NPS-CS-03-001

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Phase I Report on Intelligent Software Decoys: Technical Feasibility and Institutional Issues in the Context of Homeland Security

by

James B. Michael, Neil C. Rowe, Hy S. Rothstein,
Mikhail Auguston, Doron Drusinsky, and Richard D. Riehle

10 December 2002

Approved for public release; distribution is unlimited.

Prepared for: U.S. Department of Justice Office of Justice Programs and Office of Domestic Preparedness,
under the aegis of the Naval Postgraduate School Homeland Security Leadership Development Program

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

RADM David R. Ellison, USN
Superintendent

Richard Elster
Provost

This report was prepared for Naval Postgraduate School Homeland Security Leadership Development Program and funded by the U.S. Department of Justice Office of Justice Programs and Office for Domestic Preparedness under interagency agreement no. 2002-GT-R-057.

This report was prepared by:

James B. Michael
Associate Professor/Principal Investigator

Neil C. Rowe
Professor/Co-Principal Investigator

Hy S. Rothstein
Associate Professor/Co-Principal Investigator

Doron Drusinsky
Professor/Co-Principal Investigator

Mikhail Auguston
Associate Professor/Co-Principal Investigator

Richard D. Riehle
Visiting Professor/Investigator

Reviewed by:

Peter J. Denning
Chairman, Department of Computer Science

Released by:

D. W. Netzer
Associate Provost and
Dean of Research

REPORT DOCUMENTATION PAGE	Form approved OMB No 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** 10 December 2002	**3. REPORT TYPE AND DATES COVERED** Technical Report

4. TITLE AND SUBTITLE
Phase I Report on Intelligent Software Decoys: Technical Feasibility and Institutional Issues in the Context of Homeland Security

5. FUNDING
2002-GT-R-057

6. AUTHOR(S)
James B. Michael, Neil C. Rowe, Hy Rothstein, Mikhail Auguston, Doron Drusinsky, Richard Riehle

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Code CS
Monterey, CA 93943-5118

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-CS-03-001

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Department of Justice
Office of Justice Programs
810 Seventh St., NW
Washington, DC 20531

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
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.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words.)

The purpose of this project is to explore the technical feasibility and institutional issues associated with applying software-based deception techniques as part of Homeland defense. At present, we refer to the embodiment of software-based deception as *intelligent software decoys*, although this name may change in the next phase of our research. The key idea that we are pursuing is that software-based deception can be used to harden software assets against attack. An important novel aspect of our research is that we introduce the concept of conducting counterintelligence and intelligently employing countermeasures in cyberspace via software-based deception. The owners of computing assets may have to deploy intelligent software decoys with such capabilities in order to counter attacks conducted by technology-savvy terrorists and criminals, in addition to information warriors from rogue or enemy nation-states; conventional countermeasures will likely be ineffective against the sophisticated arsenal of cyber weapons at the disposal of such attackers, and any countermeasure will be difficult to deploy without reliable counterintelligence, particularly if the users of countermeasures intend to avoid becoming cyber war criminals.

In this report, we summarize our research and its relevance to Homeland security, and briefly discuss our plans for furthering our work under Phase II of the Naval Postgraduate School's Homeland Security Research & Technology Program. The initial results of our work indicate to us that software-based deception could play a pivotal role in protecting the U.S. critical information infrastructure and critical software applications that rely on that infrastructure.

14. SUBJECT TERMS
Deception, Homeland Security, Information Operations, Software

15. NUMBER OF PAGES
35

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Phase I Report on Intelligent Software Decoys: Technical Feasibility and Institutional Issues in the Context of Homeland Security

J. Bret Michael, Neil C. Rowe, Hy Rothstein, Mikhail Auguston,
Doron Drusinsky, and Richard Riehle

Naval Postgraduate School

Abstract

The purpose of this project is to explore the technical feasibility and institutional issues associated with applying software-based deception techniques as part of Homeland defense. At present, we refer to the embodiment of software-based deception as *intelligent software decoys*, although this name may change in the next phase of our research.[1] The key idea that we are pursuing is that software-based deception can be used to harden software assets against attack. An important novel aspect of our research is that we introduce the concept of conducting counterintelligence and intelligently employing countermeasures in cyberspace via software-based deception. The owners of computing assets may have to deploy intelligent software decoys with such capabilities in order to counter attacks conducted by technology-savvy terrorists and criminals, in addition to information warriors from rogue[2] or enemy nation-states; conventional countermeasures will likely be ineffective against the sophisticated arsenal of cyber weapons at the disposal of such attackers, and any countermeasure will be difficult to deploy without reliable counterintelligence, particularly if the users of countermeasures intend to avoid becoming cyber war criminals.

In this report, we summarize our research and its relevance to Homeland Security, and briefly discuss our plans for furthering our work under Phase II of the Naval Postgraduate School's Homeland Security Research & Technology Research Program. The initial results of our work indicate to us that software-based deception could play a pivotal role in protecting the U.S. critical information infrastructure and critical software applications that rely on that infrastructure.

Understanding of tactics of deception

On the topic of understanding the tactics of deception, we focused on understanding deceptive tactics in warfare, and prepared a paper summarizing our discoveries. We

[1] Some reviewers felt that the term "software decoy" conveyed the idea to them that the software component instrumented with deceptive behavior was a fake component, even a honey pot, rather than what we meant it to mean which is a unit of operational software.

[2] We use the term "rogue states" to refer to nation-states that either explicitly permit or do not actively combat cyber terrorism within their own borders, or do not cooperative with other nation-states to counter cyber terrorism or cyber crime.

examined much of the literature on conventional deception and the best ways to facilitate it. There is a large literature on conventional deception, and several useful frameworks exist to help analyze historical examples. Fowler and Nesbitt suggest six general principles for effective tactical deception in warfare:

- Deception should reinforce enemy expectations.
- Deception should have realistic timing and duration.
- Deception should be integrated with operations.
- Deception should be coordinated with concealment of true intentions.
- Deception realism should be tailored to needs of the setting.
- Deception should be imaginative and creative.

Dunnigan and Nofi elaborate on the above principles by introducing specific categories of deception. These categories are:

- Concealment ("hiding your forces from the enemy").
- Camouflage ("hiding your troops and movements from the enemy by artificial means").
- False and planted information (disinformation, "letting the enemy get his hands on information that will hurt him and help you").
- Lies ("when communicating with the enemy").
- Displays ("techniques to make the enemy see what isn't there").
- Ruses ("tricks, such as displays that use enemy equipment and procedures").
- Demonstrations ("making a move with your forces that implies imminent action, but is not followed through").
- Feints ("like a demonstration, but you actually make an attack").
- Insight ("deceive the opponent by outthinking him").

We used the six general principles offered by Fowler and Nesbitt to examine several historical examples. For example, one of the most famous World War II deception operations was "Operation Mincemeat." In the spring of 1943, with the campaign in North Africa coming to a successful conclusion, the Allies began to consider options for the invasion of Hitler's "Fortress Europe." Everyone agreed that the most beneficial target was Sicily. Strategically located in the Mediterranean, Sicily not only would provide a springboard for the invasion of the European mainland, but also would eliminate the Luftwaffe's presence and make the Mediterranean safe for allied shipping. However, three major obstacles faced the Allied command. First, Sicily is a mountainous island that heavily favored the defenders. Secondly, the Axis knew that the invasion of Sicily was logically the Allies next move and therefore they could augment the already staunch defenses even more. Finally, the actual invasion of Sicily would require a massive arms build up which would most certainly be detected by the Germans. In order for the invasion to be a success, the German high command would have to be deceived.

Other historical cases we examined were:

- Concealment - Taliban and al Qaeda success in evading US targeting efforts after the disintergration of the Taliban regime in the ongoing war on terror.
- Camouflage - North Vietnamese Army and the Viet Cong use of manmade hides to melt into the terrain to defy technologically superior forces.
- More sophisticated measures such as aircraft equipped with muffled engines and devices to dissipate engine heat signatures as well as paint to counter infrared detection devices.
- False and planted information - Project Forae, a MACVSOG deception operation during the war in Vietnam.
- Lies - Soviet strategic deception during the Cold War regarding their stated policy of “no first use” of nuclear weapons.
- Displays - Iraqi WMD deception.
- Ruses - The Brandenburg Commandos case.
- Demonstrations – Deception operations during Desert Storm.
- Feints - The Normandy deception.
- Insight - The 1973 Yom Kippur War.

Web-browser decoy experiments

One consequence of our study of deception was our first implementation of a software decoy (Julian, 2002). This was implemented as a dynamic Web page using Java servlet technology. Normally, the servlet responded to keyword requests for images in the DoD. But if it saw something suspicious in the keyword input, it would invoke various delaying tactics. Suspicious activity was defined to be large input in terms of either the number of characters or the number of words, and constructs suggesting source code in the language C. Delaying tactics studied included stalling for a period of time before returning a correct answer, and displaying new windows on the user's screen that appear to be new interfaces to the operating system. We showed these decoys to a number of human subjects, and they seemed to be surprised and deceived at least in part.

Architecture for decoys

Initial considerations helped us determine four high-level functionality involved in software decoy concept: intrusion detection, analysis, tracking and response. While making use of existing intrusion detection systems will provide significant support, integrating these systems with analysis, tracking and response mechanisms introduces several concerns because of the critical role of deception in the overall scenario. That is why architecture of decoys is expected to be a hybrid model combining both layered and repository architectures.

Runtime decision-making process plays a vital role in maintaining a consistent and continuous deception strategy. Therefore, architecture for decoys must be flexible enough to modify existing deception strategy when necessary. Although a flexible decision-making process allows us to manage unknown attacks to some degree, it is essential to deploy mechanisms to keep policy and rule repositories up-to-date either at runtime or off-line. This task suggests that "human-in-the-loop" component can be included in the architecture in order to manage unexpected intrusions that might occur. Another issue is that deception also introduces a dependency on policies and rules, indicating the need to protect integrity of these repositories.

From a structural point of view, software architecture for decoys represents a modular structure. Imposing high cohesion and low coupling provides sufficient level of flexibility and adaptability in order to maintain the system, especially the deception strategy. Independency among system components provides important advantages in fault tolerance, error detection and correction and survivability. Current software architecture practices allow us to accomplish desired goals by building the architecture for decoys on a combination of well-known patterns: Behavioral, Creational, and Structural as described by Gamma, et al.

Specification of intrusion detection rules and deception strategies

During the first three months of this project, we conducted experiments for testing the following two hypotheses regarding decoy specification and instrumentation: (i) that our specification language named CHAMELEON is rich enough for use in stipulating intrusion detection rules and deception strategies for defending against such attacks and (ii) these high-level formal specifications can be automatically transformed into the low-level system-call wrappers which carryout in a computationally efficient manner the deceptions against the attacker's software agents (i.e., attack programs). Our experimental apparatus consisted of a host machine running the Linux operating system, the DARPA-funded NAI Software Wrapper Toolkit, and real-world attack programs.

Our experiments yielded confirming evidence for both of the hypotheses, the implications of which are of paramount importance for Homeland Security in that the approach and technology we propose can, in theory, be used by watch officers and other types of information warriors to do the following: *specify intrusion-detection rules and deception strategies without having to know all of the low-level details* about how these rules and strategies are to be implemented and enforced by operating systems; *rapidly adapt* an information system's intrusion-detection and deception *defenses* via our automated tools for software instrumentation; and perform *real-time system-level intrusion detection and deception* (in contrast to today's post mortem analyses with relatively long response times) with an *acceptable levels of computational overhead*—typically less that six percent. In addition, we envision CHAMELEON as serving as a uniform formalism to be used throughout the homeland-security community; this would ease the burden of information warriors who need to both specify and interpret intrusion detection and response rules across heterogeneous distributed systems—in particular the United States' critical information infrastructure.

In addition to the accomplishment listed in the preceding paragraphs, we further refined CHAMELEON, creating a stable version of the language as a basis for starting our efforts to build a compiler that will translate CHAMELEON specifications into a representation that the NAI Software Wrapper Toolkit needs so that it can generate system-call wrappers; in our experiments, we had to manually perform the compiler-task. Additionally, we refined the high-level architecture for software decoys to reflect the lessons learned from using CHAMELEON and conducting the experiments. Lastly, we submitted a technical paper to a well-respected forum on information security; the paper discusses the results of the aforementioned experiments.

During 2003, we will build and test the CHAMELEON compiler. Simultaneously, we plan to develop the human-machine interface to the software-decoy specification-and-monitoring suite of tools; we refer to the interface as a *management console*, the purpose of which is to provide users with a uniform interface to the suite of tools while hiding the low-level details of how to invoke the tools. We also plan to write online documentation and provide examples of how to use the tools, and incorporate an import-export capability so that users can import data, rules, strategies, and other artifacts to and from other tools. One aspect of the suite of tools we have not addressed, but plan to do so during 2003, is that of specifying in a computational form policy and doctrine about intrusion detection and deception such that automated tools can be used by information warriors to determine under what circumstances, if any, new or existing intrusion-detection rules or deception strategies can be utilized. We are currently looking into the feasibility of using the XML-based policy-specification formalism developed under the aegis of DARPA's OASIS Project. We are also exploring the representation issues in terms of legal considerations, in particular, the law of information conflict. We submitted an article to a well-established workshop on the topic of the legal and ethical considerations of applying cyber weapons for defensive and offensive purposes; our paper discusses the legal implications of applying software decoys in an operational setting.

Integration of decoys with intrusion detection

Intrusion detection systems are software that attempts to detect events in an information system that suggest unauthorized activities such as gaining administrator-level access to the system. In FY02 we began work on interfaces from three commercial off-the-shelf intrusion-detection tools to our decoy methods. So far we have investigated two open-source programs: Snort, a network-based tool, and LIDS, a host-based tool. We established a small laboratory on a protected subnetwork to test them with particular attacks. The data from the attacks went via temporary files to an interface to temporal-logic rules.

With the suspicious-event data provided by an intrusion-detection system, our software decoys must detect and analyze a set of discrete time-sequence events for patterns of disrupting a computer system. This requires combining weaker clues as to an attack, something commercial software does not provide, and furthermore provides an additional line of defense for an information system. The most natural way to express such behavior is via a set of rules. One approach that we are investigating for such rules is to use temporal logic, a precise mechanism for reasoning about time and events. In FY02 we investigated Linear-time Temporal Logic (LTL) in particular, a formal specification language for reactive systems that extends propositional logic with four future time operators: *Until*, *Eventually*, *Always*, and *Next*, and four dual past-time operators. LTL has the property of being intuitively similar to natural language specification and capable of describing many interesting properties of reactive systems.

In particular in FY02, we investigated the applicability of DBRover, an LTLD run-time monitoring tool, for implementation of decoys. It consists of a GUI for editing temporal assertions, a simulator, and an execution engine. DBRover builds and executes temporal rules for a target program or application; in run-time, the DBRover listens for messages from the target application and evaluates corresponding temporal assertions. Monitoring is performed on-line, in tandem with the target program, and re-evaluates assertions every cycle. The DBRover uses an underlying algorithm that does not store a history trace of the data it receives; it can therefore monitor very long and potentially never ending target applications with no performance degradation over time.

Work in FY03 will address the issue, given that we are under attack, of what are the most effective ways to thwart that attack deceptively. The Rowe and Andrade paper in the Appendix exhibits the underlying technical theory that we are developing to use for this problem. Thwarting an attack is a form of counterplanning, planning with the goal of obstructing an opponent's plan. A counterplan can be built from atomic “ploys” or discrete deceptive tactics. The objective of counterplanning is to find the best set of such ploys, taking into account both the difficult of implementing those ploys and the likelihood of deceiving the attacker. But some ploys need to be done consistently with other ploys. This is a nonlinear optimization problem that can be addressed with a variety of methods, but it needs to be formulated carefully.

Papers with presentations

C. Artho, D. Drusinsky, A. Goldeberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, and W. Visser, Experiments with test case generation and runtime analysis. 2003 Abstract State Machine Conference, Sicily, Italy (invited).

J. B. Michael, M. Auguston, N. C. Rowe, and R. Riehle, Software decoys: intrusion detection and countermeasures. IEEE Information Assurance Workshop, West Point, N.Y., June 2002.

J. B Michael, On the response policy of software decoys: Conducting software-based deception in the cyber battlespace. Twenty-sixth Annual Computer Software and Applications Conference, Oxford, England, August 2000, pp. 956-962.

N. C. Rowe and S. F. Andrade, Counterplanning for multi-agent plans using stochastic means-ends analysis. IASTED Artificial Intelligence and Applications Conference, Malaga, Spain, September 2002, pp. 405-410.

N. C. Rowe, J. B. Michael, M. Auguston, and R. Riehle, Software decoys for software counterintelligence. *IAnewsletter*, 5, 1, Spring 2002, pp. 10-12.

Presentations alone

J. B. Michael, Intelligent software decoys: A glimpse at advanced research on information operations at the Naval Postgraduate School." Naval Information Operations and Warfare (NIOW) Symposium and Technology Exposition, Little Creek, Va., June 2002 (invited).

N. C. Rowe, Software decoys: Automated deception strategies for information operations. Active Network Defense Workshop, Linthicum Heights, Md., August 2002 (invited).

Student theses

D. Julian, Delaying-Type Responses for Use by Software Decoys, Master's thesis in Computer Science, Sept. 2002, co-advisors B. Michael and N. Rowe

G. Fragkos, An Event-Trace Language for Software Decoys, Master's thesis in Computer Science, Sept. 2002, co-advisors B. Michael and M. Auguston

APPENDIX A.

DRAFT: Submitted on 23 Oct. 2002 to the IFIP WG 9.6/11.7 IT Misuse and the Law Workshop on Security and Control of IT in Society (SCITS) III (Athens, Greece, May 2003).

Lawful Cyber Decoy Policy

James Bret Michael and Thomas C. Wingfield

Naval Postgraduate School, Department of Computer Science, Monterey, California USA
Aegis Research Corporation, Falls Church, Virginia USA

Abstract: Cyber decoys provide a means for automating, to a degree, counterintelligence activities and responses to cyber attacks. Like other security mechanisms for protecting information systems, it is likely that cyber decoys will in some instances be misused. In the United States, criminal law provides us with analogies for preventing or punishing improper state use of deception, and criminal and civil law give us a range of tools to use against private actors. However, in addition to states, nongovernmental entities and individuals can employ cyber decoys. In this paper we present a principled analysis of the use of cyber decoys. We explore the absolute minima in terms of customary principles for what might be considered to be acceptable use of deception.

Key words: Deception, Law, Computer security

Deception in Cyberspace

In [1], Michael *et al*. propose to use software-based deception as a means for hardening operational systems against attack. Critical units of software are wrapped with “decoying” rules, which are the cyber embodiment of both the doctrine and policy of an organization or individual for conducting counterintelligence and applying countermeasures against attackers. The rules are contained in wrappers placed around critical units of software (*e.g*., a component or method) to be protected. By critical, we mean units of software that are integral to the continued survivability of an information system and the correct enforcement of the policy embedded in the system.

When a wrapper detects a suspicious pattern of system calls by one or more computer processes, it begins to conduct counterintelligence tasks and initiates countermeasures; pattern recognition is performed at runtime. The wrappers, referred to as “decoys,” conduct counterintelligence by allowing the interaction with suspicious processes to continue, collecting information about the nature of the processes’ behavior. The wrappers respond to requests for service from the processes by applying countermeasures, with coordination of their responses provided by “decoy supervisors.” The countermeasures include actions taken to shield the wrapped software from any ill effects of the interaction, and the responses to the processes that are needed to deceive the attacker into concluding that his or her computer processes are successfully carrying out their mission. As new patterns of suspicious behavior are discovered, the database of rules for counterintelligence and countermeasure actions is updated.

A potential "homeland security" application

Homeland security within the United States encompasses, among other things, the protection of public and private cybernetic property against espionage and sabotage, especially if such a compromise would have a significant adverse effect on the national security of the United States.

Let's make the discussion of software decoys more concrete by considering how they can be used to protect a particular type of cybernetic property—a public-switched telephone network (PSTN). Within a PSTN, software units that authenticate subscribers to the network are necessary for enforcing policy against unauthorized eavesdropping on conversations. In addition, the survivability of a PSTN is contingent on the continued correct functioning of the software that implements the Signaling System 7 (SS7) protocol. Thus, by our definition, these software units are system-critical.

Software decoys can be created for both the subscriber-authentication and SS7 software units. For instance, these software units can be wrapped so that they discover patterns of system-level events that are indicative of attempts to cause exceptions to be raised in normally infrequently-called methods of these software units—such invocations of methods constitute a form of suspicious behavior. On detecting a sequence of invocations, such as one that would cause a buffer overflow, the decoys would begin gathering information about the nature of the calling processes' behavior. If a process continues to try to raise exceptions, the decoys can apply deception by, for example, providing a faked error-handling message, with the aim of making it appear to the process (and ultimately the owner of the rogue process) that the exception was raised and not handled. The goal of the decoy at this point is to maintain interaction with the process, providing the decoy with the opportunity to gather more information about the nature of the interaction. If analysis of the interaction is indicative of an attack, the decoy may be able to discern the sources and methods of the attack, using this information to make decisions about whether to applying passive (*i.e.*, strictly defensive on the attacked system) or active (*i.e.*, offensive, either just identifying the source of an attack or also directing a destructive force against the attacker) countermeasures. Likewise, the decoys may discover the interaction is non-malicious in nature and just notify the owner of the process of his or her egregious use of the software units; this addresses, to some extent, the need in information operations to correctly handle false positives.

Potential for misuse of decoys

The users of software decoys need to employ counterintelligence and countermeasures in a judicious manner, so as to prevent their misuse. For instance, software decoys, like any other software, can behave in unanticipated ways due to the presence of unknown software defects; defects can cause side effects that result in the generation of inappropriate responses. Similarly, the decoys may be poorly designed in terms of the breadth of responses, or in terms of the fidelity with which they implement the owning organization's policy. Alternatively, the decoys may not have built-in controls to prevent users or their decoys from inadvertently contravening an organization's policy on the use of countermeasures and counterintelligence; the foregoing examples exemplify the *technical misuse* of decoys.

Suppose that a public telephone company instructs the decoys used in conjunction with its SS7 software to provide deceptive responses, such as exaggerated delays, to the communication devices used by customers of competing telephone service providers, with the aim of providing those users with a degraded level of service. In the United States, injecting such delays is legal as there is no general duty on the part of nongovernmental entities to tell the truth: suboptimal performance is rarely, if ever, unlawful *per se*. In the eyes of some, the exaggerated delays represent a misuse of the technology in that the company might gain an unfair competitive advantage. We call this *intentional lawful misuse* of decoys.

Further, suppose the federal agencies within the United States employ software decoys. If the National Security Agency were to use the decoys to collect information about attackers who turn out to be U.S. citizens, this would be a violation of federal law. We refer to this as an example of *unintentional unlawful* use of decoys. One means proposed in [2] for countering this and the other types misuse is to make the decoy supervisors responsible for checking whether the rules for conducting counterintelligence and applying countermeasures are consistent with doctrine and policy.

Lawful Cyber Decoy Policy

Policies, along with doctrine, can provide guidance within an organization on how to properly use software-based deception technology. For instance, the telephone company in the preceding example could have a policy that all of its networks must require knowing intelligent waiver by the user of certain privacy rights after reasonable notice has been given to both legitimate users and intruders that software deception is in use would protect the company, absent any other egregious behavior on its part, from being held legally responsible for damage incurred by the user due to the user's interaction with the software decoys.

Criminal law already goes a long way toward giving us analogies for preventing or punishing improper state use, and criminal and civil law give us a range of tools to use against private actors. However, there are gaps in the law. For example, what if corporations start using software decoys within acceptable limits and contract out those aspects of deception that would cross the line (*i.e.*, be unlawful) while maintaining plausible deniability? (*N.B.*: Nations often contract out covert operations to civilians.)

The view of deception in society

Deception (referred to as "ruses of war") is permissible in military campaigns, and only runs afoul of the law when it rises to the level of "perfidy," the treacherous misleading of an enemy about his—or your—status under the law. However, there is a cultural bias in the United States against the use of deception by any level of government, as evidenced by the recent reluctance to institutionalize deception by quashing the effort to create the U.S. Office of Strategic Influence, whose charter was to conduct perception management across agencies, including disseminating misinformation to foreign journalists in support of the war on terrorism [3]; there is also a strong legal and cultural predisposition against using domestic U.S. journalists for active deception (vice selective withholding information, which can be enormously effective in crafting the desired conclusion), including formal guidance within the intelligence community against using them in covert operations. Hence, it is possible that proposals by the Department of Homeland Security and other government agencies, to defend against terrorist attacks on cyber property, may also fall victim to negative public sentiment.

Even discounting overwrought libertarian extremists at each end of the political spectrum, there are enough mainstream concerns about civil liberties to render a potentially intrusive program politically unpalatable. Thus, we propose that individuals and organizations apply principled analysis in assessing the legality of using software-based deception.

Principled analysis of decoy usage

Principled analysis of the use of deception involves taking into account the value of the target, the nature and immediacy of the threat, the identity of the actors, the degree to which any state is supporting them, etc. For instance, consider the principle of proportionality, as it pertains to *jus in bello*, or the law which operates between belligerents in time of war: "[t]he principle of proportionality requires the military commander to balance the collateral damage (against civilians and their property) of a planned attack against the concrete and direct military advantage expected to be gained." [4] In other words, while civilians and their property may never be targeted as such, the amount of permissible collateral damage varies with the immediate importance of the military target. This applies to digital in cyberspace as certainly as it does to kinetic warfare in realspace. The three additional customary principles of the law of armed (and information) conflict—discrimination, necessity, and chivalry—as they pertain to military use of software decoys to protect semantic webs, can be found in [2].

Let's go to the example of software decoys generating unwanted side effects due to the presence of software defects in the decoys. The principle of proportionality applies here: if the government fails to allocate adequate resources to test and validate its decoys, it would be difficult to conduct a proper proportionality analysis in the heat of information operation in a time of war. Numerous other legal problems, particularly under the principle of necessity, also arise, potentially generating legal liability up to and including the status of "war criminal" for information operators, mission planners, military commanders, and civilian approval authorities. With appropriate advance work, these potential consequences—certainly those due to advance negligence or recklessness—may be virtually eliminated.

Concluding Remarks

It is possible that software decoys can be used as an airlock between the technology and the law in that the decoys can be programmed with a wide spectrum of options for taking action. Software decoys provide for anticipatory exception handling. In other words, the decoy anticipates the types of inappropriate interaction between the calling process and the wrapped unit of software, providing in advance rules for learning about and evaluating the nature of the interaction, in addition to rules for response. One could envision developing policy that places boundaries on the extent and type of deception to be employed, but providing some degree of latitude to the user of decoys to inject creativity into deceptions so as to increase the likelihood that the deceptions will be effective. The boundaries could be used to delineate the thresholds that if breached could result in the misuse or unlawful use of decoys. That is, principled analysis can be used to meet all domestic legal criteria, and set absolute minima in terms of the four customary principles of discrimination, necessity, proportionality, and chivalry.

Lastly, the U.S. Department of Homeland Security will be responsible for coordinating the protection of both public and private cybernetic property using cyber weapons. There are gray areas in the law regarding how to coordinate counterintelligence activities and countermeasures that need to take place at the intersection of law enforcement, intelligence collection, and military activity. Principled analysis can help here too, but public policymakers will need technically and legally sophisticated advice to manage the best technological defense available within the framework of the law.

Acknowledgements

Conducted under the auspices of the Naval Postgraduate School's Homeland Security Leadership Development Program, this research is supported by the U.S. Department of Justice Office of Justice Programs and Office of Domestic Preparedness, under interagency agreement no. 2002-GT-R-057. 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.

References

[1] Michael, J. B., Auguston, M., Rowe, N. C., and Riehle, R. D. Software decoys: Intrusion detection and countermeasures. In *Proc. Workshop on Inf. Assurance*, IEEE (West Point, N.Y., June 2002), 130-138.

[2] Michael, J. B. On the response policy of software decoys: Conducting software-based deception in the cyber battlespace. In *Proc. Twenty-sixth Annual Computer Software and Applications Conf*., IEEE (Oxford, Eng., Aug. 2002), 957-962.

[3] Pentagon closed besieged strategic office, *Los Angeles Times*, 27 Feb. 2002, A6.

[4] Wingfield, T. C. *The Law of Information Conflict: National Security Law in Cyberspace*. Falls Church, Va.: Aegis Research Corp., 2000.

APPENDIX B.

DRAFT: Submitted on 16 Oct. 2002 to the 18th IFIP TC-11 Int. Inf. Security Conf. (Athens, Greece, May 2003).

An Experiment in Software Decoy Design

Intrusion Detection and Countermeasures via System Call Instrumentation

J. Bret Michael, Georgios Fragkos, and Mikhail Auguston

Naval Postgraduate School, Department of Computer Science, Monterey, California USA
Hellenic Army, Athens, GREECE
New Mexico State University, Department of Computer Science, Las Cruces, New Mexico USA

Abstract: This paper presents an implementation of integrated intrusion detection and system response based on system call instrumentation. We introduce the notion of an intelligent software decoy as a means for detection and response to patterns of suspicious behavior. A prototype of such a system has been developed using NAI Labs' Generic Software Wrapper Toolkit. We present a case study of an ftp-based intrusion.

Key words: Automatic instrumentation, Deception, Intrusion detection, Intrusion response

Introduction

Two types of strategies for defending against attacks in cyberspace are in wide use: identifying and fixing known vulnerabilities of an information system, and detecting attacks before they inflict significant damage on an information system or legitimate users of the system.

These strategies are not sufficient to ensure either the survivability or the intrusion tolerance of critical information systems, such as those comprising the information infrastructure of a nation state. These systems have to both survive and tolerate attacks perpetrated by highly trained aggressors, known as information warriors, who unlike script-kiddies, continually customize their existing arsenal of attack programs and create new ones in order to both avoid detection and achieve the maximum desired effect.

Michael *et al.* introduced in [11] a different approach to defending information systems, founded on the notion of *intelligent software decoys*, to counter the attacks of information warriors. The approach has both a protection and counterintelligence component. The decoy consists of one or more software wrappers placed around a unit of software (*e.g.*, component or method), with each wrapper consisting of a set of rules for detecting and responding to suspicious behavior. Instead of indicating to the attacker that he has been detected, the decoy keeps the attacker occupied by creating the illusion for the attacker that the attack is progressing as expected, using techniques ranging from fake error messages to redirecting the interaction with the attacking computer process to a virtual sandbox (*e.g.*, via "dazzlement" [6]). The goal is threefold: to gather information about the nature of the attack, adjust the system's defenses based on the intelligence information, and cause the attacker to experience an opportunity cost (*e.g.*, waste attack resources that could have been better applied, or expose sources and methods).

There are two basic requirements for this approach to be successful: being able to detect the attack, and responding without human intervention. Michael *et al.* in [1] propose the use of an event-based language to meet these two requirements. This language uses event patterns to define suspicious behavior and the actions to be taken when the events occur.

In this paper, we describe the next step, which was to design a prototype system and then select an exemplar attack, use our high-level language to specify the decoys to be used to counter the attack, manually translate the high-level specifications into a representation that could then be automatically converted into executable kernel modules via NAI's Generic Software Wrapper Toolkit [9], and test the decoys against the attack program.

Decoy Specification language

The high-level decoy specification language, introduced by Michael *et al.* in [1], provides for defining detection-and-response actions based on computations over event traces; it encompasses the characteristics of a cross section of the six classes of attack languages identified by Vigna *et al*. [15].

An *event* is any action that can be detected during program execution. An example of an event is a system call, such as `read`. Events can have attributes. For instance, the following statement declares that the `read` event has two attributes: `buf` (the buffer) and `nbyte` (the size of the buffer).

```
event read (buf, nbyte)
```

If we want to refer to one of these attributes we use the syntax `buf(read)`.

Two binary relations are defined for events, precedence and inclusion. An event may occur before another event or an event may be included inside another event. For example, the following axiom specifies that an event of type `open_running_processes` contains an event of type `EnumProcesses` followed by a set of one or more unordered events of type `OpenProcess`.

```
open_running_processes::(EnumProcesses {OpenProcess}+)
```

These two relations suffice to describe a program execution as a partially ordered set of events, that is, an *event trace*. The set of all events during a program's execution is contained in an event called `execute-program`.

An expression containing events and conditions on their attributes is an event pattern. The following event pattern matches any `read` system call with a buffer size larger than 1000.

```
x: read & nbyte(x) > 1000
```

In the above example the name x is associated with the specific instance of the event read.

Events have duration, a beginning, and an end. We can select from a trace only those events that match a specific pattern using the keyword detect. For example, the following statement selects a read event that has a buffer size larger than 1000 from the set of events that occur during the program execution.

```
detect x: read & nbyte(x) > 1000 from execute-program
```

A detect statement can also contain a *probe*. A probe is a Boolean expression containing event attributes, subroutine calls, or a combination of the two and is executed immediately after the previous event pattern has been successfully matched. Probes can be used to specify additional conditions for filtering events. For example, the following expression specifies that a user other than root attempts to write to the password file.

```
x: write & filename(x) == ‘/etc/passwd’
    probe (user != ‘root’)
```

An event pattern combined with an action forms a rule. When an event that matches the event pattern is detected, the action is performed. The action part of the rule is specified with the keyword do and contains C-like statements. The following rule specifies that each time a read event is detected, and the buffer contains the string “SITE EXEC”, then the value “NOOP” should be assigned to the buffer.

```
detect x: read & post (buf(x) == “SITE EXEC”)
from execute-program  do buf(x) = “NOOP”
```

A sequence of decoy rules comprises a decoy specification; the sequence determines the order in which events will be detected and responses will be generated. The complexity of the attacks and the intricacy of deception tactics make it impractical to develop decoy specifications using only a plain sequence of rules. The following statement specifies that first we detect rule_1 and then either rule_2 or rule_3.

```
rule_1 (rule_2 | rule_3)
```

In the statement shown below, we expect rule_1 first, then one or more occurrences of the sequence (rule_2 rule_3), then rule_4 might be detected, and finally rule_5 might be detected zero or more times.

```
rule_1 (rule_2 rule_3)+ rule_4? rule_5*
```

In our approach, the decoy specifications are transformed into kernel modules. These modules run in kernel space and have unrestricted access to the entire operating system. The compilation of the decoy specifications produces wrapper definitions for the Generic Software Wrapper Toolkit, which in turn produces the kernel modules.

Case study

We demonstrate how we would instrument system calls to try to detect and respond to an attack. We start with a description of a variant of a well-known attack script, and then walk through the specification of actions we would like the software decoys to take to try to both detect and respond to the attack while simultaneously deceiving the attacker about the true nature of the effects of the attack on the decoy-enabled units of software.

Attack

Certain versions of Washington University's ftp server (wu-ftpd) contain an input-validation vulnerability associated with the `SITE EXEC` command (*vid*. [17]), which if exploited, can give root privileges to a remote user. Due to the widespread use of wu-ftpd, many programs have been developed that automate the exploitation of this vulnerability. Information warriors would also create their own variants of wu-ftpd attack programs, for instance to improve the speed of execution of the attack or to make it difficult for the adversary to link the attack program back to its creator. One of these programs is autowux.c [5], which via a series of specially formatted `SITE EXEC` commands overwrites the return address on the stack and executes arbitrary commands as root. The attack itself consists of eight steps.

Logging In (Step 1)

The attacker logs anonymously into the ftp server. Anonymous ftp does not require a specific password from the user, so when asked for a password the attacker sends a special string called *shellcode* that is treated as a series of machine language instructions, which if executed by the ftp server, can spawn a shell. Commands executed through this shell will have root privileges. The two steps are to first store the shellcode somewhere in the memory of the targeted computer (the anonymous user's password is stored in the server's memory) and then try to force the target to execute the shellcode.

Checking Vulnerability and Finding Buffer Address and EIP Location (Steps 2 - 6)

Next, the command `“SITE EXEC %.f”` is sent to test whether the server is vulnerable; if the server is not vulnerable, the attack script terminates itself.

The attacker sends a series of specially formatted `“SITE EXEC”` commands to find the location in the stack where the shellcode and the execution instruction pointer (EIP) reside. When a program calls a subroutine, this address is stored in the stack. After the subroutine ends, the EIP value is fetched from the stack. By changing this value before it is fetched, the attacker can execute the instructions stored in the location pointed to by the new value.

Exploiting and Starting the Shell (Steps 7 - 8)

Based on the information collected in the previous steps, the attacker sends a `“SITE EXEC”` command that substitutes the value in the location where the EIP is stored with the address where the shellcode is stored; the shellcode will be executed spawning a shell with root privileges. If the attack is successful, the attacker interacts with the shell instead of the ftp server. To confirm success, the attacker sends the `“id”` command. This command should be executed giving back the expected result.

At this stage the attack is considered to be complete. The attack program enters an infinite loop sending the user input to the server and handling the responses.

Foiling the attack via deception

We applied a simple deception strategy for the purpose of demonstrating our approach to specification and instrumentation: make the attacker believe that the attack proceeds as expected, while simultaneously protecting the server from executing any dangerous commands. We first specify, using our high-level language, the detection and response actions to be performed by the software decoy. We then demonstrate how these specifications would be mapped from the high-level specification to the equivalent representation that the NAI Generic Software Wrapper Toolkit needs to generate the wrappers around system calls (*i.e.*, kernel modules) for the underlying operating system; we intend to implement a compiler to automate the translation process.

Since this is a remote attack, the targeted system only has access to the network traffic exchanged between the ftp server and the attacker: commands and responses. The only means by which a decoy can intervene during the attack is by intercepting and modifying this traffic. The ftp server communicates with the attacker with the help of two system calls: `read` and `write`. The wrappers can give access to these system calls and their parameters. The decoy can intercept these two calls and change the contents of the buffer passed as a parameter, substituting simulated commands and faked responses. The decoy must both substitute the commands before they reach the ftp server and the responses before they are transmitted over the network. One of the wrapper's features is the ability to intercept system calls either just before they are executed or after execution is complete, indicated by the keywords `pre` and `post`, respectively.

As mentioned above, the attack starts with the shellcode being sent as the password. We assume that each time the decoy detects the shellcode during a read operation, it needs to treat the corresponding slice of the event trace as an attack; it is unlikely that a benign user would submit a shellcode as a password. If the event slice is not detected, then the decoy does not proceed to the next step. The following is the wrapper definition code.

```
 /* STEP 1 */
op{read} && step (((char *)$iobuf) =~ m|shellCode| )
     post {wr_printf("FTP Wrapper: STEP 1.\n");
     attackStep = 1; };
```

In the second step, the attacker sends the command `"SITE EXEC %.f"` to test if the ftp server is vulnerable to the attack. When executed, this command returns two lines of response, which the decoy must emulate. The decoy must also modify the value of `$sizeret`, which is the value that `read` returns, indicating the number of characters read: this value must include an extra newline character at the end.

```
   /* STEP 2a */
op{read}&& step((((char *)$iobuf)=~ m|"^SITE EXEC %.f"| ))  post {char *
newbuf;
wr_printf("FTP Wrapper: STEP 2\n");
attackStep = 2;
/* Replace SITE EXEC command w/ a harmless command */
/* that causes the server to respond w/ two lines  */
newbuf = wr_strdup("NOOP\n\n");
delete $iobuf;  $iobuf = newbuf;
   $sizeret = 6; /* Change the return value of the */
```

```
              /* read system call to reflect the */
                 /* changes we made to the buffer  */  };
```

Once the decoy has intercepted the `read` system call, and substituted the buffer, the decoy must intercept the response from the ftp server and modify it before it reaches the attacker. The substitution must be done in such a way that the attack program receives exactly what it expects. The decoy rules we created to do this are shown in steps 2b and 2c.

```
   /* STEP 2b */
op{write} && step ((((char *)$iobuf) =~ m|"^200"|)
           && (attackStep == 2)) pre {char * newbuf;
/* Replace the error message with what the attacker expects. */
newbuf = wr_strdup("200--2\r\n");
delete $iobuf;  $iobuf = newbuf; $sizeret = 8;  };

/* STEP 2c */
op{write} && step ((((char *)$iobuf) =~ m|"^500"|)
          && (attackStep == 2)) pre { char * newbuf;
/* Replace the error message with what the attacker expects. */
newbuf = wr_strdup("200  (end of '%.f')\r\n");
   delete $iobuf; $iobuf = newbuf; $sizeret = 21; };
```

Steps 3 to 5 work in a way similar to step 2. The decoy intercepts the ftp commands before they reach the ftp server and substitute them. Likewise, the decoy intercepts the response from the ftp server and substitutes it with what the attack program expects. This way, the ftp server never executes any commands that could compromise it, and the attack program is deceived into believing that the attack proceeds as expected.

Step 6 poses a challenge, because, during this phase of the attack, the server is expected to crash several times. One way of simulating the crash is to make the ftp server close the connection. If during a `read` operation the decoy changes the value of `$sizeret` to zero, the ftp server will determine that it has reached the end of file (EOF) and close the connection.

```
   /* STEP 6a */
case op{read} && step (((char *)$iobuf) =~ m|"^SITE EXEC %"| ) post
{ char * newbuf;
if ((((char *)$iobuf) =~ m|"^SITE EXEC %3093$x"|)
      || (((char *)$iobuf) =~ m|"^SITE EXEC %3094$x"|)
         || (((char *)$iobuf) =~ m|"^SITE EXEC %3127$x"|)
      || (((char *)$iobuf) =~ m|"^SITE EXEC %3144$x"|)
      || (((char *)$iobuf) =~ m|"^SITE EXEC %3145$x"|)
      || (((char *)$iobuf) =~ m|"^SITE EXEC %3150$x"|)
      || (((char *)$iobuf) =~ m|"^SITE EXEC %3151$x"|)
      || (((char *)$iobuf)=~m|"^SITE EXEC %3152$x"|)){
      $sizeret = 0; /* EOF */
} else {
/* Replace SITE EXEC command with a harmless command */
      newbuf = wr_strdup("NOOP\n\n");
      delete $iobuf; $iobuf = newbuf; $sizeret = 6; } };
```

High-level specification

The high-level specification follows exactly the sequence of steps of the low-level wrapper specification. The most important parts of the specification are discussed here; the complete specification is given in [8].

We declare two events: `read` and `write`.

```
  event read(iobuf, sizeret)
  event write(iobuf, sizeret, nytes)
```

Next, we specify a rule for each step in the wrapper definition. Step 1 is a rule with no action part. It only has a `detect` part. When the specified event pattern is detected, the decoy proceeds to the next step waiting for the next event to be detected.

```
step_1:: detect x: read & post (iobuf(x) == shellcode)
      from execute-program
```

Step 2 contains three rules. The logic in these rules is the same as in the wrapper definition.

```
step_2a:: detect x: read
& post (iobuf(x) == "SITE EXEC %.f")
from execute-program
do {iobuf(x) = "NOOP\n\n"
    sizeret(x) = 6  }

   step_2b:: detect x: write & pre (iobuf(x) == "^200")
   from execute-program
   do {iobuf(x) = "200--2\r\n"
    sizeret(x) = 8 }

step_2c:: detect x: write & pre (iobuf(x) == "^500")
from execute-program
do {iobuf(x) = "200  (end of '%.f')\r\n"
       sizeret(x) = 21 }
```

Steps 3 through 8 are specified in a similar manner. The part of the decoy specification that does the actual work is the last one, where we specify a composite rule, named `ftp_decoy`, consisting of all the previous rules.

```
   ftp_decoy::
step_1 step_2a step_2b step_2c
(step_3_4_a step_3_4_b step_3_4_c)*
step_5 (step_5a step_5b step_5c)*
step_6_I (step_6a_I step_6b_I step_6c_I)*
step_6_7_II (step_6a_II step_6b_II step_6c_II)*
   step_8a step_8b step_8c step_8d
```

Although this wrapper deceives the attacker into believing that the attack was successful, the deception ends when the attacker tries to interact with the shell. The shell functionality is not simulated and so the attacker will discover that something went wrong and possibly suspect that the targeted ftp server utilizes a deception mechanism. There are solutions to this problem. For instance, a library could provide all of the shell functionality, and be used to maintain the deception. Alternatively, the decoy could carry on the deception by transferring the attacker to another virtual machine where everything is simulated (*e.g.*, a virtual sandbox).

We submitted the low-level specification to the Generic Software Wrapper Toolkit, which produced a decoy for use with the Linux operating system. We then ran the attack script against the decoy-enabled ftp server; the decoy handled all of the system calls, as expected.

Related work

Sekar *et al*. [12] and Vankamamidi [14] developed the high-level Auditing Specification Language (ASL), with the aim of making information systems survivable. Their goal was to make systems capable of operating and offering their services even in the presence of vulnerabilities. This is achieved by detecting attacks in real-time and isolating them before they can cause significant levels of damage.

A program's intended behavior is described in ASL as a set of assertions. Any behavior not conforming to these assertions is treated as an intrusion. A process' behavior is observed through the system calls that it makes. An ASL specification involves a series of rules. Each rule consists of an event pattern and an action. The ASL specifications are compiled into C++ classes, which are then used to generate detection engines.

Eckmann *et al*. in [7] designed the STATL language to be extensible; it can be expanded via extension modules that contain domain-specific types, variables, and events. STATL supports the State Transition Analysis Technique [16]for detecting intrusions. Attack scenarios are described as a series of states and transitions. Each transition has an action associated with it. In order to abstract away the details of the modeled attacks, only the events without which the attack would fail are used. Further abstraction is achieved by describing the actions using higher-level representations, so that actions with the same effect, but different implementations, can have the same representation. The attack scenarios are compiled into dynamically linked modules called "scenario plugins."

Ko *et al.* [9] used the Generic Software Wrappers Toolkit to integrate intrusion detection and response functions into the kernels of operating systems. They treat software wrappers as state machines that intercept system calls. Wrappers are defined using the Wrapper Definition Language (WDL). The language describes the events that are going to be intercepted and the actions that the wrapper will take when these events are detected. The WDL specifications are compiled with the help of the Wrapper Compiler (WrapC) into native code.

The Wrapper Support Subsystem (WSS) facilitates the configuration and management of the wrappers. The wrapper modules are inserted into the kernel dynamically. Once a wrapper is loaded into the kernel it can wrap any process according to activation criteria defined by the administrator. Wrappers can be layered. Additionally more than one wrapper can be associated with a program at the same time, each one implementing a different detection technique.

Finally, Templeton *et al.* in [13] argues that current intrusion detection techniques fail to detect multi-staged attacks, unknown attacks, or variations of known attacks. Their approach treats attacks as a set of capabilities. An attack is described as a set of abstract attack concepts. Their language, called JIGSAW, can be used to model the abstract concepts. Their approach can be used to discover new attacks or coordinated attacks across many systems.

Conclusion

The notion of intelligent software decoys has only begun to be explored. Nevertheless, their potential value, in protecting critical information systems, is apparent. The work presented in this paper addresses one aspect of decoy technology: the automatic instrumentation of software with detection-and-response capabilities.

New constructs and new concepts were added to the high-level decoy specification language, improving its richness and expressiveness. The case study serves as the basis for demonstrating the expressiveness of the high-level language to specify decoy actions. The results of this experiment were satisfactory, since the kernel modules respond to the attack program as specified.

The Generic Software Wrapper Toolkit proved to be a valuable tool for controlling the interaction, in the form of system calls, between the autowux.c program and the decoy-enabled ftp server. The problem is that the wrapper definitions are low level, and implementing them requires knowledge about system calls and their parameters. The proposed high-level decoy specification language reduces the complexity of the definitions, making it easier for people to write and understand these definitions. We are currently working through more case studies to identify additional language features before beginning work on building a compiler that will automatically createwrapper definitions (in NAI's Wrapper Definition Language) from our high-level specifications of decoy rules.

Acknowledgements

Conducted under the auspices of the Naval Postgraduate School's Homeland Security Leadership Development Program, this research is supported by the U.S. Department of Justice Office of Justice Programs and Office for Domestic Preparedness, under interagency agreement number 2002-GT-R-057. 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.

References

[5] Anonymous. Software program named autowux.c - wu-ftpd remote root exploit for x86/linux up to version 2.6.0, dated May 4, 2001.

[6] Cohen, F. and Koike, D. Leading attackers through attack graphs with deceptions. Unpublished manuscript, Sandia National Laboratories, Livermore, Calif., May 29, 2002.

[7] Eckmann, S. T., Vigna, G., Kemmerer, R. A. STATL: An attack language for state-based intrusion detection. *J. Computer Security* 10, 1/2 (2002), 71-104.

[8] Fragkos, G. An event-trace language for software decoys. M.S. thesis, Dept. Computer Science, Naval Postgraduate School, Sept. 2002.

[9] Ko, C., Fraser, T., Badger, L., Kilpatrick, D. Detecting and countering system intrusions using software wrappers. In *Proc. Ninth USENIX Sec. Symp.*, USENIX Assn. (Denver, Colo., Aug. 2000), 145-156.

[10] Michael, J. B., Auguston, M., Rowe, N. C., and Riehle, R. D. Software decoys: Intrusion detection and countermeasures. In *Proc. Workshop on Inf. Assurance,* IEEE (West Point, N.Y., June 2002), 130-138.

[11] Michael, J. B. and Riehle, R. D. Intelligent software decoys. In *Proc. Monterey Workshop: Eng. Automation for Software Intensive Syst. Integration*, Naval Postgraduate School (Monterey, Calif., June 2001), 178-187.

[12] Sekar, R., Bowen, T., and Segal, M. On preventing intrusions by process behavior monitoring. In *Proc. Workshop on Intrusion Detection and Network Monitoring*, USENIX Assn. (Santa Clara, Calif., April 1999), 29-40.

[13] Templeton, S.J., Levitt, K. A requires/provides model for computer attacks. In *Proc. New Security Paradigms Workshop*, ACM (Cork, Ireland, Sept. 2000), 31-38.

[14] Vankamamidi, R. ASL: A specification language for intrusion detection and network monitoring. M.S. thesis, Department of Computer Science, Iowa State University, Dec. 1998.

[15] Vigna, G., Eckmann, S. T., Kemmerer, R. A. Attack languages. In *Proc. Inf. Survivability Workshop*, IEEE (Boston, Mass., Oct. 2000).

[16] Vigna, G., Eckmann, S. T., Kemmerer, R. A. The STAT tool suite. In *Proc. DARPA Inf. Survivability Conf. and Exposition*, vol. 2, IEEE (Hilton Head, S.C., Jan. 2000), 46-55.

[17] Widespread exploitation of rpc.statd and wu-ftpd vulnerabilities. Incident note IN-2000-10, CERT Coordination Center, Carnegie Mellon University, Pittsburgh, Penn., Sept. 15, 2000.

APPENDIX C.

This paper Appeared in the IASTED Artificial Intelligence and Applications Conference, Malaga, Spain, September 2002, pp. 405-410.

COUNTERPLANNING FOR MULTI-AGENT PLANS USING STOCHASTIC MEANS-ENDS ANALYSIS

Neil C. Rowe
Modeling, Simulation, and Virtual Environments Institute, U.S. Naval Postgraduate School
Code CS/Rp, 833 Dyer Road, Monterey CA 93943 USA

Sylvio F. Andrade
Brazilian Navy
Av. Canal de Marapendi 1700/1306, Barra da Tijuca, Rio de Janeiro, RJ, CEP 22631-050 Brazil

Abstract

We have developed a hierarchical planning method for multiple agents in worlds with significant levels of uncertainty. This has resulted in expert-system tools (MEAGENT) for analysts and planners without background in artificial intelligence. MEAGENT is particularly useful in analysis of counterplanning methods intended to thwart plans in complex situations. We apply heuristics to define experiments involving many runs of carefully modified simulations, use the results to quantify the effects of various counterplanning tactics, and then produce a counterplan. We exemplify our "experimental AI" approach for the domain of firefighting on ships.

Key Words: Counterplanning, planning, intelligent agents, forecasting and prediction, firefighting

1. The Planner

Our multi-agent planner for stochastic situations MEAGENT is a Prolog system using an old idea in artificial intelligence, means-ends analysis [8] as well as more modern ideas [5]. Means-ends is a form of goal-directed behavior that hierarchically decomposes plans into precondition and postcondition subtrees. This requires formal definition of actions or “operators” by their preconditions, deletion and addition postconditions, and most importantly, recommendation conditions [12]. A plan is represented as a tree and decomposed until null subtrees, or tasks involving only preconditions already achieved, are present at all leaves. Means-ends goals can be full boolean expressions with negations and disjunctions as well as conjunctions. While other planning methods can find the best solutions to planning problems, means-ends provides a more intuitive basis to planning and a good predictor of human behavior.

Means-ends planning easily handles unexpected events: Just replan with the original goal conditions. So stochastic effects of operators are naturally accommodated. Replanning can also be made efficient by caching recommended actions for subproblems as they are discovered [9], so caching of subplans is not necessary as with some other methods [12]. Stochastic effects can be a probability of failure of an action (as of trying to extinguish a fire), modification of state (as when a fire that is out flares up again), a mistake by an agent (as when someone mistakenly turns the power back on when the fire is not confirmed to be out), or just a random variable associated with the duration of each action. Such effects can also be made situationally dependent, so that the probability of an explosion is higher while the power is on. We have used random events in many tutors for procedural skills in military-training tasks, where indeterminancy teaches students how to respond to a wide variety of crises that would be costly to simulate without a computer.

We extend stochastic means-ends analysis to real-time multi-agent paradigms by associating each action with a priority list of agents qualified to accomplish it. Agents represent different people (as on a team) and physical processes (such as fire and flooding); animate agents change states by planning whereas inanimate agents change states by difference equations. Agents plan independently to achieve their goals, assuming cooperation as necessary from other agents. They have skill levels for each task which are parameters in the stochastic process of calculating a duration and success probability for an action. Agents have resource limitations in that they cannot do more than one operator in the same time interval, and certain resources (such as a fire hose) cannot be shared. Arbitrary computations may be embedded to define the state changes of inanimate agents.

Agents can have both preassigned responsibilities (the electrician is responsible for electrical devices) as well as dynamically assigned responsibilities (like holding the fire hose). An agent can be active (doing a task), idle (if its goals are achieved), or waiting (if its goals are not achieved but it has nothing to do). Since we are primarily interested in modeling task-related teams, dynamic assignment is done in our model by an order-report paradigm. A superior gives an order to a particular idle subordinate to accomplish a particular set of goal conditions. The order "wakes up" the subordinate agent if it is idle. It then constructs its own plan to accomplish the goals, executes the plan, reports back to its superior, and the initiating order is deleted from the state. The order-report paradigm permits modeling of incomplete knowledge by agents. In some cases the subordinate agent may require the help of an assistant (as in trying to extinguish a fire using a bulky hose) which requires sub-orders to the assistant.

The simulation of actions also adjusts the states with results of concurrent actions that terminated during the time interval of the original action. It permits actions to be aborted by other actions in high-priority circumstances and when preconditions become false. For instance, if a fire reignites when a crewman is ventilating a compartment, the crewman must cease ventilation and initiate extinguishment.

MEAGENT is easy to adapt to new applications by providing definitions of new operators and agents through our tools [9]. They eliminate the need of experts to program to implement a simulation. A Prolog reasoning engine provides essential automatic backtracking during plan construction. MEAGENT is suitable for many team activities, such as office-task automation, manufacturing, routine military procedures, and sports teams.

2. An Example: Ship Firefighting

We describe an application of MEAGENT to firefighting on U.S. Navy ships, a critical skill with which all ship personnel must be familiar [11]. Since a physical experiment is risks human lives and expensive resources, agent-based simulation is essential to find bottlenecks and potential problems. (Critical-path analysis does not work well when there are significant stochastic effects or complex logical dependencies between actions, both characteristics of this and many other emergency-response tasks.) Generally the cost of a firefighting plan is assessed as the total time to complete it.

The agents in this simulation are the fire-team leader (or “scene leader”), the members of the fire team including at least an electrician, nozzleman, and hoseman, the command center monitoring alarms and giving orders to the team leader, and the fire itself. Figure 1 gives the simplest deterministic means-ends tree when no interruptions or surprises occur (although this is rare in both real firefighting and our tailored simulation) [6, 10].

Random events include casualties, availability of a medic, communication failures, equipment malfunctions, and whether the oxygen is safe when tested after the fire is out. Action durations are random variables depending on the skill level of the assigned agent, the fire size, the kind of tool used, smoke intensity, and water magnitude. For instance, the time to desmoke is a uniform random variable with mean 0.8*(Smoke/Skill) and standard deviation 0.75 of that.

The fire agent uses a stochastic epidemic model of fire growth that is applied to one-minute time steps. Its parameters are the fire type (there are four standard ones) and amount of flammable material, ignition and burnout rates, and the conditions affecting burnout (self-extinguishment) and flashover (sudden combustion of all flammable materials when the temperature gets very high). If the fire size exceeds a low threshold, an alarm sensor is signaled in the Command Center of the ship, and they order the fire team to the scene. Extinguishing is modeled by a negative ignition rate except when the wrong method is being used. Rates have random fluctuations to reflect the variety of different materials available to burn and the complexities of fire spread.

3. Plan Analysis for Counterplanning

Obstructive counterplanning is planning to interfere with or frustrate an existing plan [3]. There are two important kinds: Counterplanning in an adversarial situation, as in military defensive planning, and planning in an educational environment, as in the choice of obstacles presented by the system to the user in game-like tutors. In both cases the counterplan can be rated on the induced change to the cost of the plan, typically measured in time, physical effort, mental effort, or some combination thereof. In adversarial counterplanning the objective is to find the counterplan with largest benefit (negative effect on plan cost) minus counterplan cost, taking any uncertainty into account via decision theory. In educational counterplanning the objective is usually to provide accidents and unexpected conditions that require replanning that is neither too easy nor too hard, as well as manifesting a degree of variety. [2] endorses hierarchical planning much like means-ends as a good basis for planning in adversarial situations.

3.1 Analysis of Skills Criticality

One way to attack a multiagent plan is to interfere with the ability or readiness of agents to participate in it. But a good counterplanner should apportion their resources carefully since not all abilties are equally important. MEAGENT provides an tool to conduct experiments to guide such resource allocation.

For example, experiments assessed teams with different skill levels in firefighting [1]. 100 trials were done for 27 skill combinations (of high, medium, and low values) for a fire team of four members where the nozzlemen and the hosemen have the same skill levels. Two kinds of fire locations were used, a highly inflammable compartment and a more typical compartment. We measured performance of a team as the time elapsed between the first appearance of the fire and the recording of the completion of debriefing by the Command Center. An upper limit on simulation duration of 400 minutes was enforced for when the fire recurs numerous times. Figure 2 shows typical data, showing that the tail of the distribution is not Gaussian.

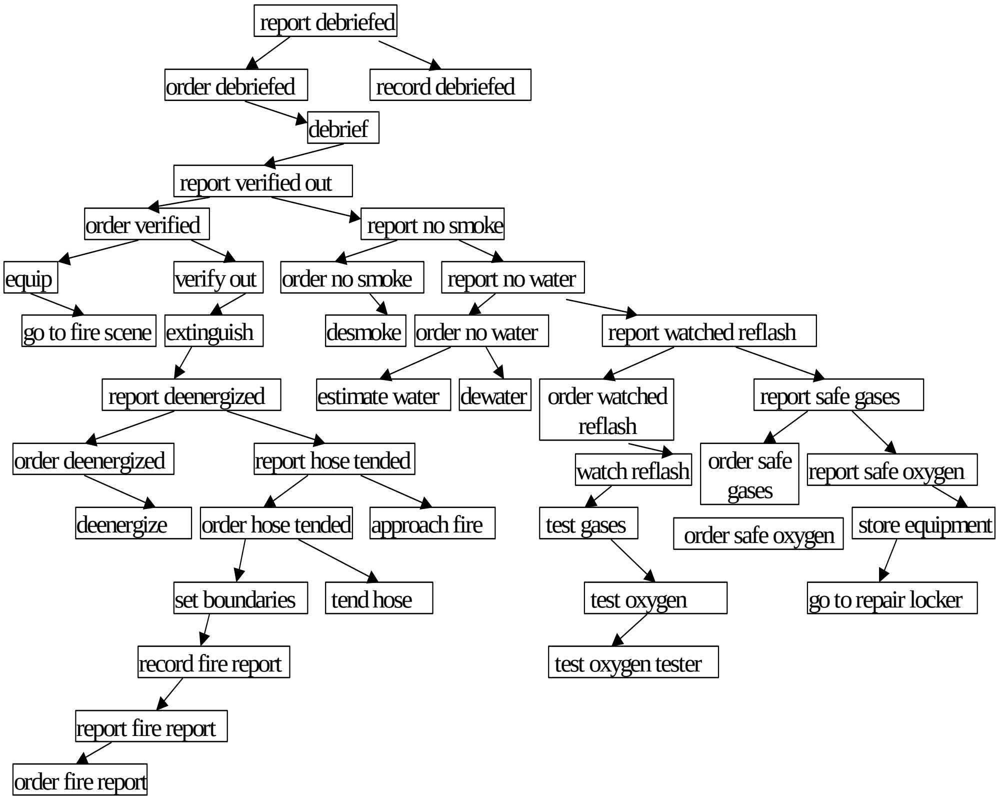

Figure 1: Plan tree for firefighting in a single compartment with no unexpected events.

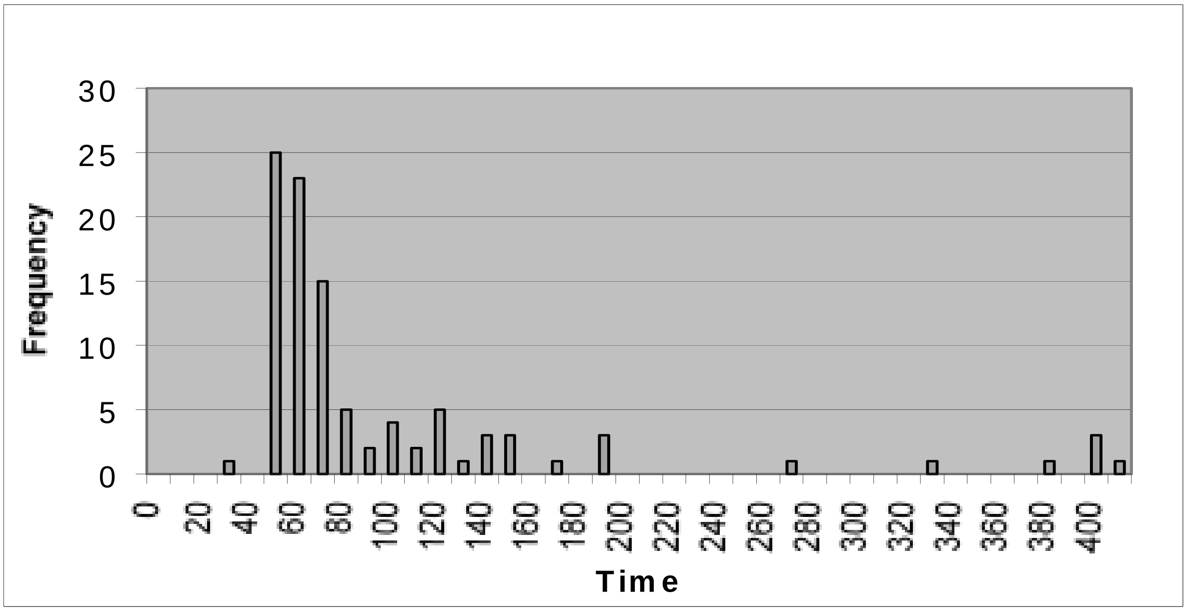

Figure 2: Histogram for total time for 100 runs of firefighting for ignition rate=0.5, burn-out rate=0.25, and all members with skill level 0.1.

Experiments permitted concrete guidelines for staffing policies for Navy fire teams:

- When the fire spreads faster, and the amount of material that burns out is very high, the percentage of intact inflammables at the end of action is going to be low, no matter what the composition of a team. Otherwise, performance is sensitive to team skill levels.
- The skills of the electrician did not much affect overall team performance; skills for the hosemen/nozzlemen were critical; and skills of the scene leader were important but not critical.

3.2 Systematic Counterplanning

But in general, counterplanning should try to change nonnumeric aspects of a plan, as fixing matters by the planner often then requires both additional thinking and time. MEAGENT provides a way to do such “systematic counterplanning”: (1) Create a standard "base" plan; (2) change states within the plan by one fact each in every possible way (i.e., investigate counterplanning "ploys"), and replan to accomplish each agent's goals after each ploy; (3) infer the degree of damage due to each ploy; (4) construct counterplans to accomplish each ploy and calculate their costs to the counterplanner; and (5) select the best cumulative counterplan that accomplishes the best set of ploys at the minimum cost. Step (2) is analogous to the idea of partial derivatives of a continuous function. If the base plan involves S states, and the average state can be modified in M ways, we must replan MS times to assess the effects of each change. A replanning from state K from the end of the base plan involves selecting an action at each of K states, so systematic counterplanning requires about $0.5MS^2$ action selections for a single linear plan. This approach is much more suitable for machine implementation than the very-general top-down approach of [3], which is mainly intended for understanding narrative accounts of counterplanning, or the general criteria for organizational maladaptivity of [4], which propose subgoals like that of discouraging intra-organization communication without suggesting specific mechanisms to do so.

To find all possible ploys, pairs of opposite facts can be inferred if the opposite fact is always added whenever an operator or random change deletes the original fact, with the additional condition that the two facts never appear together in the same state or operator condition list. For example, "safe(gases)" is an opposite of "unsafe(gases)". Otherwise, facts that are deleted by at least one operator or random change are considered facts deletable from any state in which they occur, and facts that are added by at least one operator or random change are considered facts addable to any state. In firefighting, "raging(fire)" is deletable by extinguishment, and "smokey" is addable by the fire agent for a new fire.   In addition, a good counterplanner will try if possible add facts that have not been considered in any of the planner's plans, such as fires in new locations or physical obstructions, but a good planner should be robust and have anticipated such changes, so we do not consider them here.

A problem in inferring possible changes to states is that good planning generally involves generalized reasoning with methods using variables. Generality can be preserved with deletions, but for additions and changes to states we must instantiate most variables since states must be concrete. We do this by inferring, for each variable that can appear in a state, its possible instantiations. For instance in firefighting, "ordered(F,P1,P2)" specifies that person P1 has ordered person P2 to make fact F become true; P1 and P2 must be instantiated to people in a superior-subordinate relationship, and F must be instantiated to a fact or its negation that can be accomplished. Using the firefighting specifications for instance for states for a single fire at a single location and its cleanup, we found after instantiation around 93 possible changes to a random state consisting of around 6 changes of facts to their opposites, 26 deletions of facts, and 61 additions of facts.

Since MEAGENT is intended for stochastic models, we must conduct many runs with the same starting state and goals to assess a change – at least 100 for firefighting, as discovered in the skills-effect analysis. We also need to try different base plans created by different random choices on which to make changes to states, since qualitatively different states are encountered in different plans -- in firefighting we found 100 runs were necessary on the average to see one flashover event. Since the average firefighting plan involves 41 animate-agent actions and 96 states (the fire and random occurrences creates state changes on their own), full counterplanning analysis of firefighting involves 0.5*93*96*96*100*100 = 4,285,440,000 action selections. At our typical 0.05 seconds per selection in Gnu Prolog on a two-year-old machine, this requires an impractical 2290 days, although we could sample the change space to approximate a solution. This is just for a single fire, and many more states are possible for multiple fires.

But if we proceed with this approach, after each set of runs with the same parameters we will have a distribution of costs associated with each kind of change. Cost distributions that are significantly different from one another are likely to reflect meaningful logical distinctions worth investigating. The technique of analysis of variance can identify such pairs if they are normally distributed. Otherwise, as for emergency-oriented planning like that shown in Figure 2, we may be able to partition the normal part of the distribution from a tail and analyze it separately. Doing that for firefighting by making changes to states at representative times in a representative run and then replanning, we found 69% of the changes resulted in behavior more than two standard deviations away from normal. An equal number were cost-increasing and cost-decreasing, with the latter becoming more common towards the end of the plan.

3.3 Efficient Counterplanning

Systematic counterplanning can be made much more efficient by reasoning to eliminate redundant experiments. One idea we use is to collapse analysis of identical states in different runs, building a Markov state model with state-transition probabilities. State identity can be qualitative to further reduce the possibilities. In firefighting for instance, differences in fire, smoke, and water size do not affect planning. So we found only 2,496 distinct states in 60,360 actual states in 500 runs (and 2,166 in 250 runs), which shortens our work by a factor of 24.2.

A second efficiency idea is to logically infer sets of states that require identical responses to a counterplanning ploy. Typically, a single ploy affects only a few actions in a plan. Thus the ploy will not affect planning within a period of time after the state to which it is applied – this is what makes good planning still valuable in environments with significant uncertainty. Let ploy C be a single change made to a state, either a deletion of a fact D, an addition of a fact A, or change of D to its opposite A. Let the "fix plan" be the plan to respond to ploy C by directing the agents back to a known state. Then three inference methods apply. (1) *"Forward temporal inheritance": Action X in previously completed plan P is unaffected by prior ploy C at state S if X and all actions between S and X do not require D, do not require not (A), do not add fact D, or do not delete fact A. (2) "Forward temporal ploy cancellation": Actions in a previously completed plan P have a null fix plan for ploy C if they follow or are identical to an action X after C that adds fact A or deletes fact D. (3) "Plan following": States after ploy C resulting from performing one action in its fix plan have a new fix plan which is the rest of the actions in the original fix plan.* For example, consider the ploy, in firefighting after the fire is out, of a saboteur preventing the electrician from reporting to the scene leader that the fire area is desmoked. Such a report is only necessary as a precondition to having the fire team return to the repair locker, so the "fix plan" would be to wait until that point when the missing information is realized and have the scene leader find the electrician and ask about the smoke (or alternatively check the fire area themselves, but this would undercut the electrician's responsibilities and would be an inferior fix plan). This fix plan will temporally inherit forward from any state after the fire is out up to the action of returning to the repair locker, including all states resulting from random events. The fix plan will be cancelled if a random event occurs that the electrician manages to report the desmoking. A plan-following inference would occur if some possible action before returning to the repair locker results in finding the electrician, subsequent states of which can then infer a simpler fix plan of just asking the electrician if the fire area is desmoked.

Our inference rules apply to several kinds of fix-planning strategies. The two most obvious are to immediately plan to undo the effects of C before resuming the original plan (a "lazy repair") or to plan to achieve the original goals from the new state ("radical"). Repair fix-plans will be significantly faster to build than radical fix-plans in most cases, but may result in suboptimal overall behavior because ploy C may affect reasons for the actions in the original plan, while not affecting its adequacy. Adequacy usually suffices because people are creatures of habit and prefer to persist with now-suboptimal original plans. However, if C results in the final state of the plan not satisfying the original goals after repair, more replanning must be done even with lazy repair.

The inference methods suggest a ploy taxonomy. Ploy C can be ignored after a state if temporal inheritance of an empty fix plan extends to the end of the plan. For instance in firefighting, deletion of the fact that the oxygen tester itself has been tested makes no difference after the fire is out and the oxygen is found to be safe. Ploy C can also be ignored if it reduces the cost of the plan. For instance, turning off the power at the fire scene is doing something that always needs to be done and does not hurt doing early. Depending on the application, ploy C may also be ignored if it is undoable by a single action (i.e., if the lazy repair plan is trivial). For instance, the ploy of turning the power on can be easily undone by turning it back off. All ploys not in these three categories can be thus considered nontrivial to address.

For our example of firefighting, we found 93 possible ploys, which interfered with 4.3% of the states in an average plan; 2.8% of these involved violation of conditions associated with an action applied to that state, and 1.5% were cancelled by an action. Our three kinds of inferences found an average of 1.2 additional states for which a fix plan could be used; in addition, fix plans need only be calculated to reach known states, reducing the fix-plan planning effort to 36% of the typical original planning effort at a state. This reduction in effort of 16.2% combined with the reduction due to the Markov state model reduced the effort of counterplanning to 0.67% of that of the systematic approach, reducing expected time from 2290 to a 15.3 days, making it practicable.

3.4 Using Costs in Counterplanning

Finally, after promising changes to a plan have been determined and their costs computed, we can formulate a counterplan to most cost-effectively foil the plan. We cannot generalize too much here because counterplanning requires predominantly different operators than planning (including deception), as suggested by the heuristics in [3] and illustrated in attacks on computer systems [10]. For instance, a firefighting saboteur can turn the power off, make equipment inoperable, pour gasoline around, disconnect the hose during extinguishment and plead ignorance, report incorrectly that they have checked oxygen, or relay incorrect or fake orders to subordinates. But such actions are only possible surreptitiously, thus requiring costly waiting time until circumstances are favorable and a particular probability of failure to account for. There are also potential nonlinear effects for the counterplanner to deal with, such as that one successful sabotage may make the next one harder as agents become aware that something is wrong, or that minor independent changes to a state may cause an agent to question their memory and become overly cautious. These nonlinear effects can be very significant; so since communications in crises are always problematic, communications ploys like ignored orders may be among the most successful deceptions.

However the tactics are selected, we can evaluate the counterplan costs and compare them to the benefits of sets of damaging changes to the plan. A* search is a classic way to do this tradeoff in combinatorial problems, and game theory can be used to plan for counter-responses to counterplans, and so on. Besides firefighting, we are currently investigating models of aircraft-carrier flight-deck operations and attacks by hackers on computer systems.

References

[1] S. Andrade, N. Rowe, D. Gaver, and P. Jacobs, Analysis of shipboard firefighting-team efficiency using intelligent-agent simulation, *Proc. Command and Control Research and Technology Symposium*, Monterey CA USA, June 2002.

[2] C. Applegate, C. Elsaesser, and J. Sanborn, An architecture for adversarial planning, *IEEE Transactions on Systems, Man, and Cybernetics*, 20 (1), January/February 1990, 186-194.

[3] J. Carbonell, Counterplanning: A strategy-based model of adversary planning in real-world situations, Artificial *Intelligence*, 16, 1981, 295-329.

[4] K. Carley, Inhibiting adaptation, *Proc. Command and Control Research and Technology Symposium*, Monterey CA USA, June 2002.

[5] K. Carley, J. Kjaer-Hansen, A. Newell, and A. Prietula, Plural-Soar: A prolegomenon to artificial agents and organizational behavior, in M. Masuch and G. Massimo (Eds.), *Distributed Intelligence: Applications in Human Organizations* (Amsterdam: North-Holland, 1992), 87-118.

[6] Federal Emergency Management Agency, United States Fire Administration, *Developing Effective Standard Operating Procedures for Fire & EMS Departments*, no date.

[7] A. Law and D. Kelton, *Simulation Modeling and Analysis*, *Second Edition* (New York: McGraw-Hill, 1991).

[8] A. Newell and H. Simon, GPS, A program that simulates human thought, in E. Feigenbaum and J. Feldman (Eds.), *Computers and Thought* (Berlin: R. Oldenburg KG, 1963).

[9] N. Rowe and T. Galvin, An authoring system for intelligent procedural-skill tutors,*" IEEE Intelligent Systems*, 13(3), May/June 1998, 61-69.

[10] N. Rowe, and S. Schiavo, An intelligent tutor for intrusion detection on computer systems, *Computers and Education*, 31, 1998, 395-404.

[11] U.S. Navy, *Naval Ships' Technical Manual Chapter 555 – VOLUME 1-Surface Ship Firefighting*, sections 1-9, 1-13, 1-22, 1-23, 1-24, and 9-1, 1998.

[12] G. Weiss (Ed.), *Multiagent Systems* (Cambridge, MA: MIT Press, 1999).

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
8725 John J. Kingman Rd., STE 0944
Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 013 2
Naval Postgraduate School
Monterey, CA 93943-5100

3. Research Office, Code 09 1
Naval Postgraduate School
Monterey, CA 93943-5138

4. Darrell Darnell 1
U.S. Department of Justice
Office of Justice Programs
810 Seventh St., NW
Washington, DC 20531
darnelld@ojp.usdoj.gov

4. Paul Stockton, Code 04 1
Naval Postgraduate School
Monterey, CA 93943
pstockton@nps.navy.mil

5. Ted Lewis 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943-5118
tlewis@nps.navy.mil

5. J. Bret Michael, Code CS/Mj 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943-5118
bmichael@nps.navy.mil