View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Royal Holloway - Pure

Agent Interaction and State Determination in SCADA Systems

Thomas Richard McEvoy[2] and Stephen D. Wolthusen[1,2]
[1] Norwegian Information Security Laboratory,
Department of Computer Science,
Gjøvik University College, Norway
[2] Information Security Group,
Department of Mathematics,
Royal Holloway, University of London, UK
T.R.McEvoy@rhul.ac.uk stephen.wolthusen@rhul.ac.uk*

Abstract In critical infrastructure environments, we argue that both adversaries and operators will utilize agents to manage dynamic attack/defence interactions in future. Agent behavior and, in particular, agent interaction require adequate modelling tools to reason over such situations in distributed environments where the state (malicious or non-malicious) of a channel or process can vary dynamically depending on the actions of opposing sides in attack and defence. For this purpose, we propose an extension to applied π-calculus to model agent behavior. We apply this extended calculus to the formal analysis of a class of agent-based attacks and its detection to demonstrate its utility..

1 Introduction

Critical infrastructure systems are key targets in cyber attacks [1]. But remote attackers face problems in determining state due to limitations on communication [2,3]. At the same time, operators need to respond to sophisticated attacks in real-time and may have to operate systems in a compromised state with only partial knowledge of state. The outcome of such interactions may depend on the state (or knowledge of state) of a single channel or process. Such situations require the deployment of software agents by both sides.

We propose an extension to applied π-calculus [4] to enable us to reason over agents, defining attack and defence categories. This extension augments previous models of adversary (and operator) capability [2]. As an example application, we provide a formal model of a co-ordinated attack using agents and its detection by a set of observers, working on behalf of the operator.

Section 2 outlines related work. Section 3 describes the problem and our approach. Section 4 outlines our proposed extension to the π-calculus. Section 5 outlines the coordinated attack. Section 6 demonstrates its detection. We conclude and set out future research directions in section 7.

2 Related Work

Software agents may act on autonomous decisions based on their perception of the environment [5], possibly including learning. In recent years, malicious code has exhibited increasingly agent-like behavior. This trend is likely to continue [6,7].

Software agency also permits the attacker to launch more sophisticated, including *coordinated*, attacks. There are advantages in implementing such attacks [8,9]. Dealing with such attacks also requires operators to make dynamic interventions in the face of changing adversary behavior. This requirement is underlined by the intrinsic nature of critical infrastructure systems [10,11] which need to continue to operate 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[12] in relation to a wide range of security tasks provides a distinct advantage in countering agent-based attacks.

The pi-calculus provides a proof technique over process actions [2,13,14,4]. Algebraic techniques enable us to apply formal reasoning, including automating proofs [4]. We make use of IP Traceback algorithms in our example [15,14].

3 Motivation

Understanding agent-based attack and defence strategies will be a necessity in the defence of critical infrastructure systems. Reasoning over such interactions represents a complex set of problems, particularly where multiple co-operating agents exist and such agents can "recruit" normally trusted processes to work on their behalf [2].

We seek to show that, by extending the pi-calculus, we can enable reasoning over such complex scenarios. We also extend the adversary capability model presented in [ibid.] to allow us to consider a distinct class of attacks – not dependent on direct adversary intervention, but on malicious software agents.

4 Agent Calculus

We describe the $G\{\pi\}$-calculus which is the *goal transform π-calculus*. The basic π calculus is described in [4], while the applied version we use as a basis for this extension was introduced in [2].

$||G||_{AgentName}$ is defined by a set of inter-related goals. If G is a goal then

$$G ::= \mathbf{0}|\pi.G|\nu zG|G.G|G + G|G \oplus G|G|G'|!G|[\mathcal{L}]G \tag{1}$$

where the possible actions α of G are defined in Table 1.

where $\mathcal{L}$ is a first order logic with equivalence and ordered relations and π is a capability of the π-calculus which we will define.

Goal actions are defined by the capabilities of the applied π- calculus.

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

Table 1. $G\{\pi\}$-Calculus Syntax

$$\pi ::= \bar{x}\langle z\rangle_{\tilde{c}} | x(z)_{\tilde{c}} | \lambda | f(\tilde{v}_{\tilde{c}'}) \supset \tilde{v}'_{\tilde{c}} | [\mathcal{L}]\pi. \tag{2}$$

The respective meaning given to these capabilities is defined in Table 2.

Term	Semantic
$\bar{x}\langle z\rangle_{\tilde{c}}$	Send a name with characteristics
$x(z)_{\tilde{c}}$	Receive a name with characteristics
λ	A *silent* function
$f(\tilde{v}_{\tilde{c}'}) \supset \tilde{v}'_{\tilde{c}}$	A function over a set of names
$[\mathcal{L}]\pi$	Conditional execution of a capability

Table 2. π-calculus Terms

Constants, variable and function labels belong to the set of names which we define. Goals execute themselves until they invoke another goal, at which point they terminate. G_0 is a reserved label which represents the null goal. $\tilde{z}$ indicates a vector of names. $\coprod$ indicates a set of concurrent goals or processes. $\sum$ indicates a sum M over capabilities. A label which is an inaction represents a process action which may not be directly observable. For example, let $P := M + \lambda$ be a process(where M is a sum) then $\lambda, P \supset 0$ is an inaction or *silent function*of P. A key characteristic in our model is that processes may be overwritten by messages from another agent. Hence we need to define precisely the outcome of those message. For example, let m be a message and $P := M + \Omega|Q$ then $\Omega, m, P \supset P'$ where P' may be defined arbitrarily (though, in general, P' behaves like P except under certain conditions where it executes a different behavior useful to the adversary).

In this example, destination addresses are characteristics, so that $\langle z\rangle_{X_j}$ is assumed to route the name z to the process X_j. Routing is conditional on the characteristic and is displayed as follows $\bar{x}_i\langle z\rangle_{[X_j]}$ rather than by a more conventional $[z.r = X_j]\bar{x}_{X_j}\langle z\rangle_{X_j}$ to save ink, where $z.r$ is a dotted name which indicates the characteristic routing address of the name z. Hence any name with

the destination address $z.r = X_j$ as one of its characteristics will be routed by $\bar{x}_i$ even where $\bar{x}_i$ is not the final destination.

A proof reduction is indicated using dotted notation $||Goal.Subgoal.Action||_{Agent} \rightarrow ||Goal.Subgoal.NextAction||_{Agent}$ where $NextAction$ is any capability or a goal invocation. Goals (hence agents) interact by communicating over names which are channels. A proof reduction considers all possible reductions and any claim – for example, regarding security properties – requires to be shown for all cases in the reduction.

5 Coordinated Attack

We model a coordinated attack and its detection. Using six agents, the adversary seeks to set three valves so as to cause a critical failure, while concealing the result through data manipulation [3].

$$
\begin{aligned}
&||Stage1.SendMalwareToNode_4||_{Launch1} \quad (i = 4)\\
&||Stage2.SendMalwareToNode_{12}||_{Launch2} \quad (j = 12)\\
&||X_4.ReceiveMessage|X_{12}.ReceiveMessage|X_1 \ldots |X_{15}||_{System}\\
&\rightarrow \text{ Send malware to both nodes}\\
&||Stage1.PollSuccess||_{Launch1}\\
&||Stage2.PollSuccess||_{Launch2}\\
&||X_4.BecomeMaliciousAgent|X_{12}| \ldots ||_{System}\\
&\rightarrow \text{Check results}\\
&||Stage1.ReportSuccessToLaunch3||_{Launch1}\\
&||Stage2.SendMalware||_{Launch2} \quad (j = 11)\\
&\rightarrow \text{First attack succeeds, next attempt}\\
&||X_4^{'}||_{Agent1}\\
&||X_4|X_{12}.ReceiveMessage|X_1 \ldots |X_{15}||_{System}\\
&\rightarrow \text{Await outcome and launch if successful}\\
&||Stage3.WaitForLaunch1and2 + [Success]LaunchFinalAttack||_{Launch3} \quad (3)
\end{aligned}
$$

We start by using English-like mnemonic goal labels for scenario planning [1] Three *Launch* agents, which we define, send malicious software as a name into the system which overwrites various network nodes to transform them into additional malicious agents working on behalf of the adversary.

There are four outcomes – success for both agents, failure for one or other or failure for both. A full proof requires us to define the mechanics of attack

[1] In fact, without interference in channels and processes, this *goal* calculus would be sufficient to prove the outcome of any interactions, provided the goals were defined precisely.

and we show (in part) how we may refine the reduction in equation 4 where X_i represents a system node, g_2 is a message which infects the system and s is a boolean variable. Here we see the launch agents $L1$ and $L2$ send malicious names by two channels X_9 and X_10 into the system. In turn, these are routed to target nodes i and j and the success or failure of subversion. Once this initial subversion succeeds, the newly formed agents flag their success to agent $L3$ which launches the final part of the attack. However, the messages regarding success or failure may arrive in any order.

$$
\begin{aligned}
&||Stage1.\bar{x}_9\langle g_2\rangle_{X_i}||_{L1} \quad (i=4)\\
&||Stage2.\bar{x}_{10}\langle g_3\rangle_{X_j}||_{L2} \quad (j=12)\\
&||X_9.x_9(z)||_{System}\\
&||X_{10}.x_{10}(z)||_{System}\\
&\rightarrow \text{Send malicious messages to the system}\\
&||Stage1.x_{S1}(s)||_{L1}\\
&||Stage2.x_{S2}(s)||_{L2}\\
&||X_9.(\bar{x}_i\langle m_2\rangle_{X_i}.\bar{x}_{S1}\langle\top\rangle \oplus \phi.\bar{x}_{S1}\langle\bot\rangle_{S1})||_{Agent0}\\
&||X_{10}.((\bar{x}_j\langle m_2\rangle_{X_j}.\bar{x}_{S2}\langle\top\rangle \oplus \phi.\bar{x}_{S2}\langle\bot\rangle_{S2})||_{Agent00}\\
&||X_i.x_i(z)||_{System}\\
&|X_j.x_j(z)||_{System}\\
&\rightarrow \text{Case 1 } * \text{ Both attempts succeed}\\
&||Stage1.x_{S1}(s)||_{L1}\\
&||Stage2.x_{S2}(s)||_{L2}\\
&||X_9.\bar{x}_{S1}\langle\top\rangle_{S1})||_{Agent0}\\
&|X_{10}.\bar{x}_{S2}\langle\top\rangle_{S2})||_{Agent00}\\
&||{X_i}'|G||_{Agent2}\\
&||{X_j}'|G||_{Agent3}\\
&\rightarrow \text{Case 2.1 } X_i \text{ succeeds, while } X_j \text{ fails and is re - attempted} \qquad (4)
\end{aligned}
$$

We need to show this does not affect a planned outcome and that the third launch agent $L3$ will respond appropriately (by starting the final attack or aborting it) whatever the message order. In equation 5, we show that the messages update a boolean predicate a and the final attack only launched when it holds $TRUE$. The order in which it is updated is not relevant to the outcome. The end result of case 1 is that the final part of the attack is launched by $L3$. at this point, $L3$ will sets its target valve to $Steady$ and signal the other two agents in turn to set their target values to $Open$ and $Closed$, while concealing the attack by manipulating the signal from the controllers - equation 6 again shows this in part.

$||InitialSuccess.x_{Ad}(u)||_{L3} \quad (k = 0)$

$\rightarrow$ (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 (5)

$||X_k{}'|SetSteady.set(u, Steady) \supset u'||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$
$\rightarrow$ Define the desired 'steady' value for the valve
$||X_k{}'|SetSteady.Send.\bar{x}_s\langle u'\rangle_{C2}||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$
$\rightarrow$ Send the instruction to the Contoller
$||X_k{}'||FlagSteady||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$
$\rightarrow$ Receive messages
$||X_k{}'|FlagSteady.x_s(u)||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$
$\rightarrow$ Detect whether the value has changed.
$||X_k{}'|FlagSteady.[u = Steady]\bar{x}_s\langle s\rangle_{X_5}.G_0||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$
$\rightarrow$ Once the valve is set, signal the next agent to act
$||X_k{}'.Send(u)|Conceal||_{Agent1}$
$||X_i|Waitfor1.x_s(s)||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$
$\rightarrow$ Note, fake signals conceal the true server state
$||X_i|OpenValve||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$
$\rightarrow$ Next agent opens the valve and signals the third agent
$||X_j|CloseValve||_{Agent3}$
Attack completes (6)

6 Distributed Detection

The operator employs observer agents to make a state determination over trusted routes, alerting on critical conditions. As described elswhere [13,15,14], 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.

Our other contribution is this paper is provide a formal definition of an algorithm for *observer* agents who use this information to make a determination over state [2]. We show how the observer algorithm uses messages and copies to determine a trusted set of paths. State determination is subsequently restricted to considering messages on trusted paths. An observer is defined in equation 7. We show the initial reduction of the $Observer$ in equations 10 and 8.

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

$$
\nu \quad \tilde{k}\tilde{c}, STORE, STATE, CRITICAL, C_iTREE \quad || \bullet Observe || \tag{7}
$$

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 neighbouring node in order of communication as untrusted. It can also remove marks where a node is returned to a trusted state. We can represent marked messages using a graph, defined algebraically for the purposes of the proof. For example, equation 9 shows that X_8 is no longer a trusted node by placing a bar over the node.

[2] In [14], the IP Traceback algorithms were used for detecting the location of malicious agents which is a different goal from the one set out in this paper.

$$
\begin{aligned}
&|X_i.Send|| \quad (i = 1, 2, 3) \\
&||Observe.x_{ob_j}(z)||_{Observer_j} \quad (z \in C) \\
&\rightarrow \\
&||UpdatePath.Compare(u, z, \tilde{k}, STORE) \supset w||_{Observer_j} \\
&\rightarrow \text{ No discrepancy, message on unmarked path} \\
&||Observe||_{Observer_j} \\
&\rightarrow \text{ No discrepancy, message on previously marked path} \\
&||UpdatePath.UnMarkPath(u, z, C_iTREE) \supset C_iTREE'||_{Observer_j} \\
&\{w = TRUE \wedge Marked(z.path)\} \\
&\text{or } \rightarrow \text{ Discrepancy found} \\
&||UpdatePath.MarkPath(u, z, C_iTREE) \supset C_iTREE'||_{Observer_j} \quad (w \neq TRUE)
\end{aligned}
\tag{8}
$$

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 travelling by the agent node will not be trusted for state determination.

$$
\begin{aligned}
C_1TREE = Op + (&X_1 + (X_4.X_9.X_{12}.C_1) \\
&+ (X_5 + (\bar{X}_8 + X_{10}).X_{13}.C_1)) \\
&+ (X_2 + (X_5 + (\bar{X}_8 + X_{10}).X_{13}.C_1) \\
&+ (X_6.X_9.X_{12}.C_1)) \\
&+ (X_3 + (X_6.X_9.X_{12}.C_1) \\
&+ (X_7.X_{10}.X_{13}.C_1)))
\end{aligned}
\tag{9}
$$

The second case is a 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 path 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 signalled by the $Alert$ goal.

$$
\begin{aligned}
&|X_i.Send|| \quad (i = 1, 2, 3) \\
&||Observe.x_{ob_j}(z)||_{Observer_j} \quad (z \in M) \\
&\rightarrow \\
&||UpdateState.Store(z, STORE) \supset STORE'||_{Observer_j} \\
&\rightarrow \text{ Only if message is trusted} \\
&||Store(z, STATE)||_{Observer_j} \\
&\rightarrow \quad (p > rand()) \\
&||Observe||_{Observer_j} \\
&\rightarrow \ (p \leqslant rand()) \\
&||EvaluateState.Evaluate(STATE, \tilde{c}) \supset CRITICAL'||_{Observer_j} \\
&\rightarrow \text{ Not in a critical state} \\
&||Observe||_{Observer_j} \\
&\rightarrow \text{State is critical} \\
&||Alert||_{Observer_j}
\end{aligned} \tag{10}
$$

$$
\begin{aligned}
&||X_{10}.Mark.\bar{x}\langle y\rangle_{Op}|X_{10}.x_{10}(z)|C_1.\bar{x}_{12}\langle z\rangle_{Op}|X_{12}.x_{12}(z)||_{System} \\
&\rightarrow \\
&||X_{10}.Mark.\bar{x}_8\langle y\rangle_{Op}|X_{12}.Mark.\bar{x}_9\langle y\rangle_{Op}| \\
&X_9.Mark.x_9(z)|C_1.\bar{x}_{13}\langle y\rangle_{Op}|X_{13}.x_{13}(z)||_{System} \\
&||X_8{}'.x_8(z)||_{Agent2} \\
&\rightarrow \\
&||X_8{}'.Mark.\bar{x}_5\langle y\rangle_{Op}||_{Agent2} \\
&||X_{13}.Mark.\bar{x}_{10}\langle y\rangle_{Op}|X_9.Mark.Observe(y)| \\
&C_1.\bar{x}_{12}\langle z\rangle_{Op}|X_{12}.x_{12}(z)|X_{10}.x_{10}(z)|X_5.x_5(z)||_{System} \\
&\rightarrow \\
&||X_{10}.Mark.\bar{x}_7\langle y\rangle_{Op}|X_5.Mark.\bar{x}_2\langle y\rangle_{Op}| \\
&X_{12}.Mark.\bar{x}_9\langle y\rangle_{Op}|X_9.Mark.\bar{x}_6\langle y\rangle_{Op}| \\
&C_1.\bar{x}_{13}\langle y\rangle||_{System} \\
&\rightarrow \text{ Case (1) } X_8 \text{ and } X_6 \text{ are marked} \\
&\coprod_i ||UpdatePath.MarkPath(c, y, C_1TREE)||_{Ob_i} \\
&\rightarrow \text{ Case (2) Neither node is marked} \\
&\rightarrow \text{ Case (3) } X_6 \text{ is marked but not } X_8
\end{aligned} \tag{11}
$$

Finally, we should demonstrate that dynamic behavior such as agents migrating can be accommodated, say if X_8 and X_6 change status - see equation

11. This ability to track and demonstrate the possible range of dynamic system behavior is a key aspect of this proof technique. For example, considering the probable outcomes of any state determination during a changeover in node state.

7 Conclusion and Future Work

In this paper we have discussed an extension to the adversary capability model – based on the π-calculus – proposed in [2] using a goal-based syntax and semantics to capture the operation of software agents explicitly in SCADA and critical infrastructure environments. We argue that both attackers and operators have well-founded motivations for employing software agents in such environments. This extension enables us to model a greater range of attack and defence capabilities compared with our previous approach [ibid.], in particular, coordinated attacks and defences, and to reason about complex attacker/defender interactions based on the use of software agents at a granular level. We illustrated our approach by providing a formal proof of a novel algorithm for state determination using trusted paths in a SCADA system under co-ordinated attack. Future work will concentrate on more complex analysis and research of operator/adversary interactions in critical infrastructure environments by competing coalitions of agents. For example, considering how observers may interact with each other. We will also seek to extend the approach to incorporate learning behaviors and timing considerations.

References

1. Chen, T.: Stuxnet, the Real Start of Cyber Warfare? Network, IEEE **24**(6) (november-december 2010) 2 –3
2. McEvoy, T.R., Wolthusen, S.: A Formal Adversary Capability Model for SCADA Environments. In Xenakis, C., Wolthusen, S., eds.: Proceedings of the Fifth International Workshop on Critical Information Infrastructures Security (CRITIS 2010). Volume 6712 of Lecture Notes in Computer Science., Athens, Greece, Springer-Verlag (September 2010) (in press).
3. Genge, B., Siaterlis, C.: Investigating the Effect of Network parameters on Coordinated Cyber Attacks Against A Simulated Power Plant. In Bologna, S., Wolthusen, S., eds.: Proceedings of the Sixth International Workshop on Critical Information Infrastructures Security (CRITIS 2011). Lecture Notes in Computer Science, Springer-Verlag (September 2011) in press.
4. Sangiorgi, D., Walker, D.: π-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York, NY, USA (2001)
5. Wooldridge, M.: An Introduction to MultiAgent Systems. 1st edn. John Wiley & Sons (June 2002)
6. McLaughlin, S.: On Dynamic Malware Payloads Aimed at Programmable Logic Controllers. In: Proceedings of the 6th USENIX conference on Hot topics in security. HotSec'11, Berkeley, CA, USA, USENIX Association (2011) 10–10

7. Patsakis, C., Alexandris, N.: New Malicious Agents and SK Virii. In: International Multi-Conference on Computing in the Global Information Technology, 2007. (march 2007) 29
8. Braynov, S., Jadliwala, M.: Detecting Malicious Groups of Agents. In: IEEE First Symposium on Multi-Agent Security and Survivability, 2004. (aug. 2004) 90 – 99
9. Braynov, S., Jadliwala, M.: Representation and Analysis of Coordinated Attacks. In: Proceedings of the 2003 ACM Workshop on Formal methods in Security Engineering. FMSE '03, New York, NY, USA, ACM (2003) 43–51
10. Boyer, S.A.: Supervisory Control and Data Acquisition. 4th edn. ISA (2010)
11. Krutz, R.L.: "Securing SCADA Systems". Hungry Minds Inc (2005)
12. Tesauro, G., Chess, D.M., Walsh, W.E., Das, R., Segal, A., Whalley, I., Kephart, J.O., 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. AAMAS '04, Washington, DC, USA, IEEE Computer Society (2004) 464–471
13. McEvoy, T., Wolthusen, S.: A Plant-wide Industrial Process Control Security Problem. In Butts, J., Shenoi, S., eds.: Critical Infrastructure Protection V: Proceedings of the Fourth Annual IFIP Working Group 11.10 International Conference on Critical Infrastructure Protection. Volume 367 of International Federation for Information Processing Advances in Information and Communication Technology., Hanover, NH, USA, Springer-Verlag (March 2010) 47–56
14. McEvoy, T.R., Wolthusen, S.: Defeating Node-Based Attacks on SCADA Systems Using Probabilistic Packet Observation. In Bologna, S., Wolthusen, S., eds.: Proceedings of the Sixth International Workshop on Critical Information Infrastructures Security (CRITIS 2011). Lecture Notes in Computer Science, Springer-Verlag (September 2011) in press.
15. Song, D.X., 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. Volume 2. (2001) 878 –886