Mission Aware Cyber Policy Representation

Samuel N. Hamilton¹, Sarah L. Muccio², Dean Lee³, and Allen Ott⁴

¹ Distributed Infinity, Inc., shamilton@distributedinfinity.com
² Air Force Research Laboratory, sarah.muccio@rl.af.mil
³ Distributed Infinity, Inc., dlee@distributedinfinity.com
⁴ Distributed Infinity, Inc., aott@distributedinfinity.com

Abstract—One of the great challenges in cyber policy representation is that when the network supports critical services such as in a military environment, standard cyber policy often comes into conflict with mission requirements. In this paper, we present a policy representation that overcomes this challenge by disambiguating policy enforcement questions within the context of the cyber policy and mission.

Index Terms—policy representation, mission representation, cyber policy

I. INTRODUCTION

Traditionally, cyber policy representations have been composed of a series of propositional logic statements that define what actions or conditions are allowable or prohibited in the network. This allows situational awareness tools and automated policy enforcement tools to understand when there has been a violation, but leaves enforcement questions undefined. The problem is, that for mission critical networks, situations can arise where there is a conflict between what is necessary for mission success and what is defined in the cyber policy as good security practice. In this paper, we present a policy language that allows the intermingling of mission and cyber network policy such that policy enforcement actions can be derived that take cyber security concerns into account within the context of maximizing the chance for mission success.

Additionally, both local and global policy must be resilient and effective during periods of significant network condition ambivalence. Unfortunately, this is quite challenging for formal modeling policy representations traditionally utilized within the cyber domain, which generally use subsets of propositional logic to represent both network state and network policy. For certain policy requirements this approach is ideal in that it allows strict and unambiguous definitions. For many real-world situations, however, network conditions are decidedly unclear, and the appropriate security posture given this lack of clarity is equally poorly defined. In these situations, network models and policies that represent and reason with Boolean assurance are a poor match for reasoning about the complex real-world tradeoffs that come up on a daily basis on operational networks.

Furthermore, the definition mechanics for current policy representations (such as defining series of Boolean logic or first-order logic statements using predefined formats) are not intuitive for the majority of personnel operationally responsible for maintaining network security.

Our policy representation addresses these challenges. To address situation ambiguity, the policy representation is designed to interact effectively with either fuzzy or symbolic knowledge representations and reasoning systems. Additionally, statements are defined to be more intuitive to operators not intimately familiar with predicate logic declarations. In limited field-testing, it has proven intuitive and effective for non-programmers to read existing policies or generate new ones. Furthermore, our policy language maintains the representation capability of more traditional first-order logic based representations both for compatibility with enforcement vehicles and to consistency enforcement vehicles inherent in those structures.

The paper is organized as follows: in section II we present a brief overview of relevant previous work in the area. Section III describes our overall policy design principles. Section IV gives the language semantics, and we present example usage in section V. Section VI includes a description of our conclusions and experience with this work to date.

II. PREVIOUS WORK

The goal of enforcing a set of rules that is consistent with both the productivity and the security needs of a large distributed network has been of interest to the networking community since the widespread usage of computers began, but has been gaining more relevance and attention by the research community in the last dozen years as the seriousness of the security risks posed by improper network usage (either through malicious or non-malicious activities) has become better recognized in terms of the breadth and depth of the risks and threats. Policy representation and enforcement has therefore been receiving more and more attention as an important research area.

A great deal of this research has concentrated on the subset of this problem related to access control. In [1] they define an Authorization Specification Language (ASL) that represents sets of users, and predicate actions such as ?ando? over the set of users. By restricting the complexity of their predicate
set, they can detect paradoxes in the representation, and set up conflict resolution rules such as 
\( \text{?denial take precedence}\) to establish a consistent policy. This comes up only in an open policy, which differs from a closed policy in that it allows both positive and negative authorizations to be defined. The original work in this area largely ignored the time complexity issues associated with maintaining extremely large policies, but recent extensions to this work \([2, 3]\) have addressed this issue successfully using a solid base of logical operators formatted to maintain polynomial (quadratic) time complexity for the default logic.

Logic based systems for policy representation need not be limited to access control issues. Propositional logic representations are also by far the most common approach to representing general policy. In \([4]\) this was done using first order logic with SDL (Standard Deontic Logic) with an emphasis on efficient conflict detection and querying. These two functions are quite relevant, as they come up when adding a rule or checking whether an action should be allowed.

Based on this and similar work, a number of RELs (Right Expression Languages) have been developed, along with some supporting tools \([5, 6, 7]\). Tools developed in support of these generally support only a subset of first order logic operations when used for operations such as conflict resolution for time complexity reasons. As with most work in this area, there is a strong reliance in these systems on the true or false nature of each property in the system, the result of which is a certain brittleness of the system where a single policy violation can have serious repercussions not well modeled within the system. Some attempts to overcome this brittleness have been made by representing user roles and rights using fuzzy logic \([8]\). The nature of the operators defined in this work was not conducive to supporting strict policy requirements as might be required in a military network, and was recommended by the authors for addressing non-military enterprise policy requirements related to information privacy, integrity and availability.

Relatively little work in this area has been focused on representing or enforcing time related policy. This was done in \([7, 9]\) for sequencing of individual actions in order to be able to represent a TLA (temporal logic of actions). This is necessary in a dynamic access system of any size, since rights can potentially come and go during access requests, and to support temporal conditions, though more general time relationships would of course be preferable.

### III. POLICY LANGUAGE DESIGN

In order to intermingle policy and enforcement within the context of mission, we must assume sensor and actuator support to gather information and enforce decisions. In this paper, we will refer to these as agents. Additionally, our policy language should support the representation of uncertainty reflective of network conditions, as well as maintain firm consistency with unbendable policy constraints. Given this, it is important that our policy definitions are compatible with both traditional symbolic reasoning and representation systems, and fuzzy systems. The formal policy language should be generic and flexible. It should work on sets, easily supporting the concept of \( \forall \) (for all), while still allowing focused specification where necessary.

One challenge, given these requirements, is that many of the personnel responsible for maintaining network security on operational networks do not find large segments of first order logic statements an easy and intuitive method of interaction. To address this, we designed a system that allows for simple, intuitive policy statements that could be tailored significantly by the user.

For example, let us say that it was against policy for a printer to be visible externally from a local enclave. A simple way to describe this would be:

\[
\text{IF (host.isPrinter AND host.isVisibleExternally) THEN DeployAgent(PrinterPolicyViolation)}
\]

The intention of the statement is to define an action for all printers visible outside of the locally defined network. Thus, it is intrinsically defining and manipulating sets (a union of the set of hosts that are printers and the set of hosts that are visible externally from the network). We have found this type of statement to be quite intuitive to network administrators, who seem to pick it up much faster than more traditional representations.

Interestingly, personnel with a strong programming background find this approach less intuitive, since in most programming languages, an IF statement usually refers to a single instance of a single variable. To check a condition for all hosts, you would loop through each host, and check conditions. Such looping is implicit in this statement, not explicit. Fortunately, people with this background are facile with most logic-based representations, and are able to generate complex policy statements within a few minutes of training.

One significant difference between an IF/THEN policy statement approach and a more traditional one is that the action associated with a policy violation is explicitly defined within the policy. In other policy languages, it is traditional to define the ideal network state, deviation from which is considered a policy violation. The action taken after a violation is left undefined. This is dangerous in a mission critical network environment, where mission/policy conflicts need to be noticed and resolved during the policy definition process. By combining policy enforcement and policy definition as we do here, we allow disambiguation of potential conflicts during the policy definition process. In fact, it is practically required, as there is no mechanism for defining a policy for which there is no violation consequence defined.

Note that while the policy segment above demonstrates operators such as \( \forall \) and \( \cup \) in an equivalent way to previous first-order logic policy representations, it does not address our previously identified need to support fuzzy operations and representations. The need for supporting these abilities is crucial for compatibility with a number of critical reasoning and representation schemes that may be deployed in various portions of defended networks. We enable this compatibility by allowing users to locally redefine concepts and operators.
within policy scopes. This gives users extreme flexibility in terms of the knowledge representations and reasoning systems they can support. For example, if you are using a knowledge representation that uses a single fuzzy value system, you could define FALSE for your policy scope with a statement such as:

host.isVisibleExterna I (FALSE, 0.2),

which would mean that any value less than 0.2 would be considered false for the purposes of evaluating whether a host is visible externally. To complement this, the policy compiler and policy enforcement engine can accept links to fuzzy operators, allowing redefinition of AND using fuzzy parameters.

The envisioned application of this approach is that most policy users would NOT be defining their own fuzzy values and operators. Requiring all users to be able to choose effective fuzzy reasoning values is simply unrealistic. This feature is intended to support advanced users who want to tailor their environment to local systems of cyber tools and resources. And it does this in a way that allows those that define policy to do so independent of the particular tools and techniques of any individual enclave, its mission, and the tools, techniques, and personnel deployed to support that mission.

In the subsequent portions of this paper, the examples illustrating our policy language will use a two value fuzzy system as described in [10], since this is what is currently in use by the authors in the current prototype system. The grammar proposed here has been implemented and tested, and has demonstrated effective simultaneous support for network tools using symbolic and fuzzy network state representation and reasoning systems.

IV. SEMANTICS
First we show the general flow of the policy language with an example. The policy below is designed to catch anomalous activities, and to take appropriate actions. Please note that this policy example is not intended for actual use, merely to demonstrate semantics.

POLICY: findAnomalousActivities RULE: highCPUUsage
IF ( Host hostA := (Host.totalCpuUsage( > 0.8 ) & Host.OSName( ==, "Windows" )))

DEPLOYAGENT( disableMacros, hostA );
DEPLOYAGENT( investigateHost, hostA );
}
RULE: unusualLogins
IF ( Host vulnerableHosts := ( suspiciousUsers.hostsLoggedInto )

DEPLOYYPAYLOAD(shutdownHost, vulnerableHosts);
)
IF ( Host suspiciousHosts := (usersOnVacation.isLoggedOn ( ==,TRUE ))

DISPLAY( "The users $0 are currently logged-in on hosts $1", local-admin.usersOnVacation, suspiciousHosts )
DEPLOYAGENT( investigateHost, suspiciousHosts );
)

In the highCPUUsage rule, the set of Windows machines with an abnormally high cpu usage is identified; the policy responds by first disabling the macros on these machines, and then starts an investigation to identify the problems. In the unusualLogins rule, there are two types of logins that can arouse suspicion. If any of the known suspiciousUsers are logged into the system, then these hosts are immediately shut down. If any of the users are known to be on vacation, but are still actively logging into machines in the system, then a warning is displayed to the admin console, local-admin, and these machines are investigated. The notations used in this example will be explained in later sections.

A. Policy and Rule Statements
The POLICY keyword is followed by an optional name. In the last example, the policy name is findAnomalousActivities. Similarly, the RULE keyword is followed by an optional name. Every POLICY contains at least one RULE, and every RULE contains at least one IF statement.

Policies and Rules can also be given an optional number, which specifies the frequencies at which the policies and rules should be checked. For example,

POLICY: mypolicy, 0.2

specifies a policy that will be checked once every 0.2 time units. In our implementation, a time unit is defined as a second. The reason time units do not have a fixed definition at this point is that agents are intended for deployment on a wide variety of platforms, up to and including custom hardware. In some instances, standard time measurements such as seconds may be inappropriate, and standards such as clock cycles may be more useful. By providing a generic time-unit for usage throughout the knowledge representation and policy, a basis of comparison is maintained without inaccuracies introduced by rounding errors independent of platform, while comparisons of time between platform measurements remain possible as long as a time translation to any standard unit (such as a second) is provided somewhere within the overall knowledge base.

B. Variables and Types
We make a distinction between variables and state variables. A variable is an object in the language whose value is a set of objects from the system, and a variable can only be declared from the policy language. A state variable is an attribute of the system, and the value of a state variable can be of many different types, such as string, int, real, boolean, set, etc.; furthermore, state variables are defined independently of the policy language.

The value type of any variable is the type of value held by the variable. A variable is an object whose value type is a set. There is also a type associated with each variable in the language. Furthermore, each variable type has its own name-space.

On the other hand, a state variable is an attribute. For example, OSName is a state variable. The values of OSName are strings, since OS names can be described by strings such
as “Windows” or “Linux”. One can also imagine a state variable whose value type is a set, boolean, int, etc. Suppose that usersLoggedOn is a state variable, then its value may be the set \{ dlee, amoir \}, where dlee and amoir are users in the system.

State variables are accessed through variables in the language. Suppose there are three hosts in the system: dii-dlee, dii-amoir, and dii-mngov, then the value of OSName on dii-dlee is stated as dii-dlee.OSName in the policy language. Note that the value type of dii-dlee.OSName is string. Let us suppose further that dii-dlee and dii-amoir belong to a set setA := \{ dii-dlee, dii-amoir \}, then we can refer to the OSName attribute of dii-dlee and dii-amoir by setA.OSName; in this case, the value type of setA.OSName is a set of strings. Finally, Host.OSName is the OSName attribute of all hosts in the system.

In our current instantiation, a state variable stateVar has a confidence and plausibility value associated with it. This is defined by the two value fuzzy system referred to earlier. The system chosen can be redefined by linking alternative fuzzy value systems and operators to the compiler and policy enforcement engine. While not the focus of the material presented here, we present a brief description of the two value fuzzy system in use in order to make our examples easier to understand.

A confidence value ranges from 0 to 1, is the certainty on stateVar having a particular value v, stateVar == v. Furthermore, as the confidence value approaches 1, it says that we are confident that stateVar == v is true. When the confidence value approaches 1, we are more confident that stateVar == v is true. When the confidence value is near 0, then we are not sure whether stateVar == v is true or not.

The plausibility value, which ranges from 0 to 1, is the judgment that one has on stateVar == v and its confidence value. More specifically, if one determines that a confidence value c when stateVar == v is highly accurate, then the plausibility value is near 1, whereas if stateVar == v with confidence value c is highly inaccurate, then the plausibility value is near 0.

C. Condition Statements

Condition statements are denoted by parentheses (), which describe a set of objects of a particular type, with the conditions specified inside the parentheses. A condition can either be a simple condition, or a complex condition. A simple condition is the specification of comparison operations imposed on a state variable. A complex condition is made up of one or more simple conditions.

Since there is a confidence and plausibility value associated with each state variable, the full description of a simple condition is:

stateVariable( comparison operator, value, comparison operator, confidence value, comparison operator, plausibility value)

The possible comparison operators are >, <, ==, <=, >=, and !=. An example is Host.isServer( ==, TRUE, <, 0.3, >, 0.8), which is a set of hosts with attribute isServer value equal to TRUE, confidence value less than 0.3, and plausibility greater than 0.8. If only one comparison operator and one value are specified, then it is assumed that the plausibility value comparison is omitted. For example, Host.isServer( ==, TRUE, >, 0.8) returns a set of servers with confidence value greater than 0.8. The plausibility values on these hosts are not taken into account. Furthermore, Host.isServer( ==, TRUE) returns a set of servers in the system where their confidence and plausibility values are not considered.

We define an empty condition () to resolve to TRUE. Therefore, when a simple condition is stated without any arguments, the value of the state variable is returned. Suppose usersCurrentlyLoggedIn is a state variable that is a set of users currently logged into a particular host, then the simple condition hostA.usersCurrentlyLoggedIn returns a set of users who are logged in to hostA.

A complex condition is made up of one or more simple conditions joined via logical operators, possibly combined with other complex conditions. The logical operators are | and &. For example,

(Host.isServer( ==, TRUE ) & Host.isBroadcasting( ==, FALSE ) )
| (Host.isClient( ==, TRUE, >, 0.8 ) ) is a complex condition, where Host.isServer( ==, TRUE), Host.isBroadcasting( ==, FALSE), and Host.isClient( ==, TRUE, >, 0.8) are simple conditions and Host.isServer( ==, TRUE) & Host.isBroadcasting( ==, FALSE) is a complex condition.

Since we are dealing with sets, we can think of the & and | operations as intersection and union operations, respectively. Furthermore, the logical operations are legal only if the operands are of the same type. For example,

Host hostA := Host.isServer( ==, TRUE ); Host hostB := \{ dii-moir \}; Host hostC := ( hostA & hostB );

are valid statements since hostA and hostB are sets of Host type.

D. Variable Declaration

In general, variables are declared by specifying condition statements, or by defining a set containing objects from the system. Furthermore, the type of these objects must match the type of the variable.

An example of a variable declaration with condition statement is

Host hostA := (Host.isServer( ==, TRUE ) & Host.isBroadcasting( ==, TRUE ));

where hostA is the variable name, Host is the type of hostA, := designates assignment, and the parentheses () denote that a set of hosts will be produced by the conditions defined inside the parentheses. In this case, hostA is the set of hosts that have attribute values isServer equals TRUE and isBroadcasting equals TRUE. Host is a type defined in the system, therefore must be checked at compile time to make sure that the type is valid.

Variables can also be declared via explicit use of named objects in the system. For example,
Host hostB := { dii-dlee, dii-amoir, dii-mngov };

then hostB is the variable name, Host its type, "=" designates assignment, which when followed by curly brackets {} denotes that everything included inside is a reference to names used in the system. In this example, hostB is a set containing hosts dii-dlee, dii-amoir, and dii-mngov. The ANY keyword can be used in variable declarations where explicit object names are expected. For example,

Host hostC := { ANY };

means that hostC is a set containing all hosts in the system.

E. IF Statements

The IF statements start with the IF keyword, followed by a condition statement, which is followed by curly brackets {} that enclose all actions that are to be executed when the condition statement evaluates to a non-empty set. Since all condition statements produce a set, all IF statements are associated with a set. In order to use this set, we must explicitly associate it with a variable name to avoid ambiguity. The following example illustrates this point:

IF ( Host.hostA := ( Host.isServer( ==, TRUE)) ) { DEPLOYPAYLOAD( disableMacros, hostA ); };

Here hostA is explicitly declared in the condition statement. Nested IF statements are also supported.

F. Variable Scopes

In general, a POLICY keyword opens a scope, and the scope closes when the parser encounters the next POLICY keyword, or when the end of file is reached. A RULE keyword opens a scope and the scope closes when it encounters a RULE keyword or a POLICY keyword, or when the end of file is reached. Scoping rules for IF statements are similar to those in Java or C.

G. Variable Evaluation

Suppose we have a state variable of type Host

Host hostA := { ANY };
hostA.OSName( ==, "Windows" );

The evaluation of hostA.OSName( ==, "Windows" ) results in a set of Windows hostnames. The original value of hostA remains unchanged after the evaluation. To capture the result of this evaluation, we must declare another variable of the same type, such as

Host windowsHosts := hostA.OSName( ==, "Windows" );

H. Action Statements

Currently there are four types of action statements: variable declarations, assignments of values to state variables, payload deployments, and display statements. Variable declarations were discussed in a previous section. An example of state variable assignment is the following:

hostA.isServer := ( TRUE, 0.2, 0.3 );

Here isServer is assigned a value of TRUE with confidence value 0.2 and plausibility 0.3. Technically, a state variable is a -tuple of values, hence the strange looking assignment statement. The confidence and plausibility arguments are optional.

A payload is a piece of code that is deployed from one host, usually the command center, to another host on the network, and is executed by that host. Agent deployment is specified via the DEPLOYAGENT() action in the policy language. Host hostB := { dii-dlee } ; DEPLOYAGENT ( shutdownHost, hostB );

In this example, the shutdownHost payload is deployed to dii-dlee. Note that the arguments to DEPLOYPAYLOAD() depend on the actual payload being deployed, and this will be checked at compile time. The DISPLAY statement displays messages on a target host, and has the following format:

DISPLAY( DisplayString, host, optArgs ) where host is the name of the machine on which the message will be displayed, and optArgs is a list of n variables, indexed from 0 to n − 1. If in the DisplayString there is a S symbol followed by a non-negative number i, then the string value of the ith variable in optArgs is substituted into the DisplayString.

V. USE CASE EXAMPLE

In this section we present a short example policy, and demonstrate how the language accommodates potential interactions with mission elements. We start with a short list of policy statements for our target network:

1. Port XYZ should be closed on all computers.
2. No data from database servers should be sent to unauthorized or unverified sources.
3. No malware or bots should be present on any computer
4. Security should be informed of any major network changes or threats, including firewall rule changes or malware.

As well as a critical mission condition:

5. The database servers must be up, running, and connected to outside enclaves at all times.

This policy example is not meant to be comprehensive, but simple illustrative of how the policy language can be used. The policy would be defined as follows:

POLICY: KeepPortXYZclosed
Port PortOpen:=Port.isopen(==, TRUE);
RULE: ActivityFromPortXYZ
IF( Port PortXYZ:=PortOpen(==, "portxyz") ) { DEPLOYAGENT( closeport, "portxyz" ); };

POLICY: KeepDataBaseSecure
User AuthorizedUser:=User.ispermitted(==, TRUE);
RULE: DataRequested
IF( User RequestingUser:= AuthorizedUser.isrequesting(==, FALSE)) { DEPLOYAGENT ( blockuser, RequestingUser );

1192
DEPLOYAGENT ( sendreport, Security);

POLICY: NoMalwareAllowed
RULE: DetectBot
IF(Host UnknownUser := Host containbot(==, TRUE)) {
DEPLOYAGENT ( diagnoseBotandRemove);
DEPLOYAGENT ( sendreport, Security);
}

As can be seen from this example, the resulting policy definition are not simply a one to one mapping of statements, because the IF THEN format requires the intermingling of consequences with policy violation conditions. Thus, when cyber policy (such as no malware) and mission elements (such as database must always be up) conflict, appropriate de-conflicted enforcement actions must be embedded. Not only does this encourage user de-confliction during definition, but it makes it much clearer to security personnel examining the policy what will happen in practice should conflicts arise. Also note that compatibility with advanced situational awareness or policy enforcement technologies is maintained by allowing the invocation of agents containing them. This language is not meant to replace such technologies, but to enhance them by providing an environment they can be safely and appropriately invoked in given the necessary conditions.

VI. CONCLUSIONS

The policy language described herein captures conditions defining policy within the context of mission. The primitives in the language are sets of objects that provide an intuitive representation of the state of the system at any given moment. As a result, policies can be specified via manipulations on sets using well-defined set operations, such as intersection and union. In addition, policy is not simply a set of declarations of what the ideal network condition is (such as in the case of almost all other policy representations), but incorporates an explicit definition of what actions should be taken in case of a policy violation. This action could include some combination of automated policy enforcement and including human decision makers in the process as defined by local network policy definitions. By entwining policy definition and policy enforcement, we enable mission relevant decisions to be taken into account during the policy definition. This provides a mission sensitive policy definition platform, which is critical for networks used to support delicate operations where blindly following cyber policy may result in mission disruption with potential for huge financial losses or in some cases even loss of life.

Through its use of sets as primitives, the language is highly extensible. It is compatible with both symbolic and fuzzy logic systems, providing users the option of defining their own fuzzy value systems and operators. This allows different portions of a large network (such as the GIG) to enforce policy as dictated by local needs and capabilities, and allows those that define high-level policy to do so without the burdensome requirement of understanding the language of each defensive toolset that may be deployed in individual enclaves.

The result has numerous additional benefits. Appropriately designed and utilized policy can automate many security tasks, freeing the human element to concentrate on more pressing issues of network security. Additionally it provides the human element a powerful trusted tool to enforce security and protect their assets. While use of this language has proven highly efficient and intuitive to date, it is worth emphasizing that deployment has been limited to small, distributed networks, using relatively simple policies. The authors anticipate that as we receive feedback from deployments on larger networks with more complex policy needs the language will need to be adjusted and expanded based on user feedback. We believe, however, that the core definitions presented here, in particular the intuitive statement definitions, and our incorporation of the ability to interact with a heterogeneous set of symbolic and fuzzy reasoning systems, provides a solid basis from which to support future expansions of this approach to policy representation.

ACKNOWLEDGMENT

This research was supported in part by the U.S. Air Force Research Laboratories/ Information Directorate. All opinions expressed in this paper are the authors and do not reflect the official policy or position of the Air Force Research Laboratory, the United States Air Force, Department of Defense, or the United States Government.

REFERENCES