On the Safety of Enterprise Policy Deployment

Yudong Gao Xu Chen Ni Pan Z. Morley Mao
University of Michigan - Ann Arbor
{stgyd,chenxu,nipan,zmao}eecs.umich.edu

Abstract

Enterprise policy management is challenging and error-prone. Compared to existing work that focused on analyzing misconfigurations, our work is the first to address the issues that arose during policy deployment, i.e., *effecting policy changes. In this paper, we demonstrate that naive approaches to policy deployment can easily create security vulnerabilities, such as granting access of sensitive resources to unprivileged users or temporarily allowing malicious traffic to critical network infrastructure. To systematically solve this problem, we formally define secure and insecure intermediate states, and further propose an efficient algorithm to find a deployment procedure without insecure intermediate states. We implemented and evaluated our algorithm on Group Policy framework, while only harnessing existing support and requiring no modification to the current infrastructure. Our evaluation shows that our solution adds minimal overhead to the overall deployment time while provably eliminating insecure intermediate states.*

1 Introduction

Managing enterprise networks is extremely challenging. While there are many different aspects involved, such as hardware maintenance, topology design, middlebox placement, in this paper, we focus on policy management, which spans across resource access control, network security management, *etc.* Broadly speaking, policy management [6, 29, 25] in an enterprise network environment needs to meet the requirements of security, performance, manageability, and failure resilience. The actual realizations may include restricting a machine containing sensitive data to be accessed by only a small group of users, preventing external traffic from directly reaching internal databases, *etc.* As easy as it may sound, specifying a correct network policy is surprisingly difficult, due to the enormous number of users and end-hosts managed and the complicated policy for each entity. Although millions of dollars are spent every year by large enterprises on IT support, misconfigurations still cause significant network downtime [18]. In today's networks, policy related misconfigurations are prevalent [6, 18] and can directly lead to leakage of sensitive information, impact on critical network infrastructure, performance degradation, or other undesirable consequences.

Although there is a conceptual design goal for an enterprise network, the actual policy realization is usually decomposed into many pieces, in the form of policy objects. Each policy object controls a specific type of configuration, such as the IPsec configuration for all computers, or fine-tunes certain policy for a targeted group of entities, such as setting the network proxy server for all sales personnel. Two main reasons contribute to this status quo. On the one hand, the sheer number of entities to manage and the variety of roles each entity plays is large, and continues to grow for any active enterprise. On the other hand, the policy to manage in enterprises is becoming increasingly complicated as the policy management system advances [9]. In modern enterprises, administrators need to configure for numerous types of setups, including access control, software installation, system preference, *etc.* As such, it is hard, if not impossible, to come up with a holistic policy setup to cover everything. Instead, generating decomposed and specialized policy objects is much more scalable in terms of manageability, allowing the distribution of the management workload among multiple administrators.

Enterprise policy management systems today indeed rely on the specification and integration of policy objects [26, 24]; however, this approach has several limitations: 1) *Conflict resolution* is a procedure that determines the actual policy when multiple choices are available. Although it is a general problem for policy management, it is particularly relevant when policy objects are used, because the policy setups in different objects may not agree with each other and thus require a resolution procedure. 2) *Transactional update* is needed when multiple objects together fulfill a policy design goal. To reflect a design change, those objects need to be modified in an atomic fashion without exposing intermediate states. While conflict resolution is a relatively well-studied topic [17, 1, 4, 31, 16, 30], in this paper, we focus on the issues associated with update atomicity.

Existing policy management systems usually have a central location for storing policy objects, which are frequently read by the managed entities. Unfortunately, support for transactional update of multiple objects is intentionally missing in favor of performance in current policy management systems, because policy reads are more frequent than policy modifications (thus a write lock would negatively impact performance). As a result, a policy design change that relies on modifying multiple policy objects has to be carried out in multiple steps. Without transaction support, end-hosts can retrieve a transient, inconsistent intermediate policy, easily leading to misconfigurations. This problem is exacerbated by end-hosts only updating policy objects infrequently, thus keeping the transient wrong policy setups for a surprisingly long time. The persistently connected end-hosts could use a wrong policy for hours before the next regularly scheduled update. For sporadically connected end-hosts, *e.g.,* laptops that are only occasionally connected to the enterprise network, this inconsistency window could be days or longer. The fact that policies change very frequently for large enterprise networks makes the problem more severe. Such changes are necessary for reasons including department hierarchy adjustment, personnel transfer, computing resource introduction, removal and re-purpose.

Compared to most existing work [13, 20, 23, 32, 22] that focuses on the correctness of *static* policy specifications, detection and correction of misconfigurations, such as blocking legitimate access or allowing malicious traffic, we are the first to study the *dynamic* aspect of policy deployment. This is the process of committing the new policy, usually to the central policy storage and management server. This process is usually overlooked by administrators and can potentially be exploited by stealthy adversaries.

We first make a case for the necessity of preserving policy integrity, *i.e.,* enforcing the administrators' intention throughout the deployment phase, by showing how errors and inconsistencies can easily occur. To systematically solve this problem, we formally model the general policy deployment problem. Within our formulation, we define secure and insecure intermediate states, and further propose an efficient algorithm to find a deployment procedure free from insecure intermediate states. Such insecure states are inconsistent with administrators' intentions and can thus result in security vulnerabilities. To the best of our knowledge, this is the first work to systematically deal with potential problems of policy deployment in enterprise network environment. We implemented and evaluated our algorithm on Microsoft's Group Policy framework [26, 7], while only harnessing existing support and requiring no modification to the current infrastructure. Although currently we are targeting Group Policy framework, our model is general enough to be applicable to other policy models to solve similar deployment problems. The evaluation shows that our algorithm adds minimal overhead to the overall deployment time while provably eliminating insecure intermediate states.

1.1 Group Policy Background

We base our study on group policy [26] under the active directory framework [7] because it is the de facto, widely adopted, role-based policy management systems in enterprise environment [14]. The basic concepts of group policy, as we describe next, are comparable to other frameworks, which can be studied using our formulation and algorithms. This section provides some background information about active directory and group policy.

Active directory is an important infrastructure in the Microsoft Windows Server family for managing enterprise networks. An entire enterprise network is divided into domains according to organizational or geographical terms. Objects in a domain, including users and machines, are grouped into containers called Organizational Units (OUs), which are organized as a hierarchical tree structure (Figure 1).

Group Policy is an infrastructure for designing and deploying desired configurations or policy settings to different sets of target objects. In particular, administrators specify network policies at a central location (on domain controllers), and the policies are downloaded, filtered and enforced on clients by a set of client-side extensions.

In the group policy framework, policy settings are stored in Group Policy Objects (GPOs) [8], which are the management units of group policy framework. GPOs control settings in Windows systems such as IPsec policies, registry table entries, software settings (*e.g.,* Internet Explorer or Word configurations). A GPO can be linked to one or more OUs, whose members will apply the settings specified in the GPO. Note that the members of a child OU will also apply the GPOs linked to the parent OUs. As a result, a GPO can be linked to multiple OUs, while one OU can have multiple GPOs attached - this reduces the management overhead by maximally reducing the amount of overlapping policy configurations.

For example, in the setup illustrated in Figure 1, GPO_1 is linked to the domain so it is applied to all the members inside this domain; GPO_2 is linked to two OUs: OU_A and OU_B; OU_D have two GPOs attached: GPO_3 and GPO_4. In this case, $User_1$ receives GPO_1 and GPO_2; $User_3$ and $Mach_1$ receive GPO_1, GPO_2, GPO_3 and GPO_4.

Different GPOs may try to configure the same aspect on target computers, *e.g.,* a domain-wide GPO tries to configure a default IPsec policy, while a GPO linked to *Servers* OU specifies an IPsec policy with higher security requirements. In these cases, GPOs in child OUs will override the same configurations specified in GPOs from parent OUs. For GPOs linked to the same OU, the one with the highest

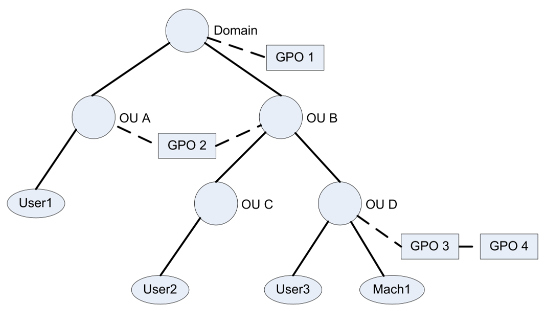

Figure 1. Organization of Group Policy Framework

precedence value is effective.

All the GPOs in one domain are stored in centralized domain controllers. For redundancy and reliability purposes, multiple domain controllers can run simultaneously, each holding a copy of the GPOs. When a domain-joined computer boots up, it downloads the GPOs applicable to the computer object from one of the domain controllers. During user login process, the GPOs applicable to the user object are downloaded and applied. A background process on each computer periodically, with a default period of 90 minutes, queries domain controllers to download and update modified GPOs.

1.2 An Example Misconfiguration Caused by Policy Deployment

Here we perform a case study about how errors and inconsistencies can be easily introduced if the policy deployment is not carefully inspected, despite the correctness of the policy specification.

Consider the case of assigning access to critical resources, *e.g.,* server machines, to a small group of privileged users while preventing access from other users. A common approach is to first define a background policy to block all access to these resources, and then override it with a more specific policy which grants access to only privileged users. While each specific policy is inserted one by one, the transient state right after committing the background policy but before committing the specific policy for privileged users, creates a misconfiguration. In this state, legitimate access of privileged users is temporarily disabled.

In large enterprises, policy settings are constantly updated for a variety of reasons, *e.g.,* handling department hierarchy change, implementing new software policies, and preventing recent attacks. Unfortunately, due to a lack of transactional support, security vulnerabilities can be exploited when a policy change is carried out as multiple successive policy modifications. These modifications are usually manually performed, thus leaving a very large vulnerability window. Between the start and end of a modification sequence, an entity that happens to query the policy storage server would receive an intermediate state with a subset of the modifications in effect. This intermediate state could break software integrity or even open up security holes, *e.g.,* new email application installed but without proper network filters specified. The severity of this problem is exacerbated by the fact that typical enterprise networks have enormous number of user and computer objects.

For the sake of illustration, let the duration of an insecure intermediate state be T, the average policy update interval for each machine be I, and the number of computers in the domain be N. The expected number of computers which download this insecure state is $S = N \times T/I$. With a very conservative estimation of $T = 10$ sec, $I = 90$ min, $N = 20,000$, we still get $S = 37$. In a large enterprise with frequent policy modifications, hundreds or even thousands of machines can download insecure states in one day. In our lab experiments with the group policy framework, deploying a GPO with a few settings takes several seconds. In practice, a GPO has hundreds of settings, and an update usually involves multiple GPOs, leading to a much longer T. Manual updates can easily increase T to be on the order of minutes. Thus many more computers can be impacted. To make matters even worse, machines with insecure settings do not perform another policy update until the next policy update event, causing them to stay in a potentially insecure state for hours or even days.

2 Related Work

In a broad sense, our work of enterprise policy deployment touches on a vast array of research projects including firewall policy analysis [33, 3], policy configuration modeling [23, 12, 13, 20], and policy misconfiguration detection [17, 16, 5, 22, 2, 15, 30].

The safety issues in policy deployment have received little attention. To the best of our knowledge, Zhang *et al.* [33, 3] presented the first work on the safety issues in firewall policy deployment. They proposed an efficient algorithm with provable safety guarantees for firewall policy deployment, eliminating unsafe intermediate states. Our work, on the other hand, covers generic policy setup for complex enterprise network environment, where firewall policy setup is a small subset. A detailed comparison can be found in §3.3.

To assist systematic analysis of policy configurations, many models have been proposed [23, 12, 13, 20]. The concept of role-based access control was introduced [23] to ease policy management for collections of users and machines. A role-based policy specification language called Ponder was proposed in [12, 13]. Ponder is declarative and object-oriented, and it also supports event triggered obligation policies. McDaniel *et al.* [20] created a general-purpose policy model and concluded that reconciliations for three or more policies are intractable. These models are too general since they try to cater to all kinds of application environment, preventing them from widely deployed. In contrast, the Group Policy framework provides a straightforward model for managing Windows machines, and is thus widely deployed in enterprise environment.

Identifying misconfigurations and reconciling conflicts [17, 16, 5, 22, 2, 15, 30] in policy setup are enduring topics. Jaeger and Zhang proposed using the concept of Access Control Spaces [17] to discover the policy coverage issue and identify vulnerable or unspecified policy spaces automatically. Also related is work by Hicks *et al.* [16] on modeling the SELinux Multi-Level Security with a logical specification, testing properties such "read-down" and "no write-down" as well as the compliance of two policies. Their work relies on high-level security assignments and constraints, which are difficult to identify in real environment. Bauer *et al.* [5] employed a machine learning approach on access log data to discover association rules. However, this approach can only detect cases in which a user is denied access to a given object. Our work does not rely on the high-level information about the design intention.

There is also related work which applies rule reasoning and graph-based algorithm to policy analysis. MulVAL [22] determines the security impact of software vulnerabilities on a particular network. It also uses a modeling language and reasoning rules to check for violations with given control policies. Hamed and Al-Shaer proposed applying graph-based Boolean function manipulation to distributed policy analysis [2] and using taxonomy to detect conflicts among policies running on network security devices [15]. Wang *et al.* [30] captured dependencies between policy components with directed acyclic graphs and proposed a linear algorithm for reconciliation. Unlike most work in this space, we do not focus on identifying conflicts in policy specification, but instead ensure secure intermediate states during the deployment process.

3 Safe Deployment Formalization

To systematically analyze the problem of policy deployment, in this section, we describe a model that captures the fundamentals of policy expression and processing in the Group Policy framework. This model is general and can be easily extended to other frameworks, as we later demonstrate in §3.3.

Table 1. Notation Table

Notation	Meaning
Uppercase Letter (*e.g.,* A)	A GPO
Lowercase Letter (*e.g.,* k)	A key
$A[k]$	Key k's value defined in GPO_A
OL	Original GPO list
TL	Target GPO list
OS	Original state
TS	Target state
IS	Intermediate state
$Last_{OL}[k]$ ($Last_{TL}[k]$)	The last GPO that defines a value for key k in OL (TL)
$<_O$ ($<_T$)	$GPO_A <_O (<_T) GPO_B$ means that A has lower precedence than B in OL (TL),
$\rightarrow$	"happen before" relationship
L	sum of the sizes of OL and TL
K	number of different keys
F	number of different filters used

3.1 A Generalized Model

GPO Setup 1

Key	Values in GPO_A
Windows Firewall: Protect all network connections	Enabled
Access to command prompt	Disabled

In our model, each GPO is viewed as mappings from **keys** to **values**. For example, in GPO Setup 1 that defines GPO_A, "Windows Firewall: Protect all network connections" and "Access to command prompt" are keys, and "Enabled" and "Disabled" are the corresponding values. The mapping from a key to a value, such as "Windows Firewall: Protect all network connections = Enabled", is called a **setting**. If two GPOs map the same key to different values, the settings in these two GPOs are in **conflict**, and the settings in the GPO with higher precedence would override the settings with lower precedence. Each GPO has an **effecting scope**, which is a set of objects that the GPO can be applied to. By default, when a GPO is linked to an OU, its effecting scope contains all the objects in the OU. **Filters** can be linked to a GPO to restrict its effecting scope.

We use an uppercase letter to denote a GPO, and a lowercase letter to represent a key, and $A[k]$ to denote key k's value defined in GPO_A. For example, in GPO Setup 1, $A[\text{"Windows Firewall: Protect all network connections"}] = \text{"}Enabled\text{"}$. If GPO_A does not define any value for key k, then $A[k] = undefined$. Table 1 summarizes the notation used in this paper.

We restrict our discussion to *a single* OU for brevity. For the cases where modifications across multiple OUs are made, our algorithm can be applied multiple times. Each run of our algorithm will generate a sequential list of modifications for each impacted OU. All the GPOs linked to the OU constitute a **GPO list**. They are applied sequentially, where GPOs appearing **later** in the list have **higher precedence**. All the GPOs inherited from the ancestor OUs are not modified during the update process, since we are targeting a single OU and all the modifications are related to the GPOs directly linked to this OU. So the inherited GPOs can be consolidated into a single GPO, which is linked to the front of the list and given the lowest precedence. Our update scenario can thus be defined as follows: given an **Original GPO List (OL)**: $O_1, O_2, \ldots, O_n$, we need to update it to the **Target GPO List (TL)**: $T_1, T_2, \ldots, T_m$.

The result of a GPO list is a **state**, which also consists of mappings from keys to values. The state's key set is the union of key sets defined in the GPOs in the list; the value of a key equals to the value that takes the highest precedence in the list.

For example, GPO Setup 2 is a GPO list of A, B in which B has higher precedence. The resulting state of this list is shown in the last column. Note that A's value "90 min" is overwritten by that of B.

GPO Setup 2

Key	GPO_A	GPO_B	**State**
Group Policy refresh interval for computers	90 min	180 min	180 min
Access to command prompt	Enabled		Enabled

The resulting state of the original GPO list (OL) and the target GPO list (TL) are called the **original state (OS)** and **target state (TS)**, respectively.

An **intermediate state (IS)** is the resulting state of the runtime GPO list at a particular time instant during the update process.

We first assume that each key is independent so that the effect of one key is independent of the values of the other keys. Then we extend the model to cover **dependent keys** later.

When all keys are independent, each particular behavior of the system is determined by the value of one corresponding key. For example, in GPO Setup 2, the key "Group Policy refresh interval for computers" and the key "Access to command prompt" are independent. We assume that both OS and TS are stable and secure, since their key values are carefully assigned by the administrator. (The problem of telling whether there is misconfiguration in OS or TS is outside the scope of this paper.) So as long as the value of each key in an intermediate state equals to that in the OS or TS, the system will behave according to the expectation of the administrator, and we say it is in a secure state, since it will not create any unexpected security hole.

Definition 3.1. IS is a **Secure Intermediate State** if and only if for each key k in IS, its value equals to either the value in OS, or the value in TS, *i.e.,* $IS[k] = OS[k]$ or $IS[k] = TS[k]$. Otherwise it is an **Insecure Intermediate State**.

The values $OS[k]$ and $TS[k]$ are **secure values** for key k. Any other values for key k are **insecure values**. Insecure intermediate states may either block legitimate access or mistakenly elevate privilege, impacting usability or creating security holes. These cases are not differentiated in our model, all of which are judged as insecure values and should be prevented.

Definition 3.2. A **Secure Update** is an update process in which the intermediate state after each operation is secure.

Now we extend the model to consider dependent keys. In a real environment, a set of keys can be dependent, meaning that together they control one particular behavior of the system. In this case, it is the combined values of these keys, instead of each individual value, that matters. Since modifications to any key result in new combinations with unknown effect, the intermediate state can be insecure.

Definition 3.3. Suppose that there is a set D in which all keys are dependent, an intermediate state is secure if and only if $IS[k] = OS[k]$ for each key k in D, or $IS[k] = TS[k]$ for each key k in D. Otherwise, it is an insecure intermediate state.

Dependent keys exist in real policy management systems, although they may not be prevalent. For example, in Group Policy, the key "Windows Firewall: Allow authenticated IPSec bypass" instructs the firewall to allow IPSec traffic. The value set to this key makes sense only when "Windows Firewall: Protect all network connections" is enabled, which turns on the Windows firewall. Suppose that currently the firewall is off so all traffic is accepted, and we need to turn on the firewall to block all traffic except those authenticated with IPSec. If the firewall is enabled first, authenticated IPSec traffic will be blocked temporarily until the key to allow IPSec bypass is set.

As another example, the key "Group Policy refresh interval for computers" specifies how often group policies for a computer are updated while the computer is in use, but if the key "Turn off background refresh of Group Policy" is enabled, background refresh will never happen no matter what value is set for the first key. In both cases, one key depends on another to take effect, and if only one of them is updated, the intermediate state is unknown and potentially insecure. So all the operations that update the values of dependent keys from $OS[k]$ to $TS[k]$ should be executed simultaneously in order to preserve secure state.

Overall, our goal is to find a secure update using the existing APIs provided by the Group Policy framework without any modification to the platform itself. These APIs perform basic policy operations, which are also available on other policy management platforms. In most cases this can be achieved by merely reordering the update operations, causing no overhead to the update process. However, in some cases additional GPOs must be inserted during the update process to guarantee secure update, which will inevitably introduce extra overhead. Details are presented in §4.

3.2 Available Update Operations

In §3.1 we presented the policy model, now we discuss how to incorporate our model with the Group Policy framework, so that the model can extract input from the framework and invoke the APIs to perform secure policy update. Similar policy operating APIs are also provided by other policy management platforms, to which our model can be ported easily.

In our policy deployment scenario, we assume that information about the OL and TL, and the settings of GPOs in both lists are available before each deployment. When an administrator finishes staging and planning an update, this information can be trivially collected from the planning and management tools, *e.g.,* the Group Policy Management Console [10] in the Group Policy framework. These are the input to our model.

To update the GPO list from OL to TL, two types of operations are necessary: list operations (operations on the GPO list) and key operations (operations on the GPO key settings).

If a GPO T_i in TL does not appear in OL, "ADD GPO" operation is needed to add T_i to the list and assign it with a proper precedence. "MOVE GPO" operation is used to change the precedence of T_i. Similarly, if O_i is not in TL, a "REMOVE GPO" operation is needed to remove it. These list operations automatically assign a proper precedence to each GPO according to its position in TL.

If T_i and O_j refer to the same GPO, but with different key settings, then key update operations on O_j are required. There are three kinds of key operations: "ADD Key $T_i[k]$" if k is defined in T_i but not O_j, "REMOVE Key $T_i[k]$" if k is defined in O_j but not T_i, or "SET Key" if both O_j and T_i define k but $O_i[k] \neq T_i[k]$. Therefore, for each GPO there is at most one list operation; for each of its key setting k, there is at most one key operation, since the update operations are determined only by the original and the target state.

These update operations are all available in the Group Policy framework. Each of them is supported by a corresponding API, which performs the operation **atomically**. The intermediate states during the execution of these APIs are not exposed.

3.3 Comparison with the Firewall Deployment Model

To the best of our knowledge, Zhang *et al.* [33] proposed the first and the only work to formally analyze the policy deployment problem for firewall polices. This is our closest work. But the firewall policy model lacks the expressiveness to describe enterprise policies which are much more general and comprehensive, making it impossible to apply their algorithms to enterprise policies deployment, as we will demonstrate later.

Next we briefly introduce the firewall policy model and then compare it with our policy model.

In the firewall model, a firewall policy is an ordered list of rules with a "first match" semantic. Each firewall rule r specifies an accept or deny action on a *filtering set*, which is a set of flows; a packet p matches r if p belongs to a flow in r's filtering set. For example, in the setup of the following policy

(1) Deny TCP 10.1.1.0/24 80

(2) Deny TCP 10.1.2.0/24 any

(3) Permit IP 10.0.0.0/8 any

HTTP traffic from 10.1.1.0/24 first matches rule (1) and gets blocked, while the traffic from 10.0.0.0/24 matches rule (3) and gets permitted.

In this model, policy deployment is a process to update the running policy from the initial policy to the target policy. They defined the concept of *safe deployment*, during which any packet that is permitted/denied by both the initial and the target policy is always permitted/denied. For policy modification languages which support only append and delete rule operations, they prove that safe deployment is not always possible. For more powerful languages supporting additional move operations, they show the existence of a safe deployment containing the minimum number of operations.

Our proposed model is similar to this firewall model. An OU is linked with a list (analogous to a policy in firewall) of GPOs (analogous to rules in a firewall policy), and each GPO has a precedence (analogous to the rule's position in the list). And for a particular key (analogous to a packet), the value (analogous to the action on the packet) is decided by the GPO with the highest precedence (analogous to the first rule that matches the packet) among all the GPOs that set a value for this key.

Note that our model is more comprehensive:

(1) GPOs are reused across OUs *i.e.,* a GPO can be linked to multiple OUs[1];

(2) Key operations are supported to update the settings in GPOs;

(3) Filters and related operations are supported to provide fine-grained control over the effecting scope of each GPO.

In contrast, for the firewall model:

(1) Firewall rules are not reused since only one firewall list is considered;

(2) There is no update operation for rules, because an update operation can be viewed as removing the old rule and inserting a new one as long as rules are never reused;

(3) There is no concept of filter or scope, and each rule in the list applies to all packets.

In summary, the firewall model can be considered as a simplified special case of our model, if we view each firewall rule as a GPO without key update operation. So the algorithms proposed for the firewall deployment model do not apply, and a new algorithm to guarantee safe deployment for the GPO model is required.

4 Algorithms for Secure Policy Deployment

Based on the model described in §3, we present our algorithm that aims to find a secure update from OS to TS, *i.e.,* an ordered sequence of operations that guarantees secure intermediate state after each operation. We first introduce the concept of "happen before" relationship among update operations, which should be enforced to prevent insecure states. A set of rules are defined to identify all relevant "happen before" relationships, described by a directed graph. Then by breaking strongly connected components and performing topological sort on the graph, we can find an order in which the update operations should be executed for a secure update. Note that it is possible that there are multiple orders which all achieve secure updates. In this case our algorithm will return the one with minimal deployment cost. We also prove that with any given OS and TS, our algorithm can always find such an order for secure update.

[1]Although our update scenario targets one particular OU at a time, our algorithm takes into account that the GPOs linked to the target OU can also be used by the other OUs within the same domain, and updates to the GPOs affect the states of those OUs as well.

4.1 "Happen before" Relationship

As defined in §3.1, an intermediate state is insecure when it contains insecure values, which do not occur in either OS or TS. During the update process, there can be time periods, *e.g.,* after the original secure value $OS[k]$ is removed and before the new secure value $TS[k]$ is added, during which the insecure values are exposed externally in the intermediate state, making the policy setting potentially insecure.

For example, suppose we are going to update an original state described in GPO Setup 3 to the target state described in GPO Setup 4.

GPO Setup 3

Key	GPO_A	GPO_B	GPO_C	**State**
key i	1	2	3	3
key j	4	5		5

GPO Setup 4

Key	GPO_A	GPO_B	GPO_C	**State**
key i	1			1
key j	4	5		5

Both the key settings $B[i] = 2$ and $C[i] = 3$ should be removed. $B[i] = 2$ is an insecure value since it is overridden by $C[i] = 3$ in OL, and is removed in TL. During the update, if $C[i] = 3$ is removed, before $B[i] = 2$ is removed, we would enter into an insecure intermediate state in which $B[i] = 2$ is unmasked and takes precedence, as demonstrated in GPO Setup 5.

GPO Setup 5

Key	GPO_A	GPO_B	GPO_C	**State**
key i	1	2		2
key j	4	5		5

To prevent this kind of situation, some operations should be executed first to introduce secure values to the list or to remove insecure values, guaranteeing that insecure values are always masked by secure values. This is what we called a *"happen before"* relationship. [2]

Definition 4.1. If operation X should occur before operation Y in order to maintain the secure intermediate state, we say that X should **"happen before"** Y.

In the previous example, insecure value $B[i] = 2$ is originally masked by the secure value $C[i] = 3$, and both of these two values are to be removed. To prevent $B[i] = 2$

[2]Note that this is also similar to the concept of "order constraint" in partial order planning [21] in AI area. We refer to it as "happen before" which is a more general idea [19].

from appearing in the intermediate state, its removal should "happen before" the removal of $C[i] = 3$.

The basic idea of our algorithm is to identify "happen before" relationships (HBRs) among update operations in order to prevent insecure intermediate states. We define several rules to identify all the HBRs.

Basically, insecure intermediate states occur when insecure values take precedence at a particular moment. An operation may change a secure state into an insecure state in three ways:

(1) Add insecure values which take precedence in the intermediate state;

(2) Elevate the precedence of existing insecure values so they take precedence in the intermediate state;

(3) Remove secure values so the originally masked insecure values now take precedence.

All these situations are prevented by the 6 rules proposed later in this section. Rule $2, 3, 4, 5(a)(b)$ target situations (1) and (2). They guarantee that before each possible operation is executed, the secure values in TS, which are necessary to mask the to-be-added or to-be-elevated insecure values, already exist in the list and take precedence. For situation (3), Rule $1, 5(c), 6$ guarantee that either the proper secure values to mask the insecure values already exist, or the insecure values are removed. Consequently, these rules prevent insecure values from appearing in any intermediate state, guaranteeing a safe update process.

Before moving to the detail of each rule, for ease of exposition, we introduce several notations depicted in Table 1. $Last_{OL}[k]$ represents the last GPO that defines key k in OL, and its k value takes precedence in OL and is thus secure, *i.e.,* $Last_{OL}[k][k] = OS[k]$. Here $Last_{OL}[k]$ refers to a GPO instead of a value, so two brackets are used to refer to the key value. "$GPO_A <_O GPO_B$" means that GPO_A appears earlier than GPO_B in OL, therefore, A has lower precedence. $Last_{TL}[k]$ and "$GPO_A <_T GPO_B$" are the corresponding notions for TL. The symbol "$\rightarrow$" is used to represent the HBR constraint.

When a GPO is added or moved via "ADD GPO" or "MOVE GPO", it is assigned a proper precedence according to its position in the TL. We set this precedence higher than those of unprocessed GPOs in the OL. For example, to update GPO list GPO_A, GPO_B, GPO_C to GPO_C, GPO_B, GPO_A, we first move GPO_B, and the list becomes GPO_A, GPO_C, GPO_B; then we move GPO_A and get GPO_C, GPO_B, GPO_A. This property is important, and the following rules depend on this property to guarantee secure intermediate states.

Now we introduce the rules one by one. Each rule comes in the form of "$Operation_X \rightarrow Operation_Y$", meaning that if both $Operation_X$ and $Operation_Y$ exist in the update process, then $Operation_X$ should be executed before $Operation_Y$ to guarantee secure intermediate state. If either $Operation_X$ or $Operation_Y$ is missing, the rule is just ignored. As discussed in §3.2, the operations in the update process are predetermined by OL and TL. What's more, GPO_A refers to each possible GPO while key k refers to each possible key.

The first rule means that if a GPO is removed in TL, its new values should be added or modified only after the GPO is removed.

1. `REMOVE` GPO_A $\rightarrow$ `ADD/SET Key` $A[k]$;
   `REMOVE` GPO_A $\rightarrow$ `REMOVE Key` $A[k]$
   `if` $A = Last_{OL}[k]$

If GPO_A is to be removed from the GPO list (still exists in the Active Directory), all the values added or set are insecure since these values never appear in OS or TS. So these values should be added or set safely after GPO_A is removed. No insecure value will be introduced in this case.

If $A = Last_{OL}[k]$, $A[k]$ is a secure value which can be used to maske insecure values, and keeping it in the list does not introduce any insecure state. So it is beneficial to keep it longer in the list. If it is to be removed, we remove it only after A is removed.

The second rule deals with the cases where there are key operations related to a to-be-added GPO. This is similar to Rule 1.

2. `REMOVE/SET Key` $A[k]$ $\rightarrow$ `ADD` GPO_A

If GPO_A is to be added, it is not in the OL. So all the values associated with A, which are to be removed or modified, are insecure since they do not show up in TS either. So these values can be removed or modified safely before GPO_A is added.

Rule 3 guarantees that when a GPO is added or moved, it already contains all of its secure values which can mask insecure values. This rule is commonly referred to by the other rules.

3. IF $A = Last_{TL}[k]$,
   `ADD/SET Key` $A[k]$ $\rightarrow$ `MOVE` GPO_A,
   `ADD Key` $A[k]$ $\rightarrow$ `ADD` GPO_A;

If $A = Last_{TL}[k]$, $A[k]$ is a secure value since it takes precedence in TL. We safely add $A[k]$ to GPO_A before moving or adding A. So when A is added or moved, $A[k]$ can be used to mask insecure values.

In some cases the first three rules are redundant, *e.g.,* when there is a secure value for k in IS, it is safe to break Rule 2 by adding GPO_A first and then removing $A[k]$, since

$A[k]$ is masked anyway. But these rules help facilitate later discussions while introducing no extra overhead in the update. None of them results in unnecessary cycles in the dependency graph, which is discussed later in §4.3.

Rule 4 is the most important and commonly used rule. It ensures that before a GPO is added to TL or moved to the right position, all the secure values needed to mask its insecure values are already in the right position in the GPO list.

4. For each key k in GPO_A, including those keys to be added with "ADD Key $A[k]$",

```
If $A \neq Last_{OL}[k]$ and $A \neq Last_{TL}[k]$ ,
  ADD/MOVE GPO $Last_{TL}[k]$
    $\rightarrow$ ADD/MOVE $GPO_A$
```

Each GPO that exists in both OL and TL has an associated "MOVE GPO" operation. Rule 5 deals with this kind of to-be-moved GPOs and their related key update operations. Each kind of key update operations is covered in one sub-rule.

5. For each GPO_A that is in both OL and TL, and $A \neq Last_{TL}[k]$:

```
(a) For "ADD Key $A[k]$",
  ADD/MOVE GPO $Last_{TL}[k]$
    $\rightarrow$ ADD Key $A[k]$
(b) For "SET Key $A[k]$",
IF $A \neq Last_{OL}[k]$,
  ADD/MOVE GPO $Last_{TL}[k]$
    $\rightarrow$ MOVE $GPO_A$,
  ADD/MOVE GPO $Last_{TL}[k]$
    $\rightarrow$ SET Key $A[k]$;
IF $A = Last_{OL}[k]$,
  ADD/MOVE GPO $Last_{TL}[k]$
    $\rightarrow$ SET Key $A[k]$;
(c) For "REMOVE Key $A[k]$",
IF $A \neq Last_{OL}[k]$,
  REMOVE Key $A[k]$ $\rightarrow$ MOVE $GPO_A$;
IF $A = Last_{OL}[k]$ and $Last_{TL}[k] \neq NULL$,
  MOVE/ADD GPO $Last_{TL}[k]$
    $\rightarrow$ REMOVE Key $A[k]$,
  and for each GPO $X$ that $Last_{TL}[k] <_T X <_T A$
and sets $k$
  REMOVE Key $X[k]$ $\rightarrow$ REMOVE Key $A[k]$;
IF $A = Last_{OL}[k]$ and $Last_{TL}[k] = NULL$, for
each GPO $X <_T A$ that sets $k$ and
  REMOVE Key $X[k]$ $\rightarrow$ REMOVE Key $A[k]$;
  REMOVE GPO $X$ $\rightarrow$ REMOVE Key $A[k]$
  if no "REMOVE Key $X[k]$" and $X \notin TL$
```

Sub-rule (a) guarantees that masking values already exist in the list before "ADD Key" operation introduces insecure values. For to-be-added GPOs, its insecure values will take effect only after the GPO is added, so their masking values need only be added before the GPO is added. This is covered in Rule 4. But it is insufficient for to-be-moved GPOs, since the GPO is in the original list, and a new value will take effect immediately when it is added. So the masking values should also be added before adding $A[k]$. Note that Rule 3 guarantees that before $Last_{TL}[k]$ is added or set, it already contains all its secure values.

Sub-rule (b) is the same as the rule for "ADD Key", except that if $A = Last_{OL}[k]$, the masking values only need to be added before the setting of $A[k]$, since $A[k]$ is secure and it can appear in intermediate states.

In sub-rule (c), if $A \neq Last_{OL}[k]$, $A[k]$ is a insecure value and it masks no other values in OL. So it can be safely removed before GPO_A is moved without exposing any insecure values.

But if $A = Last_{OL}[k]$, $A[k]$ is a secure value, and it may mask some insecure values in OL. So before removing $A[k]$, we should make sure that a new secure value, if exists ($Last_{TL}[k] \neq NULL$), is already in the IL to mask these insecure values. So "MOVE/ADD GPO $Last_{TL}[k]$" should be executed first. But there can be cases that in TL, $Last_{TL}[k]$ has lower precedence than A, and $Last_{TL}[k][k]$ cannot mask the to-be-removed k values[3] defined in GPOs between $Last_{TL}[k]$ and A. To prevent them from being unmasked, these to-be-removed values should be removed before the removal of $A[k]$. Note that since processed GPOs always have higher precedence than unprocessed and to-be-removed GPOs, $Last_{TL}[k]$ can mask all the other insecure values of k.

Consider the example demonstrated in GPO Setup 3,4,5. $A = Last_{TL}[i]$ but it has the lowest precedence in TL, so although $A[i]$ takes precedence, it cannot mask $B[i] = 2$ in the intermediate state. In this case, $B[i] = 2$ should be removed before $C[i] = 3$ is removed.

It is also possible that there is no $Last_{TL}[k]$ ($Last_{TL}[k] = NULL$) since all settings for key k are removed in TL, and $TS[k] = undefined$. In this case, we guarantee that all the insecure values masked by $A[k]$, including the values to be removed by "REMOVE Key" and "REMOVE GPO" operations, are removed before the removal of $A[k]$.

The situation that $A = Last_{TL}[k]$ is already covered in Rule 3.

The last rule is for "REMOVE GPO" operations. It guarantees that before executing the "REMOVE GPO" operations and their associated "REMOVE Key" operations, either there are new secure values in the list, or all the insecure values originally masked by the to-be-removed GPO are removed. "ADD/MOVE Key" operations are covered in

[3] If these value are not to-be-removed, they would override the current $Last_{TL}[k]$.

Rule 1.

```
6. For each $GPO_A$ in $OL$ but not in $TL$, and $A = Last_{OL}[k]$,
   IF $Last_{TL}[k]$ exists,
     MOVE/ADD GPO $Last_{TL}[k]$
       $\rightarrow$ REMOVE $GPO_A$
   IF $Last_{TL}[k]$ does not exist,
   for each GPO $X <_O A$ that sets key $k$,
     REMOVE Key $X[k]$ $\rightarrow$ REMOVE $GPO_A$;
     REMOVE GPO $X$ $\rightarrow$ REMOVE $GPO_A$
     if no "REMOVE Key $X[k]$" and $X \notin TL$;
```

This is similar to the "REMOVE Key" case in Rule 5. If $A = Last_{OL}[k]$, $A[k]$ may mask key k's insecure values, which should be removed before the removal of $A[k]$. We do not need to consider the case that $A \neq Last_{OL}[k]$ independently, since it does not mask any insecure value and the presence of A in the list does not affect the value of key k.

4.2 Extension to Support Filters

In the previous discussion, we assume that no filter is used. In this section, we extend our proposed algorithm to support filters. A filter defines a scope consisting of users and machines. When a filter is linked to a GPO, only the users and machines in the scope of the filter can apply to the GPO. Without filters, each GPO is applied to all the objects (users and machines) in the OU by default. In a real environment, filters are widely used to limit the effecting scope of each GPO, providing further flexibility in management. For example, the following filter limits a GPO's targets to only computers running Windows 2000 Server:

```
Select * from Win32_OperatingSystem
where Caption = "Microsoft Windows 2000
Server"
```

The global scope in a domain is divided into a set of *disjoint scopes*, each of which contains one or more objects. A filter consists of a set of disjoint scopes. A GPO can be linked with at most one filter,[4] which limits the GPO to apply to only the objects within the filter's scopes. If a GPO is not linked with any filters, it has a global scope and it would be applied to all the objects in the OU by default.

Accounting filters in our model introduces a new kind of update operations: "LINK Filter" links a filter to a GPO, replacing its original filter if exists; "UNLINK Filter" removes the filter linked to a GPO; "MODIFY Filter" changes the scope of a filter. These operations also have corresponding atomic APIs in the group policy framework.

There are additional HBRs among filter related operations, and between filter related operations and GPO list/key operations.

GPO Setup 6

Key / **Filter**	GPO_A / NULL	GPO_B / NULL	GPO_C / NULL	**State**
S_x key i	1	2	3	3
S_y key i	1	2	3	3
S_z key i	1	2	3	3

GPO Setup 7

Key / **Filter**	GPO_A / NULL	GPO_B / F_r	GPO_C / F_s	**State**
S_x key i	1	2		2
S_y key i	1		3	3
S_z key i	1			1

For example, there is an original state described in GPO Setup 6. Initially, no filter is linked to the GPOs, so the GPOs are applied to all the scopes (S_x, S_y, S_z). According to the preferences of the GPOs, each object in the OU receives "$i = 3$". Now suppose we need to perform the following operations:

(a) LINK Filter F_r to GPO_B, to limit its scope to S_x (*e.g.,* only Windows XP machines)

(b) LINK Filter F_s to GPO_C, to limit its scope to S_y (*e.g.,* only Windows Vista machines)

Scope S_x and S_y are disjoint, and the rest of the objects in the OU constitute scope S_z. The target state is shown in GPO Setup 7.

Consider the order in which we execute these two operations. If filter F_s is linked to GPO_C before filter F_r is linked to GPO_B, we would get an intermediate state in which the objects in S_z have GPO_A and GPO_B applied, and receive an insecure value "$i = 2$".

When we consider filter operations and list/key operations together, the cases are much more complicated because there are more possible combinations.

We first define three rules to identify HBRs among filter related operations.

```
7. MODIFY Filter $F_r$ $\rightarrow$ LINK Filter $F_r$.
```

A filter should be updated before it is linked to a GPO, preventing insecure values in the GPO to be introduced to the scopes defined in the filter's old value. For example, in GPO Setup 8, there are three disjoint scopes: S_x, S_y, S_z. Initially, no filter is linked to GPO_A so it is applied to all

[4]This is consistent with the existing group policy framework, and more importantly, it is unnecessary to use multiple filters. Given that the OU hierarchy already provides coarse-grained scope control, a single filter with expressive specification language *e.g.,* SQL in the group policy framework, is flexible enough to meet the need for fine-grained scope control. Support of multiple filters only increases the complexity of policy management and debugging.

scopes by default; filter F_s is linked to GPO_B to limit its scope to S_x.

GPO Setup 8

Key Filter	GPO_A NULL	GPO_B F_s	**State**
S_x key i	1	2	2
S_y key i	1		1
S_z key i	1		1

Now suppose we want to perform the following operations:

(a) LINK Filter F_t (initially specifies S_y) to GPO_B;

(b) MODIFY Filter F_t from S_y to S_z.

If filter F_t is linked to GPO_B before its scope is modified, GPO_B would be applied to S_y in the intermediate state, causing S_y to get an insecure value "$i = 2$" from GPO_B. Therefore, "MODIFY Filter F_t" should be executed before "LINK Filter F_t to GPO_B".

8. `UNLINK Filter` F_r $\rightarrow$ `MODIFY Filter` F_r.

If a filter is to be unlinked from a GPO, it should be updated after unlinking, so as to avoid the GPO from being improperly applied to the scopes defined in the filter's new value.

9. For each GPO with filter F_r linked, and is to be linked with a new filter F_s,
   `LINK Filter` F_s $\rightarrow$ `MODIFY Filter` F_r.

If a filter linked to a GPO is to be overridden by a new filter, the new filter should be linked before the value of the old filter is updated, preventing the new value of the old filter from introducing insecure values.

For example, consider another scenario in which we are going to make a different update on the initial setup in GPO Setup 8. Now the new update operations are:

(a) LINK Filter F_t (initially S_y) to GPO_B;

(b) MODIFY Filter F_s from S_x to S_z.

Now filter F_s is to be modified instead of filter F_t. If F_s is modified before F_t is linked to GPO_B, GPO_B will be applied to S_z, and its insecure value "$i = 2$" will take precedence. So "LINK Filter F_t to GPO_B" $\rightarrow$ "MODIFY Filter F_s".

Now we consider HBRs between filter related operations and list/key operations using a perspective from each disjoint scope.

To a particular disjoint scope, the effect of linking or unlinking a filter to a GPO is equal to that of removing/adding this GPO from/to the list. For example, in GPO Setup 6, initially S_z has GPO_A, GPO_B, GPO_C applied when no filter is linked; linking filter F_r to GPO_B excludes S_z from GPO_B's effecting scope, leaving only GPO_A, GPO_C applied to S_z, just as GPO_B is removed from S_z's list. In contrast, if we then unlink filter F_r from GPO_B, GPO_B would be applied to S_z again, appearing like adding GPO_B back to the list of S_z.

Therefore, each filter operation can be viewed as list operations that add, move or remove GPOs within a number of scopes. If a filter operation extends GPO_A's effecting scope to cover S_x, it is equivalent to an "ADD GPO_A" in S_x; if a filter operation excludes S_x from GPO_A's effecting scope, it equals to a "REMOVE GPO_A" in S_x. In the example shown in GPO Setup 6 and 7, we have:

(a) "LINK Filter F_r to GPO_B" == "REMOVE GPO_B in S_y and S_z";

(b) "LINK Filter F_s to GPO_C" == "REMOVE GPO_C in S_x and S_z".

Now each particular scope has only list operations and key operations, which is the same as the previous model without filters. We can harness the rules discussed previously to identify HBRs for the operations within each scope, and then map the list operations back to the original filter operations to get the real HBRs for the target OU. In the previous example, for S_z, "REMOVE GPO_B" $\rightarrow$ "REMOVE GPO_C", so for the corresponding operations, "ADD Filter F_r to GPO_B" $\rightarrow$ "ADD Filter F_s to GPO_C".

Rules proposed in §4.1 identify all the HBRs within each scope. By applying these rules to each disjoint scope and combining the identified HBRs, we guarantee secure intermediate states for all scopes during the update process.

4.3 Graph-based Update Sequence Discovery

Using the rules discussed in §4.1 and §4.2, we can identify all the HBRs among the update operations. These HBRs can further be used to find a secure update sequence that ensures secure intermediate states. To describe these HBRs, a directed graph $G = (V, E)$ is used. Each vertex represents an update operation. If operation $O \rightarrow P$, we add an edge from O to P.

In this graph, a vertex should "happen before" all the vertices that can be reached from it along the edges. So *Topological Sorting* can be used to find an ordered sequence of operations that conforms to all the HBRs, thus guaranteeing secure intermediate states. But before executing the topological sorting, *Strongly Connected Components (SCCs)* should be eliminated first. A SCC is a set of vertices such that each pair of vertices in this set are reachable from each other. In a SCC, each vertex V should "happen before" all the vertices, including itself, since all of them are reachable from V. Therefore, all the corresponding operations in this SCC should be executed simultaneously, since every possible sequential order of these operations breaks some HBRs. We call this a *circular dependency situation (CDS)*. For example, suppose we have operations O, P, Q in a SCC and "$O \rightarrow P$, $P \rightarrow Q$, $Q \rightarrow O$". No matter what order is used, some rules are always broken. For example, the order "O, P, Q" breaks the rule "$Q \rightarrow O$" and the derived rules "$P \rightarrow O$, $Q \rightarrow P$". To eliminate these SCCS and find a secure update sequence, we propose using *auxiliary GPOs*, which are extra GPOs temporarily added to the list to mask insecure values.

The key to eliminate SCCs is to remove some edges between vertex pairs in the SCCs, namely invalidating some HBRs. It is impossible to remove edges using only the existing GPOs, otherwise the corresponding HBRs are unnecessary and would never exist in the graph. So the possible approach is to introduce auxiliary GPOs with secure values for some keys, and assign them higher precedence to mask the insecure values that would be exposed by invalidating the HBR.

Each HBR exists to prevent insecure values of one or more keys from appearing in the intermediate state. We denote this set of keys that are *"protected"* by the HBR as P. Consider the example in GPO Setup 3 and 4, "REMOVE Key $B[i]$" $\rightarrow$ "REMOVE Key $C[i]$" is used to prevent the insecure value $B[i] = 2$ from occurring in the intermediate state, so it is used to protect key i, and its P set is $\{i\}$.

If we temporarily add a GPO which sets every key in P with a secure value (equal to the value in IS or TS)[5], and give this GPO higher precedence than all the existing GPOs, its values can mask all the insecure values. And obviously it would not introduce any insecure value. In the example shown in GPO Setup 3 and 4, if we add a new GPO_D setting "$D[i] = 1$" and assign it the highest precedence at the very beginning (so the list become A,B,C,D), we can execute the two remove key operations safely in arbitrary order, since the secure value "$D[i] = 1$" always takes precedence and masks any other value of i. At last we can remove GPO_D after all the operations are completed, again without introducing any insecure value.

Using auxiliary GPOs eliminates HBRs, breaks SCCs, and translates the graph into an acyclic one. Then a secure update sequence can be found trivially with a topological sort.

Using auxiliary GPOs inevitably introduces overhead in the update process, since extra operations should be used to add, set, and remove the auxiliary GPOs. The overhead includes longer latency of the update process, additional traffic in the network, and computation overhead in the server. Recall that without using auxiliary GPOs, our algorithm just reorders the necessary update operations, adding no overhead except the negligible computation time. To minimize the overhead of using auxiliary GPOs, we employ a greedy algorithm to find the minimal set of keys that need to be added to the auxiliary GPOs. Basically all SCCs are identified first, and all the edges between vertex pairs in these SCCs are sorted according to the sizes of their P sets. Then the first edge is removed, and the keys in its P set are added to the auxiliary GPOs; if other edges share some keys in the P set with this removed edge, the shared keys can also be safely removed from the P set of those edges; an edge is removed if its P set becomes empty, since the edge is not used to "protect" keys anymore. This process iterates until all the SCCs with more than one node are eliminated.

4.4 Support of Dependent Keys

The previous algorithms assume that all keys are independent. But as we mentioned in §3.1, there are dependent keys which control one particular behavior of the system together. These keys should be updated simultaneously, or the insecure intermediate states of their combination may result in unknown behavior.

Identifying dependent keys requires domain knowledge of the keys (*e.g.,* the meaning and effect of the Group Policy settings), which is beyond the scope of this paper. Basically, two approaches can be used to extract dependent keys: (1) Use controlled experiments to test the effect of the keys; (2) Parse the definition documents.

To update a set of dependent keys simultaneously, we add them to an auxiliary GPO, and set their values either as those in the OS, or as those in the TS, according to the operations to be performed on them. Here we assume there is no such case where only a (nonempty) subset of the dependent keys are defined in the OS or TS, because the combined result with undefined values are unknown. So either all the dependent keys in the set are defined, or all of them are undefined. If all the keys are undefined in both OS and TS, the existing HBRs prevent their values from changing into any defined value during the update process, so we do not need to consider them independently. Otherwise, if all the keys are undefined in OS and defined in TS, we set the values in the auxiliary GPO as those in TS, then these keys are updated simultaneously when the auxiliary GPO is inserted. In contrast, if all the keys are undefined in TS, we set the values as those in OS, so these keys are updated simultaneously when the auxiliary GPO is removed.

[5]If $IS[k]$=$TS[k]$=undefined, the secure value of k cannot mask any insecure value. But note that in this case k can only exist in "REMOVE GPO_A" $\rightarrow$ "ADD Key $A[k]$" or "REMOVE Key $A[k]$" $\rightarrow$ "ADD GPO_A". And since "ADD Key $A[k]$" and "REMOVE Key $A[k]$" can take part in at most one HBR, k should never appear in any SCC.

4.5 Proof of Completeness

Completeness. For any given OL and TL, our algorithm can always find an update sequence that guarantees a secure update process.

Proof. For any given OL and TL, we can always trivially find a set of operations to update OL to TL, which consists of list, key, and filter operations. Our proposed rules can identify all the necessary HBRs among this set of operations to guarantee secure intermediate state during the update. Using the algorithm proposed in §4.3, we can always find a secure update sequence in which each HBR is either satisfied, or its functionality is fulfilled by auxiliary GPOs. This secure update sequence can update OL to TL guaranteeing secure intermediate states.

4.6 Algorithm Complexity

Let K be the number of different keys used in all the GPOs involved, L be the sum of the sizes of OL and TL, F be the number of different filters used. The number of list operations is less than L, since for each GPO there is at most one list operation to ADD, REMOVE or MOVE it. The number of key settings is at most KL, when each GPO sets all possible keys. The largest possible number of key operations is also KL, since there is at most one GPO key operation for each key setting. $(L + F)$ is the largest possible number of filter related operations, since each GPO has at most one corresponding "LINK/UNLINK Filter" operation, and each filter has at most one corresponding "MODIFY Filter" operation.

Calculating $Last_{TL}[k]/Last_{OL}[k]$ for all keys takes $O(KL)$ time. In the worst case, only the first GPO in the list defines key k, and the whole list should be scanned. Applying Rule $1, 2, 3$ takes $O(KL)$ time, since each rule scans all the key operations; applying Rule $4, 5, 6$ takes $O(KL)$ time, since each rule scans all the key settings; applying Rule $7, 8, 9$ takes $O(L(L + F))$ time, since each rule scans all filter related operations $(L + F)$ and each scan goes through the GPO list(L) in the worst case.

Usually $L + F \ll K$, so the complexity to identify HBRs for one scope is $O(KL)$. We further define $M = max$(Number of objects in the global scope, 2^F) which represents the number of disjoint scopes in the worst case. So identifying HBRs for all the scopes takes $O(KLM)$ time.

Finding SCCs and executing topological sorting take $O(V + E)$ time, where V is the number of vertices, *i.e.,* the number of update operations, and E is the number of edges, *i.e.,* the number of HBRs. Suppose that the number of edges in the SCCs is E', the greedy algorithm of finding the minimal set of keys for auxiliary GPOs takes $O(E' \lg(E'))$ with a sorting. In the worst case, $E' = (L + F + K)^2 \approx K^2$, when each pair of operations has a HBR, and all the vertices are reachable from each other in the graph. This is very rare. And since circular dependency situations are also rare, $O(E) \approx O(V)$. So overall the algorithm takes approximately $O(KLM)$ time. We demonstrate later that the runtime overhead is negligible compared to the time used to deploy the update to the production environment.

4.7 Implementation

The implementation is divided into three parts, which are written in Perl with about 4,000 lines of code. In the center is the Core Engine which implements the algorithm proposed in §4. It identifies all the HBRs from the input using the defined rules, and generates a secure update sequence. The input for the core engine includes: OL, TL, key settings of each GPO, key operations, filter settings, GPO to filter links, filter operations and sets of dependent keys. The core engine is platform independent, and it can be used to find secure update sequence in other platforms using a similar model.

To integrate the core engine into the group policy framework, we implement a GPO Parser, which can either dump GPO settings from the domain controller or parse the settings from the XML backup files. The other input for the core engine, such as the TL, key update operations, and filter operations, are provided by administrators, usually in the form of backup files retrieved from the Group Policy Management Console.

The last part is a Script Generator that translates the secure update sequence into Powershell [11] scripts, which invoke GPO APIs to perform updates automatically.

5 Evaluation

In §4.6 we analyzed the runtime complexity of our algorithm, and argued that it is negligible compared to the deployment time; in §4.3 we mentioned that using auxiliary GPOs inevitably introduce overhead. In this section, we demonstrate that the computation overhead is negligible and the overhead of auxiliary GPOs is small in common cases, using test cases generated according to tuned parameters. These parameters control the complexity of GPOs setups, varying from common cases to extreme cases.

5.1 Time Overhead Quantification

We first quantify the overall deployment time and the insecure time. Suppose a deployment task consists of a sequential list of n operations: $O_1, O_2, \ldots, O_n$, which would be executed one by one. Let t_i be the time needed to execute O_i (to deploy the update) and w_i be the interval between O_i and O_{i+1}. Then the time needed to deploy the task is: $t_1 + w_1 + t_2 + w_2 + \cdots + t_n$.

If the intermediate state after O_i is insecure, the system would stay insecure at least during the period of $w_i + t_{i+1}$, until O_{i+1} is completed and can probably bring the system back to a secure state. Suppose we have a set S that each operation $O_i \in S$ leads the system to an insecure state after O_i, the overall time that the system stays insecure during the deployment would be $\sum\{w_i + t_{i+1}, \text{ for each } O_i \in S\}$.

5.2 Evaluation Metrics and Testbed Setup

We generate test cases of different update scenarios and deploy them on a real server to measure:

1. Computation time of our algorithm (*sec-exec*);
2. Time to deploy the update to the production environment using a random update sequence, simulating that by default, administrators or management tools do the update in arbitrary order (*rand-overall*), and the corresponding time that the system stays in insecure states (*rand-insec*);
3. Time to deploy the update using the secure update sequence generated by our algorithm (*sec-overall*), and the corresponding time that the system stays in insecure states (*sec-insec*), which is always 0.

For each test case, we generate a powershell [11] script to initialize the original state on the server. Then we run our algorithm and a random algorithm, generating a secure update sequence and a random update sequence, respectively. Each update sequence is executed on a simulator first. This simulator tests whether the intermediate state after each operation is secure, and generates the set S, the operations which lead to insecure intermediate state after execution. Update sequences are further translated into powershell scripts, in which each operation is mapped to a corresponding API call to deploy the update to the Domain Controller. Each API call is coupled with a timing command to capture its execution time. The powershell scripts are then executed, and the t_i for each operation is collected. When the operations are executed back to back sequentially in scripts, w_i is so small that it is ignored. At last, combining S and the t_i for each operation, the overall time to deploy the update and the time that the system is insecure during the deployment can be calculated.

The testbed is a Windows Server 2008 R2 RC running on VMWare Workstation 6.5.0, which in turn runs on a Lenovo X200 machine. This virtual machine serves as a Domain Controller, on which we also do the update directly, so the network delay is overlooked. In a real environment where network delay comes into play, the time to deploy the update, as well as the insecure time, will increase, exacerbating the vulnerabilities introduced during the deployment.

Table 2. Parameters for GPO and Key Settings

n = 1..9	sizes of the test cases without filters and dependent keys
num_gpo=10 $\times$ n	# of GPO
num_key=30 $\times$ n	# of different keys
avg_key_use=2	average # that a key is used
stdev_key_use=2	standard deviation of # that a key is used
num_ol=5 $\times$ n	# of GPO is the original list
num_tl=7 $\times$ n	# of GPO in the target list
num_add_key=10 $\times$ n	# of GPO Add key operations
num_rmv_key=10 $\times$ n	# of GPO Remove key operations
num_set_key=10 $\times$ n	# of GPO Set key operations

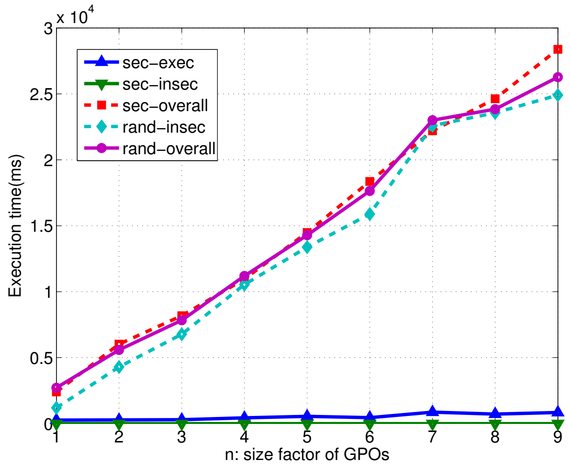

Figure 2. Execution time without filters

5.3 Experimental Results

This section presents our experimental results. For each figure, the X-axis is the size factor which controls the size and complexity of the test cases, and the Y-axis is the time in ms. For each size, 5 different test cases are generated, and the results shown in the figures are average values. For each test case, we measure the execution time of our secure update algorithm (*sec-exec*), insecure time during deployment (*sec-insec*) and overall time of the update process (*sec-overall*) using our secure update sequence, in comparison to the insecure time during deployment (*rand-insec*) and the overall time of the update process using a random update sequence. Each line in the figure depicts the result of one metric.

We first consider the situations without filters and dependent keys. Test cases are generated according to the parameters listed in Table 2.

Table 3. Parameters for Filter Settings

$f = 1..5$	size of test case with filters
num_filter=$8 \times f$	# of filters
num_scope=$4 \times f$	# of disjoint scopes
avg_scope_use=$3 \times f$	average # that a scope is used
stdev_scope_use=2	standard deviation of # that a scope is used
rate_gpo_link_filter=0.5	probability that a GPO has a scope linked
num_link_filter=$3 \times f$	# of link filter operations
num_unlink_filter=$5 \times f$	# of unlink filter operations
num_modify_filter=$5 \times f$	# of modify filter operations

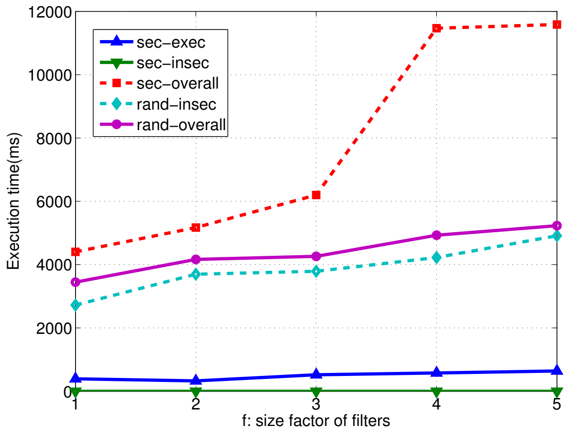

Figure 3. Execution time with filters

The parameter n is used to control the size of the GPO setup in each test case. If t different GPOs set values for the same key k, we say k is used t times. In the test cases, the key usage behavior conforms to normal distribution, with an average value of avg_key_use and standard deviation of $stdev\_key\_use$. These parameters directly affect the number of HBRs. avg_key_use is set to 2 to investigate the effect of key overriding.

The result is shown in Figure 2. The execution time of our algorithm is always negligible (less than 1% of the deployment time), even when the test case is very large. And deployment times using secure (generated by our algorithm) and random (simulating default admin or management tool's action) update sequence are nearly the same, both of which increase almost linearly with the sizes of the test cases. But the random update sequences lead to insecure states during $80-95\%$ of the deployment period, while the secure update sequences eliminate all insecure states completely.

For evaluation with filters, we set $n = 2$ and use the parameters listed in Table 3 to control the filter settings. The result is shown in Figure 3. The X-axis is the parameter f which controls the frequency that filters are used in the test

Table 4. Parameters for Dependent Key Settings

d = 0..9	size of test case with dependent keys
avg_dep_set_size=$1 + 3 \times d$	average size of a dependent key set
stdev_dep_set_size=3	standard deviation of the size that a dependent key set
num_dep_sets=$3 \times d$	# of dependent key sets

cases. The execution time of our algorithm is still small across all test cases.

Using random update sequences (no overhead), the deployment time increases slowly, since all the test cases share the same parameter settings in the number of list and key operations, and only the parameters related to filters are changed according to f. In contrast, the deployment with secure update sequence incurs perceivable overhead, due to the auxiliary GPOs for addressing circular dependencies. We observe considerable number of circular dependencies in the test cases, especially in the extreme cases 4 and 5 with a large number of filters, scopes and overlapping scopes among filters. Filter link/unlink operations for almost every GPO and modifications for almost every filter also contribute to more circular dependency situations. As a result, the overhead in test cases 4 and 5 is around 130%. These can be viewed as an approximate upper bound using filters, since the largest possible filter settings for the given GPO/key setups are used. In a real environment, the numbers of filters and scopes are limited to preserve manageability [27] and performance [28]. Consequently, in common cases, the overhead is around 30%, which is acceptable.

Note that when filters are introduced, the system suffers from even longer insecure periods (85%-96%) using random update sequences, while the secure update sequences still keep the system in secure states during the entire deployment.

Next we further consider dependent keys in the test cases. Similarly, we set $n = 2$ for the GPO setups, and in order to quantify the overhead introduced by dependent keys, we first set the number of filters to 0. Dependent keys are described with *dependent key sets*, in which the keys depend on each other. So in addition to the parameters in Table 2, three more parameters in Table 4 are used to control the number and sizes of dependent key sets.

The result is shown in Figure 4, and the X-axis is the parameter d, which determines how frequent the dependent keys occur in the test cases. Using random update sequences (no overhead), the deployment time is approximately constant with slight variation, since the number of

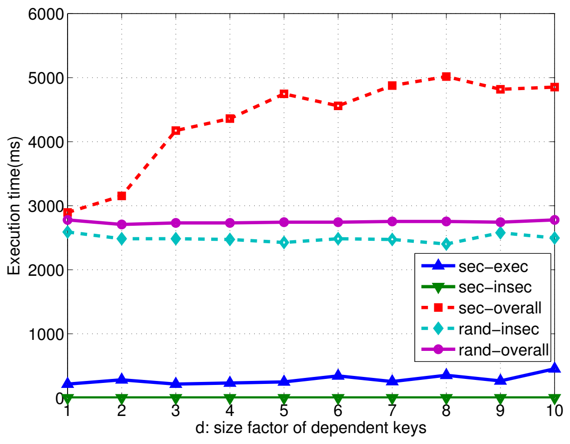

Figure 4. Execution time with dependent keys

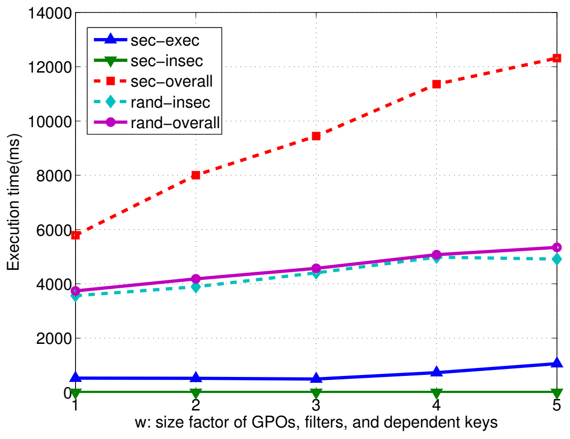

Figure 5. Execution time with dependent keys & filters

update operations is not affected by the size of dependent key sets. But almost the entire deployment period ($> 90\%$) is insecure for each case. With secure update sequences, the deployment time first increases with d and then saturates when d is large enough. In these extreme cases, almost every key belongs to one or more dependent key sets and thus should be added to the auxiliary GPOs to preserve secure intermediate states.

But according to our experience on extracting dependent keys from Group Policy framework by parsing the official definition documents, less than 100 out of 3000 keys are identified as dependent keys. Although there could be false negatives in our approach, the percentage of dependent keys should still be small, since these internal dependencies complicate the policy design process and make it more error-prone. Consequently the overhead of our algorithm is small in normal cases.

At last we combine filters and dependent keys to evaluate the overhead of using secure update sequences in the worst case. The result is shown in Figure 5. n is still set to 2, and the X-axis is w, which controls the parameters for filters and dependent key sets ($f = d = w$) at the same time. In the worst case when $w = 5$, the overhead is around 130%, which is the upper bound of using secure update sequences. Note that these cases are extreme and rare, and the overhead is much smaller in real scenarios.

In summary, we demonstrate that naive random update sequences put the system into insecure states 80%-95% of the time. The secure update sequences, generated by our algorithm with negligible computation overhead, eliminate all insecure states. Without considering filters and dependent keys, circular dependency situations are rare, and the secure update sequences add nearly no overhead to the deployment. In contrast, considering filters and dependent keys results in a considerable number of circular dependency situations, which require using auxiliary GPOs to prevent insecure states. This leads to deployment overheads of less than 30% in common cases and an upper bound of 130% in extreme cases.

6 Conclusion

In this paper, we demonstrate that unsophisticated approaches to enterprise policy deployment will create insecure intermediate states on client machines, leading to potential security vulnerabilities. Unfortunately no existing mechanisms can ensure safe deployment for enterprise policy deployment. To address this problem, we propose a model generalized from the Group Policy framework, define the concept of secure intermediate state, and present an efficient algorithm to find secure update sequences, based on the idea of identifying "happen before" relationships between update operations. Our algorithm relies only on existing interfaces provided by the underlying policy management platform to achieve transactional safety guarantees, without requiring any modifications to the platform itself. The rules we used are proven to be complete in identifying all the possible "happen before" relationships. Our evaluation shows that the computation time of our algorithm is negligible compared to the deployment time, and the secure update sequences generated by our algorithm eliminate all insecure states while maintaining an acceptable overhead in common cases.

Although we focus our discussion on Group Policy platform, our model is general enough to be easily ported to other policy model to solve similar deployment problems. For example, it can be trivially used for firewall policy deployment, since the firewall model is a special case of our model. Extending our model to other policy management frameworks will be our future work.

Acknowledgments

We gracefully acknowledge the feedback we received from the anonymous reviewers. We appreciate the inspiring discussions and comments from Feng Qian, Zhiyun Qian, Qiang Xu, Yunjing Xu and Xinyu Zhang. This work was supported in part by NSF CNS-0643612, DARPA, and Department of Army.

References

[1] E. Al-Shaer and H. Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *Integrated network management VIII: managing it all: IFIP/IEEE Eighth International Symposium on Integrated Network Management (IM 2003), March 24-28, 2003, Colorado Springs, USA*, page 17. Kluwer Academic Pub, 2003.

[2] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, 2004.

[3] E. Al-Shaer and H. Hamed. Modeling and management of firewall policies. *IEEE Transactions on Network and Service Management*, 1(1):2–10, 2004.

[4] F. Baboescu and G. Varghese. Fast and scalable conflict detection for packet classifiers. *Computer Networks*, 42(6):717–735, 2003.

[5] L. Bauer, S. Garriss, and M. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM SACMAT*, pages 185–194. ACM New York, NY, USA, 2008.

[6] M. Casado, M. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 1–12. ACM New York, NY, USA, 2007.

[7] W. S. T. Center. Active directory domain services. http://technet.microsoft.com/en-us/library/cc770946.aspx.

[8] W. S. T. Center. Group policy objects. http://technet.microsoft.com/en-us/library/cc775691(WS.10).aspx.

[9] W. S. T. Center. What's new in group policy. http://technet.microsoft.com/en-us/library/dd367853(WS.10).aspx.

[10] W. T. Center. Group policy planning and deployment guide. http://technet.microsoft.com/en-us/library/cc754948%28WS.10%29.aspx.

[11] M. Corporation. Windows powershell. http://www.microsoft.com/windowsserver2003/ technologies/management/powershell/default.mspx.

[12] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, pages 18–38, 2001.

[13] N. Dulay, E. Lupu, M. Sloman, and N. Damianou. A policy deployment model for the Ponder language. In *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings*, pages 529–543, 2001.

[14] K. Forster. Windows & .net magazine - windows active directory/group policy study. http://windowsitpro.com/article/articleid/44239/complete-results-of-the-group-policy-survey.html.

[15] H. Hamed and E. Al-Shaer. Taxonomy of conflicts in network security policies. *IEEE Communications Magazine*, 44(3):134–141, 2006.

[16] B. Hicks, S. Rueda, L. Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the 12th ACM SACMAT*, pages 91–100. ACM New York, NY, USA, 2007.

[17] T. Jaeger, X. Zhang, and A. Edwards. Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):327–364, 2003.

[18] Z. Kerravala. Configuration management delivers business resiliency. *The Yankee Group*, 2002.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. 1978.

[20] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. *ACM Transactions on Information and System Security (TISSEC)*, 9(3):259–291, 2006.

[21] P. Norvig. *Artificial intelligence: a modern approach*. Pearson Education, 2003.

[22] X. Ou, S. Govindavajhala, and A. Appel. MulVAL: A logic-based network security analyzer. In *14th USENIX Security Symposium*, 2005.

[23] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[24] SELinux. Selinux project wiki. http://selinuxprojet.org/page/Main_Page.

[25] Y. Sung, S. Rao, G. Xie, and D. Maltz. Towards systematic design of enterprise networks. In *Proceedings of the 2008 ACM CoNEXT Conference*. ACM New York, NY, USA, 2008.

[26] M. S. TechCenter. How core group policy works. http://technet.microsoft.com/en-us/library/cc784268.aspx.

[27] M. Tulloch. Best practices for designing group policy. http://www.windowsnetworking.com/articles_tutorials/Best-Practices-Designing-Group-Policy.html.

[28] M. Tulloch. Optimizing group policy performance. http://www.windowsnetworking.com/articles_tutorials/Optimizing-Group-Policy-Performance.html.

[29] D. Verma, I. Center, and Y. Heights. Simplifying network administration using policy-based management. *IEEE network*, 16(2):20–26, 2002.

[30] H. Wang, S. Jhat, M. Livny, and P. McDaniel. Security policy reconciliation in distributed computing environments. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings*, pages 137–146, 2004.

[31] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE Symposium on Security and Privacy*, pages 199–213. Citeseer, 2006.

[32] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and Analysis. In *IEEE Symposium on Security and Privacy*, pages 199–213, 2006.

[33] C. Zhang, M. Winslett, and C. Gunter. On the Safety and Efficiency of Firewall Policy Deployment. In *IEEE Symposium on Security and Privacy*, pages 33–50, 2007.