The Zodiac Policy Subsystem: a Policy-Based Management System for a High-Security MANET

Yuu-Heng Cheng, Scott Alexander, Alex Poylisher
Telcordia Technologies
{yhcheng,salex,sher}@research.telcordia.com

Mariana Raykova, Steven M. Bellovin
Columbia University
{mariana,smb}@cs.columbia.edu

***Abstract*—Zodiac (Zero Outage Dynamic Intrinsically Assurable Communities) is an implementation of a high-security MANET, resistant to multiple types of attacks, including Byzantine faults. The Zodiac architecture poses a set of unique system security, performance, and usability requirements to its policy-based management system (PBMS). In this paper, we identify theses requirements, and present the design and implementation of the Zodiac Policy Subsystem (ZPS), which allows administrators to securely specify, distribute and evaluate network control and system security policies to customize ZODIAC behaviors. ZPS uses the Keynote language for specifying all authorization policies. We lso present a simple extension of the Keynote language to support obligation policies.**

***Index Terms*—policy-based management; MANET; security;**

I. Introduction

With the development of increasingly complex systems, policies are widely used to allow users a level of customization and automation of a system without the need for rebuilding or restarting. Existing policy-based management systems (PBMS) define customization for access control (e.g., [1], [2]), obligated behavior ([3], [4]), flow control, but do not address architecture security.

In this paper, we describe the Zodiac Policy Subsystem (ZPS) which supports all of the above policy types. Additionally, ZPS must operate within an environment designed to heighten security in networks including MANETs. Most existing policy systems are defined with centralized policy control (e.g., [5]), where security is provided by the network authentication process, though some may utilize authentication information as part of the policy condition.

This material is based upon work supported by the Defense Advanced Research Projects Agency and Space and Naval Warfare Center, San Diego, under Contract No. N66001-08-C-2012.

In Section II, we describe the architecture of Zodiac, and in Section III the requirements of a policy-based management for Zodiac. In Sections IV and V, we describe the architecture and ongoing implementation of the Zodiac Policy Subsystem (ZPS). Section VI summarizes the problems addressed by the current design and indicates the directions of future work.

The main contributions of this paper are:

- identification of the security, performance and usability requirements of PBMS for high-security MANETs;
- description of major architectural and design solutions to meet most requirements; and
- description of selected aspects of the implementation.

II. The Zodiac High-security MANET

The DARPA Intrinsically Assurable MANET (IAMANET) program [6] aims to develop a "clean-slate" approach to mobile ad hoc networking emphasizing security. The approach must support network information integrity, availability, reliability, confidentiality, and safety. The network should enforce authentication and authorization of all actions with deny by default, resistance to Byzantine faults and insider threats, and define a selected set of functionality implemented in trusted hardware.

Zodiac (Zero Outage Dynamic Intrinsically Assurable Communities) [7] is our solution that addresses the IAMANET requirements. Zodiac is based on a novel security and communication building block, the Dynamic Community of Interest (DCoI). A DCoI is a dynamic group of networked nodes running an instance of a distributed application or a supporting service.

DCoIs are implemented as virtual machine containers within a node. This design limits the effect of a successful attack within one DCoI as resources are allocated

per container and processes have no direct access to the processes or files and data outside of their container.

The DCoI concept is illustrated in Figure 1, where nodes 1 and 2 belong to DCoI A, nodes 2 and 3 to DCoI B, and nodes 1, 2 and 3 belong to DCoI C. A container defines a security boundary around the node resources for a single DCoI. *Infrastructure* is the device-level resource controller for the actual CPUs, memory and lower layers of the network stack (MAC and below) of the node and containers. A single application and its supporting Zodiac subsystems, such as Group and Cryptographic Services (GCS), Routing, Naming Service, etc., run within a DCoI container. The rationale for DCoIs is described in detail in [7] and is outside of the scope of this paper.

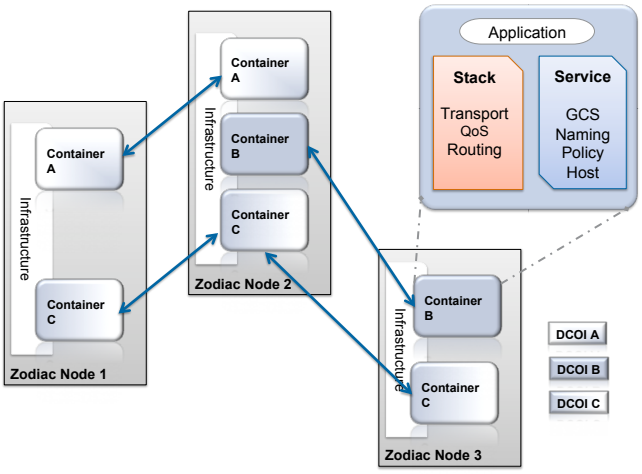

Fig. 1: The Zodiac Architecture

The communication between Zodiac nodes within a DCoI is protected by its encryption, authentication and authorization mechanisms. A node ignores communications which it is not a member of. The encryption and membership is managed by the GCS in each DCoI.

III. Zodiac Policy Requirements

The behavior of the Zodiac stack and services must be automatically customizable at run-time as manual reconfiguration is impractical in the target operating conditions. While there are certain "rules" that must be statically built into the stack and services (e.g., all traffic must be encrypted), other aspects (e.g., the list of potential DCoI members) are specific to the deployed system and usage of a particular DCoI. The need to provide customization to a high-security MANET requirement lead to a unique set of requirements for the Zodiac Policy Subsystem (ZPS):

- low CPU, memory and bandwidth use,
- support of the dynamic nature of DCoIs,
- minimal dependence of a node on another node for policy enforcement,
- operations are denied by default,
- policy author and recipient must be authenticated,
- policy integrity must be ensured in storage and distribution,
- the operation of the policy-based management system must not breach exfiltration constraints, and
- access to resources used both by containers and within a container must be policy-managed.

Both computational (CPU, memory) and communication resources in Zodiac can be very limited. Thus, ZPS's processing footprint must be small, and policies themselves must be encodable in concise form.

The ZPS user interface (UI) needs to allow administrators to create and update policies under the stress. Apart from UI design, this necessitates a highly usable policy language.

In addressing Byzantine attacks, it is very undesirable to allow a management system on a compromised node to execute operations on a remote node. Each node must have the authority and responsibility to enforce policy to protect itself and the network.

In a permit-all, explicit-deny authorization scheme, only the the known operations can be denied, so the set of permitted operations is unknown, which opens attack possibilities. Conversely, in a deny-all, explicit-permit scheme one can permit the exact minimal set of operations necessary. Additionally, unanticipated actions are denied, thus avoiding the issue that unanalyzed actions may result in security holes in the system. This is basic rationale for a deny-all scheme in a secure environment.

It is important to authenticate the author because only an explicit set of users are trusted to create policies. All the Zodiac nodes need to have the same perspective of which set of users are considered as trusted.

In policy distribution, individual policies must not be corrupted and be up-to-date. The set of policies enforced in all containers of the same DCoI must be identical and self-consistent. This comprises the policy integrity requirements.

The last requirement is to minimize the opportunity for exfiltration between DCoIs, so policy distribution should only occur within DCoIs.

ZPS also relies upon other Zodiac subsystems for its own operation. For example, policy distribution is conducted using the Zodiac stack and policy signing and

verification relies on GCS to provide the keys.

IV. ZPS Design

We have designed ZPS to address the requirements above. In this section, we describe the design decisions in detail, including the types of policies supported, architecture, components, and mechanisms for policy integrity protection and distribution.

A. Types of policies

ZPS supports both authorization and obligation policies [3]. Because of the default deny-all authorization requirement, only positive authorization is supported. Policies are further categorized into:

- *DCoI policies* which affect only a single DCoI, e.g., the membership list for the DCoI.
- *Shared resource policies* which affect the use of shared resources within a node required for a DCoI, e.g., bandwidth allocations among DCoIs for QoS assurance. [1]
- *Node policies* which determine the behavior of an entire node, e.g. the maximum number of DCoIs that can be instantiated on a given node.

The category is predetermined and orthogonal to the authorization/obligation categorization. In addition to the information assurance benefits (e.g., policy distribution), the categorization helps us determine the conflict domains of the different policies.

B. ZPS Architecture

ZPS is a fully decentralized policy system. Each ZPS instance evaluates its own set of policies and makes decisions to control the managed components, within a container. The only communication over the network is the policy contents. An alternative solution is to dispatch policy decisions from one or several locations over the network. For Zodiac, a fully decentralized solution is preferred as:

- policy contents do not change nearly as frequently as policy decision results,
- low network latency is far less critical for policy contents as it is for policy decisions, and
- it is easier to support need-to-know policy evaluation and Byzantine fault tolerance.

Policy content is, of course, sensitive on its own. Knowing the access control policies, for example, may reveal the usage of a DCoI. Because of this sensitivity, we not only protect policies in transit, but also ensure that even if our protections fail, the set of policies to which an attacker has access is minimized.

As described above, each DCoI is instantiated inside of a container. ZPS is duplicated in each container as in Figure 2. This design reduces the ability of an attacker both to inspect policies from other containers and to affect the behavior of containers other than the one to which she has gained control.

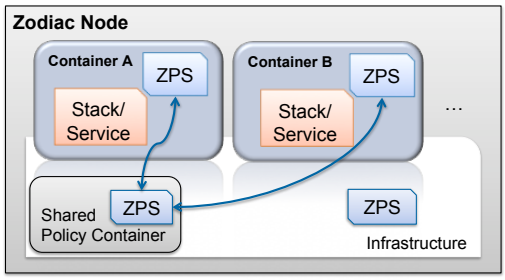

Fig. 2: Components within a Zodiac node

The *Infrastructure* on a Zodiac node serves as the operating system that controls access to the network and manages creation/deletion of DCoIs. Communications between ZPS instances are constrained to occur along the paths created and enforced by Infrastructure. This helps reduce the opportunity for unauthorized access to policy information.

The ZPS instances that reside in a container, the shared policy container, and Infrastructure evaluates policies for the DCOI, shared DCOI resources, and the node respectively.

C. ZPS Components

The basic functional components of the ZPS are the same as in the proposed IETF policy-based management architecture [5], which includes:

- A *Policy Decision Point* (PDP): evaluates the stored policies when triggered by a request (passive) or an event (proactive). The decision made is then passed to the PEP.
- A *Policy Enforcement Point* (PEP): enforces policy decisions. Usually the PEP resides in the subsystem that the policy system manages.
- *Policy Repository* (PR): stores the policies used in the policy system.

Figure 3 depicts the information flow for the basic functional components. The feedback loop allows the PDP to retrieve information from the managed system

[1]Due to the complexity of security concerns on information exfiltration, the current implementation does not support conflict detection for policies that control shared resources.

when evaluating policies and automate behaviors of the managed system; though not part of the IETF-proposed architecture, it is commonly used, particularly with obligation policies.

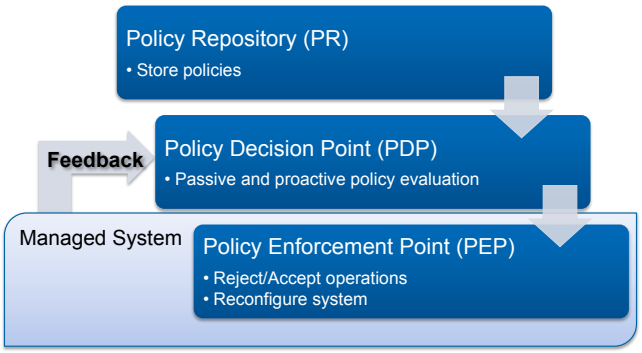

Fig. 3: A general-purpose policy-based management system with feedback

ZPS adopts this general architecture with three additional components:

- A *Policy User Interface* (PUI) allows policy authorities (administrators) to create and modify policies. In a military deployment, the PUI is enabled only on specific nodes determined by the mission.
- A *Conflict Detection Point* (CDP) detects conflicts between a newly created or modified policy against the exiting set of stored policies.
- A *Policy Distribution Interface* (PDI) is used to securely distribute and receive policies over the network.

Figure 4 shows the ZPS components and their relationships with other Zodiac subsystems within a container. The *policy management components* (collectivly is the ZPS) contain the logic to control the *policy-managed components* (Zodiac stack and services). Each policy-managed components implements the PEP that enforces the operations imported by the ZPS. These operations were carefully determined in the Countermeasure Characterizations (CMC) analysis [8].

The PDP manages the Zodiac system by reacting to requests and events from other subsystems, evaluating matching policies and directly invoking PEP operations. The requests include authorization and configuration requests. Services that started later than the ZPS can acquire its settings via a configuration requestion. Additional meta-information are used for ZPS for conflict detection which is described in Section V.

Events are defined to provide situation awareness for the Zodiac system. They serve as feedback information to the PDP. The usage of the events needs to be carefully designed. Carelessly designed events can cause system instability. For example, if an event causes the system to publish the same event, the system will be trapped in a endless loop. Currently ZPS only recognizes the event that identifies the current information operations conditions (INFOCON). We specify obligation policies to apply different communication mechanism for the DCoI based on different INFOCON level.

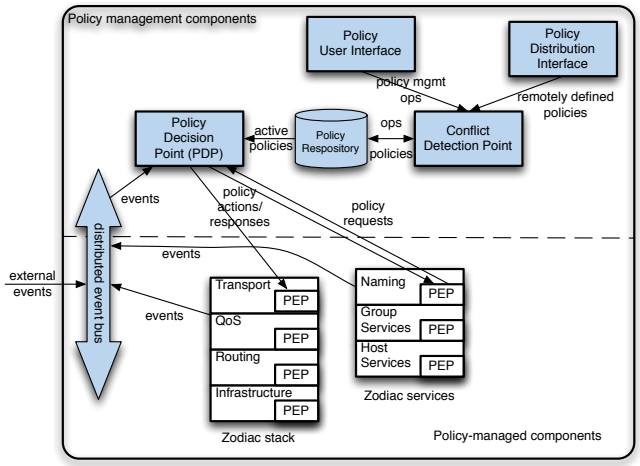

Fig. 4: ZPS components within a container

Policies are inputed into ZPS from either the PUI or PDI and are stored in the Policy Repository after the Policy Deconfliction Component ensures that the policies are conflict-free.

D. Policy Signing and Revocation

An attacker could subvert the system by providing her own policy to facilitate an attack if ZPS accepts policy from any source. Therefore ZPS should be able to identify and trust the author of each policy. ZPS also needs to support over-the-air policy updates by the commanders and their delegates in the field.

In order to protect policy integrity and authenticate the policy author, all Zodiac policies are signed directly or indirectly by trusted entities. We assume that the trusted entity creates policies that follow the operational intent. Nevertheless, some operator errors are prevented by policy conflict detection.

In the PDP engine, an implicit condition for matching policies is the validity of the policy signature. The validity of both the signature and the signer's privilege is verified upon each policy request since trusted entities may change over time (due to change of roles for adversary action, e.g., node capture). The former check

relies on the existing encoding algorithms, and the latter check is implemented using a certificate revocation list [9]. ZPS relies on GCS to perform these operations as part of its responsibility for all the cryptographic and key management in Zodiac.

For a system with a significant number of policies, verifying the policy signature for each policy upon each request may not be efficient. The policy evaluation performance may be optimized by removing policies with invalid signatures before storing the policy or upon revoking a certificate.

E. Policy Distribution

After initial network deployment, administrators may need to create a new policy or modify/remove an existing one. A modified policy set then needs to be distributed to all nodes belonging to the same DCoI. Since MANET connectivity can be unstable, the distribution mechanism should be able to handle intermittent connections.

When a node is temporarily out of reach, that node should obtain the modified policy set when connectivity is resumed. ZPS uses the reliable transport services provided by Zodiac to ensure that a policy set is correctly delivered once it is determined that distribution is required.

Since ZPS manages Zodiac based on the policy contents, the policy set for a DCoI needs to be consistently duplicated among all DCoI members. Thus, policy distribution needs to be integrated with the group membership maintenance, i.e., GCS. When a node joins a DCoI, GCS notifies ZPS to distribute the policy set to the new node after a successful security protocol exchange. When a node leaves the DCoI, the ZPS in that node will remove all policies as the container is also destroyed. The security protocol is described in [10].

During the lifetime of a DCoI, policy updates are distributed via multicast to all members with simple synchronization mechanisms for tolerating unstable network connectivity. In a future implementation, we also plan to re-key when policy distribution occurs. In order to get the new key, a node must have a copy of the current policy set. In current implementation, Zodiac statically positions policies with the correct signatures.

V. IMPLEMENTATION

In this section, we discuss the ZPS policy language, implementation of the policies in Zodiac and the specific policy structure that the language entails. We also describe policy evaluation, detection of policy conflicts and the subsequent deconfliction steps.

A. Policy Language Selection

The policy definition syntax aims to satisfy the requirements in Section II. We adopted an existing policy language and extended it to suit the needs of ZODAIC as opposed to developing a new policy language from scratch because an existing languages could be leveraged to meet our requirements. This approach gave us a head start on higher-level design issues.

Three existing policy language that we considered are Keynote [2], Ponder1 [3] and XACML [1]. Although all three provided the basic functionality for the policy subsystem modulo some extensions, we considered Keynote the best fit, based on the following considerations.

To address the dynamic nature of ZODAIC DCoIs, policy syntax usability is an important design criterion. Specifically, policies should have a format intuitive for a human operator under stressful conditions. The syntax and the representation of policies in XACML is quite verbose in plain text and is not designed for human reading or transmission over limited bandwidth. Defining an abstract XML syntax incorporated with XSLT can simplify the language; compressing the plain text can reduce the required bandwidth. However, this also implies additional computing power required for policy evaluation and distribution. This does not fit either the usability requirement or the low resource consumption requirement.

The policy syntax of both Keynote[11] and Ponder1 is quite intuitive. Additionally, the code size of the Keynote PDP is comparatively small [12] and has low memory and CPU requirements.

Another important consideration in the ZPS design that we want to be able to verify the integrity of policies and their authorized issuers and thus prevent any injection of faulty policies that can corrupt the behavior of the managed system. Therefore we require that all policies are signed. Although signing can be done independently of the policy syntax and verification can be done outside the evaluation process, a better approach is to incorporate the signatures as part of the policy syntax and make verification inseparable from the evaluation. While Ponder1 does not facilitate policy signatures in its syntax, Keynote allows policies to be signed by their issuer and enforces signature verification at evaluation time.

Following the above reasoning, we chose Keynote as a base for the ZPS implementation. It offers an evaluation environment where the default state is denial and policies specify the authorizations and actions allowed in the

system. We extended Keynote to allow specification of obligation policies. Based on occurrence of particular events, policy evaluation returns corresponding multi-dimensional vectors containing specifications for configuration parameter changes or actions that need to be executed by PEPs.

Listing 1 is an example of an authorization policy in Keynote. The authorizer's public key has the privilege to create policies for GCS; the corresponding signature is given at line 12. The "Conditions:" label (lines 8-11) states that node identified as from "blue track" and belonging to "group A, B, or C" is allowed to join the DCoI named "Chat." I.e., when the authorization request presents attributes that match the condition listed before →, the response to this request is "true."

Listing 1: An authorization policy

```
KeyNote-Version: 2
Comment: Join policy for Chat DCoI
Authorizer: GSKEY
Local-constants: GSKEY = "rsa-base64:MEgCQQCzLyQcAxiREa74XwuG\
                          7nU9YiAvICDew6GzeW8D2sAZvlvld8kol\
                          xrPvOTODOHNinVmE 90tg8bPYqrQqwEgj\
                          61jAgMBAAE="
Conditions: (DCOI == "Chat") &&
            (group == "A" || group == "B" || group == "C") &&
            (track == "blue") &&
            (request == "join") -> "true";
Signature: "sig-rsa-md5-base64:DT24V+XKt/hcQ1oerIjFBJTl6QNES\
            +XgRDzE00vx2/LGM4ZC8RCGS34zw60nW\
                          WgSP3alGkv1Kuse+y/Y/UiMfA=="
```

Original Keynote supports only authorization policies; we extended the implementation to support obligation polices. An example is shown in Listing 2. Following the same structure as that for the authorization policies, the condition on the left of → denotes the attribute matching (lines 7 and 9); the list on the right describes enforcement of specific settings in this container.

When the alert level is "ALPHA," use the settings for routing described in line 8; when the alert level is "BRAVO," use the settings for routing described in line 10. The attributes in lines 8 and 10 are implicit in the order of: dispersity degree, dispersity level, multicast type, routing within DCoI only, flooding degree, and maximum TTL.

Listing 2: An obligation policy

```
KeyNote-Version: 2
Comment: Routing policies for responding to different system status
Authorizer: GSKEY
Local-constants: GSKEY = "rsa-base64:MEgCQQCzLyQcAxiREa74XwuG7\
                          nU9YiAvICDew6GzeW8D2sAZvlvld8kolxrPv\
                          OTODOHNinVmE90tg8bPYqrQqwEgj61jAgMBAAE="
Conditions: (app_domain == "routing") && (alert_level == "ALPHA")
                -> ["1"; "0"; "flood"; "flood"; "yes"; "0"; "16"];
            (app_domain == "routing") && (alert_level == "BRAVO")
                -> ["3"; "2"; "flood"; "flood"; "yes"; "1"; "32"];
Signature: "sig-rsa-md5-base64:VEciojrcxNvWPRdGj5tXxd0t3+SR2\
            FvuhagloHb3g3fUzWwCPBB/EwG3P/xzxSxTrvhf6/hYD9spzo\
            4/PbJFGw=="
```

Though the Keynote language supports any combination of conditions, a potential security concern is using the negate operation (syntactically, the exclamation character '!') in the condition. This is because ZPS policies are permissive policies, that is, policies specify only the outcome or the operations that are permitted. Unexpected outcomes and declined operations are simply not mentioned in the policies. To illustrate this concern, we change the condition in the policy of Listing 3; specifically, the condition for track is changed to "not purple." In a system with only "blue" and "purple" tracks, the change makes no difference. However, when a new track, say "red", is introduced to the system, the policy implicitly permits both the "red" and "blue" tracks to join the DCOI, which is potentially undesirable.

Listing 3: Example of negate condition

```
...
Conditions: (DCOI == "Chat") &&
            (group == "A" || group == "B" || group == "C") &&
            (track != "purple") &&
            (request == "join") -> "true";
...
```

Though we make negative conditions in Keynote a sytax error, from a usability perspective, a better UI can assist the user by reverting the negative condition to a possitive set when editing a policy.

B. Policy Conflicts

Although we generally expect that policies are specified in a consistent manner, in a complex system where multiple operators can modify policies at runtime, conflicting policies may be distributed and cause a undesirable system behavior. As discussed in [13] and [14], there may be various sources of conflicts for policies. We analyze what type of conflicts may occur for ZPS and how they may be resolved.

The first important point when considering policy conflicts in Zodiac is that we are dealing only with positive authorization policies. This alone eliminates some types of conflicts that occur between negative and positive authorization policies [13]. However, this does not resolve all possible conflicts.

The simplest type of conflict for ZPS is having different policy decisions for the same attribute in different policies (syntax-level conflict). This can result from policy definitions and/or updates coming from different sources.

Another potential conflict type that goes beyond the syntax is the use of overlapping attribute domains, especially for authorization policies. The attributes used in policies may be based on different characteristics such as organizational divisions, roles, and common attributes. These attributes provides the policy author a

convenient and flexible language set to define policies and improves usability. However, conflicts occur when different attributes refer to the same context.

Policy 1: unit=="A" && operation=="access" && building=="B" -> "true"
Policy 2: department=="Y" && operation="access" && floor=="3" -> "true"

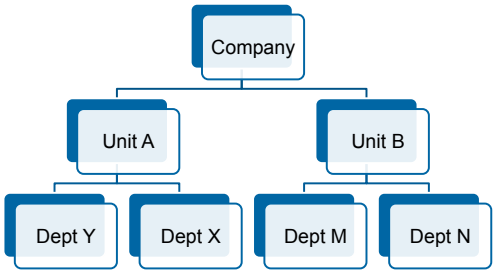

Fig. 5: Organization Example for Meta Policy

For example, two policies are given for a company which has two units and several departments in each unit. Figure 5 illustrates the unit and department relationship. The first policy states that unit A is allowed to access only floors in building B, and the second states that department Y is allowed to access floor 3 of any building. Department Y is in unit A. This type of conflict (overlapping domains) can be detected when the PBMS has the knowledge of the organizational structure and building/floor containment (attribute relationships), as discussed in [13]. The relationships between the attributes are considered as meta-policies in ZPS implemenation.

The Keynote policy condition syntax is DNF, which, when parsed is represented as parse trees. The internal nodes of the tree include logic (**AND, OR**) and assignment operators ($==, <$, etc.) The leaf nodes are condition variables and their values as shown in Figure 6. Syntax level conflicts can be detected by looking at the subtrees rooted at **AND** nodes and detecting conflicting assignments of the same variables (marked with ellipses).

In the case of overlapping domain conflicts, the domain structure can be represented with a similar DNF parse tree that has as its **AND** subtrees any maximal combination of overlapping domains. Searching for intersections of size at least two between the **AND** subtrees of the policy parse tree and the domain structure tree, we can detect conflicts arising from overlapping domains.

The policies in ZPS specify actions allowed in the managed system under certain conditions or the configuration parameters for the system. However, some combinations of conditions and/or parameter values may not be allowed because they do not conform to the desired behavior of ZODAIC components. Examples

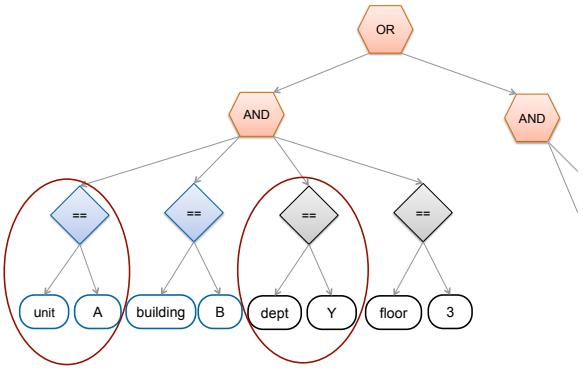

Fig. 6: Policy condition parse tree

include: specification of zero bandwidth allocation to a service and, at the same time, permission for traffic generation; or setting a flooding degree for routing to 7 for a DCoI that has only 6 potential members. We call these incompatible combinations of conditions and parameters policy interdependencies for the ZPS. They represent a conflict both when they exist in the same policy and when they are defined across policies with overlapping domains. Here we can represent again the combination of conflicting parameter assignment with parse trees and look for an overlap between them the domain trees and the policy trees as a method of conflict detection.

C. Other issues

Zodiac system requires deny by default. But some operations, e.g. bootstrap, needs to be permit by default and denied afterwards. These specific operations are not part of the general policy system.

VI. Conclusion and Future Work

A high-security MANET poses several new challenges for the design of its PBMS. First, strict enforcement of communication restrictions necessitates a highly decentralized network management, even within a node, with a PBMS instance for each DCoI, shared resources and secure infrastructure. This in turn, creates more pressure on the resource consumption of a given instance, as node resources are typically scarce. Secondly, high security requires explicit protection of policy integrity in transit, authentication of policy creators and authorization before a policy is enforced. Thirdly, policies must be made available to the PBMS instances in a timely manner to ensure enforcement. Given the dynamic nature of DCoI creation and the intermittent connectivity in MANETs, this requires appropriate policy distribution mechanisms.

Fourthly, the general policy deconfliction problem is complicated by the fully decentralized nature of policy creation and enforcement, and by the need to deconflict DCoI-only policies against shared resource and node policies within each node.

Some of the above challenges have been addressed in the presented ZPS design as summarized next. It is likely that some solutions that fit the security-related requirements of ZPS are applicable to other network environments, e.g. wireline networks.

- *Authentication and Authorization*: Each Zodiac node is given a certificate assigned by a centralized trusted entity. The identity information includes the role of the node in the management hierarchy. The certificate is used to build a chain of trust to acquire other credentials and authorizations (e.g., DCoI membership) and to provide an identity to its peers.
- *Integrity*: This includes the integrity of policies themselves and the integrity of PDP decisions. All policies are signed by their author or modifiers which enforces policy integrity in transit. PDP decision integrity is enforced by considering only authenticated policies and by placement of PDPs locally rather than across the network. ZPS utilizes a certificate revocation list to maintain the authorized set of policy signers. Policies that are deployed with a proper signature can be revoked from the repository.
- *Confidentiality*: Confidentiality indicates that specific information is known only by the information provider and its intended receiver. Besides network encryption for policy distribution, each DCoI container has its own PBMS instance to process policy information.
- *Availability*: MANET link layer communication can be highly unreliable. As policies need to be distributed to all nodes in a DCoI, ZPS uses appropriate reliable protocols (unicast or multicast). Though ZPS denies all operations by default, bootstrap processes are permitted to ensure the ability to setup initial communication between nodes.

Addressing deconfliction for DCoI policies that concern shared resources within in a node and the security requirements for information exfiltration is an on-going effort. Additional work is planned on the automated suggestion of potential conflict resolutions to facilitate policy authors' decision making, an intuitive user interface, and further updates to the Keynote language and PDP. In the near future, we plan to leverage the Zodiac secure transport mechanisms to distribute policies.

REFERENCES

[1] "eXtensible Access Control Markup Language (XACML), Version 2.0, OASIS Standard," 2005. [Online]. Available: http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The KeyNote Trust-Management System Version 2," RFC 2704 (Informational), Sep. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2704.txt

[3] N. Damianou and N. Dulay, "The Ponder Policy Specification Language," in *Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 18–38.

[4] R. Chadha, Y.-H. Cheng, J. Chiang, G. Levin, S.-W. Li, A. Poylisher, L. LaVergne, and S. Newman, "Scalable policy management for ad hoc networks," in *In Proceedings of MILCOM 2005 : the Military Communications Conference*, 2005.

[5] R. Yavatkar, D. Pendarakis, and R. Guerin, "A Framework for Policy-based Admission Control," RFC 2753 (Informational), Jan. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2753.txt

[6] "Intrinsically Assurable Mobile Ad-Hoc Network (IA-MANET)," DARPA BAA. [Online]. Available: http://www.darpa.mil/sto/solicitations/IAMANET/

[7] S. Alexander, B. DeCleene, J. Rogers, and P. Sholander, "Requirements and architectures for intrinsically assurable mobile ad hoc networks," in *Military Communications Conference, 2008. MILCOM 2008. IEEE*, 2008.

[8] H. O. Lubbes, "Countermeasure characterizations: building blocks for designingsecure information systems," in *DARPA Information Survivability Conference and Exposition II, 2001. DISCEX '01. Proceedings*, vol. 1, 2001, pp. 103–115.

[9] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), May 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[10] H. Harney, U. Meth, A. Colegrove, and G. Gross, "GSAKMP: Group Secure Association Key Management Protocol," RFC 4535 (Proposed Standard), Jun. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4535.txt

[11] M. Blaze, J. Feigenbaum, and A. D. Keromytis, "Keynote: Trust management for public-key infrastructures (position paper)," in *Proceedings of the 6th International Workshop on Security Protocols*. London, UK: Springer-Verlag, 1999, pp. 59–63.

[12] "The Keynote Trust-Management System," Implementation download. [Online]. Available: http://www1.cs.columbia.edu/~angelos/keynote.html

[13] E. C. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," *IEEE Trans. Softw. Eng.*, vol. 25, no. 6, pp. 852–869, 1999.

[14] J. D. Moffett and M. S. Sloman, "Policy conflict analysis in distributed system management," *Journal of Organizational Computing*, 1993. [Online]. Available: citeseer.ist.psu.edu/moffett93policy.html