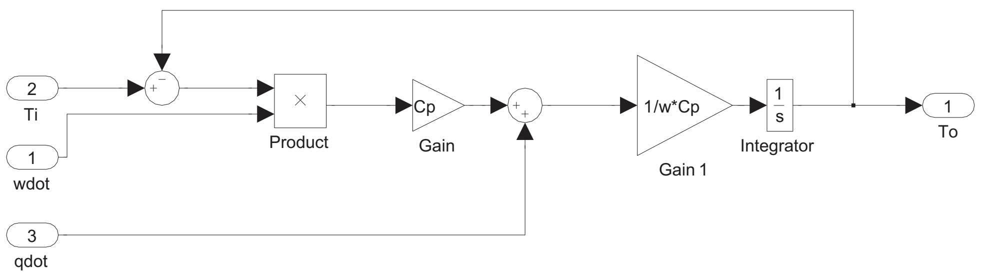


Figure 8.7: Basic Heat Exchange Calculation

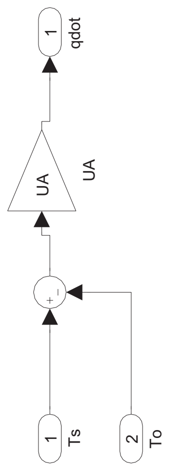


Figure 8.8: Calculate Rate of Heat Exchange

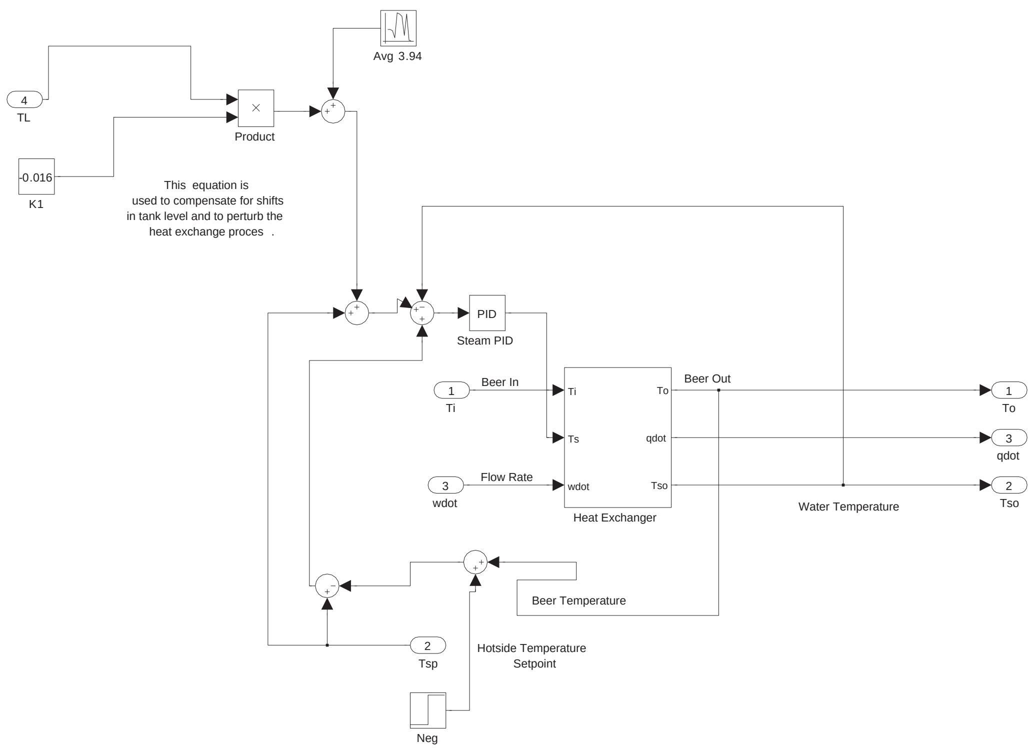


Figure 8.9: Heating Section

The coldside section uses a similar approach to simulate the action of glycol in cooling the product. "Disturbances" were introduced into the exit product temperature and the heat exchange in the refrigeration of the glycol for the coldside PID to handle.

It should be noted both the heating mechanism and the cooling mechanism were only intended to approximate the actions of heating and cooling in the model as a full representation would have required the addition of further heat exchange units and PIDs to fully model the action of these mechanisms.

Other inputs to the process are generated in a reasonably straightforward way. The initial beer temperature is around 10^{o}C and is generated using a random number generator shown in figure 8.11.

The tank level was also generated randomly as shown in figure 8.12 allowing the tank level TL to wander between its maximum and minimum values while avoiding the requirement to create a full simulation of the tank level control mechanism.

The tank level is subsequently used to calculate the flow rate shown in figure 8.13. The details of this calculation are provided in chapter 8.

It should be noted that we calculate the FR in hectoliters per minute and subsequently calculate the real flow rate RFR in hectoliters per second which is the value which is used to calculate the hotside temperature set point based on the required number of pasteurization units PU. This calculation is shown in figure 8.14.

This calculation is derived from 8.4. The final inputs to the model were per-

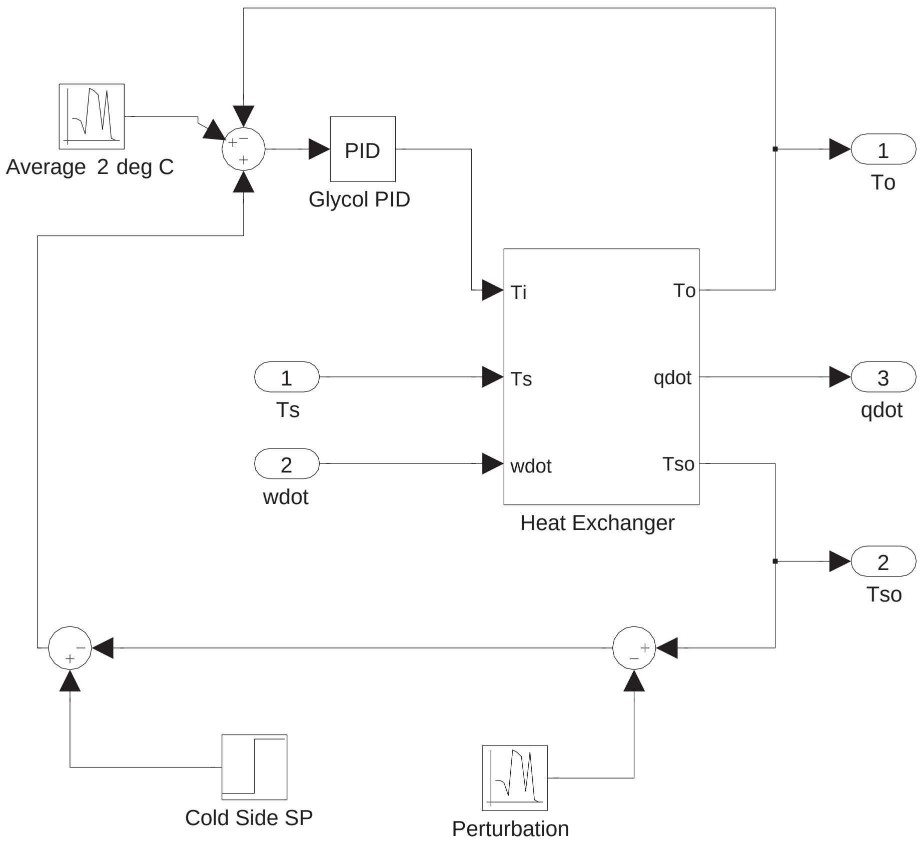


Figure 8.10: Cooling Unit

turbations simulating adversary action which are shown in Simulink blocks in the overall system model which alter tank level TL settings and pasteurization unit PU readouts to conceal alterations in the production process. These inputs were timed to occur partway through a pasteurization run.

The number of pasteurization units PU produced is calculated based on the exit temperature of the beer from the heat exchanger Ts (with a minor downward adjustment to allow for heat loss in the piping) as shown in figure 8.15.

The other outputs of interest are the exit temperatures on the hotside To and coldside Tso of the beer and the heat exchange rate *qdot* for the total heat exchange along with the hotside setpoint Tsp. All of these are measured using the Simulink *Scope* blocks shown in the complete model in figure 8.2 and their use is explained in chapter 8.

8.5.1 Model Validation and Application

The model as a whole as described is intended to allow the experimenter to simulate the action of the pasteurizer and, subsequently, to interfere in that action by altering key values while concealing the fact of alteration from operators with access to the normal output gauges. In chapter 8, we show how this limitation on

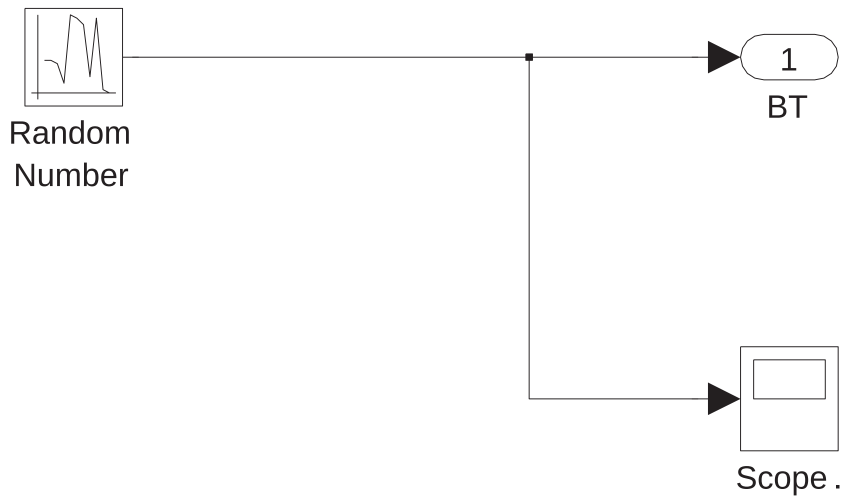


Figure 8.11: Initial Beer Temperature

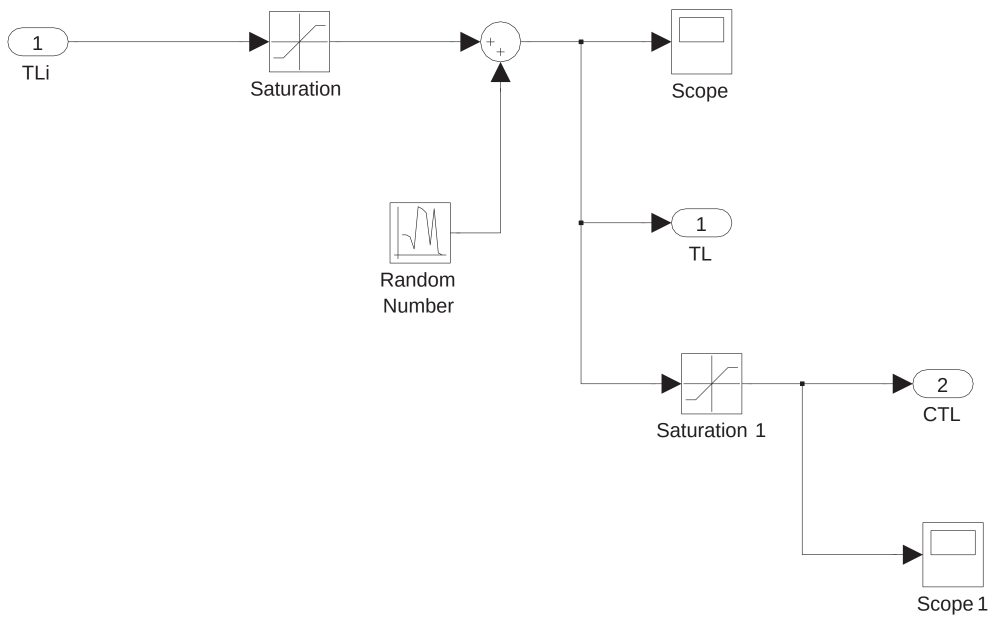


Figure 8.12: Tank Level Generation

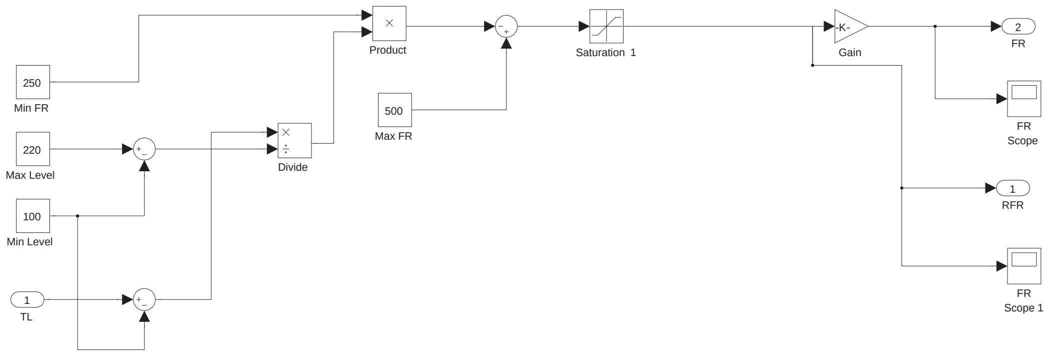


Figure 8.13: Calculate Flow Rate based on Tank Level

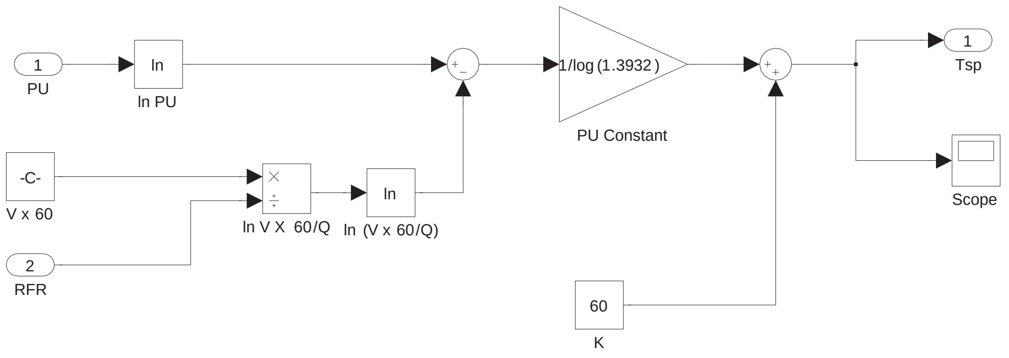


Figure 8.14: Calculate Hotside Setpoint

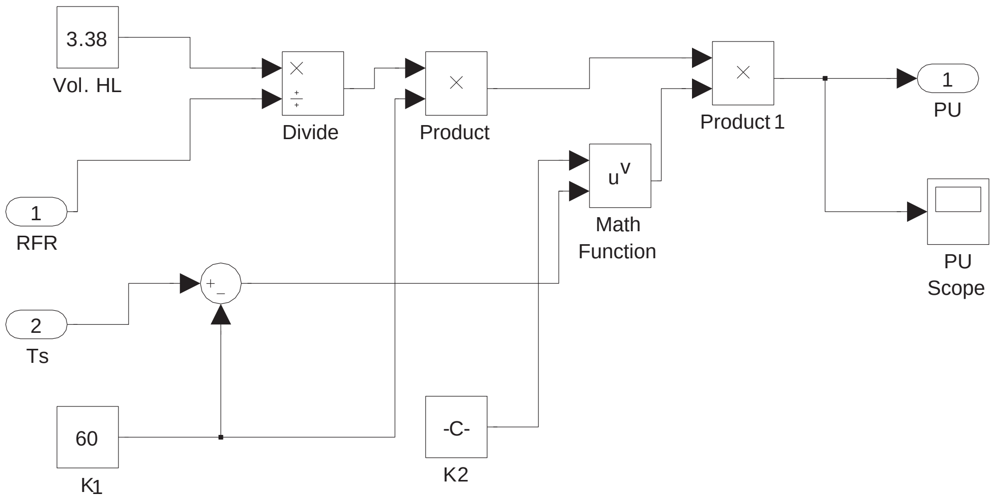


Figure 8.15: Calculate Pasteurization Units

operator knowledge can be overcome by using additional gauges to validate the model's output, assuming we can communicate these readings safely to the operator. Obviously, the solution to this secondary problem is found in chapter 5 using the proposed traceback protocol in full encryption mode.

The model was validated by Brian Furey, a control engineer at Diageo PLC, and the beer product used in the example is a globally famous Irish stout. Discussions with the engineers at Diageo PLC made it clear that the sensitivity of the product to pasteurization plays a large factor in whether or not such an attack would be successful. The non-linear nature of pasteurization and of product spoliation are both factors in this process.

8.6 Proxy Discovery

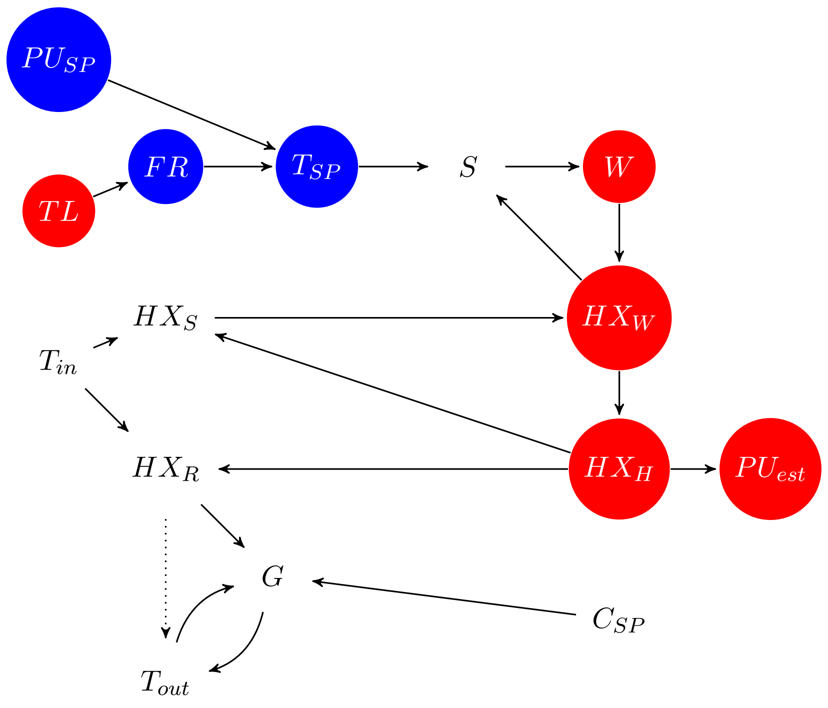


Figure 8.16: Adapted Functional Causal Model

We use an adapted functional causal model of the pasteurizer to visualize significant relations and sensor values under specific attack conditions [99] as an aid in uncovering potential substitute readings. This is a technique which we have found to be suitable for analyzing small-scale configurations. For large-scale configurations, an algebraic transform of the same technique may be used.

Let $\overrightarrow{G} = V(\overrightarrow{E})$ be a graph. Let each $v \in V$ represent a characteristic aspect of pasteurizer functionality, e.g., water temperature. Let each $\overrightarrow{e} \in \overrightarrow{E}$ be a causal relationship between distinct pasteurizer characteristics. We assume that we may ask conditional questions about the state of a node $v \in V$ where it is directly or transitively linked to a node $u \in V$, except where v is also linked to a *dominant* node w. Where a dominant node exists, its state determines the value of all slave (or *invariant*) nodes which are tail adjacent . All other relations to invariant nodes are represented as dotted lines. We assume, but do explicitly show in graph form, that each node's state is subject to unmeasured disturbances which create a probability distribution over node values. Under attack, these values will be perturbed, but the perturbation will be concealed by the attacker by manipulating computational states or using process "noise". Clearly, the attacker will conceal the value of all nodes directly implicated in determining the success of pasteurization. Supervisory access will also allow the manipulation of certain nodes. Potentially manipulated nodes are shown in blue, whilst probabilistically concealed nodes are colored red. All colored nodes belong to a the set of "covered" nodes whose values may not be knowable under attack.

Figure 8.6 represents the pasteurizer under attack. Here, PU_{SP} is the pasteurization unit set point, TL the current tank level, FR the flow rate, T_{SP} the hot-side temperate set point, S the steam temperature, W the water temperature, T_{in} the initial product temperature, HX_S the product temperature after heating in the splitter. HX_W is the product temperature after being heated by the water, HX_H the product temperature at the end of the holding pipe and pre-regeneration. PU_{est} is the estimated PU value. HX_R is the product temperature post regeneration, T_{out} the product temperature at the end of the process after glycol cooling, and finally C_{SP} is the cold-side set point and G the glycol temperature.

We observe temperature and flow rate determine the pasteurization value (see equation 8.4). We also observe that these values are central to material and energy balances in the heat exchange (eqs. 8.1, 8.2 and 8.3). These observations suggest that it may be possible to estimate pasteurization values from heat exchange performance and vice versa. Finally, we note that the heat exchange output values are *uncovered* at HX_S and HX_R, potentially enabling their use as potential measurements for determining the success of pasteurization. We proceed to show in section 8.7 how we could use supplementary sensors to detect an attack.

8.7 Experiments

Based on assumptions about attacker capability and assuming no insider collusion, there are three possible attack strategies w.r.t our specimen process:

- Lower the PU by spoofing a lower tank levels, thus increasing flow rates relative to temperature, using negative error values
- Lower the PU by dropping the water and hence product temperature relative to flow rates, using positive error values, or equivalently by resetting the PU set point
- Combine these tactics in a single attack

For the first two attack strategies, the attacker is required to disguise all relevant sensors so that they appear to show consistent values. In the third strategy, the attacker may omit to disguise the hot-side temperature as a stepped approach to jointly lower water temperatures and raise relative flow rates can be concealed in process noise, i.e., the product temperature remains constant, but pasteurization is still degraded as a process. We therefore need to show that we can use the proxy measurements we have identified in to estimate both the flow rate and the temperature and hence the pasteurization units.

As a preliminary step we show that there are distinct temperature to flow rate ratios for each PU. Solving equation 8.4 for temperature [82], we plot temperature values against flow rates for PU values of 40 (nominal value) and 20 (the fail or *divert* value) and determine the PU value for 1^o C alteration downwards for the nominal value of $PU = 40$ which represents a significant loss of quality, i.e., we also solve for $T - 1$ with the result $PU = 28.7$. The results are shown in figures 8.17.

The temperature of the cooled (pasteurized) product leaving the regenerator tank allows us to estimate flow rates. To show this, we plot this temperature for four distinct tank levels $T = \{120, 140, 160, 180\}$, representing distinct flow rates, against the nominal pasteurization rate (40), the divert value (20) and the quality loss value (28.7). The distinct banding of temperatures for the cooled product (color coded by flow rate) temperatures shows that the flow rate and hence the tank level can be estimated using this value – see figure 8.18. Confidence intervals are estimated at ± 10 HL/HR. It follows that we should be able to detect the first attack tactic used to alter flow rate by spoofing false tank level readings.

We now show how a concealed alteration in temperature may be detected. Setting the tank level at $TL = 180$ to lock in the flow rate, we show three different runs with distinct pasteurization profiles as before. Given the non-linear relationship between pasteurization levels, these temperature differences are on average sufficiently significant so as, in combination with flow rate estimation to allow calculation of process success with greater than 3σ confidence. Similar results are found for other flow rates. Hence, we can determine what the current PU level is modulo approximately 4 units.

Finally, we show how we can detect transitions concealed by signal "noise". We assume an attack where we drop the PU value *gradually* using a combination of lowering water temperatures and raising flow rates in an effort to hide in the process signal. We set the tank level to 220 and drop it in stages to 172, altering the flow rate upwards. We drop the pasteurization target from 40 to 28.7 (equivalent to quality loss) by altering the water temperature error signal or PU setpoint. We see in figures 8.20 and 8.21 that product temperatures do not vary from their mean value, while PU rates drop significantly in the same period. Obviously, the attacker could continue this process until the divert or abort values were achieved.

In figures 8.20, 8.21 and 8.22, we see the difference to the heat exchange profile as a result of these "concealed" adjustments. In fact, this result provides a greater contrast in heat exchange profiles compared to other attacks as both the lower and upper product temperatures alter simultaneously to create anomalous "steps" in the heat exchange profile. It follows that we can estimate (even control) actual pasteurization rates from this profile. We estimate that we could achieve confidence levels of ± 4 PU, assuming we can achieve precision of $\pm 0.5^o$ C in hot-side temperature values and ± 10 HL/HR in flow rate estimations.

In conclusion, our simulation shows that by placing additional sensors in suitable locations in a heat exchange mechanism and by paying attention to differentials in the physical process, we can detect alterations in the heat profile which may be indicative of an attack, even where the attacker is manipulating sensor signals

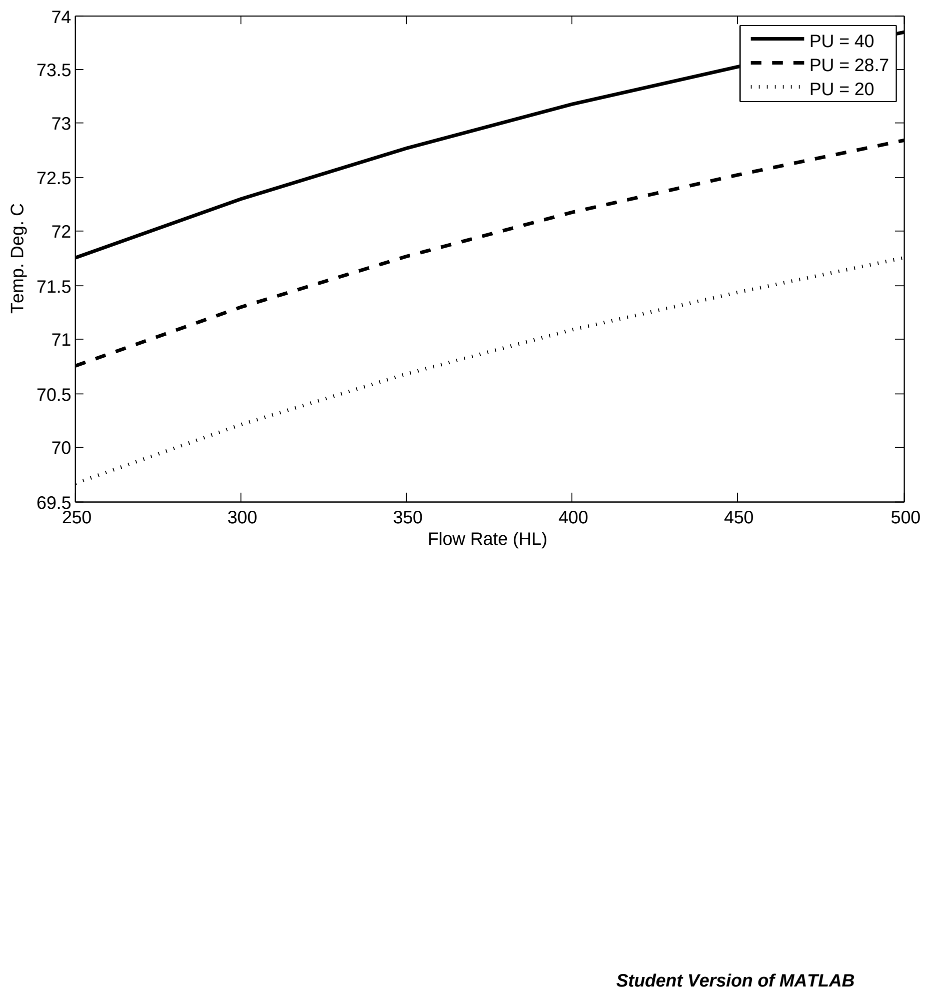
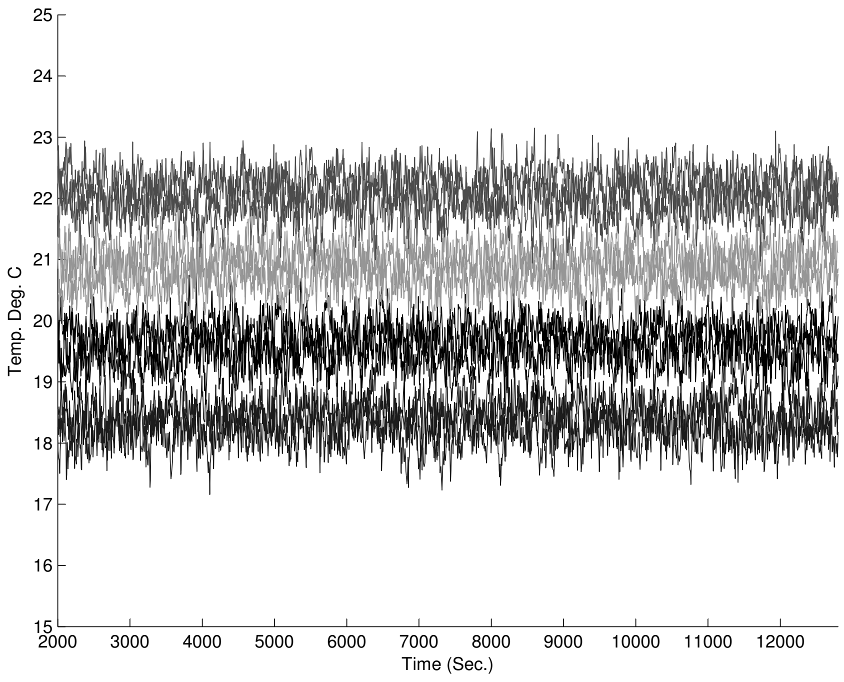


Figure 8.17: PU Profiles for Flow Rate v Temperature



Student Version of MATLAB

Figure 8.18: Cooled Product Temperature as a Proxy for Flow Rates

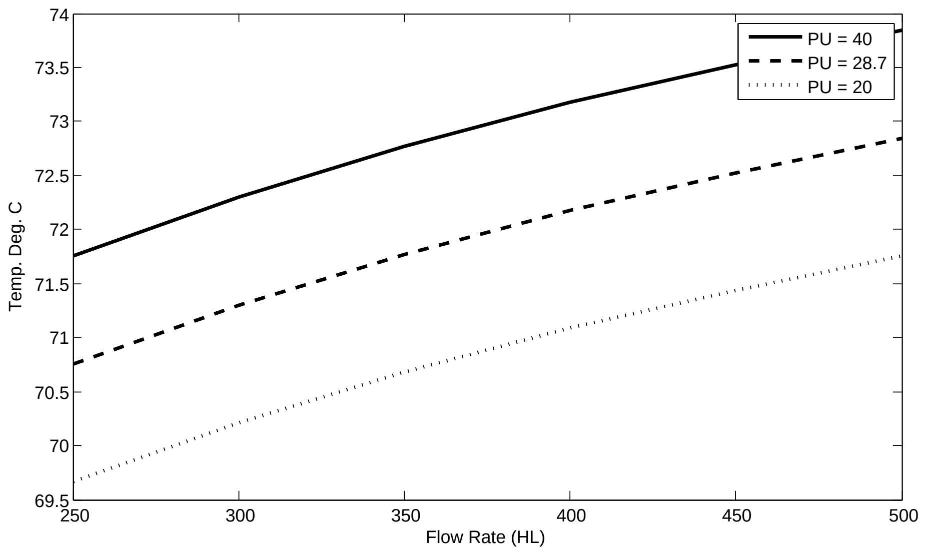


Figure 8.19: Nominal, Degraded and Divert PU Values where Tank Level = 180

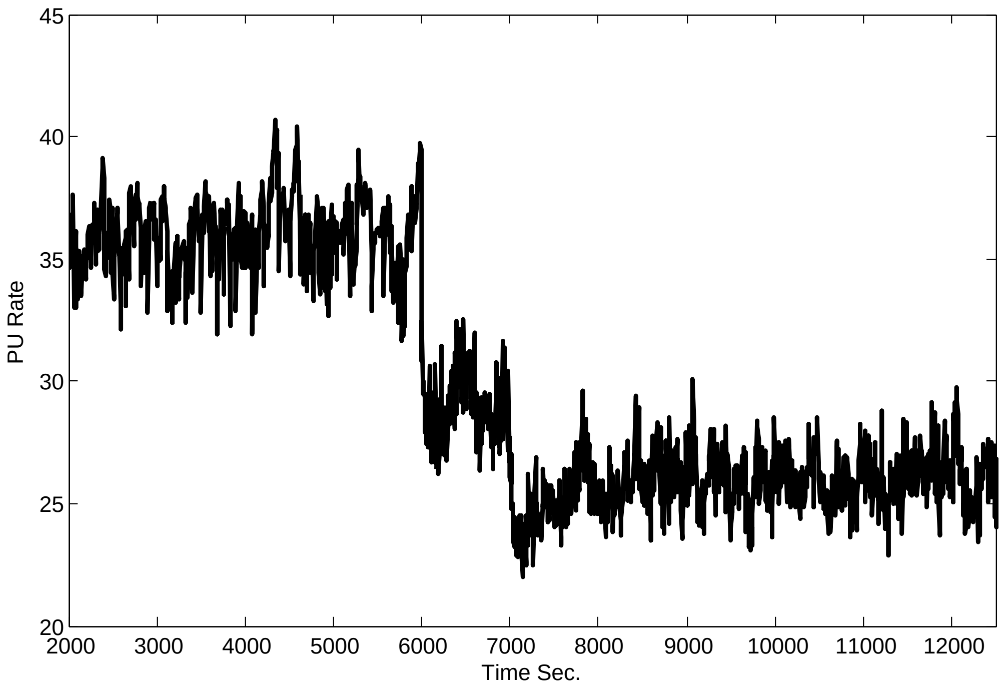


Figure 8.20: Hiding in Signal “Noise” - Pasteurization Rate Alteration

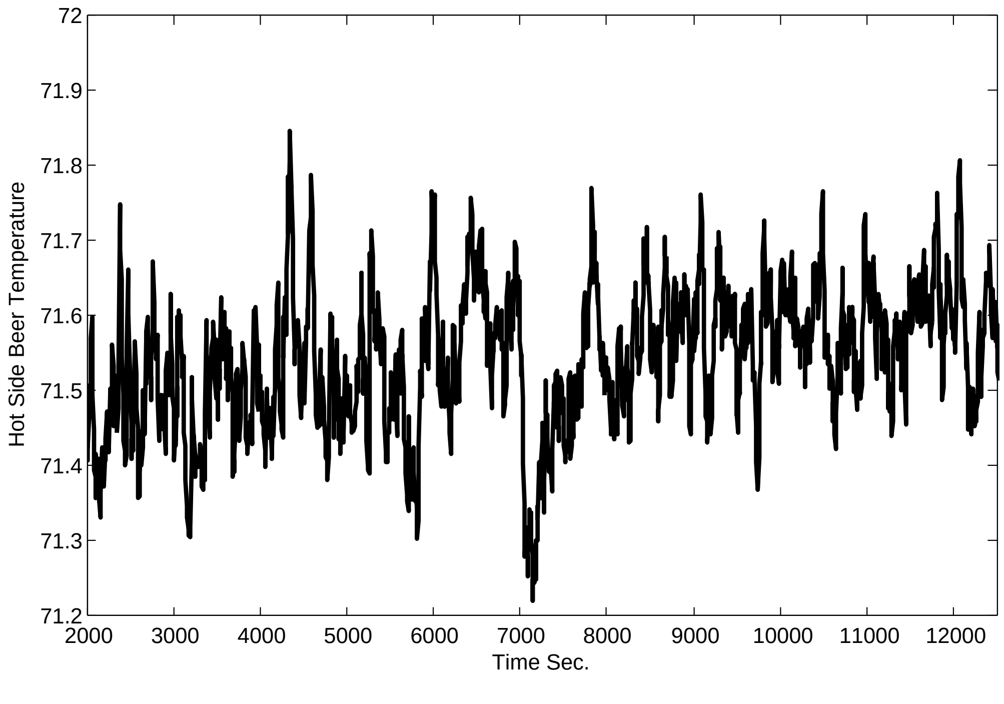


Figure 8.21: Hiding in Signal “Noise” - Product Temperature

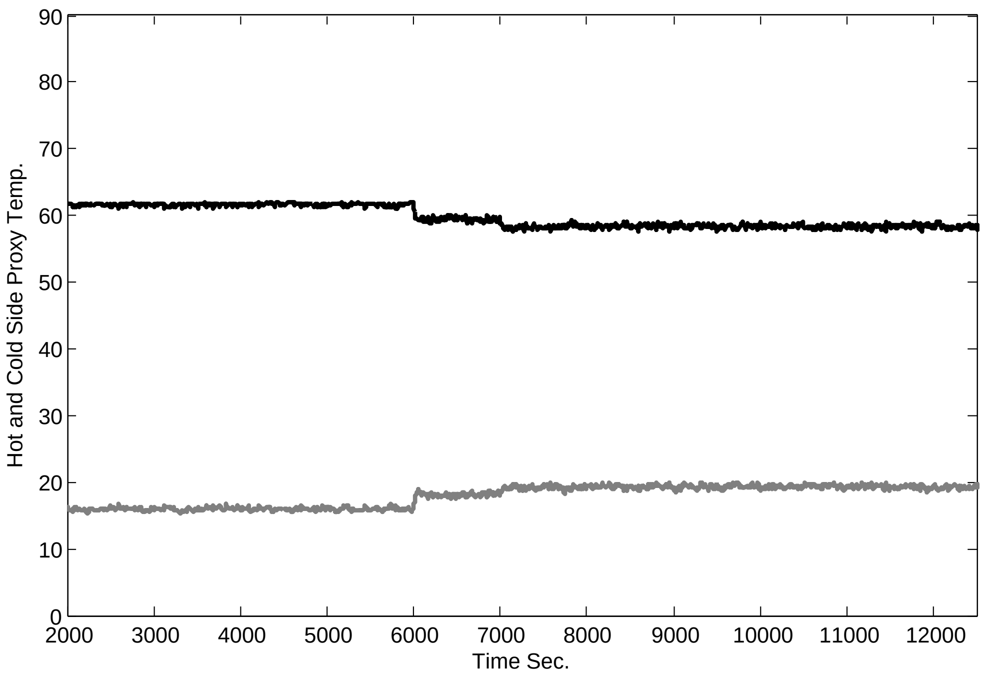


Figure 8.22: Detecting Flow Adjustments in "Noise" using Hot- and Cold- side Temperatures

and relying on the "noise" in the process to cover up his tracks. We noted, initially, that we wish to avoid using a large number of additional sensors for cost and licensing reasons. Our research here indicates that the use of two sensors would provide suitable information. Whether this is a true minimum would require additional work and may ultimately depend on the kind of plant we are dealing with.

8.8 Summary

In this chapter, we selected the example of a heat exchanger as a key process in many industrial systems. We used an exchanger associated with a pasteurization unit. While not strictly part of a critical infrastructure, many similar units would form part of such systems. We show how an attacker could obfuscate his presence in such systems by hiding alterations in the "noise" of the system. Consequently, we describe how the use of additional sensors allows us to determine the state of a plant independently of its operational signals provided we can securely communicate this information to the operator. Hence we have at our disposal an *expected behavior model* for the plant based on processing engineering methods as well as (from previous chapters) for the host and network.

Chapter 9

Conclusions and Future Work

Cat's motto: No matter what you've done wrong, always try to make it look like the dog did it.

Unknown

9.1 Preamble

In this chapter, we summarize our work and our contributions, identify any current limitations, and draw conclusions. We also discuss the future development of our research.

9.2 Research Overview

Our research was motivated by the issues raised by the problems of malicious software attacks on critical infrastructure systems. The growth in sophistication in malicious software techniques renders it increasingly difficult to observe such attacks which increasingly disguise themselves by posing as legitimate activity on systems, indistinguishable in terms of process behavior or protocol usage until after the fact, or making use of sophisticated obfuscation techniques. At the same time, the existence of multiple possible linearizations of state due to the distributed nature of such systems, at both macro- and micro- level, increases the difficulty in detection. We seek to determine the requirements for malicious software detection on such and what potential commonalities, whether advantages or disadvantages, arise in tackling such problems in distributed systems in general. We assume a "perfect" adversary who makes no egregious protocol errors and, hence, may only be detected by observing anomalies in data artifacts.

Before tackling these issues, we required to produce a formal adversary capability model (threat model) which would allow us to reason about adversary actions and their effects in distributed systems. We note that such a model may be analyzed at process level or we may apply higher order reasoning regarding agency in systems. Each process or (agent) goal may be imbued with one or more adversary capabilities which, in combination, enable it to act maliciously in systems. Our capabilities are based on the Dolev-Yao model but make different assumptions about communication and connectivity in infrastructures, which force the insertion of malicious processes or the subversion of current processes into the system by the adversary, creating a distributed foe. One or more processes exercising these capabilities may be used to construct attack scenarios. There equally existed friendly processes or agents which can also interact with the system, creating a highly complex, dynamically mutating environment.

We instantiated this model, using two closely related π-calculus variants depending on whether we wished to analyze agent or process action, taking advantage of the fact that such calculi, in the past, have been used naturally to model the action in distributed (and mobile) systems – the latter, in particular, being subject to change during a system run – and hence lent itself to our adversary definition. We further made use of functions which allowed us to represent unknowable effects (from the point of view of the operator) such as process subversion or the impacts of data manipulation attacks. This form of impact analysis allows us to demonstrate the effects of attacks, including multiple attacks and for transitive and recursive effects, and to formally prove, when necessary, that an attack may detected or prevented by some method we wish to employ. Our instantiated model represents a meta-model for each of our detective approaches – that is, we can formally prove each approach within the applied π-calculus system we apply, though, in some cases, it is trivial to do so.

We subsequently focused on developing three detective methods. Each method is characterized by the *context* created by the system – that is we made use of the fact that, in distributed systems, data artifacts in one part of the system have relations, which may be trivial (for example, equivalence) or may be complex (for example, ordering or cause and effect). These relations may also be time-bound in the sense that they are held concurrently and may alter over time in response to messages. We refer to this, therefore, as a δ-congruence which fills the δ-context. . Each method is also, as a result, characterized by the use of a concurrent or near-concurrent observation and monitoring technique.

The first method made use of techniques based on IP Traceback (though not necessarily bound to that protocol) to show how we could make use of network nodes already present in the system to provide us with potentially clean copies of messages between the plant and the operator which could be used to detect where a subverted process was located. We also adapted this method to show how software agents acting on behalf of the operator could inform him which messages to discount and which to consider when determining system state under a malicious attack. We formally proved each case using the π-calculus variants. The contextual relationship, in this case, was equivalence and the observation method was the exercise of the protocol itself.

The second method applied an analogue of the first method to observing different layers of abstraction within a multiprocessor host system from the point of view of different observation threads associated with different CPUs. The "observers" exchanged information with each other at random intervals over an observation round and noted any inconsistencies, initially, as an alteration in state. As before, this approach could detect persistent inconsistencies in the return values of function calls to key operating system components used for reporting state which would clearly indicate the presence of deception techniques in the system and we demonstrated this technique using a "proof of concept" software module in a Linux multiprocessor host. We augmented this approach by associating alterations in state with a negative specification of system behavior which would identify a sequence of changes at the semantic level that was inconsistent with the expected behavior of the system at a minimum and could, potentially, in some cases, provide a clear indication and identification of a malicious software attack. The advantage of using this cause and effect analysis was that it provided a view of action in disparate, but causally connected, subsystems of the operating system. This method would be particularly effective against DKOM attacks as well as process alteration.

The third method we introduced made use of a unique feature of process control systems which is the ability to make use of the kinetic information supplied by such systems by introducing additional sensors on secure and independent communication channels. Hence, if an adversary was successful in undetectably subverting a control unit, we could still show by examining the relationship between physical signals inconsistencies in plant operation which could potentially indicate the presence of malicious activity. We demonstrated the validity of this approach using Simulink to model the relationships in the pasteurization system we used for our example.

In each case, we sought to minimize computational complexity to fit to the low performance and capacity ICS environment and to render our approaches highly resilient to adversary attack.

9.3 Contributions

Our work makes the following original contributions:

- A formal adversary capability model, adapted from Dolev-Yao, suitable for dealing with the action of malicious processes/agents with a suitable variant π-calculus instantiation
- A traceback technique for detecting the source of integrity attacks in networks
- A causal model and associated observation mechanism for detecting kernel-level rootkits
- A “proof of concept” approach to identify aberrations in plants under control, using basic systems engineering techniques

9.4 Discussion

We discuss the nature of our contributions.

9.4.1 Formal Adversary Capability Model

Formal threat models are primarily used in relation to cryptographic analysis [74]. Formal methods have also been used to analyze the ability of attacks to penetrate systems – for example, [130]. Our focus, however, is, at least, initially, on analyzing potential adversary behavior and the possible impacts of attacks. We believe our model allows us to divorce ourselves from making unnecessary assumptions about attackers' goals, which exist in some other methods such as attack trees [75], and a narrow consideration of current system vulnerabilities (provided, for example, by the results of penetration testing) and instead consider the results of potential interventions. Although we frequently start from the position that the attacker has already penetrated the system and constructed an attack, based on the capabilities attributed to him, we are also in the position to consider impacts which depend on the *non-deterministic* success of penetration attempts. Hence we gain the capability to build up a set of possibilistic outcomes and use these to analyze the success of security protocols, system defenses and also build up a deeper comprehension of defensive priorities.

The choice of process over agency models depends on application. For example, it would be wasteful to define an agency model for the definition and proof of the working of the security protocol in chapter 6, but clearly an advantage to be able to consider the interaction of such protocols with adversary behavior in the consequent. Hence the ability to deal with agency in the model offers the opportunity not just to gather further insights into potential adversary behavior, but to meaningfully consider a range of interacting components in the system, including mobility, and also externally, by considering potential updates to agents' actions. This further augments the ability to analyze attack/defense possibilities in the model.

The adversary capability model we provide is, to some extent, over defined out of convenience. A necessary and sufficient model of attack behavior can be built while eliding the indistinguishability assumption P, learning behavior LB and COOP. The last two capabilities can be built from decision making D-MAT and the M* capabilities, while the former is an assumption adopted for our research. It is also over defined in the sense that we dealt only with integrity attacks during our investigations and it could be argued that a smaller number of capabilities is needed to define such attacks. However, the opportunity to open up further research into the use of formal models to tackle a wider set of problems than the security of key exchanges was a welcome one.

One weakness is that we elide from the definition the use of penetration methods. Penetration techniques are assumed and applied non-deterministically. This is ultimately not satisfactory. Clearly, other possible models of attacker behavior still wait to be created – including extensions of this one – which capture the complexity jointly of penetrating and impacting on a system.

9.4.2 Applied π-calculus

The algebraic approach takes inspiration from [74], but is a natural choice, given the arbitrary and dynamic nature of the environment we face, and the tasks we set ourselves with regard to defining and proving security protocols and techniques. In particular, the ability to deal with mutating processes via our syntactic and semantic extensions made this choice an easy one. We don't presume it is the only choice and we look forward to seeing other researchers develop different options. Candidate methods, in our opinion, include the use of Petri Nets, Bayesian probability graphs and functional causal models [99]. Our current method is limited to considering ordering over events. We can extend the range of attack behaviors we can model by using timing and probability characteristics associated with processes, channels and variables.

The most obvious limitation to our approach is the lack of availability of a suitable tool for model checking, automating the burden of proof. This is not to de-value human-centric proof techniques, but process algebraic methods are, in general, cumbersome to execute manually and prone to error. The most valuable contributions, in general, however, stem from the ability to combine computer- and human-based approaches together. Certain kinds of proof are likely to remain outside the scope of automation and there is no doubt that automated methods benefit from the application of thought to optimize the search space In addition, since our research interest was integrity attacks, we have not fully modeled how the loss of confidentiality or availability can be made explicit using our techniques.

Another weakness is that the syntax and semantics require refinement. In particular, we want to be able to use silent functions such as loss or delay (or any adversary function) over processes, channels and names while avoiding summation, as that can exclude an ever present possibility in the operation of a process or a communication function. A limitation is also introduced by the fact that, at this stage, we can only consider the order and not the timing of actions. In the context of a real-time system response, this is a serious limitation.

9.4.3 Traceback Techniques

As with formal threat models, the concept of traceback techniques has a long history with the primary application being attack attribution during (distributed) denial of service attacks. Our twist on the technique is not to introduce any novel methods, but rather to apply known approaches to tracing the source of integrity attacks in systems. Hence we open up a potentially large area of research into using such protocols for this purpose. It equally raises the question as to whether other areas of security – such as tracing loss of confidentiality – could also be tackled similarly. In addition, even for our application, it could easily be argued that our choice of using a messaging protocol introduces potential issues into performance sensitive ICS. But there is a vast range of other techniques in the IP Traceback space which could be considered and adapted to tracing integrity loss in distributed systems and ICS in particular. The other obvious limitation to our approach is that we have only produced a theoretical model of its application. We have not shown our approach working in relation to a real-world SCADA communication protocol. There is no doubt that this, in turn, could force compromises on our methods and lead to applying other techniques to its application. In principle, this approach is also expandable to other forms of distributed systems where paths are well-defined and the source and destination of messages is known.

9.4.4 Modeling Cause and Effect in Distributed Systems

The creation of the signature and axiomization of the process algebra we use in chapter 7 is a reasonably straightforward exercise[1]. Again, the difference lies in the application of the technique to modeling cause and effect, rather than the direct interaction of processes. A communication event between $\bar{a}$ and a could represent a large subset of actual system events $a \rightsquigarrow a' \rightsquigarrow a'' \ldots$ transitioning multiple processes and process subsystems. Nor is it necessary for the processes to belong to

[1]and often inflicted on students

the same subsystem – just that they are related by changes in state in a consistent way. So, as in the examples provided in chapter 3, an alteration of the AVC cache in SELinux can be related to changes in the access to privileged commands of a process without either the cache or the process being linked by process interactions and processes can be removed the the *task_struct* list without being removed from the process table, thus producing inconsistencies which could be potentially detectable by our approach [93]. The approach is also applicable to network systems.

The concurrent observation mechanism which we presented as a "proof of concept" puts a unique spin on the problem of defending intrusion detection systems from attackers. In nearly every other approach, the emphasis is on obscuring and, if possible, isolating monitoring mechanisms from attackers – for example, by virtual machine introspection [45] or the use of co-processors [127] – with a concomitant loss of granularity and considerable work being required to reconstruct states. In our approach, we place the observation mechanism in plain view of the attacker and in the heart of the system, making the re-construction of state trivial. We rely on numerical superiority, dissimilarities in measuring techniques and self-observation to make it difficult for an adversary to bypass the the approach. Obviously, we could equally resort to the use of co-processors, VMI techniques or the use of off-motherboard processing systems such as graphics and network cards to re-locate the loci of observation or simply to augment the approach by having both intrinsic and extrinsic observation methods working in conjunction with each other.

The limitations on our approach lie in the small number of multiprocessors on current ICS hosts which does not allow us to take full advantage of the multiplicity of observation threads – though this could be tackled using co-processors to supplement the kernel threads. We also do not at this stage tackle the construction, distribution and protection of monitors which would collate observations and identify rule violations in terms of expected system behavior. In fact, there is considerable work to do in this area to create suitable algorithms to enforce the detection of out of specification behaviors using monitors, in particular, around the dealing with missing and out of order data and the timing of events. We have really only sketched the requirements for this.

This mechanism probably represents the most sophisticated form of observation in our research and is, appropriately, tied to the most sophisticated approach to modeling system behavior. But there are clear limits on our ability to analyze such behavior . Applying our techniques even to a consideration of operating systems requires considerable and intimate knowledge of such systems to allow the appropriate placement of sensors [9] as well as (see section 9.4.4) determining which measurements to take and compare and with what frequency. The last point raises the issue of needing to fine tune the mechanism through further experimental work to determine optimal probabilities and timings for observations which we have not undertaken. There may be an element of specificity over these with regards to application or environment which could limit the application of the technique further, at least, with regards to optimizing it.

9.4.5 Augmenting Sensor Measurements

The final technique we use is firmly founded in engineering disciplines in ICS, again opening up research possibilities into using other techniques to model non-linear systems and exploring more sophisticated models of expected behavior in kinetic systems, such as the use of state estimation techniques [64], as well as being able to combine this knowledge with the information provided by the kind of techniques used in chapters 6 and 7.

We readily acknowledge limitations to this approach. Limitations may be introduced by physical constraints – for example, w.r.t. sensor emplacement (this is a common issue in process control) [13] reducing confidence in results. Distinct processes will be associated with different perturbation levels, reducing (or increasing) confidence levels. However, in most cases, even the process models used to set up the control systems are limited in precision and tuned based on experience rather than physical, chemical, etc. models. We are ultimately limited, therefore, by the precision of those models.

9.4.6 Context-based Anomaly Detection

We began our research by introducing the concept of context-based anomaly detection. The advantage of dealing with distributed systems is that multiple copies of data or related data sets may be available, making for rich pickings to mine in terms of uncovering anomalous behavior. However, the dependence of distributed systems on messages means that we need to note which messages have been sent and received by both processes and take account of delays or loss in communication and take account of these in our approach. This can create heavy computational burdens. In our selected examples, we obviously seek to minimize these burdens and place minimal reliance on requiring missing data to complete our analysis through using probabilistic reasoning.

9.4.7 General Remarks

At such an early stage of researching anomaly detection in distributed systems using context-based models, it is, of course, difficult to draw general conclusions. But some observations (forgive the pun!) may be made.

First, it is clear that context-based approaches in a distributed environment provide a rich seam of potential relations creating the possibility of measuring system behavior in novel ways. Uncovering such techniques put the adversary at a disadvantage because it increases the difficulty of subverting a system if he must consider all possible ways in which his behavior might be detectable – such as the combination of considering kinetic and computational behavior – and all parts of the system in which the data he manipulates could be available (widening the field of discovery and the potential requirement for additional subversion). This latter difficulty is expressed in very practical terms when we consider that, with the exception of our process engineering based method, we introduced large numbers of potential additional observers into the system. Even, in the process engineering example, there exists the potential for considering consistent control behavior on a plant-wide basis.

The approach is data-centric, assuming a protocol perfect adversary, and this could be considered a weakness as, in each case, the attack must already have commenced before detection can take place. Of course, we argue that we have no choice in this, given our assumptions. Still it could limit the applicability of the techniques to a specific subset of systems which are not immediately sensitive to attack impacts. The timeliness of alerting (and subsequent intervention) could also be reduced by the probabilistic nature of observation and, indeed, the resulting possibility of false negatives. This is countered by defense in depth, so that, in an ICS (or other distributed system) we would argue that all techniques we have illustrated (and others) should be used in combination. But, ultimately, we require a means of demonstrating techniques that takes account of real-time factors.

Our work, for the moment, has concentrated on building models with some basic "proof of concept" work. Obviously, practical considerations will need to be taken into greater account as we move forward with developing our methods.

9.5 Future Work

We consider future work for each of the aspects of our research.

9.5.1 π-calculus

With regards to the π-calculus variants, we believe that further work is required to tidy up the syntax and make the semantics more concrete. The scope of application of *silent* functions should be clarified to show which apply at agent, process, summation, function (including communication functions) and name level. The current use of $+$ or $\oplus$ with silent functions which characterized early papers (and has been retained for that reason to avoid extensive re-writing of proofs) is unsatisfactory because such functions may only be exercised once (unless we want to scatter them throughout our equations) until replication which may not be what is required or intended. For example, writing a potentially subvertible process as P_ω instantly clarifies the meaning and avoids unnecessary ink. A satisfactory proof which avoids the use of message priorities of the need for an adversary to act internally to a system could be provided by characterizing channels x_δ, y_δ as subject to delay and assuming (or creating) a function which encapsulates a freshness requirement that names be delayed no more than two δ's, for example. In addition, to fully realize the potential of our approach, we need to extend the capabilities of our calculus to incorporate timing and probability and to create or adapt a suitable model checking toolset. The combination of a process algebra (or calculus) with timing and the use of process engineering techniques together is our ultimate goal, allowing us both to model an attack and its impacts and determine whether an intervention by an operator or system agent can prevent or reduce the impact in real-time.

9.5.2 Adversary Capability Model

The adversary capability model is regarded as reasonably complete. However, there are opportunities to consider other means of instantiating the model such as the use of attack graphs or attack nets. We also note that the model was more powerful than we required for our research. We focused primarily on integrity issues. However, the model would have allowed us to explore loss of availability and confidentiality and these remain as outstanding problems for creating detection methods which, similar to our approach, enable us to trace the source of the attack.

9.5.3 Traceback Techniques

There are several variations of the traceback techniques we used which are worth exploring to see if they enhance our results:

Edge Observation – commonly used in IP traceback protocols, observing an edge between two adjacent nodes may offer further search efficiencies.

Initiation on System Commencement – the node could be used in its own right for anomaly detection by initiating on commencement of system operation, possibly operating with a lower probability p of observation to minimize complexity. This removes reliance from other forms of anomaly detection while making discovery trivial.

Deterministic Burst Mode – another option is, on an external detection event, initiating a deterministic version of the protocol which sends (for a short period) copies of all messages from every node in the subgraph $\mathcal{R}$. Given the persistence of message manipulation, this would lead to a more rapid revelation of subverted nodes, but at a much higher computational cost. This may not be feasible on some performance sensitive systems.

Bypass Mode – the protocol may also be used directly as a means to bypass an attacker's action on the system. In this case, the copied messages which exist before the malicious processes in the direction of communication would be considered to represent the true state of the system and be used instead of the original packets to aid in the estimation of the state of the physical plant. This would again require that the probability of observation was higher than if using the system in detection node.

In addition, there are other traceback techniques which we should consider for applicability to problems of this nature. Indeed, the consideration of other techniques is likely to be forced on us by the limitations on performance and capacity which exist in SCADA protocols.

One aspect we could explore is whether we may also exploit communication between software agents as observers who collate the information to add a further layer of validation. This would be the equivalent to the exchange of information between observer threads at host level. For example, software agents could share partially rather than fully overlapping paths, leading to requirements for collating information between agents. Nor do we consider other forms of message exchange with ICS systems such as management systems responsible for server maintenance. Finally, we could also have multiple (possibly transitive) sources and destinations of messages with different observer agents at each point all communicating via the protocol – although we need a sufficiently large scale system to be able to exploit this properly.

Naturally, all results will need to be backed up by a "proof of concept" exercise covering different kinds of SCADA protocols and purposes to show applicability. This will include consideration of different kinetic systems to determine aspects such as communication timing and probability values for observation.

9.5.4 Stochastic Algebra

We believe there are other applications for the stochastic algebra than the one shown here – or for variants of this approach. In our application, we treat messages between processes as cause and effect, which is sensible – though we may abstract from the messaging mechanism. However, we also have the option of treating cause and effect as though it were a message between systems. For example, in the context of a control system such as a smart grid, an increase in power generation might be treated as a message to other parts of the grid. This mapping of process algebraic systems to physical processes is not without precedent [6].

9.5.5 Host-based Observation Mechanism

We had the opportunity to demonstrate the host-based observation mechanism on a system with only two processors. Ideally, we would wish to explore its use with co-processors [127]. Another strand of research has already identified possibilities for using other peripheral devices with independent processing capacity, using cache snooping [111]. We could also use virtual machines [45] and use of reserved memory spaces [37], extending the principle of cross-validation of observations to create "extra-orbital" observation networks in miniature in association with each host CPU.

9.5.6 Plant Model

We used a basic process control engineering technique to provide us with a model of the plant. Obviously, there are other techniques available to exploit. In particular, we are interested in examining the use of non-linear state estimation techniques such as the use of particle filters [114] which may enable us to do away with the requirement for additional sensors and allow us to explore deviations in model behavior purely from a consideration of state outputs from the plant. Again, assuming the adversary may not subvert all process control units in a plant, the kinetic relationships which exist and are well understood could reveal the presence of the adversary and approximate their location in the system. We could then bring other methods to bear, including those example in chapters 6 and 7 to locate and eliminate adversary presence in such systems.

9.5.7 General Opportunities

Another are for research is how to make use of the conjoint operation of such techniques, establishing algorithms for collating and analyzing information from different parts of the system, gathered by our different methods to produce a holistic cross-sectional view of system behavior. Such a view would considerably raise the difficulty for adversaries seeking to subvert the system since it would, in effect, employ a “great crowd of witnesses”[2] to uncover potentially malicious behavior. Finally, there still exists the opportunity to explore context-based anomaly detection both in these and other kinds of systems such as mobile and sensor networks. This opens up considerable ground for future research and will provide validation on whether the general principles we have detected hold in other kinds of systems or there exist alternative principles or deeper themes which we can explore.

9.6 Summary

We provide an overall review of our research effort in this chapter, discuss our achievements and describe future work. We have explored methods of uncovering anomalies through concurrent observation and demonstrated that, armed with suitable models of system behavior, we can highlight deviations in expected systems behavior, both computationally and kinetically. The techniques we developed provide suitable monitoring mechanisms at network, host and sensor level, but we have not demonstrated how we could join them together to form a holistic approach to detection. We also believe there is further work to be done in exploring such techniques in other kinds of distributed or mobile systems. We believe that for each technique we have uncovered there are further variations in its implementation and application which we have yet to consider. The most important strand of work will be to join these techniques with a timing framework suitable for real-time systems. In conclusion, we regard our work as a beginning rather than something complete.

[2]Hebrews 12.1, Holy Bible, King James Version, 1664

Appendix A

Mathematical Pre-requisites

A.1 Preamble

We describe the fundamental mathematical approaches used in this thesis: state diagrams; process algebras (also named calculi) and process control. In the main matter of the thesis, we go on to describe variant approaches to utilizing these techniques.

Taking these in reverse order, process control is used by Industrial Control Systems, also known as Supervisory Control and Data Acquisition (SCADA) systems for real-time management of production values. An operator informs a controller of a *set point* value (e.g., temperature) that must be used during production and the controller uses process control which is effectively a set of differential equations sometimes labeled the *plant* to determine in real-time how to alter the settings of valves or pumps to maintain the set point. We encounter process control in chapter 5 where we describe using additional observations of an industrial process to determine if the process control model of the system is valid. We make use of MATLAB and Simulink to simulate the behavior of the plant and incorporate the plant equations into this model.

A process algebra is a model (or interpretation I) of an equational specification $\langle \Sigma E \rangle$ which describes the behavior (observable actions) of a process. Since the possible set of observable behaviors is large, a process algebra necessarily represents an abstract model of process behavior. Different behaviors may be observed for distinct purposes, leading to choices over the expressiveness of an algebra. Such choices result in different algebras, sometimes called calculi. Example choices are selection of operators, binding conventions, constants and sets over which operators range.

State transition diagrams are a graph-theoretical approach to describing process behavior which are closely related to the algebraic approaches outlined above. Indeed, it may be argued that any behavior which may be expressed in a process algebra could also be represented as automata in a state diagram. State diagrams are distinguished by a starting (*source*) state and a set of finishing (*sink*) states which are the vertices of the graph and a set of labeled transitions showing how the system may move from one state to another (for example, by sending messages) which are the edges of the graph. It is also possible to demonstrate the equivalence of state diagrams. The disadvantage of state diagrams over algebraic methods expresses itself when more complex structures (for example, involving infinite behavior or multiple interacting processes) are required. State diagrams are not economical in terms of expressing process complexity, whereas algebraic methods may be.

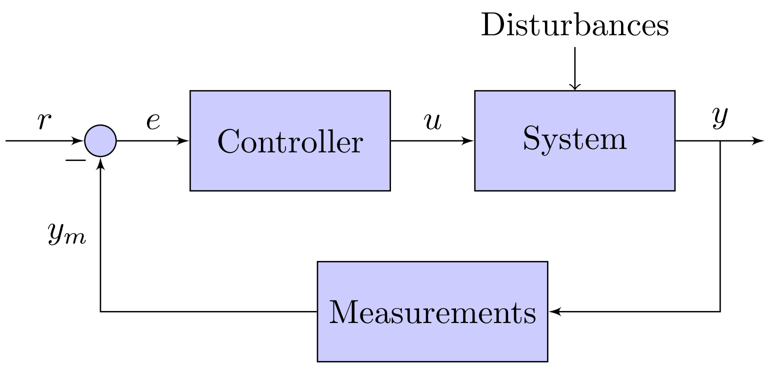


Figure A.1: Process Control Loop (Source: Tikz Examples)

A.2 Process Control

We outline what *process control* entails. In essence, each PCD obtains from the operator information on set points (e.g., temperature, pressure, flow rate) which should be maintained by a physical process. These set points are translated into process inputs by sending signals to actuators to open or close valves, turn switches off or on, set pump speeds and so forth. Signaling may be automatic or under operator control. These settings are calculated to produce a desired physical response, so they are the *controlled inputs* to the process. Each process is also subject to *disturbances* which are sometimes called "wild" signals. When outputs are measured by sensors (and sensor readings are themselves subject to both signal *noise* and measurement errors), the outputs are compared to the set points. Any significant difference between the set point and the outputs will lead to further adjustments to the input signals to bring the process response back into line.

The desired physical response of the system is calculated based on a system of differential equations describing how mass-energy balances change over time in response to inputs and measured outputs [13]. In outline, controllers use various kinds of calculation (proportional, integral or derivative), sometimes expressed by Laplace transformations of the differential equations, either singly or in combination, to control the rate at which any corrections are applied. Various types of control exist: open loop, closed loop, feedforward, feedback. These differences are not particularly important for our research at this point. We show a simple example of a controller in Figure A.1.

To simulate a controller, we make use of the MATLAB module Simulink [31]. To do this, we replicate the process followed by engineers designing the system and reproduce the mass and energy balance equations. For example, the mass and energy balance equation for a heat exchange is shown in equations A.1, A.2 and A.3:

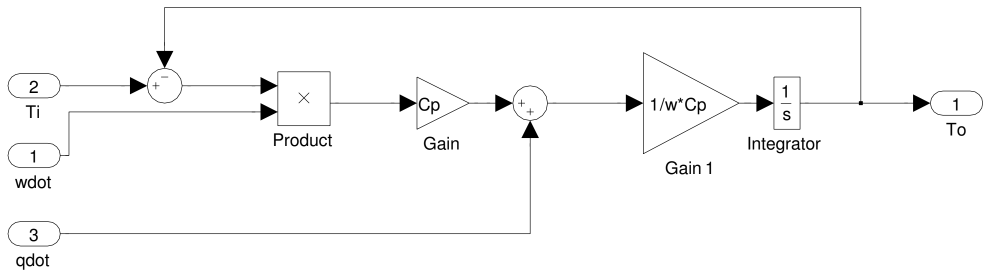


Figure A.2: Heat Exchange Block Diagram in Simulink

$$\dot{q} = UA(T_{in,s} - T_{out,c}) \tag{A.1}$$

$$wCp\frac{dT}{dt} = \dot{w}Cp(T_{in,c} - T_{out,c}) + \dot{q} \tag{A.2}$$

$$uCp\frac{dT_s}{dt} = \dot{u}CP(T_{in,s} - T_{out,s}) - \dot{q} \tag{A.3}$$

In equation 8.1 $\dot{q}$ is the rate of heat exchange, U is the coefficient of heat exchange for the construction material and A is the area, $T_{in,s}$ is the initial hot-side temperature (C^o), $T_{out,s}$ is the final hot side temperature, $T_{in,c}$ is the initial product temperature and $T_{out,c}$ is the final product temperature. Equations 8.2 and 8.3 represent the respective energy balances of the cold and hot sides where $\dot{w}, \dot{u}$ represent the hot-side and cold-side flow rates respectively, w, u the liquid volume (m^3), and Cp is the specific heat capacity ($kJ/kg - K$) of the product. The block diagram for these equations is shown in Figure A.2.

Further details on modeling and simulating process control systems may be found in [13].

A.3 Process Algebra

A process algebra is an algebra used to model process behavior, based on an *equational specification* which is a theory of process behavior. The choice in process algebra is which behaviors to model as processes may exhibit a very large number of behaviors and characteristics and it would be difficult to capture all of these in a single theory and certainly not useful. Hence such choices lead to different algebra (and calculi, the terms being synonymous) being used for distinct purposes.

A.3.1 Equational Specification

A process algebra is based on an equational specification over some arbitrary model. We formally define this notion.

Definition A.1 (Equational Specification)
An equational specification (Σ, E) *expresses a model* $\mathcal{M}$ *(which can be arbitrary) where* E *is a set of equations of the form* $t_1 = t_2$ *where* t_1 *and* t_2 *are terms and* Σ *is the signature*

$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)z = xz + yz$	A4
$(xy)z = x(yz)$	A5

Table A.1: BPA Axioms

which is the set of constant and function symbols which may appear in the specification. Σ *also gives the arity of each function symbol. Equations are often called axioms.*

A.3.2 Basic Process Algebra

BPA (Basic Process Algebra) is the theory which we use as the basis for creating our variant algebra (and, indeed, other theories). We provide the equational specification of $BPA(\Sigma_{BPA}, E_{BPA}$, set out the proof technique and give a simple example of its use.

Σ_{BPA} has two binary operators $+$ and $\cdot$ and a number of constants $a, b, c, \ldots$ dependent on use. The set of constants is denoted A and is a parameter of the theory.

E_{BPA} consists of five equations given in table A.1.

The following notational conventions are used:

1. The operator $\cdot$ is often omitted. xy means $x \cdot y$.

2. Brackets are omitted frequently and $\cdot$ binds more closely than $+$.

3. The equations in table A.1 contain universally quantified variables $x, y, z, \ldots$

We provide an intuitive interpretation of the semantics for the specification BPA. The constants $a, b, c, \ldots$ are *atomic actions* which are indivisible actions or events in process behavior.

- the operator $\cdot$ represents **sequential composition**;$x \cdot y$ is a process which first executes x then y;

- the operator $+$ represents **alternative composition**; $x + y$ is a process which either executes x or y (but not both).

The axioms of BPA can, therefore, be explained as follows:

- A1 (the **commutativity of** $+$) says a choice between x and y is the same as choice between y and x;

- A2 (the **associativity of** $+$) says that a choice between x and choosing between y and z is the same as a choice between z and choosing x or y; all three alternatives are valid;

- A3 (the **idempotency of** $+$ states that a choice between x and x is the same as x;

- A4 (the **right distributivity of $\cdot$ over $+$**) says that a choice between x and y, followed by z, is the same as a choice between x followed by z and y followed by z;
- A5 (the **associativity of $\cdot$**) should be clear.

A.3.3 Basic Terms

We define a set of terms called *basic terms* by induction:

1. every atomic action a is a basic term;
2. if t is a basic term, and a an atomic action, then $a \cdot t$ is a basic term;
3. if t, s are basic terms then $t + s$ is a basic term.

We consider terms that differ in the order of the summands to be identical working modulo axioms A1 and A2.

Proposition A.1 (Term Rewriting)
For every BPA term t, there is a basic term s such that BPA $\vdash t = s$.

PROOF

$$\begin{aligned} x + x &\to x \\ x(y+z) &\to xz + yz \\ (xy)z &\to x(yz) \end{aligned}$$

In effect, every closed term in BPA may be rewritten as a basic term. This term rewrite system is confluent and strongly normalizing. Basic terms are, therefore, useful in proofs since, to prove some statement valid for all closed terms, it is sufficient to prove it valid for all *basic* terms by reducing the closed term to normal form. This means we use structural induction as a proof technique.

A.3.4 Action Relations

Action relations provide an *operational semantics* for process expressions by which we say which actions a process can perform.

Over the set of BPA terms we define binary relations $\xrightarrow{a}$ and unary relations $\xrightarrow{a} \surd$ for each $a \in A$:

- $t \xrightarrow{a} s$ denotes that t can execute a and turn into s;
- $t \xrightarrow{a} \surd$ denotes that t can execute a and terminate.

Table A.2 provides an inductive definition. $t \xrightarrow{a} s$ if and only if it can be derived from this table.

$a \xrightarrow{a} \surd$	
$x \xrightarrow{a} x'$	$\Rightarrow x + y \xrightarrow{a} x'$ and $y + a \xrightarrow{a} x'$
$x \xrightarrow{a} \surd$	$\Rightarrow x + y \xrightarrow{a} \surd$ and $y + x \xrightarrow{a} \surd$
$x \xrightarrow{a} x'$	$\Rightarrow xy \xrightarrow{a} x'y$
$x \xrightarrow{a} \surd$	$xy \overset{a}{y}$

Table A.2: Action Relations for BPA

A.3.5 A Remark on Distributivity

We notice that the following axiom does *not* appear in BPA:

$$x(y + z) = xy + xz.$$

This axiom would give full distributivity and is rejected for BPA on the intuitive grounds that the moment of choice is different between the two terms. In the first term, x is chosen and then a choice is made between y and z. In the second, y or z is chosen immediately on selecting one or other branch starting with x.

We can see the difference in branching structure from the possible action relations. In the first instance,

$$a(b + c) \xrightarrow{x} (b + c)$$
$$\text{Case 1: } b + c \xrightarrow{b} \surd$$
$$\text{Case 2: } b + c \xrightarrow{c} \surd$$

while, in the second,

$$\text{Case 1: } ab + ac \xrightarrow{a} b \overset{b}{\surd}$$
$$\text{Case 2: } ab + ac \xrightarrow{a} c \overset{c}{\surd}$$

The difference may also be seen by drawing graphs – shown in figures A.3 and A.4:

A.3.6 An Example

We show the value of term re-writing as a proof in showing bisimilarity between processes. Let

$$s = a(b + c)d \tag{A.4}$$
$$t = a(b + c)(d + d). \tag{A.5}$$

We ask if s is *bisimilar* (i.e., an equivalent process) to t, written $s \underline{\leftrightarrow} t$. By rewriting the terms

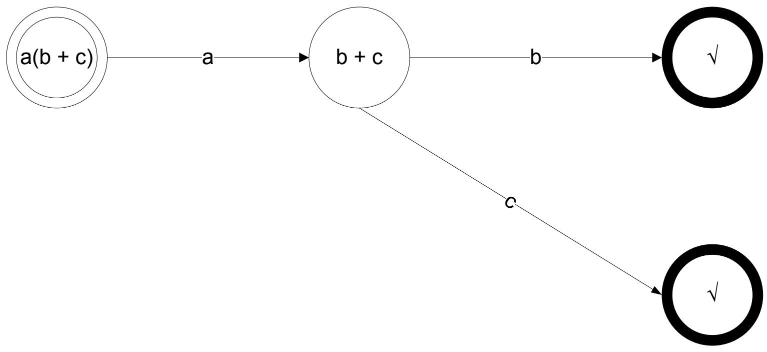


Figure A.3: Actions Relations(1)

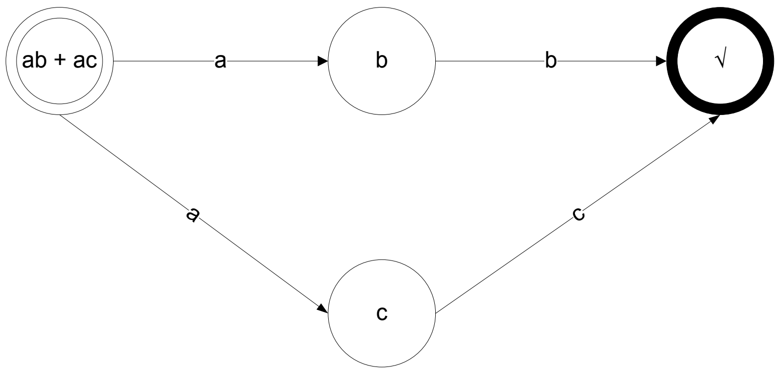


Figure A.4: Action Relations (2)

$$
\begin{aligned}
& a(b+c)d && \text{(A.6)}\\
\rightarrow\ & a(bd+cd) && \text{(A.7)}
\end{aligned}
$$

and

$$
\begin{aligned}
& a(b+c)(d+d) && \text{(A.8)}\\
\rightarrow\ & a(b+c)d && \text{(A.9)}\\
\rightarrow\ & a(bd+cd) && \text{(A.10)}
\end{aligned}
$$

which is clearly different from $abd + acd$ (which would follow if full distributivity was allowed) and the result follows.

A.4 Process Calculus: the π-calculus

Process calculus is related to process algebra (see section A.3) with similar, though parallel, beginnings [34]. It was designed to be used as a theory of mobile processes

$\mathbf{0}$	Null action: a process which does nothing
M	A sum over capabilities
$M + M'$	Alternate composition, exclusive - or
$\pi.P$	A process with a single capability; the process cannot proceed until it executes
$P + P'$	An exclusive choice between P and P'
$P\|P'$	P and P' proceed independently; but may interact by shared names
$\nu z\ P$	The scope of the name z is restricted to P
$!P$	Infinite composition of $P\|P\|P$, allowing repetition

Table A.3: π-Calculus Summands and Processes

and, as such, enables us to also model distributed systems which share many of the characteristics of mobile systems. Here we provide an outline summary of the basic π-calculus based on [ibid.].

A.4.1 Capabilities and Processes

The prefixes which are its capabilities (for action) are:

$$\pi ::= \bar{x}y|x(z)|\tau|[x = y]\pi$$

The first capability is send a name y by a name x and the second to receive any name by x. The third is the capability for *unobservable* action and the fourth is a conditional capability exercised if x and y are the same.

The processes and summations of the π-calculus are given in table A.3:

$$P ::= M|(P|P')|\nu z\ P|!PM ::= \mathbf{0}|\pi.P|M + M'.$$

The intended interpretations are:

We note that $\tilde{x}$ may be used to indicate an n-tuple of names $x_1, x_2, \ldots, x_n$. Finally, we use $:=$ as our defining operator, e.g, $P := (\nu\tilde{z})\ Q$.

A.4.2 Binding

Names in the π-calculus may be free or bound. A name is bound if it occurs within the scope of P as $x(z).P$ or $\nu z\ P$. The free names of P circumscribe its capability for action. In order for P to send x, send via x or receive via x, the name x must occur free $x \in fn(P)$ where $f(n)P$ is the set of the free names of P. The bound occurrences of a name z indicate where a name received via x may be *substituted* for z

(see section A.4.3). Because binding of the scope of the name prevents another process excluded from that scope by interacting by that name, it is possible to *extrude* the scope of a name by sending it to another process by a free name shared by two processes and using substitution[ibid.]. In addition, binding has been used to represent covert, or encrypted, means of communication in variants of the π-calculus – [2].

A.4.3 Substitution

Definition A.2 (Substitution)
A substitution *σ is a function on names that is the identity except for the finite set.*

If a name w does not occur in the process P then $P\{^w/_z\}$ is the process obtained by replacing each free occurrence of z in P by w. It is also possible to substitute for a bound name with the same constraint. The constraint may be bypassed provided a further suitable substitution is made for the bound names.

Processes which can be transformed into one another are called *α-convertible*. Such processes are identified in the π-calculus. The mismatch prefix $[x \neq y]$ is not used in the π-calculus where names represent communication channels because a substitution $\{^y/_x\}$ would invalidate monotonicity w.r.t α-convertibility – that is that substitution must not invalidate any process capabilities.

A.4.4 Operator Precedence

Parentheses are used to resolve ambiguity. We observe the conventions that prefixing, restriction and replication bind more tightly than composition and prefixing more tightly than sum. So $\pi.P|Q$ is $(\pi.P)|Q$, $\nu z\ \ P|Q$ is $(\nu z\ P)|Q$, $!P|Q$ is $(!P)|Q$ and $\pi.P + Q$ is $(\pi.P) + Q$. substitutions also bind more tightly than a process operator so $\pi.P\sigma$ is interpreted $\pi.(P\sigma)$. Parentheses may also be inserted to aid reading, for instance, $(\nu x)P$.

A.4.5 Examples of Binding and Substitution

Consider the process P where

$$P := (x(z).\bar{z}a.\mathbf{0}|\bar{x}w.\bar{y}w.\mathbf{0})|y(v).v(u).\mathbf{0}.$$

Because the first and second components share the name x, they may interact. Similarly, the second and third components contain y. But the first and third components may not interact. The second and third components may not interact immediately as the name y in the second component is under another capability. The result of the interaction of the first and second components is

$$P' := (\bar{w}a.\mathbf{0}|\bar{y}w.\mathbf{0})|y(v).y(u).\mathbf{0}$$

Note the substitution $^w/_z$. This substitution would also have been valid if the second component was defined as $\nu w(\bar{x}w.\bar{y}w.\mathbf{0})$ and the scope of the process would have been written $\nu w\ P'$. Hence we see that we may extrude the scope of a name, so that the name is shared between two processes.

The second component can now send the name w to the third component via y.

$$P'' := (\bar{w}a.\mathbf{0}|\mathbf{0}|w(u).\mathbf{0})$$

We also see that the first and third components may now further interact by w. In contrast, let Q be a process such that

$$Q := \nu z\ (\bar{z}x.\mathbf{0}|z(s).\mathbf{0})|z(t).\mathbf{0}$$

only has one action due to the restricted scope of the initial z which should be distinguished from the other z which is not in the scope of the bound name with which it shares a label.

A.4.6 Proof Techniques

While there are a number of proof techniques which may be used in relation to the π-calculus, including structural induction (see section A.3), the basic techniques in the calculus for demonstrating properties are *reduction* and the use of *labeled transitions* which are very similar in concept and operation to *operational semantics* as used in process algebra [ibid.]. Reduction describes the intra-action of processes, while labeled transitions describe the interaction of processes with their environment. The first approach makes use of structural congruence $\equiv$ to alter the composition of processes to a point where a reduction may take place using the *axioms of reduction* and the *rules of equational reasoning* . The second approach builds the proof of process action from a set of *transition rules* which are applied hierarchically to any two processes P and Q which interact with each other, first, addressing the individual transition of each process and, subsequently, showing the joint outcome. We outline these techniques and provide the rule sets. In practice, we regard reduction as more useful to our purpose because we consider in our models sets of processes which intra-act to form distributed computations whose actions undo or conserve security properties.

A.4.7 Reduction

The reduction axioms are

$$(\bar{x}y.P_1 + M_1)|(x(z).P_2 + M_2) \quad \rightarrow P_1|P_2\{^x/_y\} \tag{A.11}$$

$$\tau.P + M \rightarrow P. \tag{A.12}$$

Variant axioms which swap the order of expressions are not needed because of the structural rule

$$\text{from } P_1 \equiv P_2 \text{ and } P_2 \rightarrow P_2' \text{ and } P_2' \equiv P_1' \text{ infer } P_1 \rightarrow P_1' \tag{A.13}$$

where $\equiv$ is the *structural congruence* relation which we now define.

To define structural congruence, we define the notions of context and congruence and provide the axioms of structural congruence. First, an occurrence of $\mathbf{0}$ is *degenerate* if it is the left or right term in a sum $M_1 + M_2$ and *non-degenerate* otherwise.

SC-MAT	$[x = x]\pi.P \equiv \pi.P$
SC-SUM-ASSOC	$M_1 + (M_2 + M_3) \equiv (M_1 + M_2) + M_3$
SC-SUM-COMM	$M_1 + M_2 \equiv M_2 + M_1$
SC-SUM-INACT	$M + \mathbf{0} \equiv M$
SC-COMP-ASSOC	$P_1 \vert (P_2 \vert P_3) \equiv (P_1 \vert P_2) \vert P_3$
SC-COMP-COMM	$P_1 \vert P_2 \equiv P_2 \vert P_1$
SC-COMP-INACT	$P \vert \mathbf{0} \equiv P$
SC-RES	$\nu z\, \nu w\, P \equiv \nu w\, \nu z\, P$
SC-RES-INACT	$\nu z\, \mathbf{0} \equiv \mathbf{0}$
SC-RES-COMP	$\nu z(P_1 \vert P_2) \equiv P_1 \vert \nu z\, P_2,\ \text{if } z \notin fn(P_1)$
SC-REP	$!P \equiv P \vert !P$

Table A.4: The axioms of structural congruence

REFL	$P = P$
SYMM	$P = Q$ implies$Q = P$
TRANS	$P = Q$ and $Q = R$ implies $P = R$
CONG	$P = Q$ implies $C[P] = C[Q]$

Table A.5: The rules of equational reasoning

Definition A.3 (Context)
A context *differs from a process only in have a* hole $[\cdot]$ *in place of a non-degenerate occurrence of* $\mathbf{0}$.

A context is written $C[P]$ for the process obtained by replacing $[\cdot]$ in C by P. The replacement is literal, so names free in P may be bound in $C[P]$. A context is a syntactic entity which transforms processes to processes and contexts which are α-convertible are not identified.

Definition A.4 (Congruence)
An equivalence relation $\mathcal{S}$ on processes is a congruence *if* $(P, Q) \in \mathcal{S}$ *implies* $(C[P], C[Q]) \in \mathcal{S}$ *for every context* C.

Definition A.5 (Structural Congruence)
is the smallest congruence on processes that satisfies the axioms in table A.4 *and the rules of equational reasoning* A.5.

The axioms of structural congruence allow manipulation of the term structure to enable reduction in accordance with the *reduction rules* shown in table A.6.

In general, we omit manipulations of the term structure which can be tedious unless required to avoid ambiguity. For example, suppose $M := \bar{x}z.\mathbf{0}, N := \bar{y}w.\mathbf{0}$ and $N' := x(v).\mathbf{0}$. Then

R-INTER	$\frac{}{(\bar{x}y.P_1+M_1)\vert x(z).P_2+M_2)\to P_1\vert P_2\{^y/_z\}}$
R-TAU	$\frac{}{\tau.P+M\to P}$
R-PAR	$\frac{P_1\to P_1'}{P_1\vert P_2\to P_1'\vert P_2}$
R-RES	$\frac{P\to P'}{\nu z\ P\to\nu z\ P'}$
R-STRUCT	$\frac{P_1\equiv P_2\to P_2'\equiv P_1'}{P_1\to P_1'}$

Table A.6: The reduction rules

$$\begin{aligned}(N+N')|\nu z\ M \quad &\equiv \nu z((N+N')|M)\\ &= \nu z((\bar{y}w.\mathbf{0}+x(v).\mathbf{0})|\bar{x}z.\mathbf{0})\end{aligned}$$

using SC-RES-COMP; and using SC-COMP-COMM, SC-SUM-COMM and SC-SUM-INACT, we get

$$\begin{aligned}\nu z\ ((N+N')|M) \quad &\equiv \nu z(M|(N+N'))\\ &\equiv \nu z(M(N'+N))\\ &\equiv \nu z((M+\mathbf{0})|(M'+N)),\end{aligned}$$

where the last process is of the form of the first axiom of reduction

$$\nu z(\bar{x}z.\mathbf{0}+\mathbf{0})|(x(v)+\mathbf{0}+N).$$

In this case, we would give the first term manipulation which resolves any ambiguity with regards to the scope of z and the result of the last manipulation prior to reduction and the interim steps are considered to be a *well-normalized derivation* of the rules of structural congruence and equational reasoning. A well-normalized derivation consists of an application of R-INTER or R-TAU and any number of applications of R-PAR and R-RES, followed by an application of R-STRUCT and without folding any sub-term under a term by applying SC-REP from right to left. It can be shown [1] that there always exists a well-normalized derivation of any term. Let P and P^* be terms such that $P \equiv P^*$, we write $P \Longrightarrow P*$ for a well-normalized derivation P to $P*$. Hence, returning to our example, we would normally write

$$(N+N')|\nu z\ M \equiv \nu z((N+N')|M)$$

and

$$\nu z((N+N')|M) \Longrightarrow \nu z((M+\mathbf{0})|(M'+N))$$

.

[1]The proof is omitted due to length and technicality

A.4.8 Transition Relations

Transitions enable us to analyze the behavior of a system by dividing it into parts. We find this useful in proving the interaction of components of distributed algorithms in relation to their environment. We define the actions.

Definition A.6 (Actions)
The actions *are given by*

$$\alpha ::= \bar{x}y | xy | \bar{x}z | \tau.$$

We write Act for the set of actions.

The first action is ending the y by x. The second is receiving y via x. The third is sending a fresh name by x. The last is an internal (*unobserved*) action.

The transition relation labeled by α will be written $\overset{\alpha}{\to}$. For example, $P \overset{\bar{x}y}{\to} Q$ expresses that P can send a name y by x and evolve to Q.

Definition A.7 (Transition Relations)
The transition relations, $\{\overset{\alpha}{\to} | \alpha \in Act\}$, *are defined by the rules in table A.7, excluding symmetric rules such as SUM-R for SUM-L.*

These rules reflect the informal account of the capabilities of the π-calculus. A proof of the action of a process is given by considering the reduction of individual processes and the joint outcome in accordance with the rules.

A.5 State Transition Diagrams

State Transition Diagrams(State Diagrams) are an abstract graph-theoretical representation of the ordering of events, or states, in a process. There are various representations. Here we use a directed graph representation.

Let α (pronounced "Act") be a set of process actions, e.g., $x := 1$, if $(x == 1)$ then $y := 2$. Let $\overrightarrow{G} = V(\overrightarrow{E})$ be a state diagram. Each $v \in V$ is a vertex, generally labeled, representing one, or more, states in a process. Let each $\overrightarrow{e} \in \overrightarrow{E}$ be a directed edge, again normally labeled with an action $a \in \alpha$.

We also mark one vertex as the starting state or source state, which may be shown by an "arrow" or here by a double circle and we may mark one or more states as accepting or sink states, in this thesis, shown by a thick circle.

Two examples of state transition diagram are provided in figures A.3 and A.4. These examples show the close relationship between process modeling using state diagrams and process algebras.

OUT	$\dfrac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P}$
INP	$\dfrac{}{x(z).P \xrightarrow{xy} P\{^{y}/_{z}\}}$
TAU	$\dfrac{}{\tau.P \xrightarrow{\tau} P'}$
MAT	$\dfrac{\pi.P \xrightarrow{\alpha} P'}{[x{=}x]\pi.P \xrightarrow{\alpha} P'}$
SUM-L	$\dfrac{P \xrightarrow{\alpha} P'}{P{+}Q \xrightarrow{\alpha} P'}$
PAR-L	$\dfrac{P \xrightarrow{\alpha} P'}{P\vert Q \xrightarrow{\alpha} P'\vert Q}$ $\quad bn(\alpha) \cap fn(Q) = \emptyset$
COMM-L	$\dfrac{P \xrightarrow{\bar{x}y} P' \quad Q \xrightarrow{xy} Q}{P\vert Q \xrightarrow{\tau} P'\vert Q'}$
CLOSE-L	$\dfrac{P \xrightarrow{\bar{x}(z)} P' \quad Q \xrightarrow{xz} Q'}{P\vert Q \xrightarrow{\tau} \nu z\ (P'\vert Q')}$ $\quad z \notin fn(Q)$
RES	$\dfrac{P \xrightarrow{\alpha} P'}{\nu\ P \xrightarrow{\alpha} \nu z\ P'}$ $\ z \notin n(\alpha)$
OPEN	$\dfrac{P \xrightarrow{\bar{x}z} P'}{\nu z\ {}^{\bar{x}z} P'}$ $\quad z \neq x$
REP-ACT	$\dfrac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'\vert !P}$
REP-COMM	$\dfrac{P \xrightarrow{\bar{x}y} P' \quad P \xrightarrow{xy} P'}{!P \xrightarrow{\tau} (P'\vert P'')\vert !P}$
REP-CLOSE	$\dfrac{P \xrightarrow{\bar{x}(z)} P' \quad P \xrightarrow{xz} P''}{!P \xrightarrow{xz} (\nu z(P'\vert P''))\vert !P}$ $\ z \notin fn(P)$

Table A.7: The transition relations

Appendix B

Distributed Environments

B.1 Preamble

In this appendix, we discuss the distributed systems we have used as examples during our research. We start with a generic discussion of the nature of distributed system and some of the problems of observability and controllability. We go on to provide two examples of environments which may be categorized as distributed systems – first, multi-processor operating systems and, second, Process Control Networks(PCN).

B.2 Notation

We define the notation used in this chapter, and subsequently, to describe distributed systems in Table B.1. The notation is taken from [46].

N	The number of processes
$P_1, P_2, \ldots, P_N$	Processes
G	Global state
e, f, g	Events
s, t, u	Local states
$\prec_{im}$	Immediately precedes
$\prec$	Locally precedes
$\rightsquigarrow$	Remotely precedes
$\rightarrow$	Happened before
$\overset{p}{\rightarrow}$	Potential causality
$\parallel$	Concurrency
S_i	Sequence of states on P_i
$next(e)$	Event immediately after e
$prev(e)$	Event immediately before e
$e.p$	Process on which event e occurred
$s.p$	The process in which state s occurred
S	Set of all local states
C, D	Logical clocks
$\mathcal{C}$	The set of all logical clocks
$s.c$	The logical clock value in state s
$s.v$	The vector clock value in state s

Table B.1: Notation

B.3 Modeling Distributed Systems

The text in this section is a summary of parts of [46], highlighting any particular features of models of distributed systems relevant to our purpose.

B.3.1 Topology and Composition

We may model a *distributed system* "as a loosely coupled message-passing system without any shared memory or a global clock". The system may be described as a set P of N processes, where $P = |N|$, where each process $P_1, P_2, \ldots, P_N$ may send messages along one, or more, directed channels $C_1, C_2, \ldots, C_K$ in set of channels C such that $|C| = K \geq N$. We may represent the topology of a distributed system D as a directed graph $\overrightarrow{G} = V(\overrightarrow{E})$, where $G = D$, $V = P$ is the set of vertices which are processes and $\overrightarrow{C} = E$ is the set of directed edges which are channels. A bi-directional channel may be represented by two unidirectional edges.

Channels may be assumed to have infinite buffer and to be error free. No assumptions are made about the ordering of messages, which may be subject to arbitrary delays. The state of a channel is the sequence of messages sent along it. A process is defined as a set of states, an initial condition and a set of events. Finite processes may be described by state transition diagrams – see A.

In general, a distributed system D consists of a network of heavyweight processes which send messages to each other to achieve the goals of the *distributed computation*. Each process may be composed of multiple lightweight threads which share memory and which work concurrently (taking advantage of the efficiencies offered by parallelism in local processing). However, a distributed system can mimic a parallel system and a parallel system may, using messaging structures in shared memory, mimic distributed systems' messaging architecture. One could even argue that a parallel system is simply a special case of a distributed system because it is a distributed system with only one channel – shared memory.

B.3.2 Basic Models

There three basic models of a distributed computation to which we refer in our research - the *interleaving* model, the *happened before* model and the *potential causality* model. All three models are useful for different purposes.

B.3.2.1 Interleaving Model

In the interleaving model, a *run* is a *global sequence* of events. All events on the run are interleaved. A *global state* is a cross-product of the local states of the processes [1]. The global state function $nextG, e$ gives the next global state when the event e is executed in global state G. The interleaving model defines a total order over the set of events.

Definition B.1 (Interleaving Model)
A sequence of events $seq = (e_i : 0 \leq i \leq m$ *is a computation of the system in the interleaving model if there exists a sequence of global states* $(G_i : 0 \leq i \leq m+1)$ *such that* G_0 *is an initial state and* $G_{i+1} = next(G_i, e_i) \quad for \quad 0 \leq i \leq m$

[1]For convenience, we assume that processes also record the messages sent along channels and hence hold the state of the channels as well.

B.3.2.2 Happened Before Model

The interleaving model is important because all possible runs of a distributed computation may be expressed as a total order (or linearization) using this model. This leads to the *interleaving assumption* which effectively treats every event as instantaneous, but states that no two events may execute simultaneously. Given two events e and f, either e executes first or f executes first. But, as the previous statements imply, there may be more than one possible interleaving of events. Since, for some process p, either $e.p \prec f.p$ else $f.p \prec e.p$ or, for any two processes, either $e.p \leadsto f.q$ else $f.q \leadsto e.p$ or $e.p \| f.q$, distributed systems are more properly represented by a partial order called the happened before relation denoted by $\rightarrow$. This is defined as the transitive closure of $\prec_{im}$ and $\leadsto$.

Definition B.2 (Happened Before Relation)
The happened before relation ($\rightarrow$) is the smallest relation that satisfies:

1. $(e \prec_{im} \vee (e \leadsto f) \Rightarrow (e \rightarrow f)$*, and*
2. $\exists g : (e \rightarrow g) \wedge (g \rightarrow f) \Rightarrow (e \rightarrow f)$

Hence, a *run* or a *computation* in the happened before model is defined as a partial order over the set of events E where all events in a single process are assumed to form a total order. A happened before model may be presented as a *space-time* diagram where the relation holds if there is a directed path from e to f - see Figure B.1.

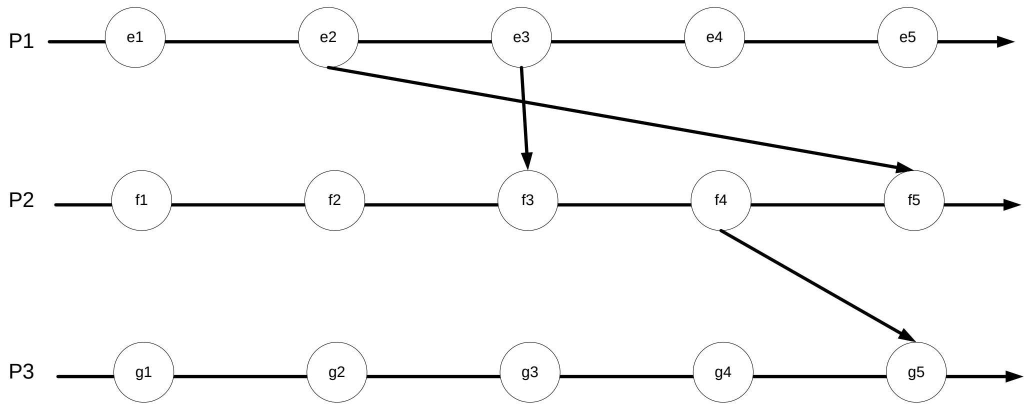


Figure B.1: Run in happened before Model

B.3.2.3 Potential Causality Model

While the order of events in a single process can be accurately determined, it does not follow that they have a causal relationship. However, it may be expensive to determine causality. Hence we can use the *potential causality* relation. This relation respects the *causality* relation, but the converse may not hold. The happened before model is a valid potential causality model because if e causes f then e must happen before f. But it is not true that a potential causality relation requires all events in a single process to be ordered. The potential causality relation implies a partial order on a single process.

Definition B.3 (Potential Causality Relation)
The potential causality relation on the event set is the smallest relation satisfying

- *If an event e potentially causes another event f on the same process then $e \xrightarrow{p} f$.*
- *If e is the send of a message and f is the corresponding receive, then $e \xrightarrow{p} f$.*
- *If $e \xrightarrow{p} f$ and $f \xrightarrow{p} g$ then $e \xrightarrow{p} g$*

Events e and f are independent if $\neg(e \xrightarrow{p} f) \wedge \neg(f \xrightarrow{p} e)$. A potential causality diagram may be used to represent a run in a potential causality model – see figure B.2.

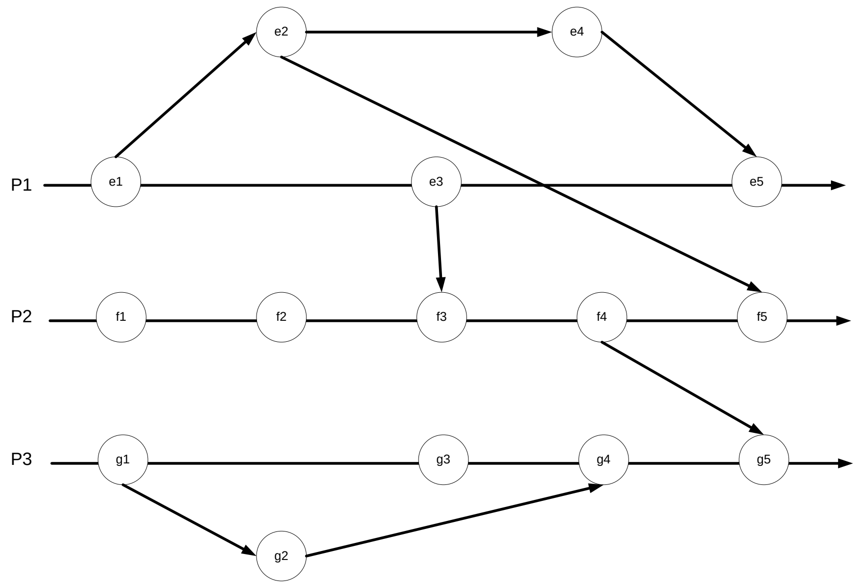


Figure B.2: Run in a Potential Causality Model

B.3.3 Models Based on States

To this point, we have concentrated on models based on events. It is also possible to use state-based models where – in a graph of the run – the nodes represent states and the edges represent events. But some care is needed since not every partially ordered set (or poset) of states represents a valid distributed computation and may contain cycles in the induced graph – see fig B.3. To avoid having to check for these, the notion of a decomposed partially ordered set (deposet) of states is introduced. The assumptions lying behind this model are –

- No event is assumed that no event is received before the initial state or sent after the final state.
- Each event is either a send event, a receive event or an internal event.
- Messages may not be sent and received as a single event.

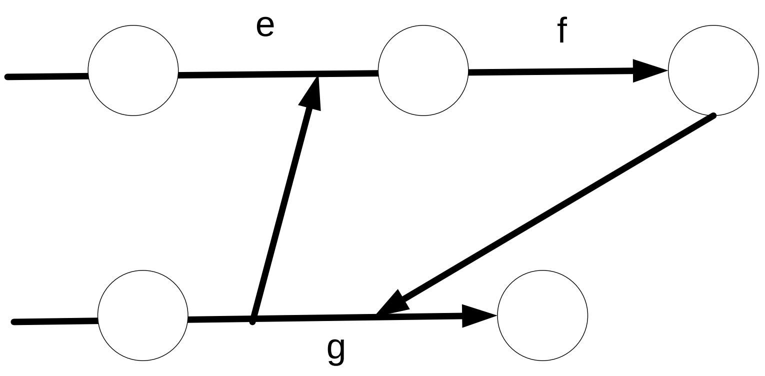


Figure B.3: A Poset on States with No Valid Event Based Poset

Let S_i be the sequence of local states in a process P_i. Let [2]

$$S = \coprod_i S_i.$$

The execution of the distributed computation, consisting of N concurrent processes $P_1, P_2, \ldots, P_N$ can subsequently be modelled as a tuple $(S_1, S_2, \ldots, S_N, \rightsquigarrow)$. This tuple represents a deposet, provided we retain the assumptions outlined. More formally, defining the initial and final states as $initial(i) := min \quad S_i$ and $final(i) := max \quad S_i$, a deposet may be defined as follows –

Definition B.4 (Deposet)
A deposet is a tuple $(S_1, \ldots, S_N, \rightsquigarrow)$ such that $(S, \rightarrow)$ is an irreflexive partial order that satisfies:

- $\forall i : \neg(\exists u : u \rightsquigarrow initial(i))$
- $\forall i : \neg(\exists U : final(i) \rightsquigarrow u)$
- $s \prec_m t \Rightarrow \neg(\exists u : s \rightsquigarrow u) \vee \neg(\exists u : u \rightsquigarrow t)$

[2]The notation differs from [46] at this point, but is consistent with the conventions used in other parts of this thesis.

B.3.3.1 Global Sequence of States

There are many total orders which are consistent with linearizations of a partial order defined by a deposet (or run). A global sequence is a single linearization which corresponds to a sequence of global states where a global state is a vector of local states. The set of global sequences $(S_1, S_2, \ldots, S_N, \leadsto)$ consistent with a run r is denoted $linear(r)$. A *global sequence* g is a finite sequence of global states denoted as $g = g_1 \ldots g_l$ where g_k is a global state for $1 \leq k \leq l$. So we define a global sequence of a run as:

Definition B.5 (Sequence of Global States)
g is a global sequence of a run r (denoted by $g \in linear(r)$ if and only if the following constraints hold:

- *$\forall i$: g is restricted to $P_i = S_i$(or a stutter of S_i, to be defined)*
- *$\forall k$:$g_k[i] \| g_k[j]$ where $g_k[i]$ is the state of P_i in the global state g_k*
- *$\forall k$:g_k and g_{k+1} differ in the state of exactly 1 process*

The initial constraint says that if an observer restricts his attention to a single process P_i, then he would observer S_i or a stutter of S_i where a stutter of S_i is a finite sequence of repetitions of the state of S_i. The second constraint requires all states to be pairwise concurrent and the third constraint states the *interleaving* assumption which is that, given two events e and f, either e occurs first or f occurs first. This allows us to represent concurrency relatively simply.

B.3.4 Modelling Time in Distributed Systems

When the behavior of a distributed computation is viewed as a total order, the actual order of events cannot be determined in the absence of accurately synchronized physical clocks. Hence we provide mechanisms to give a set of total orders in which events *could have* happened. The purpose of the clock is to provide an order not to indicate duration. We define two such mechanisms: a *logical clock* and a *vector clock* and we provide respective implementations, but omit proofs, for which we refer the reader to [46].

B.3.4.1 Logical Clocks

We begin by defining a logical clock:

Definition B.6 (Logical Clock)
A logical clock *C is a map from S (the set of all local states) to $\mathcal{N}$ (the set of natural numbers) with the following constraint:*

$$\forall s, t \in S : s \prec_{im} t \vee s \leadsto t \Rightarrow C(s) < C(t)$$

$\mathcal{C}$ is the used to denote the set of all logical clocks which satisfy the above constraint. This constraint models the sequential nature of execution for each process and the physical requirement that any message take a non-zero amount of time to transmit. This constraint may be satisfied by assigning clock values so that if two

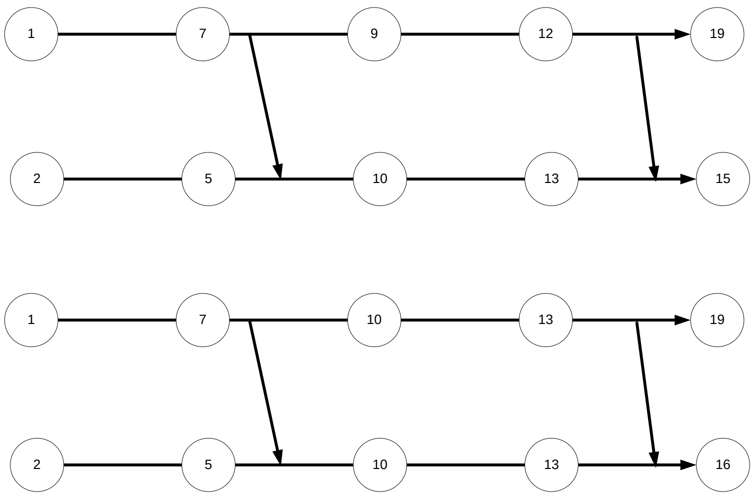


Figure B.4: An illustration of a clock assignment

local states are concurrent, then we may construct a logical clock such that both states are assigned the same time stamp.

$$\forall u, v \in S : u||v \Rightarrow \exists C \in \mathcal{C} : (C(u) = C(v).$$

An example of such an assignment is provided in figure B.4 and we also give a sample implementation.

Algorithm B.1: Logical Clock

```
1
    $P_i$ ::
    **var**
       $c$ : integer initially $0$;

    send event $(s, send, t)$;
       $//s.c$ is sent as part of the message
       $t.c := s.c + 1$;

    receive event $(s, receive(u), t)$;
       $t.c := max(s.c, u.c) + 1$;

    internal event $(s, internal, t)$;
       $t.c = s.c + 1$;
```

B.3.4.2 Vector Clocks

Logical clocks satisfy the following property:

$$s \rightarrow t \Rightarrow s.c < t.c$$

But the converse is not true; $s.c < t.c$ does not imply that $s \rightarrow t$. Logical clocks therefore do not provide complete information about the *happened before* relation. A *vector clock* may be used to overcome this lacuna.

Definition B.7 (Vector Clock)
A vector clock *is a map from S to $\mathcal{N}^k$(vectors of natural numbers) with the following constraint:*

$$\forall s, t : s \rightarrow t \Leftrightarrow s.v. < t.v.$$

where $s.v$ is the vector assigned to the state s.

A vector clock allows the partial order of the *happened before* relation $\rightarrow$ to be captured. Vectors of natural numbers are used to time stamp processes. Given two vectors x and y of dimension N we compare them as follows:

$$\begin{aligned} x < y &= (\forall k : 1 \leq k \leq N : x[k] \leq y[k]) \wedge \\ & (\exists j : 1 \leq jleqN : x[j] < x[j]) \\ x \leq y &= (x < y) \vee (x = y) \end{aligned}$$

This order is partial for $N \geq 2$.
Again, we provide a sample implementation in Figure 8:

B.3.5 Determining State in Distributed Systems

One of the difficult features of distributed systems is that no process has access to the global state of the system. Although, for many applications, it is sufficient to capture a past state, from which, for example, the computation could re-start after failure, or else a stable property – such as the loss of a token. An algorithm which captures a global state is a called a *global snapshot algorithm*. A consistent global state is not simply a product of local states, but a consistent *cut* of such states.

Definition B.8 (Consistent Cut)
Let S be a deposet. Let G be a cut. A subset $G \subset S$ is a consistent cut (or a consistent global state) if and only if $\forall s, t \in G : s \| t$ and $|G| = N$.

A global snapshot algorithm is an algorithm which computes a consistent cut (or subcut) of a system. The algorithm assumes all channels are uni-directional and satisfy the FIFO property with regard to messaging. An example algorithm, shown in Figure B.3, shows how a special message called a *marker* is sent along all channels. All processes are white before they receive the marker. Every process which is white which receives the marker changes its color from white to red and forwards the marker to other processes before sending any further messages. This rule, along with the FIFO condition, guarantees that no white process ever receives

Algorithm B.2: Vector Clock Algorithm

```
1
    $P_j$ ::
     **var**
       $v$ : array$[1 \ldots N]$ of integer
         initially$(\forall i : i \neq j : v[i] = 0) \wedge (v[j] = 1)$

    send event$(s, send, t)$ :
       $t.v := s.v;$
       $t.v[j] := t.v[j] + 1;$

    receive event$(s, receive(u), t)$ :
       **for** $I := 1$ to $N$ **do**
         $t.v[i] := max(s.v[i], u.v[i]);$
       $t.v[j] := t.v[j] + 1;$

    internal event$(s, internal, t)$ :
       $t.v := s.v;$
       $t.v[j] := t.v[j] + 1;$
```

a message sent by a red process. This in turn guarantees that local states are mutually concurrent at the point of color change. Each process also captures the state of messages in channels by recording any further messages from channels until they have also turned red.

It should be noted that while global snapshot algorithms are useful for capturing stable predicates, they are not useful for capturing unstable properties where the state of the property may alter and the changes be reversed between two snapshots. It may also have the disadvantage of creating overheads if snapshots are taken over-frequently.

B.3.6 Examples of Distributed Systems

In sections B.4 and B.5, we provide two concrete examples of distributed systems which are commonly used today. First, a *multi-processor operating system* which exists today on every computing device which can run software applications from mobile phones to supercomputers. Second, process control networks which are used throughout the modern world to control the supply of energy, manufacturing, transport, communications and so forth.

Both of these systems are, to different extents, subject to the same non-linear ordering of events and the potential for processing failure as we have described in our abstract model of distributed systems and, hence, subject to similar difficulties in determining global state. Multi-processor systems are not subject to the same

Algorithm B.3: Global Snapshot Algorithm

```
1
    $P_i$ ::
    var
      $color$ : $\{white, red\}$ initially $white$;
      // assume $k$ incoming channels
      $chan$ :  array $[1 \dots k]$ of queues of messages initially $null$;
      $closed$ :  array $[1 \dots k]$ of boolean initially $false$;

    $turn\_red()$ enabled if $(color = white)$ :
      save_local_state;
      $color := red$;
      send (marker) to all neighbors;

    Upon $receive(marker)$ on incoming channel $j$;
      if $(color = white)$ then
        $turn\_red()$;
      $closed[j] := true$;

    Upon $receive(program\_message)$ on incoming channel $j$:
      if $(color = red) \wedge \neg closed[j]$ then //append the message
          $chan[j] := append(chan[j], program\_message)$;
```

communication losses or delays which are found in process control networks.

B.4 Multi-Processor Operating Systems

In this section, we describe the principles on which multi-processor operating systems, to be specific, kernel programs are designed. We emphasize the features most relevant to our research in our discussion. The operating system used in the research was Linux, one of the Unix family, and kernel version 2.6[3] and this influences our description. But the same principles can be found in the majority of commonly available operating systems used today. This description is based on [16].

B.4.1 Basic Operating System Concepts

Every computer system includes a set of programs called the operating system. The key program in this set is the *kernel*. The kernel is the first program loaded into RAM on boot up and contains the critical processes required for the system

[3]Linux operating systems from 2.6.xx have used a number convention where if xx is even then the operating is a beta build and if it is odd, it is an official release.

to operate. Hence, the terms kernel and operating system are frequently used as synonyms – as in this study.

The operating system must fulfill two goals:

- Interact with hardware components, services all low-level programmable elements;
- Provide an execution context for the applications run on the system (the user programs).

Some (early) operating systems allowed all user programs to interface directly with hardware programs. Linux (and the majority of modern operating systems) abstract away from the operation of low-level components and hardware interactions. Hence to use an operating resource, user programs must make a request to the kernel which evaluates the request and, if it chooses to grant it, interacts with hardware components on behalf of the system. This mechanism is enforced by specialist hardware features which enforce, at least two, different execution modes in the CPU – in Unix systems, these are called *user mode* and *kernel mode*.

B.4.2 Multiuser Systems

A *multiuser* system is a computer that can independently and concurrently execute several applications belonging to one or more users. *Concurrently* means the applications can be active at the same time and contend for resources. *Independently* means the application can perform its task without regard for the other applications running on the system.

Multiuser systems must include several features:

- An authentication mechanism for verifying the user's identity;
- A protection mechanism to prevent buggy programs blocking other applications;
- A protection mechanism against software which could spy on or interfere in other user's activities;
- An accounting mechanism which limits the resources assigned to each user.

Ensuring these mechanisms requires relying on the hardware protection in the CPU to prevents user's from imposing directly on the system's circuitry and bypassing these protections.

B.4.3 Users and Groups

In a multiuser system, each user has a private space on the machine. The operating system must ensure that the private portion of a user space is visible only to its owner. In particular, the operating system must ensure that no user can exploit a system application for the purpose of violating the private space of another user.

Users can set access permissions on files in their private space which either reserve files for their own use, allow them to be shared with a group or allow them to be shared universally. Files can have read, write or execute permissions set on them for user, group or world permissions.

Operating systems also have a superuser or root account which is used by system administrators to manage user accounts and perform maintenance tasks. The root user can do almost everything, because the operating system does not apply the usual protections, including accessing any file and manipulating any running process.

B.4.4 Processes

All operating systems use the *process* as a fundamental abstraction. Multiuser system must enforce an execution environment in which several processes may be active concurrently and contend for shared system resources. Such systems are called *multiprocessing systems* and must have a scheduler program which enforces preemption (i.e. it forces processors to relinquish the CPU and ensures fairness in access to processing resources). The model of processing on such operating systems is a kernel/process model where each process operates (in User mode) with the illusion that it is the only processor on the system while the kernel manages resources amongst processors as the system operates in Kernel mode. Whenever a process makes a system call or invokes a hardware facility, the kernel activates to service this request. At the same time, it can also force the process to relinquish the CPU while the request is being serviced, allowing other processes to advance. Modern operating system kernels are *re-entrant* – that is several programs may be running in Kernel mode at the same time.

B.4.5 Kernel Architecture

Most modern operating systems in commercial use are monolithic. Each kernel layer – and each kernel consists of multiple layers of process abstraction – runs as part of a whole kernel program in Kernel mode. *Micro-kernel* systems, by contrast, have a small set of functions, synchronization primitives and an interprocess communication mechanism. Operating system functions are built and run in separate layers based on this core. Such systems are very flexible and have increased security, but the performance penalties are high. To get around this, modern kernels allow modules to be linked to (and unlinked from) the kernel. Such modules can be used to create a device driver or a file system. Modules are executed in Kernel mode and, as for the kernel process (and root user), have wide spanning access to the files and resources on the system.

B.4.6 Uniprocessor and Multiprocessor Systems

Multiprocessing can occur on both uniprocessor systems and multiprocessor systems. The advantage of the latter is clearly that more than one process can run concurrently on the system, increasing performance while minimizing power requirements. The cost of this advance is that operating systems must negotiate resource allocation requests not only between processes, but also between CPUs. In particular, memory resources may be shared or private, depending in part on both software and hardware constraints. Access to memory may also be uniform (as on symmetric multiprocessing systems) or non-uniform (NUMA architectures). Symmetric processing systems have homogeneous chip architecture. But it is also possible for multiprocessor systems to have heterogeneous chip architecture. Finally, chip utilization can vary even during run time, depending on processing and power saving requirements. [needs citation]

B.4.7 Multiprocessor Systems as Distributed Systems

We argue that multiprocessor systems may be considered to be a special case of a distributed system. Or, at least, that it is possible on a multiprocessor system to simulate the behavior of distributed systems by having multiple concurrent processes – known in Linux as *light weight threads* – which use shared memory as a communication channel. It is also possible for processes which do not share memory to communicate using files known as *pipes*. The difference between this model and the model for a uniprocessor system which may also communicate by the same means is that it is possible for communication to be concurrent, rather than wholly asynchronous.

B.5 Process Control Networks

Process Control Networks (PCN), more normally call industrial control systems (ICS), but here using this term to emphasis their commonalities, are divided into two types: First, distributed control systems (DCS) and, second, supervisory control and data acquisition systems (SCADA). The distinction between the two is sometimes blurred, depending on the application – and frequently a further term ICS (Industrial Control Systems) is preferred instead [67]. But we use the term PCN because our focus is on the control – to be specific, *who* is in control – of such systems.

Essentially, distributed control systems are largely used in factories to provide *local*, frequently granular, process control over local area networks (LAN) while SCADA systems are used to provide *remote* supervisory process control over a variety of long distance communication media such as Internet, WAN (wide area networks), telephony (including cellular and analogous systems) and radio. Having said this, a large scale factory may benefit from SCADA control. PCNs are used in all aspects of production, disposal, communications, power generation and transport making them ubiquitous in modern life.

From our point of view, we treat the networks as analogous in function (they both gather information about and issue instructions to control physical processes over communication networks and both are supervised in real time by human operators). Working at this level of abstraction, the techniques which we have uncovered in our research may be applicable to each kind of system.

We use the rest of this section to describe the salient features of PCN, their development over time. We distinguish the functions, operation, or construction of SCADA and DCS, where necessary, and discuss the security issues which affect them. The information in this section is primarily drawn from [17] and [65] with reference to SCADA and DCS. Where additional sources are used, they are cited individually[4].

B.5.1 Supervisory Control and Data Acquisition Systems(SCADA) - A Description

SCADA refers to collecting data from one or more distant facilities and the ability to send limited control instructions to those facilities. SCADA is not normally used to run factories, but some factories may be large enough to benefit from using SCADA. SCADA systems tend to be large facilities such as a group of oil and gas wells, an electric power transmission grid, or an irrigation system.

SCADA systems allow a human operator to make changes to distant controllers in a large, geographically dispersed facility. This can include making set point changes, opening and closing valves and gathering measurement information. The economic benefits of such control are considerable as plants grow in scale and particularly where a component under control is not only at a distance, but physically hard to reach.

Such technology is best applied to control processes which are geographically dispersed and are relatively simple to monitor and control, but need frequent intervention. Example uses are[17] –

- Groups of hydroelectric stations
- Oil or gas production facilities
- Oil or gas pipelines
- Electric transmission systems
- Irrigation systems
- A heavy oil upgrader [5]

This does not mean that SCADA systems cannot apply more complex forms of control (e.g., remote control or automated responses).

Typical signals gathered from remote locations include various kinds of alarms, analog and digital values relating to valves and switches and totalized measurements. Signals to SCADA systems are limited to bit and analog values addressed to a particular device at the process. A digital signal sets one of two states (e.g., a signal setting a switch from OFF to ON). An analog value might set the % aperture of a valve. It should be noted that SCADA systems are not just telemetry systems gathering data in the field, communication is always two-way with central systems interrogating perimeter devices to receive updates on the system's response and being used by operators to send instructions.

The elements of a modern SCADA system are [17] –

- Human operator
- I/O device

[4]Some of the information is drawn from the author's own experience in carrying out security assessments on SCADA systems.

[5]Controlled by DCS, but far enough away from the control room that it would be expensive to install dedicated data cables, leading to the use of SCADA communications.

- Master Terminal Unit(MTU)
- Historian
- Other computer hosts and network devices
- Communication media and devices (e.g., modem or radio)
- Process control devices (RTU)
- Field devices (sensors or actuators)

We describe each of these elements in turn.

Operator The operator is an individual trained to control the system by considering data presented to him (or her) by the I/O device from the MTU and issuing instructions to deal with any process control issues which arise.

I/O Device The I/O device is generally a computer workstation which presents data to the operator from the MTU and issues instructions from the operator to the MTU to be passed to the RTU and field devices. There may also be workstations, accessed by the operators or other IT support staff, used to monitor telecommunication, network and host devices on the local, or wide area network, but technically these are not part of the SCADA system as they are not involved in supervisory control and data acquisition. Variants on I/O devices include large scale graphic displays of the system state on control room walls and the use of lights and/or audible alarms to ensure that certain messages from the system are not ignored by operators. Terminal screen interfaces may hold text and graphics and input can be supplied by keyboard or pointing device.

Master Terminal Unit (MTU) The MTU is a computer which can monitor and control the process, using a scheduler to repeat instructions, even when the operator is not present. For example, regularly requesting an update from an RTU. Large-scale, complex SCADA systems may have more than one MTU, each one controlling a different segment of the whole process.

Historian The historian is a database server which records the data acquired from the SCADA system, for example, for trends analysis and other more complex statistical operations. It may be associated with other computer hosts which can apply such analyses and present management information (i.e., a reporting and presentation infrastructure layer) back to corporate networks associated with the SCADA system.

Computer Hosts and Network Devices Modern SCADA systems do not exist in isolation. They are generally linked to a corporate network and, as already stated, may contain an analysis, reporting and presentation layer in addition to a supervisory control layer. Other kinds of devices include not only network devices (such as routers and switches) found on any local area network, but also protocol translators which translate IP protocols into serial protocols and *vice versa* for communication with field equipment (including process control devices). There may also be security appliances such as firewalls, IDS (intrusion detection systems)and AAA (Access, Authentication, Accounting) servers. Such systems may be linked by WAN to failover sites located at alternative operation centers which are used for disaster recovery or business contingency purposes in the event of a large-scale destructive security incident at the primary operations center (such as a fire). Finally, remote access to the system may be supported where operators utilize a VPN to access workstation functionality. RAS technology as well a terminal services and/or virtual systems may support this access.

Communication Devices Example communication devices include LAN and WAN routers and switches, radio and dial-up modems, satellite links, GPRS, cellular networks, telephony and Internet VPN. Communication is two-way and is normally implemented using a master-slave protocol where the MTU initiates the communication and the RTU responds. Communication is scheduled to allow the MTU to gather information from RTUs in a timely fashion. Hence, to use an analogy, MTUs "sweep" PCDs like a radar to update their information on the state of the plant. It should be noted, however, that this *master-slave* communication is starting to be replaced by peer-to-peer techniques as PCDs become more powerful computationally. In a peer-to-peer system, PCDs initiate communication and provide updates using exception report (i.e., they report significant changes).

Process Control Devices SCADA use Remote Terminal Units (RTU) for control (which may be PLCs) and which refer to as PCD. It should be noted that PCD previously consisted of dedicated microprocessing units and these are replaced in modern systems with programmable computers which operate a variety of proprietary and open source operating systems in support of their application logic. PCD scan processes under control in real time, making continual adjustments to plant settings to maintain values and set points determined by the operator.

Field Devices PCD are connected to sensors which gather data about the plant under control. Sensors may be electronic, or use a combination of mechanical (e.g., hydraulic or pneumatic) and electronic components. In general, they send a varying current to the PCD inputs which is translated into analog or digital signals. PCD also connect to actuators which are electro-mechanical devices which act on the plant, e.g., altering the setting on a valve or changing a switch from ON to OFF. Actuators may consist of solenoids or again translate varying electrical signals to pneumatic or hydraulic forces. Both sensors and actuators may use electronics and modern trends are seeing the replacement of customized microprocessors with programmable chips.

B.5.2 Distributed Control Systems

DCS contain the following elements in common with SCADA systems –

- Human operator
- I/O device

- Master Terminal Unit(MTU)
- Historian
- Other computer hosts and network devices
- Process control devices (RTU)
- Sensors or actuators

The primary difference (these days) is that DCS, in general, act locally rather than remotely and can, therefore, exercise control at a more granular level (if required) using closed loop control because they are not limited in the same way by the difficulties of communication over long distances, whereas SCADA exercise supervisory control in general using open loop control techniques.

B.5.3 Historical Development

SCADA technology emerged from discoveries in telemetry through the development of digital communication and radio technology leading ultimately to a dependence on sophisticated computing and modern telecommunications technology [17]. Initially, electronic control systems and control devices were confined to a single local and the systems were not connected to external networks. Early systems would consist of a single minicomputer or PLC which interfaced with local controllers (normally custom-built microprocessors), sensors and actuators, i.e., a distributed control system architecture (DCS). Companies and vendors developed various communication protocols, generally proprietary, to operate these systems. As computer technology developed, business managers pushed to be supplied with real-time information about remote operations and the attractions of remote data acquisitions for organizations with geographically dispersed operations are obvious, leading to the development of SCADA [65].

The evolution of SCADA and DCS, normally called industrial control systems (ICS), has continued in the modern era as microprocessors have gradually been replaced by RTUs and PLCs which are essentially microcomputers with proprietary or open-source operating systems, running open communication protocols, or, at least, adapted versions of such protocols. For example, Modbus, a serial protocol, is being replaced by Modbus TCP/IP. At the same time, two other changes are happening. Organizational management continues to drive for information in real-time which can be incorporated into corporate decision-making. Hence PCN are being increasingly linked to corporate networks as well as using the Internet as an additional communication medium. In addition, the labor economy is changing leading to increasing use of contractors and a reduction in reliance on full-time staff [14].

B.5.4 ICS as Distributed Environments

Clearly, ICS fall into the category of distributed systems. Processing is carried out at various nodes and the results communicated with other processes to achieve the goal of the system. Distributed processing on ICS may be subject to failure and communications may be subject to both loss and delay leading to the non-linear ordering of messages and process actions.

It should be noted, however, that the potential impact of these effects is dampened by the design and operation of such systems. MTUs carry out scheduled sweeps of RTUs to gather stored information about system state (i.e., the state of the plant under control) at various points and this data collection is timed to provide meaningful information about changes in state. Loss of communication will lead to repeated attempts to garner the information (in faster time) and delays are generally compensated for by seeking or using fresh information. MTU applications are in general designed to cope with missing, or even delayed, data, using estimation techniques to compensate. The systems themselves are subject to active intervention in real time by systems administrators to compensate for equipment, operating system, communications and applications failures. In short, such systems have been designed to be quasi-deterministic (rather than non-deterministic) in their operation [65].

Nomenclature

$\alpha, \beta, \ldots'$	In π-calculus, silent functions.
$(S; \leq)$ or $(S, \leadsto)$.	In distributed systems, order relations.
$+$	In π-calculus, contingent choice.
2^A	Power set of A
$::=$	In π-calculus, process definition operator.
$\bar{x}\langle\ldots\rangle_c$	In π-calculus, send a name by channel x.
$\ddot{x}$	Second order differential product.
$\dot{x}$	First order differential product.
$\mathcal{A}$	A set whose elements are sets
$\oplus$	In π-calculus, the exclusive sum.
$\preceq$	In distributed systems, the relationship "precedes".
$\rightarrow$	In π-calculus, the "reduction" operator, used in proofs.
$\supset$	In π-calculus, the function output operator.
A	Set
ab	In algebra, the product $a.b$. In process algebra, a sequence of states.
e_i	In distributed systems, an individual event belonging to a process P_i
$P, Q, R, \ldots$	Processes
S_i, s_i	In distributed systems, set of states and an individual state belonging to process P_i
$U, T, R, \ldots$	In engineering formulas, variable names.
$x(ldots)_c$	In π-calculus, receive a name by x.
$x(s).y(z)$	In process calculus, a sequence of events are separated by the period operator.
$x, y, z, \ldots$	Variable names

Actuators	Parts of control units which act on kinetic elements such as switches or values in a plant.
CCS	Calculus of Communicating Systems, a process algebra created by Robert Milner [88].
CI	Critical Infrastructure - refers to systems which provide vital services to enable human survivability.
COTS	"commercial-off-the-shelf" refering to software applications or systems which can be bought ready made rather than developed bespoke.
DCS	Distributed control system, consisting of local control units and actuators.
Deposet	De-composed partially order set.
field system	Another name for a distributed control system.
ICS	Industrial Control Systems
IP Traceback	A technique using message routing information to trace message origins.
OOB	Out of band, referring to communication channels not used by normal network traffic and assumed secure.
PCD	Process Control Devices - a categorical reference to multiple types of control devices.
Plant	A process under control or a set of differential equations representing a process under control.
Plants	Kinetic processes in ICS, controlled by computing systems.
PLC	Process logical controller, a computational device used for process control.
Poset	Partially ordered set. A set with a relation $(P; \leq)$ defined over its elements.
Process Control Network	A way of referring to SCADA and DCS systems jointly.
RTU	Remote Terminal Unit, a computational device used for process control.
SCADA	Supervisory control and data acquisition, a system which uses telemetry and command signals to control kinetic processes.
VMM	Virtual Machine Manager

Index

Bibliography

[1] Aaraj, N., Raghunathan, A., and Jha, N. K. Dynamic Binary Instrumentation-Based Framework for Malware Defense. In *Proceedings of Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2008)* (Heidelberg, Germany, July 2008), D. Zamboni, Ed., vol. 5137 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 64–87. 17

[2] Abadi, M., and Gordon, A. D. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput. 148*, 1 (1999), 1–70. 162

[3] Al-Duwairi, B., and Govindarasu, M. Novel Hybrid Schemes Employing Packet Marking and Logging for IP Traceback. *IEEE Transactions on Parallel and Distributed Systems 17*, 5 (May 2006), 403 – 418. 19

[4] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. The landscape of parallel computing research: A view from berkeley. Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006. Available from: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html. 105

[5] Baeten, J., Basten, T., and Reniers, A. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2009. Available from: http://books.google.co.uk/books?id=P7MdQAAACAAJ. 8, 12

[6] Baeten, J., van Beek, D., Cuijpers, P., Reniers, M., Rooda, J., Schiffelers, R., and Theunissen, R. Model-based engineering of embedded systems using the hybrid process algebra chi. *Electronic Notes in Theoretical Computer Science 209*, 0 (2008), 21 – 53. ¡ce:title¿Proceedings of the {LIX} Colloquium on Emerging Trends in Concurrency Theory (LIX 2006)¡/ce:title¿. Available from: http://www.sciencedirect.com/science/article/pii/S1571066108002181. 152

[7] Baeten, J. C. M. A brief history of process algebra. *Theor. Comput. Sci. 335*, 2-3 (May 2005), 131–146. Available from: http://dx.doi.org/10.1016/j.tcs.2004.07.036. 3, 11

[8] Baliga, A., Ganapathy, V., and Iftode, L. Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Secur. Comput. 8*,

5 (Sept. 2011), 670–684. Available from: http://dx.doi.org/10.1109/TDSC.2010.38. 16, 22, 23

[9] BALIGA, A., KAMAT, P., AND IFTODE, L. Lurking in the Shadows: Identifying Systemic Threats to Kernel Data. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P 2007)* (Piscataway, NJ, USA, May 2007), IEEE Press, pp. 246–251. 15, 22, 23, 95, 99, 148

[10] BASIN, D., AND CREMERS, C. From dolev-yao to strong adaptive corruption: Analyzing security in the presence of compromising adversaries, 2009. cas.cremers@inf.ethz.ch 14301 received 12 Feb 2009, last revised 26 Feb 2009. Available from: http://eprint.iacr.org/2009/079. 12, 36

[11] BENCSÁTH, B., PÉK, G., BUTTYÁN, L., AND FELEGYHAZI, M. Duqu: A stuxnet-like malware found in the wild. Tech. rep., BME CrySyS Lab., October 2011. First published in cut-down form as appendix to the Duqu report of Symantec. 1, 14

[12] BENETTI, D., MERRO, M., AND VIGANÒ, L. Model Checking Ad Hoc Network Routing Protocols: ARAN vs. endairA. In *SEFM* (2010), pp. 191–202. 11, 20

[13] BEQUETTE, B. *Process Control: Modeling, Design, and Simulation*. Prentice-Hall International Series in the Physical and Chemical Engineering Sciences. Prentice Hall PTR, 2003. Available from: http://books.google.co.uk/books?id=PdjHYm5e9d4C. 11, 20, 149, 155, 156

[14] BIGHAM, J., GAMEZ, D., AND LU, N. Safeguarding SCADA Systems with Anomaly Detection. In *Proceedings of the Second International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security (MMM-ACNS 2003)* (St. Petersburg, Russia, Sept. 2003), V. Gorodetsky, L. Popyack, and V. Skormin, Eds., vol. 2276 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 171–182. 20, 184

[15] BISTARELLI, S., PERETTI, P., AND TRUBITSYNA, I. Analyzing Security Scenarios Using Defence Trees and Answer Set Programming. *Electron. Notes Theor. Comput. Sci. 197*, 2 (Feb. 2008), 121–129. Available from: http://dx.doi.org/10.1016/j.entcs.2007.12.021. 13

[16] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, 3 ed. O'Reilly, 2005. 11, 177

[17] BOYER, S. *SCADA: Supervisory Control and Data Acquisition*. International Society of Automation, 2009. 1, 11, 13, 180, 181, 184

[18] BRAVO, P., AND GARCIA, D. Proactive detection of kernel-mode rootkits. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on* (2011), pp. 515–520. 22

[19] BRAYNOV, S., AND JADLIWALA, M. Representation and Analysis of Coordinated Attacks. In *Proceedings of the 2003 ACM Workshop on Formal methods in*

Security Engineering (New York, NY, USA, 2003), FMSE '03, ACM, pp. 43–51. Available from: http://doi.acm.org/10.1145/1035429.1035434. 13

[20] BRAYNOV, S., AND JADLIWALA, M. Detecting Malicious Groups of Agents. In *IEEE First Symposium on Multi-Agent Security and Survivability, 2004* (aug. 2004), pp. 90 – 99. 13

[21] BYRES, E. Patching for control systems security - a broken model? In *68th Annual Instrumentation Conference* (2013). 1

[22] BYRES, E., AND HOFFMAN, D. The Myths and Facts behind Cyber Security Risks for Industrial Control Systems. Tech. rep., Department of Computer Science, University of Victoria, Victoria, BC, Canada, Apr. 2004. 17

[23] CARDENAS, A. A., ROOSTA, T., AND SASTRY, S. Rethinking security properties, threat models, and the design space in sensor networks: A case study in scada systems. *Ad Hoc Netw. 7*, 8 (2009), 1434–1447. 12, 16, 20

[24] CAVALLARO, L., SAXENA, P., AND SEKAR, R. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2008)* (Heidelberg, Germany, July 2008), D. Zamboni, Ed., vol. 5137 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 143–163. 15

[25] CHEE-WOOI, T., MANIMARAN, G., AND CHEN-CHING, L. Cybersecurity for critical infrastructures: Attack and defense modeling. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 40*, 4 (July 2010), 853 –865. 20

[26] CHEN, T. M., AND ABU-NIMEH, S. Lessons from Stuxnet. *IEEE Computer 44*, 4 (2011), 91–93. 16

[27] CHEUNG, S., DUTERTRE, B., FONG, M., LINDQVIST, U., SKINNER, K., AND VALDES, A. Using Model-based Intrusion Detection for SCADA Networks. In *Proceedings of the SCADA Security Scientific Symposium* (Miami Beach, FL, USA, Jan. 2007), pp. 127–134. 20, 49

[28] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-Aware Malware Detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P 2005)* (Piscataway, NJ, USA, May 2005), IEEE Press, pp. 32–46. 2, 15, 17, 20

[29] CHUVAKIN, A. An Overview of Unix Rootkits. White Paper, iDefense Laboratories, 2003. iDefence Inc., 14151 Newbrook Suite, Chantilly, VA 20151. 2, 14, 16, 54

[30] COUTINHO, M. P., LAMBERT-TORRES, G., DA SILVA, L. E. B., DA SILVA, J. G. B., NETO, J. C., BORTONI, E., AND LAZAREK, H. Attack and Fault Identification in Electric Power Control Systems: An Approach to Improve the Security. In *Proceedings of Power Tech 2007* (Lausanne, Switzerland, July 2007), IEEE Press, pp. 103–107. 20

[31] DABNEY, J. B., AND HARMAN, T. L. *Mastering SIMULINK*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. 127, 155

[32] DAHL, O., AND WOLTHUSEN, S. Modeling and execution of complex attack scenarios using interval timed colored petri nets. In *Information Assurance, 2006. IWIA 2006. Fourth IEEE International Workshop on* (2006), pp. 12 pp.–168. 13

[33] DAVID, F. M., CHAN, E., CARLYLE, J. C., AND CAMPBELL, R. H. Cloaker: Hardware supported rootkit concealment. In *IEEE Symposium on Security and Privacy* (2008), IEEE Computer Society, pp. 296–310. 16

[34] DAVIDE SANGIORGI AND DAVID WALKER. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001. 3, 11, 13, 44, 58, 160

[35] DAYHARSH, C. A., AND DEL VECCHIO, H. W. Thermal Death Time Studies on Beer Spoilage Organisms. *Proceedings of the American Society of Brewing II* (1952), 48–52. 124

[36] DEAN, D., FRANKLIN, M., AND STUBBLEFIELD, A. An Algebraic Approach to IP Traceback. *ACM Transactions on Information System Security 5* (May 2002), 119–137. 18, 19

[37] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS 2008)* (New York, NY, USA, Oct. 2008), P. Ning, P. Syverson, and S. Jha, Eds., ACM Press, pp. 51–62. 21, 152

[38] DOLEV, D., AND YAO, A. On the security of public key protocols. *IEEE Transactions on Information Theory 29*, 2 (Mar 1983), 198 – 208. 2, 35, 38

[39] DONG YAN, YULONG WAN, S. S., AND YANG, F. A Precise and Practical IP Tracekback Technique Base on Packet Marking and Logging. *Journal of Information Science and Engineering 28*, 3 (May 2012), 453–470. 19

[40] EMIGH, A. The Crimeware Landscape: Malware, Phishing, Identity Theft and Beyond. *Journal of Digital Forensic Practice 1* (2006), 245 – 260. 17

[41] FALLIERE, N., MURCHU, L. O., AND CHIEN, E. W32.Stuxnet Dossier. Tech. rep., Symantic Security Response, Oct. 2010. 1, 2, 14

[42] FU, Y., HE, J., AND LI, G. A Distributed Intrusion Detection Scheme for Mobile Ad Hoc Networks. *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007) 02* (2007), 75–80. 17

[43] GAMEZ, D., NADJM-TEHRANI, S., BIGHAM, J., BALDUCELLI, C., BURBECK, K., AND CHYSSLER, T. *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*. John Wiley & Sons, New York, NY, USA, 2005, ch. Safeguarding Critical Infrastructures. 1, 15, 17, 20

[44] GARBER, L. Security, privacy, policy, and dependability roundup. *Security Privacy, IEEE 10*, 6 (2012), 6–8. 14

[45] GARFINKEL, T., AND ROSENBLUM, M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network And Distributed System Security Symposium (NDSS '03)* (San Diego, CA, USA, Feb. 2003), Internet Society. 21, 148, 152

[46] GARG, V. *Elements of Distributed Computing*. Wiley-Interscience, 2002. Available from: `http://books.google.co.uk/books?id=NlVBtVPeR0QC`. 2, 11, 24, 39, 96, 97, 100, 106, 168, 169, 172, 173

[47] GENGE, B., AND SIATERLIS, C. Investigating the effect of network parameters on coordinated cyber attacks against a simulated power plant. In *Critis - 2011, 6th International Conference on Critical Information Infrastructures Security* (September 2011), vol. 6, pp. 143 – 153. 74

[48] GÖBEL, O., FRINGS, S., GÜNTHER, D., NEDON, J., AND SCHADT, D., Eds. *IT-Incidents Management & IT-Forensics - IMF 2008, Conference Proceedings, September 23-25, 2008, Mannheim, Germany* (2008), vol. 140 of *LNI*, GI. 23, 199

[49] HAO, J., HAO, Y.-J., DING, Z.-J., AND SONG, L.-T. A methodology to detect kernel level rootkits based on detecting hidden processes. In *Apperceiving Computing and Intelligence Analysis, 2008. ICACIA 2008. International Conference on* (2008), pp. 359–361. 22

[50] HEASMAN, J. Implementing and Detecting an ACPI BIOS Root Kit. Briefing at Black Hat 2005 (Las Vegas, NV, USA), July 2005. 16

[51] HOFMEYR, S. A., FORREST, S., AND SOMAYAJI, A. Intrusion detection using sequences of system calls. *Journal of Computer Security, 1998 6*, 3 (1998), 151–180. 17

[52] HOGLUND, G., AND BUTLER, J. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005. 16

[53] HRISCHUK, C. E., AND WOODSIDE, C. M. Logical clock requirements for reverse engineering scenarios from a distributed system. *IEEE Trans. Software Eng. 28*, 4 (2002), 321–339. 94

[54] HUANG, Y., STAVROU, A., GHOSH, A. K., AND JAJODIA, S. Efficiently Tracking Application Interactions using Lightweight Virtualization. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security (VMSec 2008)* (New York, NY, USA, Oct. 2008), J. Nieh and A. Stavrou, Eds., ACM Press, pp. 19–28. 21

[55] INGOLS, K., CHU, M., LIPPMANN, R., WEBSTER, S., AND BOYER, S. Modeling Modern Network Attacks and Countermeasures Using Attack Graphs. In *ACSAC '09. Annual Computer Security Applications Conference, 2009.* (dec. 2009), pp. 117 –126. 13

[56] IVAN SKLYAROV. *Programming Linux Hacker Tools Uncovered*. A-LIST, LLC, 2007, ch. 21. 16, 109

[57] IZADDOOST, A., OTHMAN, M., AND RASID, M. Accurate icmp traceback model under dos/ddos attack. In *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on* (2007), pp. 441–446. 19

[58] JIANG, X., WANG, X., AND XU, D. Stealthy Malware Detection through VMM-based "out-of-the-box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)* (New York, NY, USA, Oct. 2007), S. De Capitani di Vimercati and P. Syverson, Eds., ACM Press, pp. 128–138. 21

[59] KACHIRSKI, O., AND GUHA, R. Effective intrusion detection using multiple sensors in wireless ad hoc networks. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 2* (Washington, DC, USA, 2003), IEEE Computer Society, p. 57.1. 17

[60] KING, S. T., MAO, Z. M., LUCCHETTI, D. G., AND CHEN, P. M. Enriching Intrusion Alerts Through Multi-host Causality. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)* (2005). Available from: http://www.isoc.org/isoc/conferences/ndss/05/proceedings/papers/camera.pdf. 17

[61] KIREMIRE, A., BRUST, M., AND PHOHA, V. A prediction based approach to ip traceback. In *Local Computer Networks Workshops (LCN Workshops), 2012 IEEE 37th Conference on* (2012), pp. 1022–1029. 19

[62] KO, C. Logic Induction of Valid Behavior Specifications for Intrusion Detection. *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy 00* (2000), 0142. 17

[63] KORDY, B., MAUW, S., RADOMIROVIĆ, S., AND SCHWEITZER, P. Foundations of Attack-Defense Trees. In *Proceedings of the 7th International Conference on Formal aspects of Security and Trust* (Berlin, Heidelberg, 2011), FAST'10, Springer-Verlag, pp. 80–95. Available from: http://dl.acm.org/citation.cfm?id=1964555.1964561. 13

[64] KRAUS, T., KUHL, P., WIRSCHING, L., BOCK, H., AND DIEHL, M. A Moving Horizon State Estimation Algorithm Applied to the Tennessee Eastman Benchmark Process. In *2006 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems* (Heidelberg, Germany, Sep. 2006), pp. 377 –382. 149

[65] KRUTZ, R. *Securing SCADA systems*. Wiley Pub., 2005. 1, 11, 13, 17, 180, 184, 185

[66] KUMAR, P., AND SELVAKUMAR, S. Distributed denial-of-service (ddos) threat in collaborative environment - a survey on ddos attack tools and traceback mechanisms. In *Advance Computing Conference, 2009. IACC 2009. IEEE International* (2009), pp. 1275–1280. 3, 18

[67] LANGNER, R. *Robust Control System Networks: How to Achieve Reliable Control After Stuxnet*. Momentum Press, 2011. Available from: http://books.google.co.uk/books?id=ucWTZwEACAAJ. 1, 11, 180

[68] LAUF, A. P., PETERS, R. A., AND ROBINSON, W. H. Embedded Intelligent Intrusion Detection: A Behavior-Based Approach. *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops 1* (2007), 816–821. 17

[69] LI, F., LAI, A., AND DDL, D. Evidence of advanced persistent threat: A case study of malware for political espionage. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on* (oct. 2011), pp. 102 –109. 14

[70] LIN, Z.-S., CARDENAS, A., AMIN, S., HUANG, Y.-L., HUANG, C.-Y., AND SASTRY, S. S. Model-based detection of attacks for process control systems. In *16th ACM Computer and Communications Security Conference* (2009), ACM, p. submitted. Available from: http://chess.eecs.berkeley.edu/pubs/600.html. 20

[71] LINDA, O., VOLLMER, T., AND MANIC, M. Neural Network based Intrusion Detection System for Critical Infrastructures. In *Proceedings of the 2009 International Joint Conference on Neural Networks (IJCNN 2009)* (Atlanta, GA, USA, June 2009), IEEE Press, pp. 1827–1834. 17

[72] LIPPMANN, R. P., AND INGOLS, K. W. An Annotated Review of Past Papers on Attack Graphs, 1998. Available from: http://en.scientificcommons.org/18618950. 13

[73] LOCASTO, M. E., PAREKH, J. J., KEROMYTIS, A. D., AND STOLFO, S. J. Towards Collaborative Security and P2P Intrusion Detection. In *Proceedings of the IEEE Information Assurance Workshop (IAW)* (June 2005). 17

[74] MAO, W. A structured operational modelling of the dolev-yao threat model. In *Security Protocols*, B. Christianson, B. Crispo, J. Malcolm, and M. Roe, Eds., vol. 2845 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 34–46. Available from: http://dx.doi.org/10.1007/978-3-540-39871-4_5. 2, 3, 12, 35, 36, 38, 45, 145, 146

[75] MAUW, S., AND OOSTDIJK, M. Foundations of Attack Trees. In *International Conference on Information Security and Cryptology ? ICISC 2005. LNCS 3935* (2005), Springer, pp. 186–198. 13, 146

[76] MCEVOY, R., AND WOLTHUSEN, S. *Agent Interaction and State Determination in SCADA Systems*. Springer-Verlag, 2012. 5, 18, 51

[77] MCEVOY, T., AND WOLTHUSEN, S. A formal adversary capability model for scada environments. In *Critical Information Infrastructures Security*, C. Xenakis and S. Wolthusen, Eds., vol. 6712 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 93–103. Available from: http://dx.doi.org/10.1007/978-3-642-21694-7_8. 5, 13, 20

[78] McEvoy, T. R., and Wolthusen, S. Defeating Node Based Attacks on SCADA Systems Using Probabilistic Packet Observation. In *Proceedings of the Sixth International Workshop on Critical Information Infrastructures Security (CRITIS 2011)* (Luzern, Switzerland, Sept. 2011), C. Xenakis and S. Wolthusen, Eds., vol. 6712 of *Lecture Notes in Computer Science*, Springer-Verlag. (in press). 5

[79] McEvoy, T. R., and Wolthusen, S. Algebraic analysis of attack impacts in critical infrastructures. In *Critical Information Infrastructures Security*, Lecture Notes in Computer Science. 2012. 5

[80] McEvoy, T. R., and Wolthusen, S. D. Using observations of invariant behaviour to detect malicious agency in distributed environments. In Göbel et al. [48], pp. 55–72. 5, 22

[81] McEvoy, T. R., and Wolthusen, S. D. Trouble brewing: Using observations of invariant behavior to detect malicious agency in distributed control systems. In *CRITIS* (2009), E. Rome and R. E. Bloomfield, Eds., vol. 6027 of *Lecture Notes in Computer Science*, Springer, pp. 62–72. 5

[82] McEvoy, T. R., and Wolthusen, S. D. Trouble Brewing: Using Observations of Invariant Behavior to Detect Malicious Agency in Distributed Control Systems. In *Proceedings of the Fourth International Workshop on Critical Information Infrastructure Security (CRITIS 2009)* (Bonn, Germany, Sept. 2009), E. Rome and R. Bloomfield, Eds., vol. 6027 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 62–72. 123, 137

[83] McEvoy, T. R., and Wolthusen, S. D. An algebra for the detection and prediction of malicious activity in concurrent systems. *2010 Fifth International Conference on Systems 0* (2010), 125–133. 5, 22

[84] McEvoy, T. R., and Wolthusen, S. D. Detecting sensor signal manipulations in non-linear chemical processes. In *Critical Infrastructure Protection* (2010), T. Moore and S. Shenoi, Eds., vol. 342 of *IFIP Advances in Information and Communication Technology*, Springer, pp. 81–94. 5, 42

[85] McEvoy, T. R., and Wolthusen, S. D. Host-based security sensor integrity in multiprocessing environments. In *ISPEC* (2010), J. Kwak, R. H. Deng, Y. Won, and G. Wang, Eds., vol. 6047 of *Lecture Notes in Computer Science*, Springer, pp. 138–152. 5

[86] McEvoy, T. R., and Wolthusen, S. D. A plant-wide industrial process control security problem. In *Critical Infrastructure Protection* (2011), J. Butts and S. Shenoi, Eds., vol. 367 of *IFIP Advances in Information and Communication Technology*, Springer, pp. 47–56. 5, 18

[87] McLaughlin, S. On Dynamic Malware Payloads Aimed at Programmable Logic Controllers. In *Proceedings of the 6th USENIX conference on Hot topics in security* (Berkeley, CA, USA, 2011), HotSec'11, USENIX Association, pp. 10–10. Available from: `http://dl.acm.org/citation.cfm?id=2028040.2028050`. 13

[88] MILNER, R. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999. 3, 11, 12, 187

[89] MOLINA, J., AND ARBAUGH, W. Using Independent Auditors as Intrusion Detection Systems. In *Proceedings of the Information and Communication Security* (Heidelberg, Germany, Dec. 2002), R. Deng, S. Qing, F. Bao, and J. Zhou, Eds., vol. 2513 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 291–302. 21

[90] MOSER, A., KRUEGEL, C., AND KIRDA, E. Exploring multiple execution paths for malware analysis. *Security and Privacy, IEEE Symposium on 0* (2007), 231–245. 15

[91] MOSER, A., KRUEGEL, C., AND KIRDA, E. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (Miami Beach, FL, USA, Dec. 2007), IEEE Press, pp. 421–430. 15

[92] MOTTA PIRES, P. S., AND OLIVEIRA, L. A. H. G. Security Aspects of SCADA and Corporate Network Interconnection: An Overview. In *Proceedings of the 2006 International Conference on Dependability of Computer Systems (DepCos – RELCOMEX 2006)* (Szklarska Proeba, Poland, May 2006), IEEE Press, pp. 127–134. 17

[93] NICK L. PETRONI, TIMOTHY FRASER, AARON WALTERS, WILLIAM A. ARBAUGH . An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *Proceedings of the 15th USENIX Security Symposium* (2006). 2, 15, 22, 148

[94] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)* (New York, NY, USA, Mar. 2008), S. Eggers and J. Larus, Eds., ACM Press, pp. 308–318. 21

[95] OPLINGER, J., AND LAM, M. S. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)* (New York, NY, USA, Oct. 2002), K. Gharachorloo, Ed., ACM Press, pp. 184–196. 21

[96] PARK, K., AND LEE, H. On the Effectiveness of Probabilistic Packet Marking for IP Traceback Under Denial of Service Attack. In *INFOCOM 2001: Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies* (2001), vol. 1, pp. 338 –347. 19, 69, 73

[97] PARK, S. Malware expert: Execution tracking. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2012 Third* (2012), pp. 48–55. 22

[98] PATSAKIS, C., AND ALEXANDRIS, N. New Malicious Agents and SK Virii. In *International Multi-Conference on Computing in the Global Information Technology, 2007* (march 2007), p. 29. 13, 36, 49

[99] PEARL, J. *Causality: Models, Reasoning, and Inference*, 2nd ed. Cambridge University Press, Cambridge, United Kingdom, 2009. 20, 135, 146

[100] PETRONI, JR., N. L., AND HICKS, M. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)* (New York, NY, USA, Oct. 2007), S. D. C. di Vimercati and P. Syverson, Eds., ACM Press, pp. 103–115. 15, 21

[101] PREDA, M. D., CHRISTODORESCU, M., JHA, S., AND DEBRAY, S. A Semantics-based Approach to Malware Detection. *SIGPLAN Not. 42*, 1 (Jan. 2007), 377–388. Available from: http://doi.acm.org/10.1145/1190215.1190270. 15, 20, 22, 27, 94, 99, 103

[102] RHEE, J., RILEY, R., XU, D., AND JIANG, X. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on* (2009), pp. 74–81. 22

[103] RILEY, R., JIANG, X., AND XU, D. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)* (Heidelberg, Germany, Sept. 2008), R. Lippmann, E. Kirda, and A. Trachtenberg, Eds., vol. 5230 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–20. 21

[104] RRUSHI, J., AND KANG, K.-D. Detecting Anomalies in Process Control Networks. In *Critical Infrastructure Protection III* (Hanover, NH, USA, Mar. 2009), C. Palmer and S. Shenoi, Eds., Springer-Verlag, pp. 151–165. (Proceedings of the Third Annual IFIP WG 11.10 International Conference on Critical Infrastructure Protection). 17

[105] RUTKOWSKA, J. Beyond the CPU: Defeating Hardware Based RAM Acquisition. Defcon 2007. 16, 22

[106] SAMUEL T. KING AND PETER M. CHEN AND YI-MIN WANG AND CHAD VERBOWSKI AND HELEN J. WANG AND JACOB R. LORCH. SubVirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&#38;P'06)* (Los Alamitos, CA, USA, 2006), vol. 0, IEEE Computer Society, pp. 314–327. 2, 16

[107] SANDRA RING AND ERIC COLE. Taking a Lesson from Stealthy Rootkits. *IEEE Security and Privacy 02*, 4 (2004), 38–45. 15

[108] SATTARI, P., GJOKA, M., AND MARKOPOULOU, A. A network coding approach to ip traceback. In *Network Coding (NetCod), 2010 IEEE International Symposium on* (2010), pp. 1–6. 19

[109] SAVAGE, S., WETHERALL, D., KARLIN, A., AND ANDERSON, T. Network Support for IP Traceback. *IEEE/ACM Transactions on Networking 9*, 3 (June 2001), 226 –237. 18, 19, 64, 65, 74

[110] SCHLESSER, J. E., ARMSTRONG, D. J., CINAR, A., RAMANAUSKAS, P., AND NEGIZ, A. Automated Control and Monitoring of Thermal Processing Using High Temperature, Short Time Pasteurization. *Journal of Dairy Science 80*, 10 (Mar. 1997), 2291–2296. 17

[111] SEEGER, M., AND WOLTHUSEN, S. Observation mechanism and cost model for tightly coupled asymmetric concurrency. In *Systems (ICONS), 2010 Fifth International Conference on* (2010), pp. 158–163. 152

[112] SEN, K., ROŞU, G., AND AGHA, G. Online efficient predictive safety analysis of multithreaded programs. *Int. J. Softw. Tools Technol. Transf. 8*, 3 (2006), 248–260. 22, 98, 102, 103

[113] SHENG, S., CHAN, W., LI, K., XIANZHONG, D., AND XIANGJUN, Z. Context information-based cyber security defense of protection system. *IEEE Transactions on Power Delivery 22*, 3 (jul 2007), 1477–1481. 17, 20, 27

[114] SIMON, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley, 2006. Available from: `http://books.google.co.uk/books?id=UiMVoP_7TZkC`. 18, 20, 152

[115] SONG, D. X., AND PERRIG, A. Advanced and Authenticated Marking Schemes for IP Traceback. In *INFOCOM 2001: Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies* (2001), vol. 2, pp. 878 –886. 19, 70, 74

[116] SVENDSEN, N. K., AND WOLTHUSEN, S. D. Modeling and Detection of Anomalies in Critical Infrastructure Networks. In *Critical Infrastructure Protection II* (Arlington, VA, USA, Mar. 2008), M. Papa and S. Shenoi, Eds., Springer-Verlag, pp. 101–107. (Proceedings of the Second Annual IFIP WG 11.10 International Conference on Critical Infrastructure Protection). 16, 20, 24

[117] SZOR, P. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005. 2, 14, 21, 108

[118] TESAURO, G., CHESS, D. M., WALSH, W. E., DAS, R., SEGAL, A., WHALLEY, I., KEPHART, J. O., AND WHITE, S. R. "a multi-agent systems approach to autonomic computing". In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1* (Washington, DC, USA, 2004), AAMAS '04, IEEE Computer Society, pp. 464–471. Available from: `http://dx.doi.org/10.1109/AAMAS.2004.23`. 13

[119] TSAUR, W., AND CHEN, Y.-C. Exploring rootkit detectors' vulnerabilities using a new windows hidden driver based rootkit. In *Social Computing (SocialCom), 2010 IEEE Second International Conference on* (2010), pp. 842–848. 16

[120] VERBA, J., AND MILVICH, M. Idaho National Laboratory Supervisory Control and Data Acquisition Intrusion Detection System (SCADA IDS). In *Proceedings of the 2008 IEEE Conference on Technologies for Homeland Security* (Waltham, MA, USA, May 2008), IEEE Press, pp. 469–473. 16, 20, 63

[121] WAN FOKKINK. *Introduction to Process Algebra*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000. 3

[122] WANG, J. A rule-based approach for rootkit detection. In *The 2nd IEEE International Conference on Information Management and Engineering (ICIME)* (april 2010). 22, 23

[123] WANG, X. R., LIZIER, J. T., OBST, O., PROKOPENKO, M., AND WANG, P. Spatiotemporal Anomaly Detection in Gas Monitoring Sensor Networks. In *Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN 2008)* (Bologna, Italy, Jan. 2008), R. Verdone, Ed., vol. 4913 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 90–105. 20

[124] WATTS, D. Security & Vulnerability in Electric Power Systems. In *Proceedings of the 35 North American Power Symposium (NAPS 2003)* (Rolla, MO, USA, Oct. 2003), pp. 559–566. 17

[125] WEN, Y., HUANG, M., ZHAO, J., AND KUANG, X. Implicit detection of stealth software with a local-booted virtual machine. In *Information Sciences and Interaction Sciences (ICIS), 2010 3rd International Conference on* (2010), pp. 152–157. 21

[126] WILHELM, J., AND CKER CHIUEH, T. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID 2007)* (Heidelberg, Germany, Sept. 2007), C. Kruegel, R. Lippmann, and A. Clark, Eds., vol. 4637 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 219–235. 21

[127] WILLIAMS, P. D., AND SPAFFORD, E. H. CuPIDS: An Exploration of Highly Focused, Co-Processor-based Information System Protection. *Computer Networks 51*, 5 (Apr. 2007), 1284–1298. 21, 148, 152

[128] WONG, T. Y., WONG, M. H., AND LUI, C. S. A Precise Termination Condition of the Probabilistic Packet Marking Algorithm. *IEEE Transactions on Dependable and Secure Computing 5*, 1 (Mar. 2008), 6–21. 18

[129] WOOLDRIDGE, M. *An Introduction to MultiAgent Systems*. Wiley, 2009. Available from: `http://books.google.co.uk/books?id=X3ZQ7yeDn2IC`. 13, 31

[130] WU, R., LI, W., AND HUANG, H. An attack modeling based on hierarchical colored petri nets. In *Computer and Electrical Engineering, 2008. ICCEE 2008. International Conference on* (2008), pp. 918–921. 13, 145

[131] XIANG, Y., ZHOU, W., AND GUO, M. Flexible Deterministic Packet Marking: An IP Traceback System to Find the Real Source of Attacks. *IEEE Trans. Parallel Distrib. Syst. 20*, 4 (Apr. 2009), 567–580. Available from: `http://dx.doi.org/10.1109/TPDS.2008.132`. 19

[132] YANG, D., USYNIN, A., AND HINES, J. W. Anomaly-Based Intrusion Detection for SCADA Systems. In *Proceedings of the 5th International Topical Meeting*

on Nuclear Plant Instrumentation, Control, and Human Machine Interface Technologies (NPIC&HMIT 06) (Albuquerque, NM, USA, Nov. 2006), American Nuclear Society. 17, 20

[133] YE, F., YANG, H., AND LIU, Z. Catching "Moles" in Sensor Networks. In *ICDCS* (2007), p. 69. 19, 70

[134] YEE, B., AND TYGAR, J. D. Secure Coprocessors in Electronic Commerce Applications. In *Proceedings of the First USENIX Workshop on Electronic Commerce* (New York, NY, USA, July 1995), D. E. Geer, Ed., USENIX Press, pp. 14–14. 21

[135] ZHANG, X., VAN DOORN, L., JAEGER, T., PEREZ, R., AND SAILER, R. Secure Coprocessor-Based Intrusion Detection. In *Proceedings of the 10th ACM SIGOPS European Workshop* (New York, NY, USA, July 2002), G. Muller and E. Jul, Eds., ACM Press, pp. 239–242. 21