Chapter 8

AGENT INTERACTION AND STATE DETERMINATION IN SCADA SYSTEMS

Thomas McEvoy and Stephen Wolthusen

Abstract Defensive actions in critical infrastructure environments will increasingly require automated agents to manage the complex, dynamic interactions that occur between operators and malicious actors. Characterizing such agent behavior requires the ability to reason about distributed environments where the state of a channel or process depends on the actions of the opposing sides. This paper describes an extension to the Applied π-Calculus for modeling agent behavior in critical infrastructure environments. The utility of the extension is demonstrated via an agent-based attack and defense interaction scenario.

Keywords: Critical infrastructure, agents, state determination, π-Calculus

1. Introduction

Critical infrastructure systems have received significant attention as targets of cyber attacks [4]. Remote attackers, however, face problems in determining the system state due to limitations on communications [5, 7]. At the same time, operators must respond in real-time to sophisticated attacks and may have to make decisions based on partial knowledge of the system state. The outcomes of these interactions may depend on the state (or knowledge of the state) of a single channel or process. We argue that such situations require the deployment of software agents by both sides to automate, in whole or in part, attack and defense strategies.

This paper introduces an extension to the Applied π-Calculus [12] that provides the ability to define and classify agent-based attack and defense strategies. The extension augments a previously-proposed formal adversary capability model [7]. A model of a coordinated attack and defense scenario is presented to illustrate the utility of the extension.

J. Butts and S. Shenoi (Eds.): Critical Infrastructure Protection VI, IFIP AICT 390, pp. 99–109, 2012.

2. Background

Software agents often use learning behavior techniques and rely on their perception of the environment to make autonomous decisions [15]. Indeed, in recent years malicious code has exhibited increasingly agent-like behavior, a trend that is likely to continue [10, 11].

Software agents permit an attacker to launch sophisticated attacks, including coordinated attacks on targets. From the offensive perspective, there are several advantages to launching agent-based attacks [2, 3]. Responding to such attacks requires the operator to make dynamic interventions in the face of changing adversary behavior. The difficulty for the operator is exacerbated by the intrinsic nature of critical infrastructure systems, which may require continued operations even in a compromised state [1, 6]. Moreover, the scale and complexity of critical infrastructure systems render it unlikely for a human operator to make the appropriate responses to deal with a coordinated attack.

The ability of software agents to autonomously perform a range of security tasks provides a distinct advantage in countering agent-based attacks [14]. Understanding agent-based attack and defense strategies is key to protecting critical infrastructure systems. However, reasoning about agent-based systems presents a complex set of problems, especially with regard to multiple cooperating agents that "recruit" normally trusted processes to work on their behalf [7]. An extension to the well-known π-Calculus makes it possible to reason about such complex scenarios. The extension also provides a model to examine a distinct class of attacks that are dependent on malicious software agents and not on direct adversary intervention.

3. $G\{\pi\}$-Calculus

The π-Calculus provides a formal mechanism for modeling process actions [7–9, 12]. The associated algebraic theory facilitates formal reasoning, including automating proofs [12]. This section describes the extended goal transform π-Calculus ($G\{\pi\}$-Calculus). Interested readers are referred to [12] for fundamentals of the basic π-Calculus and to [7] for details about the Applied π-Calculus, which is used as the basis of the extension described in this paper.

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

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

where the possible actions α of G are defined in Table 1. Note that $\mathcal{L}$ is a first-order logic with equivalence and ordered relations, and π is a capability of the π-Calculus.

Goal actions are defined by the capabilities of the Applied π-Calculus:

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

with the semantics defined in Table 2. Goals execute until they invoke another goal, at which point they terminate. G_0 is a reserved label that represents the null goal.

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

Term	Semantics
$\mathbf{0}$	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 2. π-Calculus terms.

Term	Semantics
$\bar{x}\langle z\rangle_{\tilde{c}}$	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

The set of system names N comprises channels, constants, message characteristics, variables and function labels. Note that $\tilde{z}$ denotes a vector of names, $\coprod$ denotes a set of concurrent goals or processes, and $\sum$ denotes a sum M over capabilities. A label for an inaction represents a process action that may not be directly observable. For example, if $P := M + \lambda$ is a process, then $\lambda, P \supset 0$ is an inaction or "silent" function of P.

A key characteristic of the model is that processes may be overwritten by messages from another agent. Hence, the outcomes of messages need to be precisely defined. For example, if m is a message and $P := M + \Omega|Q$, then $\Omega, m, P \supset P'$ where P' may be defined arbitrarily. In general, P' behaves like P, except under certain conditions where it executes a different behavior useful to the adversary (i.e., Byzantine behavior).

Destination addresses are characteristics in the example scenario presented in this paper. Assume that $\langle z\rangle_{X_j}$ is used to route the name z to the process X_j. Routing is conditional on the characteristic and, for brevity, is denoted as $\bar{x}_i\langle z\rangle_{[X_j]}$ rather than the more conventional $[z.r = X_j]\bar{x}_{X_j}\langle z\rangle_{X_j}$ where $z.r$ indicates the characteristic routing address of the name z. Hence, any name with the destination address $z.r = X_j$ as a characteristic is routed by $\bar{x}_i$, even

Two agents send malware to selected system nodes.
$||Stage1.SendMalwareToNode_4||_{Launch1} \quad (i = 4)$
$||Stage2.SendMalwareToNode_{12}||_{Launch2} \quad (j = 12)$

The selected nodes are ready to receive the messages.
$||X_4.ReceiveMessage|X_{12}.ReceiveMessage|\coprod_{k=1...15,k\neq i,k\neq j} X_k||_{System}$

After the messages are sent, the launch agents poll for success.
$||Stage1.PollSuccess||_{Launch1}$
$||Stage2.PollSuccess||_{Launch2}$
$||X_4.BecomeMaliciousAgent|X_{12}|\ldots||_{System}$

One of the attacks succeeds and the success is reported to Agent 3.
$||Stage1.ReportSuccessToLaunch3||_{Launch1}]$

The second launch agent continues its initial subversion attempts.
$||Stage2.SendMalware||_{Launch2} \quad (j = 11)$
$||X_4^{'}||_{Agent1}$
$||X_4|X_{11}.ReceiveMessage|\coprod_{k=1...15,k\neq j} X_k||_{System}$

The third agent waits for complete success before it attacks.
$||Stage3.WaitForLaunch1and2 + [Success]LaunchFinalAttack||_{Launch3}$

Figure 1. Initial coordinated attack.

when $\bar{x_i}$ is not the final destination. A proof reduction is indicated using the notation:

$$||Goal.Subgoal.Action||_{Agent} \rightarrow ||Goal.Subgoal.NextAction||_{Agent}$$

where $NextAction$ is any capability or goal invocation. The proof reduction is identical to that of the π-Calculus [12], with the exception that goal labels are used to limit the consideration to the active (i.e., dotted) goals $|| \bullet Goal||$ of each agent.

4. Coordinated Attack

This section models a coordinated attack. Using automated agents, the adversary seeks to manipulate three valves in order to cause a critical failure, while concealing its actions. The first step is to define the goal labels for scenario planning. In fact, without interference in channels and processes, the $G\{\pi\}$-Calculus would be sufficient to prove the outcome of any interactions, provided that the goals are defined precisely.

In the coordinated attack shown in Figure 1, two defined "launch" agents send malware as a name to a system to overwrite various network nodes and transform them into additional malicious agents that work on behalf of the adversary. The scenario has four possible outcomes: (i) success for both agents;

The attack uses two routing processes to forward malware.
$||Stage1.\bar{x}_9\langle g_2\rangle_{X_i}||_{L1} \quad (i = 4)$
$||Stage2.\bar{x}_{10}\langle g_3\rangle_{X_j}||_{L2} \quad (j = 12)$
$||X_9.x_9(z)||_{System}$
$||X_{10}.x_{10}(z)||_{System}$

After the messages are sent, the launch agents wait for success.
$||Stage1.x_{S1}(s)||_{L1}$
$||Stage2.x_{S2}(s)||_{L2}$

The routing agents report either success or failure.
(ϕ is a silent function that indicates failure)
$||X_9.(\bar{x}_i\langle m_2\rangle_{X_i}.\bar{x}_{S1}\langle\top\rangle \oplus \phi.\bar{x}_{S1}\langle\bot\rangle_{S1})||_{Agent0}$
$||X_{10}.((\bar{x}_j\langle m_2\rangle_{X_j}.\bar{x}_{S2}\langle\top\rangle \oplus \phi.\bar{x}_{S2}\langle\bot\rangle_{S2})||_{Agent00}$
$||X_i.x_i(z)||_{System}$
$|X_j.x_j(z)||_{System}$

Case 1: Both attempts succeed.
$||Stage1.x_{S1}(s)||_{L1}$
$||Stage2.x_{S2}(s)||_{L2}$
$||X_9.\bar{x}_{S1}\langle\top\rangle_{S1})||_{Agent0}$
$|X_{10}.\bar{x}_{S2}\langle\top\rangle_{S2}||_{Agent00}$
$||X_i{}'|G||_{Agent2}$
$||X_j{}'|G||_{Agent3}$

Case 2: Either attempt fails and the initial attack continues.

Case 3: Neither attempt succeeds and the attack is aborted.

Figure 2. Reduction of the coordinated attack.

(ii) failure for Agent 1 and success for Agent 2; (iii) success for Agent 1 and failure for Agent 2; and (iv) failure for both agents. Based on the initial outcome, a third launch agent initiates the final part of the attack.

Figure 2 shows the reduction of the coordinated attack. Note that X_i is a system node, g_2 is a message used to infect the system and s is a Boolean variable.

In the example, the launch agents $L1$ and $L2$ send malicious names into the system using channels X_9 and X_{10}, respectively. In turn, these are routed to the target nodes i and j. If the initial subversion succeeds, the newly formed agents flag their success to agent $L3$, which launches the final part of the attack. Note that the messages indicating success or failure may arrive in any order, which may affect the planned outcome.

As demonstrated in Figure 3, the messages update a Boolean predicate a and the final attack launches if the predicate evaluates to $TRUE$. In this instance,

The third launch agent expects a success or failure indicator.
(Counter k keeps track of the number of messages)
$||InitialSuccess.x_{Ad}(u)||_{L3} \quad (k = 0)$

Case 1: Both attacks succeed and counter $k = 2$.
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k++)\ ||_{L3}$
$\ldots \rightarrow \quad (u = m_2, a = m_2 \wedge \neg m_3, k = 1)$
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a''.(k++)\ ||_{L3}$
$\ldots \rightarrow \quad (u = m_3, a = m_2 \wedge m_3, k = 2)$
$||Stage3||_{L3}$

Case 2: L3 sends a negative flag, but L2 succeeds.
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k++)\ ||_{L3}$
$\ldots \rightarrow \quad (u = m_2, a = m_2 \wedge \neg m_3, k = 1)$
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a''.(k++)\ ||_{L3}$
$\ldots \rightarrow \quad (u = \neg g_3, a = m_2 \wedge \neg m_3, k = 2)$
$||G_0||_{L3}$

Case 3: L2 and L3 send negative flags.
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k++)\ ||_{L3}$
$\ldots \rightarrow \quad (u = \neg g_2, a = \neg m_2 \wedge \neg m_3, k = 1)$
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a''.(k++)\ ||_{L3}$
$\ldots \rightarrow \quad (u = \neg g_3, a = \neg m_2 \wedge \neg m_3, k = 2)$
$||G_0||_{L3}$

Case 4: L3 succeeds, but L2 fails.
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a'.(k++)\ ||_{L3}$
$\ldots \rightarrow \quad (u = \neg g_2, a = \neg m_2 \wedge \neg m_3, k = 1)$
$||InitialSuccess.UpdateAttack.Update(u, a) \supset a''.(k++)\ ||_{L3}$
$\ldots \rightarrow \quad (u = m_3, a = \neg m_2 \wedge m_3, k = 2)$
$||G_0||_{L3}$

Figure 3. Determining if the final attack should be executed.

the update order is not relevant to the outcome. The result of Case 1 is that the final part of the attack is launched by L3. At this point, L3 sets the target valve to *Steady* and signals the other two agents to set their target valves to *Open* and *Closed*. The attack is concluded by masking the signal from the operators as demonstrated in Figure 4; this serves to conceal the attack.

5. Distributed Detection

For the defense strategy, an operator employs observer agents for state determination and trusted routes for alerts regarding critical conditions. As described in [8, 9, 13], each network node that receives a message adds its address

The first agent sets a valve to a steady value.
$||X_k{}'|SetSteady.set(u, Steady) \supset u'||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$

$||X_k{}'|SetSteady.Send.\bar{x}_s\langle u'\rangle_{C2}||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$

The instruction is sent to the controller.
$||X_k{}'||FlagSteady||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$

Messages are received.
$||X_k{}'|FlagSteady.x_s(u)||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$

After the flag is steady, the next agent is signaled.
$||X_k{}'|FlagSteady.[u = Steady]\bar{x}_s\langle s\rangle_{X_5}.G_0||_{Agent1}$
$||X_i|Waitfor1||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$

Fake signals are used to conceal the true plant state.
$||X_k{}'.Send(u)|Conceal||_{Agent1}$
$||X_i|Waitfor1.x_s(s)||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$

The next agent opens the valve and signals the third agent.
$||X_i|OpenValve||_{Agent2}$
$||X_j|Waitfor2||_{Agent3}$
$||X_j|CloseValve||_{Agent3}$

The attack completes.

Figure 4. Concealing the coordinated attack.

to mark the route. Each node may also probabilistically forward a message copy to "observer" agents for comparison.

The attributes provide a formal definition for observer agents, which use the information to make state determinations. Note that in [9], IP traceback algorithms were used to detecting the locations of malicious agents – a different goal from the one addressed in this work. Here, the observer algorithm uses

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

Figure 5. Formal specification of an observer agent.

messages and copies of messages to determine a trusted set of paths. This is demonstrated by the observer definition in Figure 5.

State determination is restricted to considering messages received on trusted paths. Figure 6 provides the initial reduction of the observer using copied messages to determine route trustworthiness. The observer receives a message and invokes the goal $UpdatePath$ to compare the message with the original. If no discrepancy is found, the observer moves to the next message. If a discrepancy is found, then it notes the route and marks the forward neighboring node as untrusted. The marked messages can be represented using a graph and defined algebraically.

The indication that the path marking algorithm responds correctly depends on whether a message is marked before or after manipulation. If the message is copied before it is manipulated, then the malicious agent node appears to deliver trustworthy messages, but any subsequent node appears untrustworthy. Hence, the next node in the communication chain is marked. Alternatively, when any previous node appears to deliver a trustworthy copy, the agent node is indicated using the bar notation. In both instances, a message traversing the agent node is not trusted for state determination.

Figure 7 demonstrates another case for examining a trustworthy message and, depending on the probability, a snapshot of state. The observer receives the original message and retains the message in $STORE$. If the message arrives on a trusted path, the message is included in $STATE$ to make a determination of the system state. If the state is detected as critical, the operator is signaled by the *Alert* goal.

Dynamic behavior such as agent migration can be accommodated. For example, a change of status for X_8 and X_6 can be represented as shown in Figure 8.

$||X_i.Send||\quad (i = 1, 2, 3)$
$||Observe.x_{ob_j}(z)||_{Observer_j}\quad (z \in C)$
$||UpdatePath.Compare(u, z, \tilde{k}, STORE) \supset w||_{Observer_j}$

Case 1 - No discrepancy, message on unmarked path
$||Observe||_{Observer_j}$

Case 2 - No discrepancy, message on previously marked path
Update paths
$||UpdatePath.UnMarkPath(u, z, C_iTREE) \supset C_iTREE'||_{Observer_j}$
$\{w = TRUE \wedge Marked(z.path)\}$

Case 3 - Discrepancy found
Update paths
$||UpdatePath.MarkPath(u, z, C_iTREE) \supset C_iTREE'||_{Observer_j}$
$(w \neq TRUE)$

Figure 6. Using copied messages to determine route trustworthiness.

$||X_i.Send||\quad (i = 1, 2, 3)$
$||Observe.x_{ob_j}(z)||_{Observer_j}\quad (z \in M)$
$||UpdateState.Store(z, STORE) \supset STORE'||_{Observer_j}$

Trusted messages are stored in STATE variable
$||Store(z, STATE)||_{Observer_j}$
$\rightarrow\quad (p > rand())$
$||Observe||_{Observer_j}$
$\rightarrow\ (p \leqslant rand())$

Evaluate plant state using trusted messages
$||EvaluateState.Evaluate(STATE, \tilde{c}) \supset CRITICAL'||_{Observer_j}$

No critical state found, continue
$||Observe||_{Observer_j}$

Alert on finding a critical state
$||Alert||_{Observer_j}$

Figure 7. Using store and alert messages to determine trustworthiness.

The ability to track the possible range of dynamic system behavior is a key aspect of the modeling technique. This provides the ability to consider probable outcomes of any state determination during a changeover in node state.

The ability to represent dynamic defense strategies facilitates reasoning about agent interaction schemes. Indeed, the modeling approach facilitates

$||X_{10}.Mark.\bar{x}\langle y\rangle_{Op}|X_{10}.x_{10}(z)|C_1.\bar{x}_{12}\langle z\rangle_{Op}|X_{12}.x_{12}(z)||_{System}$

$||X_{10}.Mark.\bar{x}_8\langle y\rangle_{Op}|X_{12}.Mark.\bar{x}_9\langle y\rangle_{Op}|$
$X_9.Mark.x_9(z)|C_1.\bar{x}_{13}\langle y\rangle_{Op}|X_{13}.x_{13}(z)||_{System}$
$||X_8'.x_8(z)||_{Agent2}$

$||X_8'.Mark.\bar{x}_5\langle y\rangle_{Op}||_{Agent2}$
$||X_{13}.Mark.\bar{x}_{10}\langle y\rangle_{Op}|X_9.Mark.Observe(y)|$
$C_1.\bar{x}_{12}\langle z\rangle_{Op}|X_{12}.x_{12}(z)|X_{10}.x_{10}(z)|X_5.x_5(z)||_{System}$

$||X_{10}.Mark.\bar{x}_7\langle y\rangle_{Op}|X_5.Mark.\bar{x}_2\langle y\rangle_{Op}|$
$X_{12}.Mark.\bar{x}_9\langle y\rangle_{Op}|X_9.Mark.\bar{x}_6\langle y\rangle_{Op}|$
$C_1.\bar{x}_{13}\langle y\rangle||_{System}$

Case 1 -X_8 and X_6 are marked
$\coprod_i ||UpdatePath.MarkPath(c, y, C_1TREE)||_{Ob_i}$

Case 2 - Neither node is marked

Case 3 - X_6 is marked but not X_8

Figure 8. Dynamic behavior of agent migration.

the analysis of agent behavior in order to determine the appropriate responses during a coordinated attack.

6. Conclusions

The π-Calculus extension described in this paper uses goal-based syntax and semantics to explicitly capture the operation of agents in critical infrastructure environments. It also provides the ability to model an increased range of attack and defense capabilities compared with previous approaches. Specifically, it facilitates the modeling of coordinated attacks and defenses, and the ability to reason about complex interactions at a granular level. The example scenario demonstrates state determination in the face of a coordinated attack by leveraging trusted paths.

Our future work will concentrate on modeling and analyzing complex operator and adversary interactions in critical infrastructure environments. We will also seek to extend the approach to incorporate learning behavior and timing considerations.

References

[1] S. Boyer, *Supervisory Control and Data Acquisition*, ISA, Research Triangle Park, North Carolina, 2010.

[2] S. Braynov and M. Jadliwala, Representation and analysis of coordinated attacks, *Proceedings of the ACM Workshop on Formal Methods in Security Engineering*, pp. 43–51, 2003.

[3] S. Braynov and M. Jadliwala, Detecting malicious groups of agents, *Proceedings of the First IEEE Symposium on Multi-Agent Security and Survivability*, pp. 90–99, 2004.

[4] T. Chen, Stuxnet, the real start of cyber warfare? *IEEE Network*, vol. 24(6), pp. 2–3, 2010.

[5] B. Genge and C. Siaterlis, Investigating the effect of network parameters on coordinated cyber attacks against a simulated power plant, *Proceedings of the Sixth International Workshop on Critical Information Infrastructure Security*, 2011.

[6] R. Krutz, *Securing SCADA Systems*, Wiley, Indianapolis, Indiana, 2006.

[7] T. McEvoy and S. Wolthusen, A formal adversary capability model for SCADA environments, *Proceedings of the Fifth International Workshop on Critical Information Infrastructure Security*, pp. 93–103, 2010.

[8] T. McEvoy and S. Wolthusen, A plant-wide industrial process control security problem, in *Critical Infrastructure Protection V*, J. Butts and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 47–56, 2011.

[9] T. McEvoy and S. Wolthusen, Defeating node-based attacks on SCADA systems using probabilistic packet observation, *Proceedings of the Sixth International Workshop on Critical Information Infrastructure Security*, 2011.

[10] S. McLaughlin, On dynamic malware payloads aimed at programmable logic controllers, *Proceedings of the Sixth USENIX Conference on Hot Topics in Security*, p. 10, 2011.

[11] C. Patsakis and N. Alexandris, New malicious agents and SK virii, *Proceedings of the International Multi-Conference on Computing in the Global Information Technology*, p. 29, 2007.

[12] D. Sangiorgi and D. Walker, *π-Calculus: A theory of mobile processes*, Cambridge University Press, Cambridge, United Kingdom, 2001.

[13] D. Song and A. Perrig, Advanced and authenticated marking schemes for IP traceback, *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, pp. 878–886, 2001.

[14] G. Tesauro, D. Chess, W. Walsh, R. Das, A. Segal, I. Whalley, J. Kephart and S. White, A multi-agent systems approach to autonomic computing, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 464–471, 2004.

[15] M. Wooldridge, *An Introduction to MultiAgent Systems*, Wiley, Chichester, United Kingdom, 2002.