Closed Reflective Networks: a Conceptual Framework for Intrusion-Resistant Autonomous Systems

Catriona M. Kennedy and Aaron Sloman
School of Computer Science
University of Birmingham
Edgbaston, Birmingham B15 2TT
Great Britain

Abstract

Intrusions may sometimes involve the insertion of hostile code in an intrusion-detection system, causing it to "lie", for example by giving a flood of false-positives. To address this problem we consider an intrusion detection system as a reflective layer in an autonomous system which is able to observe the whole system's internal behaviour and take corrective action as necessary. To protect the reflective layer itself, several mutually reflective components (agents) are used within the layer. Each agent acquires a model of the normal behaviour of a group of other agents under its protection and uses this model to detect anomalies. The ideal situation is a "closed reflective network" where all components are monitored and protected by other components within the same autonomous system, so that no component is left unprotected.

Using informal rapid-prototyping we implemented a closed reflective network based on three agents, where the agents use majority voting to determine if an intrusion has occurred and whether a response is required. The main conclusion is that such a network may be better implemented on multiple hardware processors connected together as a simple neural network.

1 Introduction and Background

If an autonomous system is to continue operating in the presence of faults and intrusions without external intervention, it requires a self-monitoring capability to recognise problems in its own operation and restore itself to normal functioning. This is already an established area in remote autonomous vehicles such as in [27]. We consider self-monitoring as a kind of *reflection* in the cognitive sense. That is to say, the system to be protected is regarded as a "whole" cognitive system which must survive in a hostile environment, in much the same way as a living system does. Related ideas are "meta-cognition" [26] and "meta-management" [4] which involve (among other things) the ability of a cognitive system to detect problems in its internal processing and modify them.

1.1 Self/nonself distinction

In order to survive, an autonomous system should make a distinction between its normal (expected) state and actual state. An abnormal state is called an *anomaly* and may be a recognised failure state or it may be unknown. Unknown states may be merely unusual (in that measurements are wildly different from expected) or "impossible" (in that measurements seem to contradict the theory on which expectancy is based). In this paper we are mostly concerned with the "unexpected" kind of anomaly. An example of "impossibility" is given in [16].

In nature, immune systems detect foreign invaders by distinguishing between "self" and "nonself". (A general overview can be found in [14]). The recognition of self happens when actual patterns (sometimes called "pattern instances" in AI terminology) match "expected" patterns already known by the immune system (the immune system's "knowledge" being equivalent to a model). Unknown micro-organisms are regarded as "nonself" and may be attacked.

The new field of *artificial* immune systems (surveyed in [8]) uses this concept of self/nonself distinction, but they have important limitations which will be discussed later. In our investigation of autonomous reflective systems, we are aiming to approximate the immune system capability of self/nonself distinction, although not necessarily with the same architecture.

A self/nonself distinction requires a model of "self". The model may be a representation of the system's own components and their normal patterns of behaviour. Any anomalous pattern may then be categorised as "nonself". In software this may be associated with a new component (such as a new version of a software package) or a different kind of user. It is therefore advantageous for the system to evaluate whether an unknown pattern is "good" or not. For example, a foreign but harmless pattern (along with a description of the object causing it) may be "accepted" into the model of "self" after a period of observation. Thus, we can identify two kinds of capability:

- *Anomaly-detection*: the system should detect unusual features when observing its own behaviour, or the absence of expected features. In immune systems terminology, it detects "nonself" (patterns that it classes as "foreign"). In the case of absence, a "nonself" is not directly observed but may be inferred as a cause of the absence.

- *Quality evaluation*: the system should evaluate the current state of the world as good, bad, satisfactory, etc. This evaluation is only useful if a foreign pattern has also been detected. If foreign patterns have not been associated with a quality deterioration and the system is "trusting", it can accept the patterns as "self" (along with the object producing the pattern if applicable). If there is a quality deterioration, the system might try to suppress execution of any unfamiliar object that can be identified from the anomalous patterns.

The simplest (minimal) situation is a "pessimistic" policy where anything unknown is assumed to be "bad" and everything familiar is "good". A more detailed discussion of the distinction between the two is given in [17].

Anomaly-detection and quality evaluation can also be used non-reflectively, for example when encountering new features in the environment. One of the most difficult problems in any form of quality-based diagnosis and response is *credit assignment* - how to correlate anomalous events with an increase or decrease in "quality". The problem in its more general form of assigning credit or blame to actions was first recognised by [25]. It is now an established subfield of research, in particular in relation to reinforcement learning. See for example, [39]. We assume that these established techniques for assigning credit can also be incorporated into our architecture.

1.2 Mutual Reflection

The simplest design for an autonomous reflective system is hierarchical with a top-level self-monitoring component. This kind of architecture tends to be assumed when designing reflective agents. In a hostile environment, however, failures or deliberate attacks can be made against *any* component of the system, including its self-monitoring and repair components. If the self-monitoring layer of a hierarchical architecture is disabled or modified, there is no component in the system that can reliably detect this problem.

In previous papers we introduced a way of "working around" this "reflective blindness" problem by distributing the reflection over multiple agents. [19] presents a minimal prototype based on two mutually reflective agents which enable an autonomous system to survive indefinitely in the presence of random damage to its anomaly-detection and repair components. This prototype was maximally "pessimistic" (with no evaluation of quality) and could recover from *omission* failures. That is, failures in which a component merely stops functioning (for example it does not lie). It may cause intermittent "lying" in some other components but it cannot cause *consistent* lying that follows the pattern of correct functioning. (Details of this assumption are discussed later).

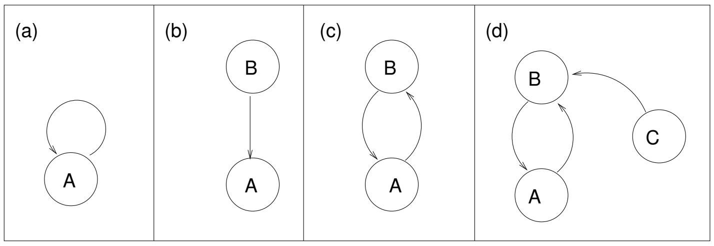
An extended prototype in [18] enabled the system to recover from certain kinds of hostile code attacks. An optimistic policy with quality evaluation was used to determine the "hostility" of the code. Any number of unfamiliar components were tolerated if they did not degrade quality.

The first part of this paper presents a conceptual framework for applying distributed reflection to any number of agents. The second part presents a 3-agent prototype implementation which can withstand certain kinds of hostile code scenarios that the two-agent architecture could not cope with.

2 Closed Reflective Networks: Conceptual Framework

It should be mentioned that this is an informal conceptual framework which is continually evolving as a result of experience gained in rapid prototyping and design of intrusion scenarios. We will explain more about the methodology later.

Possible configurations of agents are shown in figure 1. Each circle in figure 1 is an agent. An arrow



(a): A's meta-level monitors itself

(b): "Open" hierarchical reflection: a second agent B monitors A's meta-level.

(c): "Closed" distributed reflection: A and B monitor each other's meta-levels

(d): "Open" distributed reflection with an additional agent C

Figure 1: Configurations for reflective networks

indicates the monitoring relationship (which can also include capability to modify). Implicit in each agent is a meta-level which actively does the monitoring. The arrow also means "equally distributed" monitoring; that is, for any agent that is monitored, *all* of its components are monitored to the same extent using the same methods (including its meta-level).

Figure 1(a) shows the "centralised" configuration where a single agent detects anomalies in its own execution (which we have argued is not reliable). (b) and (d) are examples of "open" reflection. A configuration is "open" if there is at least one agent which is a source of an arrow but not a destination. Configuration (b) is *hierarchical* because A is unambiguously *subordinate* to B. (c) is "closed" meaning that all meta-levels are also object-levels (they are observable and modifiable from within the system itself).

Configuration properties that exclude each other are summarised in table 1. For example, according to our definition, an architecture is either "hierarchical" or "distributed" but not both. "Hierarchical" means that there is always one agent that has final "authority" in determining the system's actions, because its model of the whole system is not questioned. "Distributed" means that there are at least two agents whose models of the system have equal "value", thus giving both agents the same "authority" (We will consider later what "authority" and "equality" mean in practice).

The following combinations are unreliable:

Table 1: Architecture parameters

centralised	distributed
hierarchical	distributed
open	closed

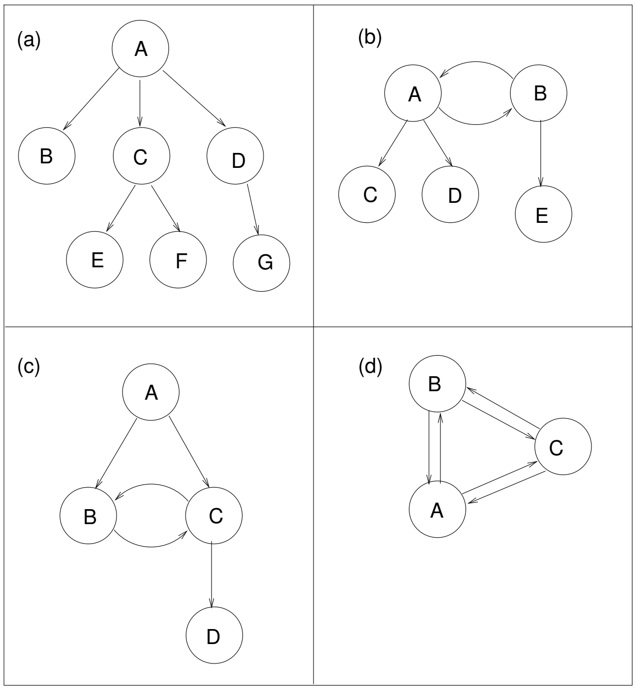
1. Centralised and closed (one agent monitors all of its components)

2. Centralised and open (one agent monitors some of its components)

3. Hierarchical and open (for two agents, one agent monitors another)

There is no fundamental difference between (2) and (3) above, since A and B in (3) can play the roles of meta- and object-levels in (2).

Note that in a two-agent system a distributed reflective system is always closed. For convenience, we call such a configuration a *closed reflective network* or CRN. Examples involving more agents are shown in figure 2. The importance of closed architectures in biological systems was first emphasised



(a): A simple hierarchy

(b): "Closed" distributed reflection involving two agents, each with subordinate agents

(c): "Open" distributed reflection involving two subordinate agents

(d): "Closed" distributed reflection involving three agents

Figure 2: More complex configurations

by theoretical biologists Maturana [24], Varela [41] and Rosen [30] but not formally defined in a way that could lead to computational realisation. Our concept of closed reflection is an adaptation of these biological concepts.

2.1 Why is this "reflective"?

One may argue that a system of mutually observing agents is not really a "reflective" system but is instead a society of agents. However, we call it reflective because the *effects* of mutually observing agents on the whole autonomous system are to enable it to distinguish between self and nonself to the degree of accuracy required to survive in its environment. (Biological immune systems only do this to a "sufficient" degree of accuracy to allow a species to survive). We can say that it satisfies the *requirement* of sufficiently accurate reflection imposed by the particular environment.

A CRN can also satisfy the requirement for "computational" reflection stated by [22] of having a causally connected self-representation. We will show later how the causality requirement is satisfied when we present an architecture.

3 Methodology

Our methodology is *design-based* [33] which aims to understand a phenomenon by attempting to build it. The "building" process involves the development of intuitions using informal rapid prototyping. The importance of such intuitive interaction with real systems as an aid to development of theory is emphasised by Chrisley [6] in the context of cognitive science. Applying this principle in software engineering means that the design process should not initially be constrained by a formal definition, although a formalisation may gradually emerge as a result of increased understanding acquired during iterated design and testing. Such an emergent formal definition may be very different from an initial one used to specify the design initially. For example we do not define precisely what a "software component" is, since the definition of a "component" evolves depending on the kind of software being designed.

Since the aim is the development of concepts, the result of the investigation is not quantitative, but is instead a statement of the environmental conditions in which an architecture is successful (in that it enables survival) and the conditions in which it fails (and why).

3.1 Architecture Schemas

Our methodology is also *architecture-based* which means that architectures, not algorithms, are the focus of the investigation. We use the term *architecture* for any software design that requires some form of concurrency; otherwise it is simply a program design (an algorithm).

An architecture may be defined recursively. That is, its components may themselves have concurrently interacting sub-components. In this paper, we assume that an architecture only contains "elementary" components in the form of sequential processes. The following notational convention applies:

- *Agent*: a single execution thread (instance) of a program which has some degree of intelligence and autonomy. For example, it may have some reasoning and adaptation capability.
- *Component*: any software module which does not contain concurrently executable subcomponents. For example, a component may be an agent, a subroutine or a collection of related rules in a rulebase.

The following taxonomy of terms were introduced in [34] and are explained in detail here.

- *Architecture Schema (AS)*: constraints on components and their interrelationships. In particular, what kind of algorithms and other components are required? What are the constraints on their interaction? Are there any constraints on how they should start themselves up? Is there any learning or adaptation involved? Architecture schemas can exist at different levels of refinement. An example of a schema is the Birmingham *CogAff* schema.
- *Architecture (A)*: a specific way of applying an architectural schema to a set of components. In particular, how should the components be mapped onto instances (executable entities)? How do the instances interact? An architecture can be translated directly into an implementation (but clearly there may be several implementations of the one architecture e.g. in different programming languages). An example of an architecture is the Birmingham *H-CogAff* architecture which is a special case of *CogAff*.
- *Architecture Implementation (AI)*: set of executable components written in one or more languages and allocated to one or more processors.
- *Instances of an architecture implementation (AII)*. Different instances of the same implementation may exist; for example, they may run on different networks.

Before it becomes an implementation instance, an architecture already contains component instances (processes) because the architecture must specify the relationships between processes during execution. However, for abbreviation we use the term "component" for a passive code component (for which the architecture might require several instances) and for an active architecture component (instance).

Any bootstrapping constraints specified in the architecture schema must be applied to the architecture. For example, the whole architecture may be bootstrapped by activation of random components at random times, or the components may activate each other or gradually increase each other's effectiveness. Some of these constraints may apply to a particular implementation only, but some may be specified on the architecture level because they may be essential for the required function of the system.

Constraints on relationships between components are informally specified. They may become formalised at a later stage, when there are many working instances of the architecture being applied to real-world problems.

3.2 Broad-and-shallow architectures

Our approach is also "broad and shallow" [3]. A *broad* architecture is one which provides *all* functions necessary for a complete autonomous system (including sensing, decision-making, action, self-diagnosis and repair.).

A *shallow* architecture is a specification of components and their interactions, where each component implements a scaled-down simplified version of an algorithm or mechanism which implements the component's function. Our final aim is not a shallow architecture, but shallowness is necessary initially in order to make the rapid-prototyping manageable. It is also necessary to make the following assumptions:

1. *Shallowness assumption*: a shallow architecture can be deepened without changing the fundamental properties of the architecture (i.e. the necessary properties are preserved). In other words, a deepening would not make the results of the shallow architecture completely meaningless.

2. *Recovery assumption*: If the system can *detect* a problem then it can diagnose and repair it. In other words, all problems are recoverable if they can be recognised. This is a consequence of the "broad" approach. Since the investigation is about reflection on the architecture level, and about compensating for reflective blindness, we are not concerned with details of diagnosis and recovery algorithms. Only minimal versions of these are implemented, in order to produce a "complete" autonomous system.

3.3 Closed Reflection as a Design Constraint

We can specify the constraints (architecture schema) informally for closed reflection as follows:

Constraint 1: There is no component in the system that is not subject to monitoring and modification by the system itself. In particular, the meta-level components are subject to *at least as much* monitoring as all other components.

Constraint 2: Monitoring and modification should include a minimal capability to continually observe a component's current state, to detect deviations from normal behaviour (anomalies) and to modify or influence the operation of the component in response to an anomaly.

Constraint 3: a model of normal behaviour should be acquired by mutual observation (bootstrapping requirement)

We emphasise that the constraints are informal and "soft" in that they are guidelines for the design process.

The highest level component is an agent (which is always a sequential process in our work). A component can be defined on arbitrarily low levels depending on the monitoring accuracy required by the particular environment.

The definition of "minimal" capability depends on the type of environment the system must survive in. For example, if the environment demands precise timing of actions and contains enemies that can modify the timing in subtle ways then a minimal capability would involve precise measurement of "normal" timing patterns.

An open reflective architecture is one for which constraint 1 does not apply; there is at least one executive or control component which is not subject to monitoring or control by the system itself or it is subject to a considerably weaker form of monitoring and control than is the rest of the system. As for the definition of "minimal" capability above, the precise meaning of "considerably weaker" depends on the design context.

4 Interacting with an Environment

The configurations in figures 1 and 2 do not show how the system interacts with an environment. We define three different types of "environment":

- The application domain: what is the environment from the point of view of the users and what are their requirements?
- The internal world: what kind of internal states are sensed by the autonomous system? How can it respond in this environment?
- Damage and intrusion: what can go wrong in the internal world and in the application domain? What kind of "enemies" exist?

We define the requirements in the application domain as the *primary task* of the autonomous system. The protection of the capability to satisfy the requirements involves *secondary tasks* such as intrusion-detection, which are typically invisible to the user. "Survival" means continuing to fulfil the primary task in the presence of faults and intrusions.

4.1 A simple virtual world

For rapid prototyping of architectures, we simulate an application domain using a virtual world called "Treasure" which is a modification of the "Minder" scenario [42]. The Treasure "environment" is made up of several treasure stores, one or more ditches and an energy source. The autonomous system is a control system for a vehicle. The vehicle's primary task is to collect treasure, while avoiding any ditches and ensuring that its energy level is kept above 0 by regular recharging. The "value" of the treasure collected should be maximised and must be above 0. Treasure already "owned" by the vehicle continually loses value as it gets less "interesting". Treasure stores that have not been visited recently are more interesting (and thus will add more value) than those just visited. The system "dies" if either collected treasure value or energy-level become 0 or the vehicle falls into a ditch. A configuration is shown in figure 3.

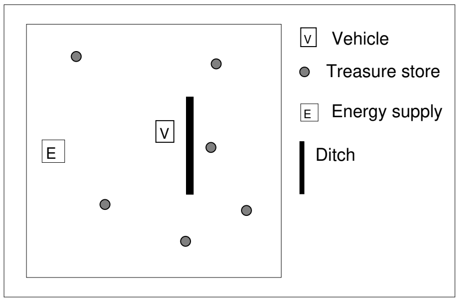
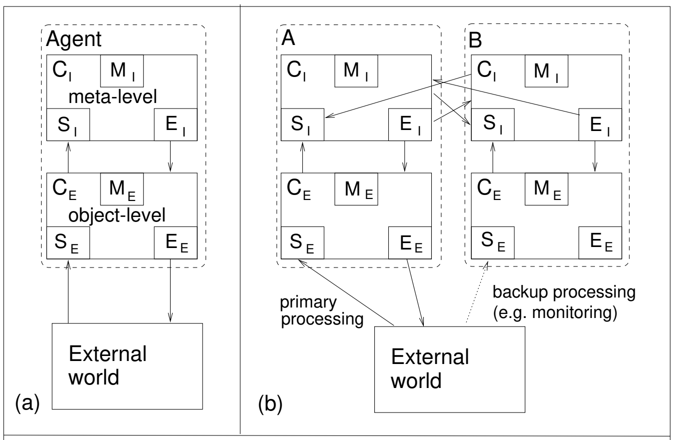


Figure 3: Treasure Scenario

4.2 Control systems

The primary task above can be achieved by a simple control system. Figure 4(a) shows a two-layer architecture schema containing an object-level and a meta-level respectively. The object-level is a control system C_E which maintains critical variables in an external world within acceptable values. For example, in the Treasure scenario, "acceptable values" mean that the vehicle's continually decaying treasure and energy levels are maintained above 0; otherwise it does not "survive". We assume that the agent has a model M_E (part of C_E) to predict the next state of the external world. The *meta-level* labelled C_I is a second control system which is applied to the agent's internal components. The model M_I (part of C_I) is used to predict the "normal" state of the internal world. Sensors and effectors (S_I and E_I) are also used on the meta-level. We call the two-layered structure a *reflective agent*.



C_I control of internal world	C_E control of external world
M_I model of internal world	M_E model of external world
S_I internal sensors	S_E external sensors
E_I internal effectors	E_E external effectors

(a) Single-agent reflection; (b) Distributed reflection (simplest case): agent is divided into primary and backup copies A and B. Only the primary changes the external world; both have read/write access to each other's meta-levels and their own object-levels

Figure 4: Reflective agent architectures

4.2.1 Causally connected self-representation

For reflective control systems of the above kind, the self-representation exists in two states: expected state and actual state of the system. Computationally reflective systems use operators to inspect and modify their own operation. They are confusingly called "reify" and "reflect" respectively, but we will call them `inspect` and `modify`.

The equivalent of `inspect` in the reflective control system architecture is provided by the internal sensors, which if accurate, should give a representation of the actual state of the system. (Monitoring of sensor-operation is also included in the monitoring of a meta-level).

With a control system, there is also causal connection between the desired state and the system's actual behaviour (in the above architecture the desired state is the "normal" expected state, although this may be different in some circumstances). In contrast to a typical reflective programming language, however, the causal connection is not guaranteed to work perfectly. The difference is in the existence of an environment. The causal connectivity may be characterised as a "pressure" or "motivation" of the system to change its own state into the ideal one.

4.3 Distributed reflective control

In figure 4(a), reflective blindness means that C_I cannot reliably detect anomalies in its own operation. For example, the anomaly-detection component of C_I may be modified so that it no longer does any anomaly-detection. Figure 4(b) shows one possible way in which the reflective control system can be distributed. Only one agent (the primary) changes the external environment (as can be seen from the dotted line from one agent only). The other is a passively monitoring backup. The primary/backup configuration is the simplest to design, since both agents are based on identical code. There are other possibilities: for example, one agent may be a specialist in treasure maximisation while the other is a specialist in safety and energy.

4.4 Instantiating the architecture-schema

Instantiation of the architecture-schema into an implementable architecture involves filling the components with content so that they can be directly programmed in a selected language.

4.4.1 External control system

Table 2 lists the main contents of the object-level C_E in figure 4(a) which enables survival in the Treasure scenario *without* hostile interference. The agent's representation of the world plays the role of M_E (for example, what does treasure normally look like?).

The architecture was implemented as a set of rules divided up into modules (rulesets). Thus the table shows a component hierarchy of individual rules, rulesets and ruleset groups according to function. The table is a schematic summary only. The practical details of implementation involves additional rules which do not relate to the architecture itself.

For example, "evaluate_state" specifies how the system's interest in treasure stores increases with

Table 2: Selected architecture components

Function	Ruleset	Sample rules
Sense	external_sensors	see_treasure
		see_ditch
		see_energy
Decide	evaluate_state	interest_growth
		interest_decay
	generate_motive	low_energy
		low_treasure
	select_target	new_target
		target_exists
Act	avoid_obstacles	near_an_obstacle
		adjust_trajectory
	avoid_ditch	near_ditch
		adjust_trajectory
	move	move_required
		no_move_required

time and how its interest in the treasure its collected treasure decreases.

4.4.2 Internal world

To define the function of the internal control system we require a definition of an internal world, which in turn requires some knowledge of the implementation. (Note that this is one area where the architecture specification actually requires knowledge of implementation details).

Rules are run by a rule-interpreter, which runs each ruleset in the order specified in the agent architecture definition. The implementation is based on the SIM_AGENT package [35]. In SIM_AGENT, an agent execution is a sequence of sense-decide-act cycles which may be called *agent-cycles*. Each agent may be run concurrently with other agents by a scheduler, which allocates a time-slice to each agent. For simplicity we assume that agents are run at the same "speed" and that exactly one agent-cycle is executed in one scheduler time-slice.

To monitor the operation of its own software the system must have a representation of internal events (e.g. in the form of execution traces) as well as access to its various components so that they can be repaired or replaced as necessary. We call this the "internal world". During each cycle of an agent execution, the agent leaves an "observable trace" that can be sensed by its internal sensors (S_I). An *event record* is generated for each rule that has its conditions checked and for each rule whose actions are executed. The trace read by S_I is a sequence of such event records. Thus the agent has a record of its own operation in the previous cycle.

The internal sensor readings would normally include the trace left by C_I itself. However, as argued above, there is no component within the single agent that can monitor this trace independently and respond reliably to problems indicated by it. For example, any anomaly in C_I's trace may mean that the anomaly-detection itself is defective. Similarly if the internal sensors are defective, and they leave an anomalous trace, these same defective sensors would have to read the trace. Consequently the inclusion of C_I's own trace in the input to C_I would not be an effective use of monitoring resources and we have excluded it. The agent is therefore only partially reflective (it monitors its object-level only).

4.4.3 Artificial immune system algorithms

The content of C_I in figure 4(a) for our prototype is similar on a high level to the algorithms used in some artificial immune systems (AIS) such as Forrest et. al. [10], [12] and Dasgupta and Forrest [9]. An AIS requires two algorithms:

- a training algorithm which runs during a protected "training phase" to acquire a model of "self" and the capability to discriminate between "self" and "nonself" patterns;
- an anomaly-detection and response algorithm for use during the "operational phase" when real intrusions can occur.

In contrast to our approach, however, AIS self-models are typically sets of subsymbolic strings which are matched with real execution patterns using numerical and syntactic string comparison methods. An abnormal string or a significant mismatch between sets of strings does not necessarily identify a component which may be causing the problem. Furthermore, it is difficult to use syntactic and numerical methods for evaluation of "quality".

In our very simplified version of an immune system, the model of "self" is a set of symbolic (semantic) statements indicating which events almost always happen (and would be anomalous if they did not happen) and which events rarely or never happen (meaning they would be anomalous if they happened). Details of training and model acquisition algorithms are different from the details of Forrest's algorithm and are given in [19]. In particular, the training phase includes the discovery of *contexts* or "modes" when observing another agent's meta-level (e.g. which components are characteristically active during training phase but never during operational phase?). Modes are themselves associated with types of activity. For example the two modes: "anomaly detected" and "no anomaly detected" are associated with "monitoring activity" of operational phase.

Examples of the kind of statements in the model might be as follows:

- "Rule X of ruleset Y of agent A almost always fires (regardless of mode)"
- "Rule X of ruleset Y of agent A almost always fires when A is in mode 1 of activity Z" (for example, when A is in training phase of immune system activity)
- "Rule M of ruleset N of agent B never fires when B is in mode 2 of activity W"
- "Ruleset Y of agent A is always active when A is in mode 1 of activity W"
- "Ruleset M of agent B is never active when B is in mode 2 of activity Z"

The first three examples are typical of the model acquired in the damage-tolerant prototype presented in [19]. The last two examples show a more coarse-grained "resolution" in the model used in later prototypes which tested specific hostile code scenarios.

This type of model is "shallow" in that it is very intolerant of noise (although it has measures of "approximately always" and "approximately never"). The idea is to focus on sharp distinctions which provide the most information, a commonly used technique in concept learning and some data mining algorithms such as [29], [32] and [21].

There is no need for the meta-level C_I to be an artificial immune system. Any kind of intrusion-detection system might be used. We have focused on immune system algorithms because their stated objective is to build a model of "self" and to make a distinction between self and nonself, which is what we require. However, as argued in [19] and [17], current artificial immune systems do not explicitly address reflective blindness as a software architecture problem. Instead they improve robustness implicitly by using distributed versions of algorithms over several nodes [13] and using more diversity in software design [11]; they do not specify constraints on interactions between algorithm versions. Furthermore AIS do not necessarily satisfy the causal connection requirement of computational reflection, unless they have an autonomous response capability as in [37].

5 Overcoming Limits of Two-agent Mutual Reflection

The two-agent system presented in earlier papers cannot cope with the following:

1. hostile code on an agent's meta-level;
2. simultaneous attacks on the meta-levels of different agents;

We now present a three-agent prototype which can resist these types of attack.

5.1 Tolerating Hostile Code on a Meta-level

For simplicity we use the term "hostile" to indicate any code which has destructive effects (not just "malicious" code). In particular we address the situation where hostile code detects a non-existent anomaly (or responds without detecting any anomaly) thus causing it to suppress correctly functioning components.

The two-agent implementation has a simple conflict-resolution mechanism which works as follows: The first agent to detect an anomaly in its neighbour begins its response by setting its neighbour's status to "failure" and taking over control from it if necessary. An agent in "failure status" has its anomaly-detection suppressed in order to avoid a situation in which agents "fight" for control of the vehicle. A conflict would take the form of an agent repeatedly overriding the control of the other and might be deliberately instigated as a kind of "denial of service" attack.

The "failure status" mechanism causes no problems if a meta-level is only *disabled* in an attack (as in the previous implementations). The worst case scenario is that the disabled meta-level will not detect a real anomaly in its neighbour and will fail to take over control when it should. However, for this to happen, some time must pass during which the neighbour can detect an anomaly in the meta-level and "repair" it before it causes any real problems. (This is due to the assumptions of a minimal time interval between attacks and only one agent being affected at a time).

By contrast, if a meta-level contains hostile code, it can *immediately* detect a non-existent anomaly and take control of the vehicle with no delay. This leaves no possibility of correction within the system itself because the only remaining agent has its monitoring turned off. (We excluded this scenario in the two-agent hostile code recovery system).

For example, suppose that hostile code in an agent A's meta-level falsely reports an anomaly in B's meta-level. Then A will suppress B's monitoring and take control of the vehicle if it is not already in control. This leaves it in a position to execute further hostile code while in control of the vehicle. Even if hostile code was not inserted on the object-level at the time of the first attack, there is an arbitrary amount of time for this to happen as part of a subsequent attack. (Note that an "accidental" false positive for which there is no hostile code does not present this problem; it would only cause an unnecessary switch from one agent to another).

As a solution to the above problem, three agents can determine a response based on a majority vote. Table 3 shows a simple example.

In the table, "1" means that an anomaly was detected in an agent's meta-level; "0" means that no

Table 3: Example of conflicting anomaly reports

Agent	A	B	C
A	0	1	0
B	1	0	0
C	1	0	0

anomaly was detected. In accordance with the distributed reflection principles, an agent's meta-level does not look for anomalies in itself, meaning that the diagonal is always 0.

The values in the table show what happens when A's meta-level contains hostile code. It wrongly reports an anomaly in B. However both B and C say that A contains an anomaly but B does not. Therefore the conflict can be resolved by majority voting. There remains the problem of how B and C know about each other's decisions and which one should do the responding. A solution to this is presented in the next section.

It may be argued that if a meta-level does not detect anomalies in itself then B's statement that it detects no anomaly in itself is invalid and hence the only valid votes are A and C, which conflict. However, the fact that both B and C detect that A is anomalous means that there is a majority of 2-1 in favour of A's statement being invalid. Similarly if A were to say that both B and C were anomalous, the same argument would hold, given that B and C agree that A is anomalous.

If we assume that only one agent of the three can be attacked with hostile code simultaneously and that the other agents are operating correctly (including their capability to send and receive messages), then a correct decision based on a majority vote is always possible if the individual agents can detect an anomaly correctly, with no "harmless" false-positives or negatives. This follows because the two situations where an agent can report a non-existent anomaly are the two cases above, where the compromised agent indicates that either one or both of the remaining agents are faulty respectively. In both cases the remaining agents must agree that the hostile agent is faulty, given the above assumptions.

A detailed analysis of the voting problem can be found in [5].

5.1.1 Assumptions

We are excluding "Byzantine" type failures [20], where an agent can lie to some agents but not to others (or it may lie in different ways). We are assuming that the communications infrastructure operates independently of the participating agents so that an "anomaly-detected" message is broadcast to all agents and the sender identifier is appended to a message in a way which cannot be influenced by the sending agent itself. This makes the Byzantine scenario very difficult, unless there is also a simultaneous attack on the communications infrastructure.

5.2 Tolerating simultaneous attacks on different agents

The following assumptions held for the two-agent system: First, a single attack only affects *one agent*. Secondly the time interval between attacks is above the minimum time required to allow detection and recovery from the problem when the software operates correctly. In other words, it should be physically possible for it to recover. The second assumption still holds, but the first one can be relaxed if the number of agents can be increased.

"Simultaneous" does not necessarily mean "at the same instant". It may mean that more than one attack happens in rapid succession. However, we assume that there is insufficient time to begin recovery from the previous attack, so that the attacks are approximately "simultaneous". This is simply the second assumption above in a different form, namely that recovery from one attack is not interrupted by a subsequent one.

The likelihood of two separate agents being attacked in rapid succession is reduced if agents are based on independently designed software.

5.2.1 Simultaneous disabling attacks

We first consider the case where an attack only causes omission failures on the meta-level; that is, the meta-level is disabled so that one or more of its components *do nothing*. Examples of omission failures in an agent's meta-level are as follows:

- it cannot detect an anomaly when it should,
- it cannot send or receive messages,
- it cannot respond to an anomaly (e.g. its suppression or repair capability may be disabled)

An omission failure cannot cause an agent to "lie" about anomalies in other agents.

If D is the number of simultaneous disabling attacks that can be tolerated and n is the number of agents, the following relationship should hold:

$$D = n - 1$$

assuming that every agent monitors every other agent as shown in figure 2(d). This only says that one agent should be left intact to diagnose and respond to the problem. We will show later how this works in practice.

5.2.2 Simultaneous hostile code attacks

The number H of simultaneous *hostile code* attacks on the meta-levels of different agents that can be tolerated is:

$$H < n/2$$

This is to enable a majority vote. It assumes that every agent monitors every other agent. For example, if n = 3, only one hostile code attack can be resisted; if n = 5 then 2 agents can give false reports and so on. If there is one hostile code attack, there cannot be a simultaneous disabling attack, since a majority vote requires the remaining agents to be operational. We will discuss later how an n-agent system determines whether to wait for a majority vote and whether to respond immediately.

Our closed reflective network is similar to a distributed fault-tolerant network where each processor tests every other one and where there is no single global observer. See for example [23]. It is assumed that faulty components can give false information in the same way as a software component with hostile code. Hence, the $H < n/2$ relation is also known in the distributed fault-tolerance literature where H is the number of faulty processors that can be tolerated in a network of n processors, assuming that each processor tests all others. (The general relation is more complex for a network in which some processors are observers but are not observed, and where the network may not be fully connected but this is not relevant to the current discussion). [36] present a diagnosis algorithm to be executed by each participating processor. Our algorithm (to be presented later) is based on the same basic principle with the addition of automated response.

5.3 Summary of restrictions

The architecture-related restrictions for an n-agent system (where $n > 2$) can now be summarised as follows (they are called *Arc1*, *Arc2*, etc. for *architecture* related).

Arc1: The maximum number of agents that can be attacked simultaneously with disabling attacks is $n - 1$

Arc2: The maximum number of agents that can be attacked simultaneously with hostile code is $n/2 - 1$.

Arc3: If hostile code intrusions are combined simultaneously with disabling attacks on other agents, then they are all regarded as hostile code attacks.

Arc4: If two "simultaneous" attacks occur in rapid succession, there is insufficient time to begin recovery from one attack before the next one occurs.

Arc5: If hostile code causes an agent to lie, the agent tells the same lie to all other agents (no Byzantine failures).

5.4 Comparison with distributed fault-tolerance

Our closed reflective network is similar to a distributed self-testing and diagnosis network of the kind analysed in [23] where there is no global observer. In such a system, each processor observes and tests a subset of the processors in the network and exchanges results with neighbouring processors. The messages of faulty processors will either be non-existent or anomalous (in that they don't agree with the majority). Such a self-testing network differs from our architecture in the following ways:

1. Components of the network do not acquire models of each other to compare against reality, but instead follow a standard fault-testing procedure; they do not collectively acquire a distributed model of "self".
2. The network is not required to take autonomous action on its own (although it may be coupled to an autonomous system which uses its results).

6 Design of a three-agent system

6.1 Training phase structure

For three agents, the training phase has to be extended to enable each agent to observe training and operational phases of all others. A training phase design for n agents is shown schematically in figure 5. An observation window is a concatenation of two segments labelled d_1 and d_2 in the diagram and

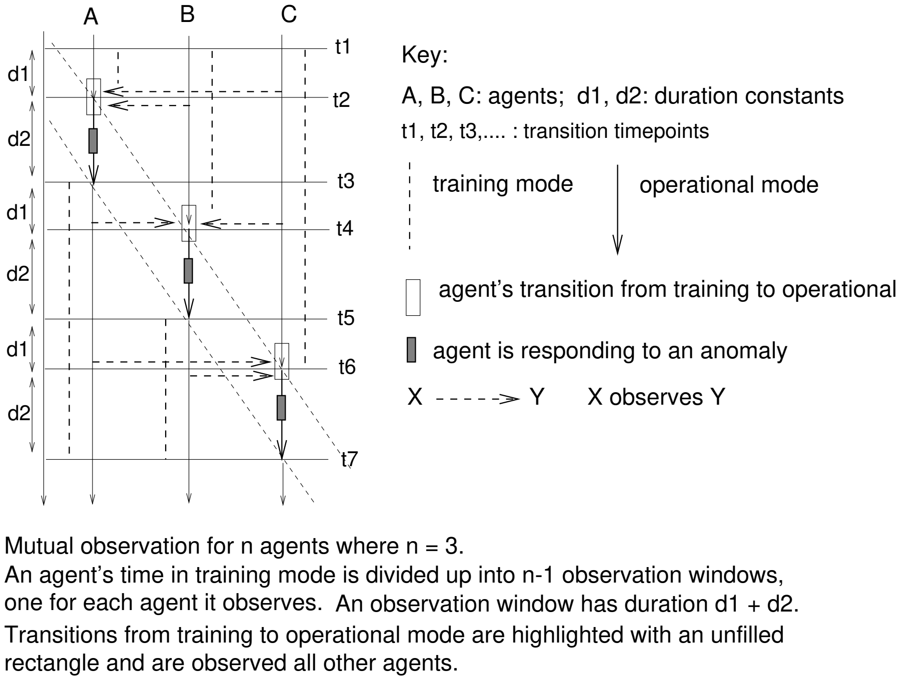


Figure 5: Staggered observation windows in training phase

represents the minimum duration of an uninterrupted training phase required to observe the different modes of an agent's behaviour. (In practice, a continuous training phase segment is often longer than this, as can be seen in the diagram).

Both segments of the window must be sufficiently long to allow observation of (a) an agent making a transition from training to operational mode and (b) the agent responding to an anomaly during operational mode. Normally $d_1 < d_2$ because we are more interested in the different submodes of operational mode than in the training mode (although this need not always be the case). d_1 only has to be long enough to enable recognition of a "mode" (as different from a fluctuation). d_2 is the duration of the "interim operational phase" (iop) and may be increased depending on the different kind of things we want to observe in this phase.

6.2 Training time scalability

If d_T is the total duration of the training phase, d_I is initial training phase duration, w is the observation window duration and n is the number of agents, then

$$d_T = d_I + nw$$

This can be seen from the diagram for 3 agents but it can easily be extended for any n agents. d_I is not shown but its ending is marked by t_1 the first transition timepoint.

6.3 Response based on majority vote

Since in the closed reflective architecture there is no single global observer, we must consider how a majority vote "causes" a response. We divide this up into the following subproblems and their provisional solutions:

- How does the voting mechanism work?
  When an agent detects an anomaly it broadcasts this to all other agents. If the agent receives a threshold T number of messages *which agree with its own decision* on whether there is an anomaly or not, then the agent believes there is sufficient support for a response.

- How does an agent know the value of T?
  The value of T is a parameter depending on the number of agents and can be varied depending on whether a fast "unthinking" response is desirable or a slow "cautious" one. It may also be learned during an extended training phase (see below). In a 3-agent system it is trivial however (T = 1).

- How does an agent know whether it should be the one to respond?
  The first agent to receive the threshold number of messages may be called the "winner". The winning agent then starts its response by inhibiting all others: broadcast to them that the situation is already being taken care of and there is no need for further counting of agreement messages etc. Then it simply acts in the same way as an agent in a 2-agent system (stochastic scheduling can be used so that the "winning" agent is non-deterministic).

The multi-agent network becomes similar to a "lateral inhibition" neural network as described in e.g. [1] in which the strongest firing neuron (the "winner") inhibits those around it.

Responding to an anomaly is non-trivial and may require identification and suppression of hostile code. In practice, it may be unlikely that several agents are ready to respond with a precise plan of action; it may be more probable that none of them can arrive at a precise identification of the hostile component. In the same way as for the two-agent architecture in [19] and [18], we make the simplifying assumption that a diagnosis and response is always possible (the "recovery" assumption). The implementation and scenarios are designed so that this assumption holds.

For clarity we use the term "inhibition message" to mean an *informative* message which simply says that the situation is already being taken care of and there is no need for further action. In contrast, "suppression" means preventing hostile code from executing and may include any means available to do this (including slowing down execution).

We assume that *only* those agents that detect an anomaly are able to take action, since the anomaly information is required to point out the hostile component (as was done in our quality monitoring implementation).
If any agent detects an anomaly it does the following:

- broadcast its decision to all other n-1 agents

- wait for other agent's results and count the number of votes that agrees with its own ("agreement" messages).

- if a threshold number of agreement messages have been received then inhibit other agents and initiate response.

- if an inhibition message has been received while there are still outstanding messages then wait for remaining messages to confirm that the inhibiting agent is supported by a majority vote.

In a 3-agent system, an example broadcast message might contain (0, 1, 0) meaning that an anomaly was detected in the second agent (B). The sending agent's identifier is given as part of the message. As stated above we assume that an agent will indicate 0 for its own status (but its field is included in the message to form a complete "fault-pattern").

If agent C detects an anomaly first and sends the broadcast, it can proceed with a response as soon as it receives one other "echoing" message (expected from agent A). The threshold is only 1 in this case. Agent C then inhibits the responses of other agents and changes the status of B to "failure". (If the execution speeds of each agent are variable, agent A could initiate the response even if it was last to detect an anomaly).

If any agent does not detect an anomaly but receives one or more "anomaly-detected" messages it does the following:

- broadcasts a "no anomaly-detected" message to all other n-1 agents;

- possibly increase its alertness, allowing it to revise its decision;

- It does not need to receive an inhibition message because it does not want to respond (it ignores any such messages).
- It should suppress any attempts to override vehicle control or put an agent into failure status until there is a majority vote.

The above assumes that there are no timeouts and that all agents comply with the rules of voting. Later we will consider what happens when we relax those assumptions.

A majority vote may be prevented by an agent's inability to to detect a real anomaly. In such a situation our implementation will allow the anomaly-detecting agent to proceed with its response in the absence of majority support, *provided there is no conflict* in the agents' diagnoses. (A scenario of this type is described below).

We now look at different hostile code scenarios in detail.

6.4 Implementation-related restrictions

One of the problems with testing of a shallow architecture is that several restrictions must be made on the intrusion scenarios which seem initially to be unreasonable. Before discussing these, we make the following assumptions:

A1: *Recurrent false-positives are hostile; intermittent false-positives are accidental*: our scenarios exclude hostile code which causes the targeted meta-level to make intermittent (apparently random) false reports of anomalies (although this is a particularly challenging problem for future work).

A2: *It is possible to distinguish between intermittent and recurrent false-positives*: This is reasonable because the current voting mechanism in the architecture can be extended to allow time for an agent's decision to "stabilise". Repeated voting to allow stabilisation is common in fault-tolerant systems. See for example [28]. Our architecture can also be extended so that repeated votes allow an agent to monitor different sources of data, resulting only in intermittent false positives (if its code is correct). Therefore any agent that produces a "flood" of false-positives contains hostile code.

Assumption A2 is a special case of the "shallowness" assumption mentioned earlier; that is, we are assuming that a certain kind of "deepening" of the architecture is possible (in this case vote stabilisation) without changing the fundamental properties of the architecture.

The following "artificial" restrictions are necessary simplifications to the intrusion scenarios and implementation.

R1: *False negatives are due to failure*: Our intrusion scenarios are designed so that their effects will always be detected as anomalous execution events provided the software is operating correctly. In other words, only a failure in the anomaly-detection software can cause a false-negative (and a failure is always due to an intrusion in our environment). This restriction is reasonable because we are not testing an individual agent's anomaly detection algorithm, but instead the interaction of components in a 3-agent architecture with voting.

R2: *Timeouts are due to failure*: Timeouts in any awaited response of an agent are caused by a component in the agent being disabled or modified. We wish to exclude other problems such as increased processing load slowing down an agent.

R3: *Final vote is available immediately*: The intrusions cause affected components to behave in a consistent way immediately, so that the first vote is the "final" or stabilised vote (unless there is a timeout). This is allowed by the shallowness assumption which states that a "deepening" of the voting mechanism (along with a more noisy intrusion scenario) will not make a fundamental difference to the architecture as a whole.

These restrictions, together with the more general assumptions can be summarised as follows:

Imp1: Any false-negative or timeout by an anomaly-detection component is due to failure of the component, which can either be an omission failure or destructive code (this is simply restating R1 and R2)

Imp2: Any false-positive by an anomaly-detection component is due to destructive code in the component (follows from assumptions and R3)

They are labelled *Imp1* and *Imp2* because they apply only to a specific *implementation* and will be referred to in the rest of the paper. For example, a consequence of both restrictions is that a correctly functioning agent A that detects only an omission anomaly in another agent B should not expect B to lie (because hostile activity cannot go undetected (*Imp1*) and omission failures cannot cause lying (*Imp2*).

7 Simple hostile code scenarios

A "simple" scenario is one in which the enemy follows the normal rules of voting, such as broadcasting its detection of an anomaly, waiting until it receives a majority in its favour and sending an inhibition message before responding.

A hostile code "attack" is implemented as in [18], where an "enemy" agent replaces a correct rule in an *object-level* component (ruleset) with a "rogue" version. We now do the same thing with a meta-level component which is specifically concerned with anomaly-detection. The aim is to cause the component to lie, by acting as if it had detected an anomaly when there is none.

7.1 Meta-level hostile code scenarios

The following hostile code scenarios were simulated and tested:

Scenario 1(a): an agent A executing hostile code states that both B and C are anomalous, and either B or C controls the vehicle. The subsequent vote should indicate that A is wrong. Since we are assuming that only one agent is attacked, the other agents will agree that A is anomalous. **Action**: the winning agent responds to diagnose and suppress hostile code in A.

Scenario 1(b): an agent A executing hostile code is already in control of the vehicle states that both B and C are anomalous. The majority vote indicates that A is wrong. **Action**: the winning agent will override A's control of the vehicle and begin its diagnosis and suppression of hostile code in A.

Scenario 2(a): an agent A executing hostile code states that B only is anomalous and B is in control of the vehicle. **Action**: majority vote and response as for 1(a).

Scenario 2(b): an agent A executing hostile code is already in control of the vehicle and states that B only is anomalous. **Action**: majority vote and response as for 1(b).

In scenario 1, the hostile code is indiscriminate about the agents it lies about. In scenario 2 it is selective (it can lie about several agents if $n > 3$). The two scenarios are summarised in tables 4 and 5. We have seen scenario 2 earlier in table 3. If we consider a deepened version of the architecture in

Table 4: Scenario 1 for 3 agents

Agent	A	B	C
A	0	1	1
B	1	0	0
C	1	0	0

which some restrictions are relaxed (for example if noise is allowed in the voting), scenario 1 may be easier than scenario 2 because the "hostile" agents are behaving in a way that contrasts most sharply with normal agents.

Table 5: Scenario 2 for 3 agents

Agent	A	B	C
A	0	1	0
B	1	0	0
C	1	0	0

7.1.1 Scenarios for n agents

Although we only implemented a 3 agent prototype, it is important to consider what happens if $n > 3$.

For $n > 3$ agents, scenario 1 would be as follows: a number of agents k where $(k > n/2)$ execute hostile code and each states that *all* remaining m agents $(m = n - k)$ are anomalous. **Action**: as for 3 agents, the majority vote should indicate that the k agents are themselves anomalous and all remaining agents are normal. The main difference is that the winning agent must suppress hostile code in k agents instead of just one. This can be seen in table 6 which gives an example with $n = 5$. Diagnosis and suppression may be not be more difficult, however, since the consistent behaviour may result from the same attack on different agents (with the same hostile code).

Table 6: Scenario 1 for 5 agents

Agent	A	B	C	D	E
A	0	1	1	0	1
B	1	0	0	1	0
C	1	0	0	1	0
D	0	1	1	0	1
E	1	0	0	1	0

Scenario 2 is more complex: a number of agents $k(k > n/2)$ execute different hostile code and each states that different subsets of agents are anomalous. **Action**: as for scenario 1, except that a clear majority vote may be more difficult in a "scaled up" version, since there may be less contrast between the "statements" of the two groups of agents. Identifying and suppressing hostile code may also be difficult if the attacks are independent and have to be diagnosed separately.

Table 7 shows a configuration of scenario 2 for five agents with two hostile agents saying different things. Agent A falsely says that B is anomalous, while D falsely says that C is anomalous. The "*" is a "don't care" symbol, meaning they can say anything about the remaining agents, including each other.

Table 7: An example of Scenario 2 for 5 agents

Agent	A	B	C	D	E
A	0	1	*	*	*
B	1	0	0	1	0
C	1	0	0	1	0
D	*	*	1	0	*
E	1	0	0	1	0

7.2 Tolerating simultaneous attacks

Returning to the 3 agent prototype, if two agents are disabled simultaneously, the remaining agent will detect an anomaly in both of them, which it broadcasts. Situations where an agent A detects an omission anomaly in B and C are as follows:

Scenario 3: A detects omission anomalies in B and C:

B and C both agree that there are no anomalies in A or they time out (they may detect anomalies in each other). **Action**: A responds either immediately or after a timeout. This does not require a majority vote because there is no conflict. Furthermore A only detects an omission anomaly, and the required response is a repair, where the worst case is an unnecessary repair (equivalent to a false positive).

For $n > 3$ agents: a subset of agents m all detect omission anomalies in k agents where $k <= n-1$ and $m = n - k$ because of *Imp1*. None of the k reportedly faulty agents detect anomalies in any of the m agents reporting the problems. **Action**: If $m > 1$ the winning agent responds, otherwise the only intact agent responds.

Scenario 3(a): (redundant) A detects omission anomalies in B and C:

B and C both state that A is anomalous indicating that its decision about B and C cannot be trusted. Within the restrictions, B and C must be correct. (If they were both wrong there are two "lying" agents, which we are excluding). The majority-triggered response happens in the normal way. This variant is redundant because it is the same as scenario 1 above.

Scenario 3(b): (excluded) A detects omission anomalies in B and C:

One of B or C states that A is anomalous but the other does not. This is excluded by the restrictions. Either B or C must be "lying" while A is stating they are both faulty, so *either* two agents are lying (A and one of B or C) *or* one agent is lying (the one that says that A is faulty) and the remaining one is disabled (as A has detected correctly). Restriction *Arc3* excludes this.

Table 8 shows scenario 3 for three agents. The "*" is a "don't care" symbol meaning that B and C may or may not detect omission anomalies in each other or they may time out. Scenario 3 requires that both B and C do *not* detect an anomaly in A (they may timeout). This scenario was tested successfully. Table 9 shows that scenario 3(a) is the same as scenario 1. Scenarios 1 to 3 may be called

Table 8: Scenario 3 for 3 agents

Agent	A	B	C
A	0	1	1
B	0	0	*
C	0	*	0

Table 9: Scenario 3(a) for 3 agents which is redundant

Agent	A	B	C
A	0	1	1
B	1	0	0
C	1	0	0

"simple" because they assume that the voting mechanism itself is not subverted. In the real world, however, more complex situations are likely.

8 More difficult scenarios

An agent may experience further anomalous behaviour during a vote. To recognise such anomalies, the training phase was modified so that it included observation of voting behaviour. A single agent is observed detecting and responding to an anomaly in the same way as in a normal training phase. However, before the agent responds it activates the same components that are active during a vote, so the characteristic signature of a vote can be learned, even if it is only an "exercise". The components include broadcasting, waiting for a majority (threshold) and sending an inhibition message. The differences between such an exercise and reality should be minimised, although they will not be eliminated. For example, the event indicating the threshold number of messages (majority vote) is replaced by a condition indicating that it is in training phase. Otherwise the component is active in the same way. (We did not include a delay during the "waiting behaviour", but this can be added if required).

We successfully tested the implementation in some scenarios involving timeouts and premature anomaly response, where an agent does not comply with voting rules. The scenarios are not intended to be exhaustive but give an insight into the complexity of the problem.

8.1 Timeouts

Some timeout scenarios are as follows:

Scenario 4(a): An agent A detects an omission or hostile anomaly in another agent B. While waiting for a majority vote, A experiences a timeout from B. **Action**: A and one other agent should agree that a response is required and the winning agent will do this.
For $n > 3$: A experiences that a number of agents less than $n/2$ have timed out. The timed out agents are also the ones that A has detected anomalies in. Action as for 3 agent case, except that several agents have to be repaired.

Scenario 4(b): (excluded) A detects *omission* anomalies in B and C. While waiting for a majority vote, A experiences a timeout from B and C. **Action**: In our current implementation, this situation is excluded because an agent in this position does not wait for any vote but immediately repairs faulty components in B and C (a majority vote is not expected to be reliable). This scenario is a special case of scenario 3, in which the omission failures cause a timeout in both remaining agents (the "*" symbol can mean this).

Scenario 4(c): (redundant) A detects an omission anomaly in either B or C. While waiting for a majority vote, A experiences a timeout from B and C because A's own message receiving capability is disabled. In this case the remaining agents will agree that it is faulty and the winner will repair its message-receiving component. This is a situation where a faulty agent can itself detect a real anomaly. The scenario is also a special case of scenario 3, in which the "*" is "1" for a faulty agent which correctly detects an anomaly in the remaining faulty agent (except that the faulty agents are not B and C as in the table 8 but instead A and another agent).

Scenario 4(d): (excluded) A experiences a timeout on receiving a vote from an agent in which no anomaly was detected. This is excluded by restriction *Imp1* above.

Scenarios 4(a) and 4(c) were tested.

8.2 Subverting the voting rules

We consider the situation in the 3-agent system where an agent B begins responding to an anomaly in C or A without waiting for a majority vote. It may comply with the voting rules initially (for example it may announce that it has found an anomaly). Premature response can happen in the following situations:

Scenario 5(a): B sends an inhibition message before it receives a majority to support its response. **Action**: none, unless there is a majority saying that B itself is anomalous. (See below for details) of this problem).

Scenario 5(b): B does not send an inhibition message but immediately activates its anomaly response after broadcasting that it has detected an anomaly in other agent(s). **Action**: see below.

Scenario 5(c): B activates its anomaly response without broadcasting any vote or sending any inhibition message. **Action**: see below.

In scenario 5(a), the inhibition-receiving agent (A) does not know whether B has received a threshold number of votes. Therefore it has no way of knowing whether B is responding correctly or not. It is possible that B's component for vote-counting and waiting for a majority is disabled and that B has "honestly" detected an anomaly in C but could not broadcast it. Our implementation is "optimistic" in that it trusts an agent that appears to be complying with the voting rules unless there is evidence that it should not be trusted. In other words, a premature response is only suppressed if there is a majority saying that B itself is anomalous. Even if the receiving agent disagrees with B's diagnosis, this is not sufficient reason to suppress it. Consequently, an inhibition message from an agent which is already regarded as suspect is not "believed", but in all other cases it is.

In scenarios 5(b) and 5(c), the lack of an inhibition message preceding a response is regarded as anomalous by both remaining agents. Hence there should be a majority vote in favour of stopping B's response. In scenario 5(c), the counter-response may not happen immediately because the initial vote (triggered by B's anomaly announcement) may not at first indicate that B is anomalous. The premature response should then be detected by one of the remaining agents which triggers a new vote, resulting in a suppression action if there is a majority.

In scenarios 5(b) and 5(c) we assume that the remaining agents have time to identify and suppress the hostile code in B before B does significant damage to their ability to do this.

We assumed that the above are suppression responses. Within our current restrictions, a premature repair response need not be suppressed because the worst case scenario is an unnecessary "repair". B may be correctly repairing another agent but its voting components may have been disabled. This is the case where two agents have disabled components simultaneously, something which is allowed by the assumptions. If B is incorrectly repairing another agent due to any hostile code, it does no more damage than a false positive (unless there is a series of repairs with maximum frequency - denial of service)

By contrast, if B attempts to suppress another agent C without a majority vote then B's suppression action must be hostile. The situation where it is suppressing C correctly is excluded because C would then have been attacked before B's disabled voting components could be repaired, a situation which is not allowed by the assumptions for the three-agent system.

8.3 Avoidance of deadlock

Deadlock is a situation where agents indefinitely wait for each other to take some action. Something similar to deadlock might happen in this implementation if an agent waits indefinitely to receive a threshold number of votes. However, this should not happen unless the timeout mechanism for all n agents is disabled or modified simultaneously. There is no other situation where an agent waits for something. Although in a scaled up version, this might change (for example, an agent's next action might depend on the result of a diagnosis from a specialist agent, which is in turn waiting for the first agent to finish its action).

9 Related Work

9.1 Societies of agents

Multi-agent teams can be used for distributed intrusion detection. An example is AAFID [38] which is based on multi-agent defence introduced by [7]. Each agent observes some aspect of network traffic and acquires a model of its normal activity during a training phase so that anomalies can be detected. However, there is no requirement for agents to observe each other. Similarly, there is no explicit concept of "self" as with artificial immune systems.

Because the system is distributed, an attack on the intrusion detection system itself (subversion) may be detected by using multiple redundancy (several agents observing the same activity) and cross-checking their results. Hence the basic approach is similar to that of distributed self-testing above. However, at the time of writing a detailed architecture for cross-checking has not been developed.

Multi-agent architectures which are strictly hierarchical, such as [2], have the same limitations of hierarchical layered intrusion-detection which corresponds to figure 1(a) and (b) and figure 4(a).

An example from the agent-society category that is most relevant to reflective blindness is "social diagnosis" (Kaminka and Tambe [15]) based on agent tracking (Tambe and Rosenbloom [40]). The idea is that agents observe other agents' actions and infer their beliefs to compensate for deficiencies in their own sensors. E.g. if an agent is observed to swerve, it can be inferred that something exists which it wishes to avoid, such as a hole in the road. In particular, an agent may discover a fault in its own operation by observing the reaction of other agents.

However, for our problem, such models are unnecessarily restricted by the "society" metaphor. In a society, the internal operation of agents are not included in the observed world. It follows that such a multi-agent network can only be a distributed reflective network in a weak sense (only insofar as an agent A_1 can detect faults in its own external behaviour by observing A_2's external reaction, hence the term "social diagnosis"). There is however the possibility of introducing multiple perspectives into such a model (i.e. different representations of the world), where one perspective may include features that are unknown in others.

9.2 Hofmeyr and Forrest's distributed immune system architecture

A distributed architecture ARTIS for an artificial immune system is presented by [13]. The architecture is very closely modelled on the natural immune system and is represented as a network of nodes called "localities", each of which runs an instance of the negative selection algorithm described above. This architecture covers the problem of self-monitoring implicitly using the decentralisation and diversity inherent in the natural immune system. However, the problem of reflective blindness is not explicitly addressed and there are no requirements on the interaction between localities.

10 Conclusions and Future Work

The prototype has shown that a closed reflective network can be implemented in practice which is not restricted to two-agent mutual reflection. However, it also points to some hard problems for future work. In particular, the voting system involves the addition of some complexity to the interaction between agents and we have seen how this complexity can introduce new types of failure or possibilities of exploitation by an attacker.

An alternative to software voting might involve "hardwiring" the voting system so that the multi-agent network looks like a simplified neural network. In such an architecture, each agent runs on a separate processor or "node". An anomaly-response by one node might be made physically impossible unless "enabled" by a threshold amount of support (or "excitation") by neighbouring nodes. The initial stage of the response involves mostly diagnostic reasoning and may be done at any time by all agents, regardless of majority support. The final stage (the actual intervention in another agent's code) may be "compiled" into an executable program in advance and be ready to be activated by sufficiently strong excitation from neighbouring nodes. This kind of compilation is similar to that used by the situated automata of [31]. In a scaled up version, the excitation should not become strong until the votes have stabilised and a *persistent* majority is shown.

The mutual observation stage of the training phase may also be more easily implemented in parallel with separate processors gathering data on each other's normal activities. It may also be advantageous to have several processors which remain in training phase after the remainder of the network has become operational. In this way the models of normal behaviour can be regularly updated to include new patterns of activity whose effects are acceptable.

Future work will involve the scaling up of the architecture to allow more "noisy" intrusions and anomaly-detection, including voting behaviour. For example, allowing time for vote stabilisation requires the learning of "normal" patterns of stabilisation to be included in the training phase. What kind of voting delay is *acceptable* may also be learned by observation or it may be directly specified. Having acquired this knowledge, an agent participating in a vote during operational phase may believe that the delay in waiting for stabilisation has reached a tolerance threshold and then broadcast an "organisational" vote that the voting procedure should be ended because it may be causing more disruption than the actual intrusion itself (if there is a real one). This is merely another form of "meta-management".

References

[1] Igor Aleksander and Helen Morton. *An Introduction to Neural Computing.* International Thomson Publishing Company, 1995. Pages 158–162.

[2] Jai Sundar Balasubramaniyan, Jose Omar Farcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. Technical Report COAST TR 98-05, Department of Computer Sciences, Purdue University, 1998.

[3] J. Bates, A. B. Loyall, and W. S. Reilly. Broad agents. In *AAAI spring symposium on integrated intelligent architectures*. American Association for Artificial Intelligence, 1991. (Repr. in SIGART BULLETIN, 2(4), Aug. 1991, pp. 38–40).

[4] Luc P. Beaudoin. *Goal processing in autonomous agents*. PhD thesis, School of Computer Science, 1994.

[5] Andrea Bondavalli, Silvano Chiaradonna, Filicita Di Giandomenico, and Lorenzo Strigini. Rational Design of Multiple Redundant Systems. In B. Randell, J.-C. Laprie, H. Kopetz, and B.Littlewoods, editors, *Predictably Dependable Computing Systems*. Springer Basic Research Series, 1995.

[6] Ron Chrisley. Non-conceptual Content and Robotics: Taking Embodiment Seriously. In K. Ford, C. Glymour, and P. Hayes, editors, *Android Epistemology*, pages 141–166. AAAI/MIT Press, Cambridge, 1995.

[7] Mark Crosbie and Gene Spafford. Active defense of a computer system using autonomous agents. Technical Report 95-008, Department of Computer Sciences, Purdue University, 1995.

[8] D. Dasgupta and N. Attoh-Okine. Immunity-based systems: A survey. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Orlando, October 1997.

[9] D. Dasgupta and S. Forrest. Novelty-detection in time series data using ideas from immunology. In *Proceedings of the International Conference on Intelligent Systems*, Reno, Nevada, 1996.

[10] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukun. Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, 1994.

[11] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[12] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[13] S. A. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473, 2000.

[14] Steven A. Hofmeyr. An interpretative introduction to the immune system, 2000. To appear in *Design Principles for the Immune System and Other Distributed Autonomous Systems*, edited by L.A. Segel and I. Cohen. Santa Fe Institute Studies in the Sciences of Complexity. New York: Oxford University Press (In Press).

[15] G. A. Kaminka and M. Tambe. What is wrong with us? improving robustness through social diagnosis. In *Proceedings of the 15th National Conference on Artificial Intelligence(AAAI-98)*, 1998.

[16] C. M. Kennedy. Anomaly-driven concept acquisition. In *Proceedings of the Workshop Machine Learning and Concept Acquisition of the 1998 German Conference on Artificial Intelligence (KI'98)*, Bremen, Germany, September 1998.

[17] C. M. Kennedy. Evolution of self-definition. In *Proceedings of the 1998 IEEE Conference on Systems, Man and Cybernetics, Invited Track on Artificial Immune Systems: Modelling and Simulation*, San Diego, USA, October 1998.

[18] C. M. Kennedy and A. Sloman. Autonomous Recovery from Hostile Code Insertion using Distributed Reflection. Technical Report CSR-02-2, University of Birmingham, School of Computer Science, 2002.

[19] C. M. Kennedy and A. Sloman. Reflective Architectures for Damage Tolerant Autonomous Systems. Technical Report CSR-02-1, University of Birmingham, School of Computer Science, 2002.

[20] L. Lamport, R. Shostack, and M. Peace. The byzantine general's problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[21] W. Lee, S. Stolfo, and K. Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14:533–567, 2001.

[22] P. Maes. Issues in computational reflection. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 21–35. North-Holland, 1988.

[23] Gerald M. Masson, Douglas M. Blough, and Gregory F. Sullivan. System diagnosis. In D. K. Pradhan, editor, *Fault-Tolerant Computer System Design*. Prentice-Hall, New Jersey, 1996.

[24] H. Maturana and F. J. Varela. *Autopoiesis and Cognition: The Realization of the Living*. D. Reidel Publishing Company, Dordrecht, The Netherlands, 1980.

[25] Marvin Minsky. Steps toward artificial intelligence. *Proceedings Institute of Radio Engineers*, 49:8–30, 1961.

[26] R. Oehlmann. Metacognitive adaptation: Regulating the plan transformation process. In *AAAI Fall Symposium on Adaptation of Knowledge for Reuse*, 1995.

[27] Barney Pell, Douglas E. Bernard, Steve A. Chien, Erann Gat, Nicola Muscettola, P. Pandurang Nayak, Michael D. Wagner, and Brian C. Williams. An autonomous spacecraft agent prototype. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 253–261, New York, 5–8, 1997. ACM Press.

[28] Dhiraj K. Pradhan and Prith Banerjee. Fault-Tolerant Multiprocessor and Distributed Systems: Principles. In D. K. Pradhan, editor, *Fault-Tolerant Computer System Design*. Prentice-Hall, New Jersey, 1996.

[29] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[30] R. Rosen. *Life Itself*. Columbia Univerity Press, New York, Complexity in Ecological Systems Series, 1991.

[31] Stanley J. Rosenschein and Leslie Pack Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73(1–2):149–173, 1995.

[32] Rudy Setiono and Wee Kheng Leow. FERNN: An algorithm for fast extraction of rules from neural networks. *Applied Intelligence*, 12(1-2):15–25, 2000.

[33] A. Sloman. Prospects for AI as the general science of intelligence. In *Proceedings of the 1993 Convention of the Society for the Sudy of Artificial Intelligence and the Simulation of Behaviour (AISB-93)*. IOS Press, 1993.

[34] A. Sloman. What are virtual machines? are they real?, October 2001.

[35] A. Sloman and R. Poli. Sim_agent: A toolkit for exploring agent designs. In Joerg Mueller Mike Wooldridge and Milind Tambe, editors, *Intelligent Agents Vol II, Workshop on Agent Theories, Architectures, and Languages (ATAL-95) at IJCAI-95*, pages 392–407. Springer-Verlag, 1995.

[36] Arun K. Somani. System level diagnosis: A review. Technical report, Dependable Computing Laboratory, Iowa State University, 1997.

[37] A. Somayaji. Automated response using system-call delays. In *9th Usenix Security Syposium*, August 2000.

[38] E.H. Spafford and D. Zamboni. Intrusion detection using autonomous agents. *Computer Networks 34*, pages 547–570, 2000.

[39] R.S. Sutton. *Temporal Credit Assignmenment in Reinforcement Learning.* PhD thesis, University of Massachusetts, School of Computer and Information Sciences, 1984.

[40] M. Tambe and P. S. Rosenbloom. Architectures for agents that track other agents in multi-agent worlds. In *Intelligent Agents, Vol. II*, LNAI 1037. Springer-Verlag, 1996.

[41] F.J. Varela. *Principles of Biological Autonomy.* North-Holland, New York, USA, 1979.

[42] I. Wright and A. Sloman. Minder1: An implementation of a protoemotional agent architecture. Technical Report CSRP-97-1, University of Birmingham, School of Computer Science, 1997.