

Intelligent Software Decoys

James Bret Michael and Richard D. Riehle

Department of Computer Science, Naval Postgraduate School
833 Dyer Road, Monterey, California 93943-5118
bmichael@cs.nps.navy.mil, rdriehle@nps.navy.mil

Abstract

We introduce an abstraction known as an intelligent software decoy for protecting objects within a component-based architecture from egregious and malicious use by mobile agents. If an agent misuses or tries to circumvent the published interface specification of an object, then the object switches from its nominal operating mode to a deception mode. While serving as a decoy, an object attempts to both deceive the agent into concluding that its violation of the interface specification has been successful and assess the nature of the violation. The interface specification is treated as a contract consisting of preconditions, postconditions, and a class invariant. Failure of a precondition triggers the transition between modes. An intelligent software decoy is adaptable, autarkic, polymorphic, and self-replicating. The decoy disguises and defends itself by modifying its contract at run-time through the use of both polymorphism and late binding. The nature and extent of any change to an object is governed by its class invariant.

Keywords: Agent, authentication, buffer overflow, component-based software architecture, contract, deception, decoy, distributed systems, encryption, object, polymorphism, security, software

1. INTRODUCTION

Suppose that there exists a distributed system of thousands of sensors, comprising part of an intelligence-gathering information system, in which each sensor is field-programmable via software hooks. An intelligence analyst could broadcast messages to the sensors instructing them to either activate or deactivate themselves, change their mode of operation (e.g., from filtering to no filtering of sensor inputs), or install and execute new software (e.g., for controlling sensing of phenomena or encrypting communication with the data-collectors). In addition, the analyst could query the status of the sensors to assess the organization's level of readiness for tracking an enemy's movement of troops and weapons.

On arriving at the software interface of a sensor, a mobile agent interacts with the sensor to reach the goal assigned to the agent by its owner. However, if the mobile agent is poorly designed, its flaws may lead the agent to try to interact with the sensor software in a way that was not intended by the creator of the agent; this is an example of an egregious but non-malicious use of the object. This could include unintentionally triggering a change in the operating mode of the sensor. If the software agent is malicious, it may try to sabotage the sensor. For example, it might try to alter the sensor software so that the movement of enemy forces will not be detected or reported by the sensor. If the software is written in Java, the agent might try to change the behavior of one of the objects or classes. In early versions of the Java Virtual Machine (JVM), such an attack was quite easy to effect due to the fact that a rogue process could insert its own class definition using the same name as the original predefined Java class [13].

The intelligence-gathering example illustrates the need for permitting mobile agents to modify the software-controlled behavior of a distributed system or a subset of the objects that comprise the system. On the other side of the coin, the object needs to be protected from egregious or malicious acts by an agent to misuse the object or modify the object in an unintended or unauthorized manner. By egregious, we mean an unintentional or non-malicious use or modification of the object's interface or behavior, while malicious refers to an attack on the object.

One approach to protecting the intelligence-gathering system is to both encrypt the messages and authenticate the mobile agents to the software objects. However, McHugh and Michael have identified some of the challenges in managing cryptographic keys in distributed systems, especially when group membership (e.g., subgroups of the sensors) changes frequently [14]. Moreover, an authenticated mobile agent may have been compromised, or its creator, who at one time was trustworthy, may no longer be so. In summary, encryption and authentication do not address the issue of discovering and responding to the goals or actions of mobile agents.

Another example of an approach to protecting distributed systems from mobile agents is to require that the agents only interact with software objects via a formal interface specification known as a software contract, as introduced in Meyer's design-by-contract model [16]. However, an agent might try to bypass the contract to modify the behavior of the targeted object. Thus, precondition assertions for controlling access to the object may only be effective at thwarting the actions of

non-malicious agents, that is, agents whose flawed design induces unintended interactions with objects through the interfaces to these objects. This is known in the epigrammatic world as “Locks are intended to keep honest people honest.”

We introduce an abstraction known as an intelligent software decoy for protecting objects within a component-based architecture from egregious and malicious use by mobile agents. If an agent misuses or tries to circumvent the published interface specification of an object, then the object switches from its nominal operating mode to a deception mode. While serving as a decoy, an object attempts to both deceive the agent into concluding that its violation of the interface specification has been successful and assess the nature of the violation. The interface specification is treated as a contract consisting of preconditions, postconditions, and a class invariant. Failure of a precondition triggers the transition between modes. An intelligent software decoy is adaptable, autarkic, polymorphic, and self-replicating. The decoy disguises and defends itself by modifying its contract at run-time through the use of both polymorphism and late binding. The nature and extent of any change to an object is governed by its class invariant.

2. SOFTWARE DECOYS

A decoy is intended to deceive something or someone into believing it is the object it advertises itself to be. Therefore, the creator of a decoy must actualize the decoy as much as possible to complete the deception. The more the external observer is deceived, the better the decoy is performing its role. Daniel and Herbig define deception as the “deliberate misrepresentation of reality done to gain a competitive advantage” [5].

When a duck hunter deploys decoys on a lake, those decoys are painted to resemble the species of duck being pursued. If the decoys can be made to move about, the deception may be more effective: the real ducks will think that the decoys are also real since the decoys appear to be paddling through the water. In this case, the effectiveness of the decoys need only be good enough so as to draw the real ducks within shotgun range.

An intelligent software decoy has some of the same properties as the physical decoy. It certainly has the same objective: deception. If the decoy is intelligent, it can continually deceive the target of the deception into action that accomplishes several goals. In the case of an attack or the deployment of countermeasures executed by an attacker, one of the goals of the owner of the decoy is to protect the actual entity being shielded from attack and anti-decoy countermeasures.

Another goal, in the context of an attack, is to ensure that every attack reveals the presence of an attacker. In this way, the decoy can use its own intelligence to deploy more decoys and to alert other objects that an attack signature has been identified. As more decoys are deployed, their creator can also alter their own characteristics so that the decoys appear to be different from the one originally attacked.

In an ideal situation, the decoys will be able to adopt a chameleon-like character that allows them to appear to be different as other decoys and attackers change form. In the context of software decoys, this model of decoys raises the concept of intelligent agents to a new level of sophistication. It requires that both the interfaces and the objects be polymorphic, that is, the contract for each object must be polymorphic. Consequently, any message to a decoy can be encrypted, but the decoy will have its own knowledge of the encryption scheme based on the parameters of the polymorphic message. Successful execution of the decoy will require satisfying the precondition, the invariant, and the postcondition. Since the postcondition is internal to the object, it is not easily compromised even with dynamic patching schemes.

3. PRIOR RESEARCH

The general notion of a software decoy is not new. For example, the term “decoy” has been used in the context of reasoning with incomplete information in multiagent systems. According to Zlotkin and Rosenshein [27],

One obvious way in which uncertainty can be exploited can be in misrepresenting an agent’s true goal. In a task oriented [*sic*] domain, such misrepresentation might involve hiding tasks, or creating false tasks (phantoms, or decoys), all with the intent of improving one’s negotiating position. The process of reaching an agreement generally depends on agents declaring their individual task sets, and then negotiating over the global set of declared tasks. By declaring one’s task set falsely, one can in principle (under certain circumstances), change the negotiation outcome to one’s benefit.

This earlier research indirectly addresses the Byzantine Generals problem [12] in that there was an attempt to construct incentive-compatible negotiation mechanisms such that “no agent designer [*sic*] will have any reason to do anything but make his agent declare his true goal in a negotiation.” In contrast to the work of Zlotkin and Rosenshein, in which interaction between agents was investigated, we explore the use of software decoys in the context of the interaction between agents and software components.

Turing introduced the “imitation game” [25], now known as the Turing test, for testing the intelligent behavior of software. The participants in the test consist of a computer, a human, and an interrogator. The goal of the interrogator, who is a human subject, is to distinguish between the computer and the human with whom he or she carries on a conversation. The

identity of the respondent, that is the computer or human, is hidden from the interrogator. The measure of intelligent behavior of the software system is the percentage of time that the interrogator cannot correctly distinguish between the response of the computer, which simulates a human response, and that of the person typing responses. Thus, the game is one of deception: programming a machine to deceive, via impersonation, a human into believing that the machine he or she is conversing with is also a human being.

In contrast to the approach taken by Turing to test for computer intelligence, Goldberg [8] attempts to address questions of intelligent reasoning by computers by arguing that a computer cannot deceive itself. His argument relies on a “common-sense view of the mind,” that is, that a computer cannot possess beliefs or self-knowledge, as can a human. However, Goldberg does not address the issue of whether one computer can deceive another. In our own work, we argue that it is possible for a software component to deceive an agent by creating a deception based on either direct inspection of the internal state of the other agent, or alternatively, assessing the intentions of the agent by monitoring the agent’s behavior. In addition, we subscribe to the theory posed by Hirstein that self-deception can be due to conflicts other than between beliefs, namely, a “conflict between two representations, a ‘conceptual one’ and an ‘analog’ one” [9]. Our conception of a decoy is one in which a decoy, agent, or other type of software can itself possess conflicting representations.

In [6], examples are presented of the use of deception in military campaigns dating back thousands of years. In [4], Cohen presents a classification of defenses for information systems, in which one of those defenses is deception:

Defence 98: deceptions. Typical deceptions include concealment, camouflage, false and planted information, reuses [*sic*], displays, demonstrations, feints, lies, and insight (Dunnigan, 1995). Examples include facades used to misdirect attackers as to the content of a system, false claims that a facility or system is watched by law enforcement authorities, and Trojan horses planted in software that is downloaded from a site. Deceptions are one of the most interesting areas of information protection, but little has been done on the specifics of the complexity of carrying out deceptions. Some work has been done on detecting imperfect deceptions.

Cohen has explored this class of defense for use in protecting computing resources in a distributed system. He refers to such protection techniques as “defensive network deceptions” [2], and has attempted to develop formal models of defensive deceptions and the types of attackers for which these deceptions are to be used. In one of these models, the attacker is characterized as an agent “who believes that information systems are vulnerable and [the attacker] has finite resources to attack” the systems. In this model, the attacker relies on intelligence reports about the information systems in order to identify and choose a specific vulnerability of the system to target, and that the attacker will not attack unless it believes that “there exists an exploitable weakness of value.” In the other model Cohen presents, the attacker and defender are both assumed to believe that all systems of positive non-zero economic worth have at least one exploitable weakness.

Cohen introduces six goals for defensive network-deceptions [2]; they are to make the following:

1. Likelihood of any individual intelligence probe encountering a real vulnerability low.
2. Likelihood of any individual intelligence probe encountering a deception high.
3. Time to defeat a deception infinite.
4. Time to detect a vulnerability once a deception is encountered from a given attack location infinite.
5. Time to detect an intelligence probe against a deception very small.
6. Time to react to an intelligence probe against a deception very small.

These goals, to some extent, have been incorporated into the Deception Toolkit (DTK) [3]. Prior to the emergence of the DTK, the most widely used type of tool for defensive network deception was the honey pot, which is still used today. A honey pot is a decoy that is placed in a highly visible location within an information system so as to draw the attention of attackers. According to Cohen, honey pots have not proved to be very effective at influencing the decision making of an attacker because each honey pot “consumes such a small portion of the overall intelligence space and has little effect on altering the characteristics of the typical intelligence probe” [2].

The DTK distributes deceptions throughout the network to be protected, with the deceptions utilizing unused network-system resources. An example of a deception that can be created using the DTK is to populate the network with IP addresses masquerading as addresses of valuable system resources: the fake IP addresses and dummy resources associated with them serve as decoys. The DTK has evolved from a simple extension to honey-pot systems to incorporate techniques to both increase the size of the search space (i.e., for a real versus decoy service) and the sparseness of actual vulnerabilities. Cohen has also used the DTK as an experimental apparatus for testing strategies to improve the quality of deceptions. The strategies he lists in [2] include the following: injecting synthetic network traffic into the network, reconfiguring the deception network over time, injecting synthetic information about the organization and its constituents into the system, and using real systems rather than software sandboxes as decoys.

Moose [17], like Cohen, has tried to model deception from a systems view. He explicitly models the evolution of pairs of stimuli and responses between the defenders of a system who are using deception techniques and that of the attackers. The

modeling paradigm is intended to capture deception and counter-deception scenarios, the plans of actors (i.e., defender and attacker), uncertainty associated with intelligence information, feedback loops, and the risk models of actors.

The Denial and deception Analyst Workbench (DAWS) [11] is an interactive system used by intelligence analysts to maintain denials and deceptions, in other words, cover stories. The workbench consists of a set of integrated tools, managed by an expert system. DAWS pre-processes raw intelligence data so that it can be automatically forwarded to analysts based on pattern matches on their information-needs profiles. The other tools help the user manage denials and deceptions that are perpetuated for a target audience. DAWS and DTK are similar in that they both are designed with the human in the loop.

The development of counter-deception techniques has been a very active area of research in the information theory community. For example, in the 1970s, Gilbert et al. [7] explored the use of codes to detect evidence of deception on the part of an opponent that tries to intercept or change messages between a transmitter and its intended receiver. The opponent tries to capture message streams on a channel without letting the original transmitter or the intended receiver know that the message has been captured. The typical attack scenario involves a rogue process, such as a Trojan horse, that redirects message traffic on trusted channels or via a covert channel (i.e., a channel that bypasses the information system's reference monitor). The opponent may raise the deception to an even higher level of sophistication by implementing a man-in-the-middle attack. In such an attack, the opponent captures a message, m , modifies the captured message, yielding m' , and makes m' look as though it has not been tampered with. The opponent impersonates the original transmitter while forwarding m' to the receiver that the original transmitter had intended m to reach.

Recent advances in information theory, such as those reported in [10, 15, 22] have produced authentication-coding schemes for detecting deception in authentication channels with single or multiple usage (i.e., without changing the key after each message is sent). The authentication codes are used to derive the lower bounds on the probability that an opponent will successfully deceive the receiver via substitution or impersonation.

Tognazzini [24] has investigated constructive uses of deception for designing human-computer interfaces. He compares the art of illusion, as practiced by magicians, to the illusions created by the designers of graphical user interfaces, that is, the virtual reality that the user of the interface perceives. Some of the techniques that he explores are misdirection, attention to detail, and the manipulation of time. He concludes his essay with a discourse on the concept of a threshold of believability (on the part of the user of a graphical user interface) and the ethics of impersonation, in the form of anthropomorphism (i.e., software agents impersonating humans).

4. A FRAMEWORK FOR INTELLIGENT SOFTWARE DECOYS

In this section we characterize the components and connections of the architecture and framework in which the intelligent software decoys reside.

Components, Named Interfaces, and Reuse

We treat intelligent software decoys as objects within components, following the usage by Szyperski of the terms "component," "object," and "interface" to describe component-based software architectures [23].

Definition (Intelligent software decoy): An object with a contract for which a violation of one or more preconditions by an agent causes the object to try to both deceive the agent into concluding that its violation of the contract has been successful and assess the nature of the violation, while enforcing all postconditions and class invariants.

The connectors between components are named interfaces. There is no requirement for the name that a decoy-equipped object advertises (i.e., the binding of its name to remote object references in the remote object table) to other components to be unique. The interface of a decoy consists of an ordered list of arguments. The arguments can be either primitive types or object classes. In the latter case, the argument supports polymorphic types. Each class is composed of its own arguments and behavior. The arguments are used to access the methods of objects within a component, either through a remote procedure call (RPC) or remote method invocation (RMI), as shown in Figure 1. All calls to procedures or invocations of methods are communicated to the object via the object's interface.

A software decoy can replicate itself, using the same name for the cloned components. Mobile agents cannot distinguish whether an object is operating in its nominal or deception mode. In order for objects to be able to distinguish amongst themselves, one could implement the architecture using a single address space operating system such as Sombrero [21], or possibly a distributed operating system that supports object-request brokers, such as StratOSphere [26].

An intelligent software decoy can change the form of its contract interface at run-time. The modification of the form of a decoy's interface is supported by polymorphism; that is, the component inherits its interface from its parent class. The modification of the interface can involve changing one or more of the following: the number of arguments, the order of arguments, or the data type or class of arguments.

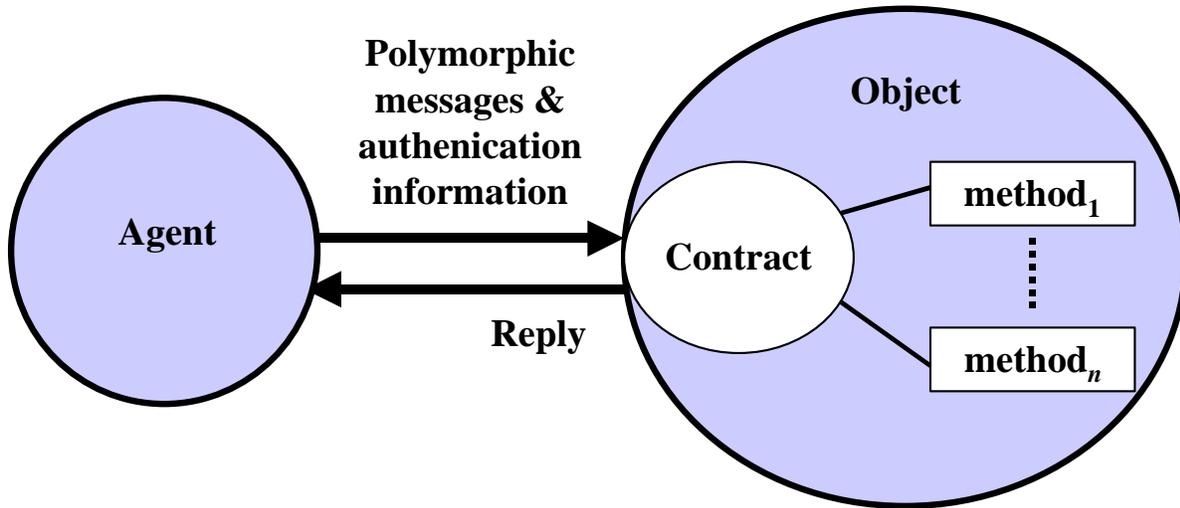


Fig. 1. Interaction between an agent and an object

The number of possible combinations of input arguments, in theory, is infinite, as is the number of class derivations. The permutation of arguments to introduce randomness into a system is not new. For example, Rothstein introduced the idea of using permuted arguments as a form of decoy in his work on message opacifiers [20].

In addition to permuting the ordering of the arguments and changing the quantity and type of arguments, randomness is injected into the interface by padding the input-argument list with one or more dummy arguments. While the total number of arguments is held constant, the position of the dummy arguments in the argument list can be changed, as can the data types of any of the arguments. The number of permutations, denoted by P , of the input-argument list for this strategy is

$$P_{m,n} = k^m \cdot (m+n)! \quad (\text{Eq. 1})$$

where m is the total number of dummy arguments, n is the total number of legitimate arguments, and k is the number of unique data types (both primitive and class-based) from which to assign a type to a dummy argument.

A mobile agent computes an argument list for an object it wants to access and passes that list along with authentication information to the interface of the target object. After the agent is authenticated to the object, the object verifies that the argument list that the agent passed to it is correct.

Definition (Correct agent-generated argument list): An agent-generated argument list is correct if and only if the number, ordering, and type of these arguments exactly match those of the target object's interface.

If the agent-generated argument list is correct, then the client where the object resides checks the access control list to determine whether the agent holds the permissions to access the method (e.g., execute the method locally or export the method for remote execution).

Protection of Object Behavior from Unauthorized Modification

Preconditions, postconditions, and class invariants govern the behavior of an intelligent software decoy. If the preconditions or postconditions fail during an interaction with a mobile agent, then the decoy either aborts the requested call, or raises an exception and unwinds to the caller. An alternative policy to raising exceptions is to retry the operation with a new set of data. The class invariants protect decoys from having their behavior modified in an unauthorized way. An agent cannot modify the behavior beyond the extent to which such modification is permitted by the parent class of the decoy.

Randomness can also be introduced into the design of the decoys by allowing the preconditions on the invocation of methods of a component to vary.

$$P_{m,n,q} = k^m \cdot (m+n+q)! \quad (\text{Eq. 2})$$

where q is the number of unique preconditions in the sample space. We do not allow for the class invariant to be permuted.

Polymorphic Types

As mentioned earlier, component interaction is based on a contract that is controlled by assertions (i.e., preconditions) as well as by a polymorphic type. The polymorphic type permits a late-binding of the message interaction. The preconditions require certain characteristics to be satisfied for each interaction to be carried forward. Preconditions are not a strong enough mechanism for all circumstances. They are particularly ineffective at guarding against mischievous action.

Polymorphic types are a little more interesting. We declare that certain parameters can have different characteristics within some accepted range of types. The types themselves may carry a set of encryption features as well as other encoding that makes them less likely to be compromised by an attacker.

An important difference in a software decoy is when the encryption error is rejected. Ordinarily, if a password fails on a routine, that routine rejects the attempt at entry. In contrast, the software decoy lets mischief proceed unnoticed by the attacker. Instead of repelling the attack, the software decoy engages it without revealing that its action is benign. This could be called the Venus flytrap model. This pleasant looking little flower lets its prey enter, enjoy the fragrance of its pollen, and encloses it for a tasty meal.

If the precondition is satisfied and the mischief is in the form of a patch, then the intelligent software decoy relies on the protection afforded by enforcement of the invariant and postcondition. Once again, if the invariant fails within the decoy, the attacker is never notified. If the postcondition fails, we apply a kind of software *jiu jitsu* within that decoy. This means we allow the attacker to believe it has been successful in overpowering the defenses while tumbling it harmlessly through the code instead of letting it forward any messages to other agents. Our approach to deception is a cross between ambiguity-increasing (A-type) deception [5], in which the decoy seeks to ensure that the “level of ambiguity always remains high enough to protect the secret of the actual operation,” and misleading (M-type) deception [5], which entails reducing ambiguity by “building up the attractiveness” of a decoy, thus causing the attacker to concentrate its resources on the decoy.

Exchange of Roles

An intelligent software decoy can operate in one of two modes: nominal or deception.

Definition (Anomalous behavior of a mobile agent): An anomalous behavior of a mobile agent is one in which a request for access to a legitimate object by a mobile agent fails the test of authentication, test for correctness of the agent-generated argument list, or the check for the necessary access permissions.

Policy 1 (Transition to deception mode): If an object detects anomalous behavior during its interaction with a mobile agent, then the object transitions from nominal into deception mode, or remains in deception mode.

Policy 2 (Transition to nominal mode): An object remains in deception mode until the object or the agent terminates its interaction with the other party.

The purpose of Policy 1 is to free up the object from processing legitimate requests so that it can take on the role of a software decoy, in particular, containing the agent and gathering information about the agent. Policy 2 provides for objects to return to operating in a nominal mode.

Observation-Inference Component

The software decoy tries to determine the nature of a mobile agent’s interaction with it in order to respond appropriately to the mobile agent. The software decoy records the messages passed to its interface by the mobile agent. The software decoy has a pattern recognition capability for distinguishing between whether an anomalous behavior exhibited by a mobile agent is due to an error in the mobile code or an attack by that agent.

Ante Chamber: The Response Component

The role of design-by-contract [16] is critical. There can be a failure of the precondition, in which case, we must have a response policy for precondition violations. In general, failure of a precondition means the agent will not do any of its work. The policy question remains for each agent: what action is appropriate when the precondition fails? A bad precondition may originate from a benign source or may represent an attempted attack. At the very least, the decoy keeps track of such failures. Failure of the invariant or postcondition intuitively represents a higher probability of an attack on the object. In particular, the failure of the postcondition should trigger the self-modifying behavior of the decoy.

The policy for responding to a mobile agent is embedded in the software decoy. The person or organization that owns the software decoy might specify the following policies:

Policy 3 (Containment by decoy): If a mobile agent, due to a software error in its code, passes an incorrect argument list to the software decoy or the real object, then the decoy should activate its containment countermeasure, rather than actively attack the mobile agent.

Policy 4 (Counterattack by decoy): If the mobile agent intended to attack the object, then the object should not under respond by treating the interaction as being due an egregious use of the object.

Policy 3 is intended to guard against an active attack on a non-malicious mobile agent; the result of such an attack could trigger a counterattack by the mobile agent or the mobile agent's coordinating agent owned by the same enterprise—a form of “friendly fire” due to an error in assessing the true nature of the violation of the contract. In contrast, Policy 4 dictates that the containment should include countermeasures that involve an active attack against not only the mobile agent, but also the applet that generated the agent, or even the process that invoked the applet.

The rules that dictate the responses of the object while it is in the deception mode are enforced within the decoy's antechamber. The antechamber serves as a waiting area for the requests initiated by an agent, that is, while the decoy assesses the nature of the contract violations and generates responses.

Testing for the failure of preconditions is not unlike acceptance testing performed by recovery blocks as part of a fault-tolerance strategy. Communicating recovery blocks (CRB), as introduced by Randell [19], provide for propagation of state recovery among recovery blocks. Likewise, the results of checking by the decoys of the preconditions for each of the chained procedure calls or method invocations (i.e., a procedure call resulting in another procedure call, or one method invocation resulting in another method invocation) are passed back to the object from which the calling procedure or method invocation originated.

5. LANGUAGE SUPPORT FOR INTELLIGENT SOFTWARE DECOYS

We believe that Eiffel is a natural choice of programming languages for implementing intelligent software decoys, at least for the purposes of initial experimentation with such decoys. In contrast to Ada95, Java, and other programming languages for which extensions have been implemented or proposed to permit the specification and use of contracts, Eiffel provides explicit support for design-by-contract in the form of built-in language constructs for specifying preconditions, postconditions, and class invariants. In addition, Eiffel's semantics provide for the checking of preconditions and postconditions at the object interface, rather than only when a method is invoked or exited.

In the example of software-controlled sensors, one could wrap the methods (e.g., `activate_sensor`) in the class named `INTELLIGENCE_GATHERING_SENSOR` with a contract as outlined in Figure 2.

```
class INTELLIGENCE_GATHERING_SENSOR
-- A sensor with an identification number and a status (on, off, unavailable)
feature
  definitions
  activate_sensor(parameter list) is
  -- routine for activating or reactivating a sensor
require
  preconditions
do
  operations
ensure
  postconditions
invariant
  invariants
rescue -- enter antechamber
  enter_antechamber(args)
end
```

Fig. 2. Outline of a contract for the class `INTELLIGENCE_GATHERING_SENSOR`

For example, a precondition could be that the sensor must already be deactivated before it can be activated, while the postcondition could be that the results of the reactivation tests performed on the sensor are not transferred to the collector in the clear (i.e., the data must be encrypted). One of the operations associated with activation of a sensor could be the generation of an encryption key. An example of an invariant is that the sensor-based data-filtering methods remain unchanged.

Moreover, Eiffel provides for inheritance of the assertions from ancestor classes by a descendant class, which is needed to preserve the integrity of the software contracts for the software decoys that are generated by a software component. However, not all Eiffel systems support the full range of the levels of run-time monitoring of assertions.

An exception will be raised when one or more of the object's assertions are violated; program control is transferred to the rescue clause. The decoy-specific Rescue clause shown in Figure 2 could be used to invoke the logic program contained in the decoy's antechamber; we are exploring the technical feasibility and efficacy of this approach. The Rescue clause, which is part of the Eiffel language, is used here to maintain the object in a safe state, in this case a decoy state, rather than terminate the current routine within INTELLIGENCE_GATHERING_SENSOR.

6. DISCUSSION

The use of intelligent software decoys within real-world systems would mark a major shift in the design of survivable systems. A decoy is not a honey pot. Instead, every object within a distributed system can be designed or wrapped with the functionality of an intelligent software decoy: there is no need to set aside a subset of all of the objects within a system as tempting bait. Further, the intelligent software decoy is founded upon the integration of formal methods (i.e., design-by-contract), deception techniques, and defensive-programming techniques. Moreover, the intelligent software decoy attempts to distinguish between egregious and malicious uses of an object's interface while also building and acting on stored and current signatures of agent-interface interaction: this is a departure from treating all violations of policy to be malicious in nature.

Intelligent software decoys can be introduced into information systems in an evolutionary manner. The owner of an information system can introduce intelligent software decoys into legacy systems without resorting to redesigning or reprogramming the existing objects in their entirety. Instead, objects in legacy systems, especially those that are necessary for the survivability of mission-critical systems (e.g., a military command-and-control system, including its infrastructure components) can be wrapped with contracts. As resources for making major modifications to the legacy systems or building new information systems become available, the owners of these systems can gradually develop reusable components containing contracts. For example, an enterprise could use Eiffel to wrap the current version of its proprietary implementation in the C programming language of the MPEG-2 compression protocol, and then completely rewrite the protocol in Eiffel with native support for contracts when the resources are available to do so.

The use of wrappers to implement contracts for commercial-off-the-shelf (COTS) software applications is also appealing because the users of such applications typically do not have access to the source code of those applications. Even if the users had access to the source code, the costs might outweigh the benefits of making changes directly to the source code because the software vendor may not support user-modified software or might change the functionality of the tool on a frequent basis. By treating an object as a black box, the user only invests in wrapping each new version of an application, rather than the potentially costly modification of the internals of the application. For example, the U.S. Department of Defense would likely benefit from the use of contract wrappers because that organization makes extensive use of COTS software applications to build information systems, some of which need to be trusted or are mission-critical.

Irrespective of whether an intelligent software decoy represents a component within the middleware or application of a system, the decoy relies on a strong foundation: the local network operating system or the distributed operating system. If the operating system cannot be trusted, then it is likely that a malicious agent will attack the weak operating system rather than directly attack the intelligent software decoys, especially in the case in which the attacker knows that the objects can operate in a deception mode. Therefore, it may be necessary to incorporate intelligent software decoys into the design of the operating system itself. For example, decoys could be introduced in an incremental manner into the Linux, Windows2000, StratOSphere, and Sombrero operating systems, starting with the objects in the kernels of the operating systems.

Furthermore, intelligent software decoys can be used to complement other approaches to protecting the components of an information system, such as the development of compilers that check for conditions that could trigger a buffer overflow. A buffer overflow is typified by the failure to specify and enforce contracts for methods that write data to buffer arrays. In essence, the calling object (or agent) is allowed to write past the bound of the target data structure, resulting in the overwriting of adjacent data structures in the same stack frame. The Return Address Defender (RAD) [1] is a patch to some existing compilers for both creating a safe area to maintain a copy of return addresses and wrapping applications with code to protect against buffer-overflow attacks. However, Chiueh and Hsu admit that RAD itself is susceptible to buffer-overflow attacks. The components of RAD and the compiler themselves could be implemented using intelligent software decoys, thus making those components more robust to buffer-overflow attacks.

7. CONCLUSION

Our approach to deception is different from that proposed in [2] in that we introduce the use of software contracts and polymorphism to create and manage intelligent software decoys. The software contracts are used to specify security policy and mediate the interaction under policy between the intelligent software decoy and agent: the postcondition and invariant

place fail-safe constraints on the behavior of the decoy, thus permitting the decoy to allow the attacking mobile agent to interact with the decoy while containing the agent. The class invariant makes it impossible for the attacker to modify the behavior of a decoy, while polymorphism permits the decoy to change its appearance, in the form of preconditions, to the attacker. Moreover, the intelligent software decoys populate the entire system space; that is, every software component can switch modes at run-time—from nominal to deception mode, and vice versa—and replicate itself. In addition, the decoys can operate in an autonomous manner, due to their autarkic nature, or they can communicate their intentions to other software components to coordinate their actions to either deceive attackers or trace the source and nature of the attack.

8. FUTURE WORK

We are in the process of refining the mathematical formulation of intelligent software decoys, in addition to extending the typing of decoys, such as distinguishing between “volunteer” and “drafted” decoys. As a first step toward demonstrating the technical feasibility of using intelligent software decoys to protect objects from the effects of egregious or malicious uses of the object or its interface, we are using the Eiffel programming language to instrument objects with the functionality of decoys.

Moreover, we are designing a commitment protocol to address the issue of transitive closure for chains of method invocations, that is, methods calling other methods. At each invocation of a method, the object determines whether the preconditions are satisfied. However, with a chain of method invocations, it is necessary to evaluate the conformance to the contract at each invocation. We are designing a data structure akin to a sandbox in order to store the intermediate results generated by each invocation within the chain of methods. As an integral part of the antechamber, the commitment protocol is used to decide whether to commit or abort the transaction; the transaction consists of the entire chain of invocations of methods. We chose to test the protocol against buffer overflows stemming from the egregious use of an object’s interface specification, in addition to the malicious use of software contracts. Buffer overflows have been used in many successful attacks on distributed systems and involve chains of method invocations. We also will address challenges in distinguishing between egregious and malicious attacks, such as false positives resulting from errors in reasoning about the temporal validity of the signatures of agent-object interaction. It is important that the decoy be able to distinguish between the types of agent-object interaction in order to minimize the likelihood of denying service to legitimate non-malicious agents.

In addition, we are exploring ways to apply intelligent software decoys in distributed databases in which lightweight objects perform queries on multidatabases. For instance, we are exploring how intelligent software decoys can be used in the DBMS-aglet framework proposed by Papastavrou, Samaras, and Pitoura [18]. For this case study, we would like to determine, for example, whether the aglets could create a successful denial-of-service attack by causing objects to replicate themselves as decoys. We are investigating this and other issues related to object persistence and garbage collection.

ACKNOWLEDGEMENTS

This research is supported by grants from the Naval Research Laboratory and the Naval Postgraduate School’s Institutionally Funded Research Program. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

We thank Bruce Allen, Terry Mayfield, John McHugh, Bertrand Meyer, Reg Meeson, Joel Pawloski, Christopher Slattery, Michael Wathen, and David Wheeler for critiquing previous versions of our conceptualization of intelligent software decoys. We also thank both the anonymous reviewers and the participants of the workshop for their comments.

REFERENCES

1. Chiueh, T.-C. and Hsu, F.-H. RAD: a compile-time solution to buffer overflow attacks. In *Proc. Twenty-first Internat. Conf. on Distributed Computing Systems*, IEEE (Phoenix, Ariz., Apr. 2001), 409-417.
2. Cohen, F. A mathematical model of simple defensive network deceptions. *Computers & Security* 19, 6 (2000), 520-528.
3. Cohen, F. A note on the role of deception in information protection. *Computers & Security* 17, 6 (1998), 483-506.
4. Cohen, F. Information system defences: a preliminary classification scheme. *Computers & Security* 16, 2 (1997), 94-114.
5. Daniel, D. C. and Herbig, K. L. Propositions on military deception. In Kaniel, D. C. and Herbig, K. L., eds., *Strategic Military Deception*. Pergamon Press, New York, 1982, pp. 3-30.
6. Dunnigan, J. F. and Nofi, A. A. *Victory and Deceit*. William Morrow & Co., New York, fifth ed., 1995.
7. Gilbert, E. N., MacWilliams, F. J., and Sloane, N. J. A. Codes which detect deceptions. *Bell Syst. Tech. Jour.* 53, 3 (1974), 405-424.
8. Goldberg, S. C. The very idea of computer self-knowledge and self-deception. *Minds and Machines* 7, 4 (Nov. 1997), 515-529.

9. Hirstein, W. Self-deception and confabulation. *Jour. Philosophy of Science* 67, 3 (Suppl. S, Sept. 2000), S418-S429.
10. Johansson, T. Lower bounds on the probability of deception in authentication with arbitration. *IEEE Trans. Inf. Theory* 40, 5 (Sept. 1994), 1573-1585.
11. Kisiel, K. W., Rosenberg, B. F., and Townsend, R. E. DAWS: Denial and deception analyst workstation. In *Proc. Internat. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, vol. II, IEEE (Tullahoma, Tenn., June 1989), 640-644.
12. Lamport, L., Shostak, R., and Pease, M. Byzantine Generals problem. *ACM Trans. Programming Languages and Systems* 4, 3 (1982), 382-401.
13. McGraw, G. and Felton, E. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley & Sons, New York, 1996.
14. McHugh, J. and Michael, J. B. Secure group management in large distributed systems: What is a group and what does it do? In *Proc. New Security Paradigms Workshop*, ACM (Caledon Hills, Ont., Sept. 1999), 80-85.
15. Meijer, A. R. Deception in authentication channels with multiple usage. In *Proc. IEEE South African Symp. on Communications and Signal Processing*, IEEE (Matieland, South Africa, Oct. 1994), 60-67.
16. Meyer, B. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, N.J., 1998.
17. Moose, P. H. A systems view of deception. In Kaniel, D. C. and Herbig, K. L., eds., *Strategic Military Deception*. Pergamon Press, New York, 1982, pp. 136-150.
18. Papastavrou, S., Samaras, G., and Pitoura, E. Mobile agents for world wide web distributed database access. *IEEE Trans. Knowledge and Data Engin.* 12, 5 (Sept.-Oct. 2000), 802-820.
19. Randell, B. System structure for software fault tolerance. *IEEE Trans. Software Engin.* 1, 2 (June 1975), 220-232.
20. Rothstein, J. Parallel processable cryptographic methods with unbounded practical security. In *Proc. Internat. Symp. on Inf. Theory*, IEEE (Ithaca, N.Y., Oct. 1977), 43.
21. Skousen, A. and Miller, D. The Sombrero single address space operating system prototype: A testbed for evaluating distributed persistent system concepts and implementation. In *Proc. Internat. Conf. on Parallel and Distributed Processing Techniques and Applications*, CSREA Press, (Las Vegas, Nevada, June 2000), 557-563.
22. Smeets, B. Bounds on the probability of deception in multiple authentication. *IEEE Trans. Inf. Theory* 40, 5 (Sept. 1994), 1586-1591.
23. Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, New York, 1999.
24. Tognazzini, B. Principles, techniques, and ethics of stage magic and their application to human interface design. In *Proc. Conf. on Human Factors in Computing Systems*, ACM (Amsterdam, Neth., Apr. 1993), 355-362.
25. Turing, A. M. Computing machinery and intelligence. *Mind* 59, 236 (Oct. 1950), 433-460.
26. Wu, D., Agrawal, D., and El Abbadi, A. StratOSphere: mobile processing of distributed objects in Java. In *Proc. Fourth Annual Internat. Conf. on Mobile Computing and Networking*, ACM (Dallas, Tex., Oct. 1998), 121-132.
27. Zlotkin, G. and Rosenschein, J. S. Mechanism design for automated negotiation, and its application to task oriented domains. *Jour. Artificial Intelligence* 86, 2 (Oct. 1996), 195-244.