Managing Access Control Complexity Using Metrics

Trent Jaeger
IBM T. J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532, USA
Email: jaegert@watson.ibm.com

Abstract

General access control models enable flexible expression of access control policies, but they make the verification of whether a particular access control configuration is safe (i.e., prevents the leakage of a permission to an unauthorized subject) difficult. The current approach to expressing safety policy in such models is to use constraints. When the constraints are verified, then the configuration is verified to be safe. However, the addition of constraints to an access control expression significantly increases the complexity of the expression, so it quickly becomes difficult to understand the access control policy expressed in the model such that future changes can be made correctly. We propose an approach whereby the complexity of each access control configuration is estimated, so the administrators can see the effect of a configuration change on the future ability to maintain the configuration. We identify metrics for making complexity estimates and evaluate these metrics on some constraint examples. Our goal is to enable the use of flexible access control models for safety-critical systems by permitting limited use of constraints that do not complicate the configuration beyond a maintainable complexity.

1 Introduction

In access control modeling, there is an inherent conflict between the need for flexibility in modeling the access control policy and the need for restricting the model in order to verify the safety of the access control policy. First, early work in access control focused on the design of access control models that enable the expression of arbitrary access control policies, such as Lampson's *access control matrix* [12]. Using an access control matrix, the access control policy is expressed as the operations that subjects (i.e., actors) can perform on objects. Each different policy that can be expressed in such a model is called a *configuration* of the model. Recent models, such as role-based access control (RBAC) [16] and Dynamically Typed Access Control (DTAC) [20], permit such general policy expression, but also introduce concepts to enable the aggregation of statements to reduce the system administrators effort.

Second, an important issue in the design of a configuration is whether the configuration enforces the *safety policy* that was intended in its design. That is, does the configuration ensure that subjects can only obtain the rights that are intended for them and no others. Unfortunately, the model described above is not sufficient to ensure that the safety of a configuration can be proven [8]. Therefore, two approaches have been taken: (1) the safety of the model is guaranteed for each configuration or (2) the safety of each configuration change is verified against an explicit safety policy. In the first case, access control models must be limited to ensure that each configuration is safe. This restriction has been achieved by reduction of expressive power (e.g., *take-grant* model [6]), limiting the means by which configuration changes are possible (e.g., the fixed models and trusted administration of Domain and Type Enforcement [7] and Bell-LaPadula [4]), or through models with subtle restrictions on how policy can be expressed (e.g., SPM [3] and TAM [2]. The most successful approach has been the second type because the type of policies and administrative models have been useful in some context. Interestingly, the third type of models have more expressive power, but were not applied in practice because the subtleties of the restrictions were difficult for administrators to understand and enforce.

In models of the second type, an additional concept called *constraints* is introduced to enable the safety of the configuration to be verified. Safety is verified by checking that each change in access control configuration does not violate any of the safety constraints. While the notion that a constraint checking can enforce safety is a theoretical solution to the

safety problem, it leaves system administrators with difficult practical problems. First, a system administrator must be able to express the system's safety properties correctly in terms of constraints. Traditionally, constraint languages are based on predicate logic [1, 11, 5]. While these languages enable great flexibility in the expression of constraints, it is quite unlikely that typical system administrators will be able to express more than a few simple constraints properly. In addition, the system administrators must also maintain the system as in evolves. This means that constraints may need to be added or removed, and the result must reflect the intended safety policy.

We believe that if system administrators can better understand and manage the complexity of their configurations and constraints, then such flexible access control models can be used in high security environments with non-trivial safety requirements. However, little attention has been paid to developing an understanding of the complexity of access control models. In this paper, we propose and evaluate metrics for access control model complexity. Our goal is to find metrics that correlate reasonably well with our intuitive notions of complexity. Given such metrics, we envision that user studies can be done to determine the level of complexity that can be managed effectively by system administrators. System administrators then will have a basis for balancing the complexity of their configuration and constraints to determine whether a safety policy can be enforced and maintained. We expect that this will result in the ability to use more flexible access control models in high security environments.

In Section 2, we first examine the possible options for a workable definition for complexity. Next, in Section 3, we define the access control model that we will use to express configurations and constraints. In Section 4, we specify a series of example separation of duty constraints each of increasing complexity to use to compare the effectiveness of different complexity metrics. In Section 5, we define a basic approach to complexity measurement and define complexity metrics. In Section 6, we apply and evaluate the effectiveness of different metrics. In Section 7, we summarize and examine future work.

2 Defining Complexity

First, we need to identify a useful definition for configuration complexity. By useful, we mean that such a definition must be representative of some real task of the system administrators and be practically measurable.

An access control configuration is used for two purposes: (1) authorization and (2) safety enforcement. Primarily, we want to use the configuration to authorize requests. Secondarily, we want to verify that the current access control configuration does not leak any permissions to any unauthorized subjects. System administrators must understand a configuration both to modify the authorization state (i.e., subjects, permissions, and authorization types) and verify that the safety requirements are upheld (i.e., verify that the assignment of authorization types and permissions to subjects satisfy the configuration's constraints). Therefore, the complexity of safety enforcement depends on the expression of both the authorization state and safety requirements.

The complexity of an access control configuration impacts system administrators when they want to modify an access control configuration. They must understand the current configuration sufficiently to make the desired modification while ensuring that the authorization and safety requirements are correct. Since different modifications are possible, in theory, complexity is the difficulty of understanding the configuration necessary to make any modification. Therefore, one possible definition of configuration complexity is the sum of the complexity of any possible modification to that configuration. This depends on the complexity of understanding the effect of a change on the authorization and safety verification behavior of the configuration before the change. While for each modification we may only need to consider local effects, the configuration complexity must take all changes into account.

Therefore, we define configuration complexity to be the complexity of understanding the authorization and safety verification operations. If system administrators understand exactly what happens in each authorization and safety verification operation, then presumably they can understand the effect of making a change. Then, the complexity of the new configuration depends on the complexity of authorization and safety computation in the new configuration.

Therefore, our goal is to identify metrics to measure the complexity of understanding the concepts in an access control configuration. First, we describe an access control model to describing the access control concepts. This model makes explicit the information that must be computed and the interactions between concepts that make the model complex. Second, we identify some examples that can be used to give us an expectation of the accuracy of the complexity metrics. We then develop complexity metrics for understanding each concept in the model and apply these metrics to the examples. The degree to which the combination of different complexities accumulate is not obvious, so we experiment with a few combina-

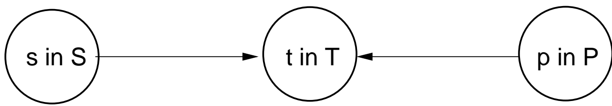


Figure 1: The basic access control model consists of subjects, permissions, and authorization types (i.e., authorization relationships, such as roles).

tion approaches for different combinations.

3 Access Control Concepts

In order to develop and test metrics for access control configuration complexity, we must have some canonical access control model in which configurations can be described. This model must include: (1) the basic access control concepts, such as subjects, objects, permissions, and authorization types; (2) some common extension concepts, such as aggregation and inheritance; and (3) a means for ensuring that the safety of a configuration can be guaranteed: constraints. This model is motivated by the graph role model work of Osborn *et al* [14, 15].

3.1 Basic Access Control Model

We use the notation of the dynamically typed access control model (DTAC) to express access control relationships. DTAC defines an authorization relation t (i.e., authorization type or role) as a data type with three functions: (1) $S(t) = \{subjects\}$; (2) $P(t) = \{permissions\}$; and (3) $N(t) = name$. In this case, a type represents an authorization relationship between sets of subjects and permissions (i.e., objects and the operations that may be performed on them). Further, permissions may be decomposed when desired into their constituent objects and operations, $O(p)$ and $Op(p)$, respectively.

Note that many access control models are isomorphic to the DTAC model at this level of abstraction. For example, roles in role-based access control models are also authorization relations with the same functions. Even multilevel security can be expressed using this model when we view the authorization relationship as a level.

Visualization of an access control policy is often useful in understanding it. Abstracting the DTAC model defined above, we get a graph as shown in Figure 1, in which elements of the set S are assigned to elements of the set T, and elements of the set P are also assigned to elements of the set T (the assignments are many-to-many).

Note that we are often also interested in the propagation of assignments across the authorization relationship. For any subject $s \in S$, we would want to determine the permissions available to that subject, $P(s) \equiv \bigcup_{t \in T(s)} P(t))$. Similarly, we can determine the subjects to which a particular permission is available. In general, any assignment to an authorization relation can be propagated to concepts on the opposite side of the relation (e.g., $O(s)$ and $S(o)$).

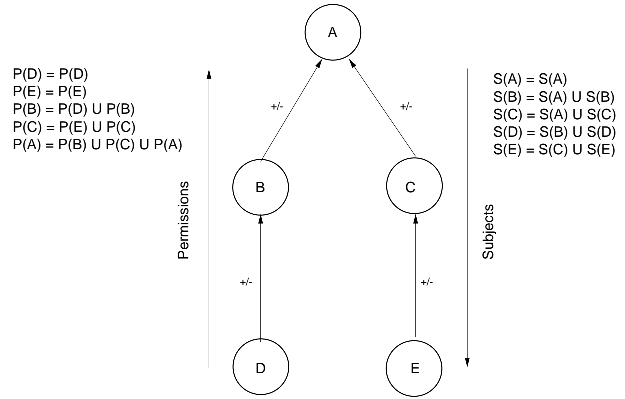


Figure 2: In an inheritance relation, subjects are aggregated in an inverse direction to permissions.

For simplicity's sake, we do not extend the model further. The full model described in [10] also includes: (1) the decomposition of permissions into objects and operations; (2) the decomposition of subjects into sessions and processes; and (3) the collection of authorization current state and history.

3.2 Aggregation and Inheritance

Aggregation applies to any system concept by generalizing each concept slightly to support a value $X(x) = \{x_1, x_2, ..., x_n\}$ where X is the concept type and x and $x_1, x_2, ..., x_n$ are instances of that type. Thus, the concepts which we used before were simply singleton sets. Note that an aggregation relationship in the model will be denoted by a '+'.

There are two types of computations available on aggregations: (1) summations over all members and (2) iterations over each individual members. The semantics of the summation concept functions for aggregations are: $f_s(A) = \bigcup f(a_i)$ for all $a_i \in A$. That is, the same functions of an aggregate union the values for the constituents of the aggregate. Since we look at each node as a set, we use this as the default semantics, so the subscript for summation does not need to be expressed.

Also, we can apply a function once for every member of an aggregation, like an iterator operator. The semantics of iterator concept functions are: $f_i(A) = \{f(a_i)\}$ for each $a_i \in A$. Iteration is explicit in each constraint.

Unfortunately, the semantics of the aggregation relation does not effectively cover all the types of concept grouping in the model. In particular, the functions in authorization types do not behave as summation and iteration functions in all cases. Another relation, called *inheritance*, signified by a '+/-', by the direction in which information is transferred by the inheritance relationship. As shown in Figure 2, permission information (e.g., permissions, operations, and objects) is aggregated in the direction of inheritance relationship (the '+'), but subject information is aggregated opposite to the direction of the inheritance relationship (the '-'). Given the direction of the aggregations, the aggregation computation semantics still hold.

3.3 Constraint Model

The approach we advocate for safety verification is to define an initial configuration of authorization relationships and to place constraints that limit the ways that the configuration can be modified. The constraints are to ensure that the authorization relationships do not grant an unauthorized permission (i.e., are safe).

Since we define our basic model using sets, the natural way to define constraints is as binary relationships between pairs of sets. We chose to limit ourselves to binary relationships for two major reasons: (1) they are easy to describe and draw as labelled edges in a two-dimensional graph, which we hope makes them easier to understand and (2) they are simpler and more compact than ternary (or higher) relationships so the algorithms and data structures are more efficient. In addition, our initial investigations demonstrated that many common constraints can be expressed using only binary relationships [18, 19, 10].

In addition to their algorithmic benefits, we believe that the minor loss of expressive power of binary relationships versus ternary relationships is beneficial to modelling: it simplifies the construction of a fixed point, that point necessary to construct constraints on constraints.

There are two broad categories of constraints. The first is based around the notion of subsets and set equality; thus for example, we have test for *equality* ($=$), *subset* ($\subset$), and *not subset or equal* ($\not\subseteq$). In addition to the standard subset operators we define two sets to be *incomparable* ($\not\sim$) if neither is a subset of the other (except in the degenerate case in which one is empty).

$$A \not\sim B \stackrel{\mathsf{def}}{=} (A \not\subseteq B) \wedge (B \not\subseteq A) \vee (A = \emptyset) \vee (B = \emptyset)$$

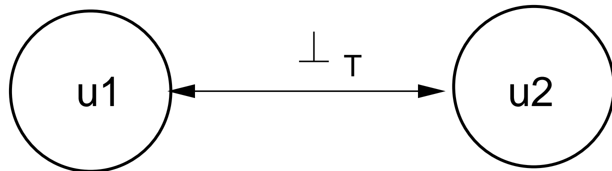


Figure 3: The graphical representation of a *user-user conflict separation of duty* constraint. Subject $u1$ may not be assigned any authorization type to which subject $u2$ is assigned and vice versa. That is, their type sets must have a null intersection.

The second is based around the notion of overlap between two sets when neither is necessarily a subset of the other, and is defined by limiting *maximal cardinality* of their intersection; so we write $|A \cap B| \leq n$ for two sets A and B. The notion of two sets having no overlap, which we refer to as being *disjoint*, is so common that we give it a special symbol ($\perp$), and write $A \perp B$ for $|A \cap B| = 0$.

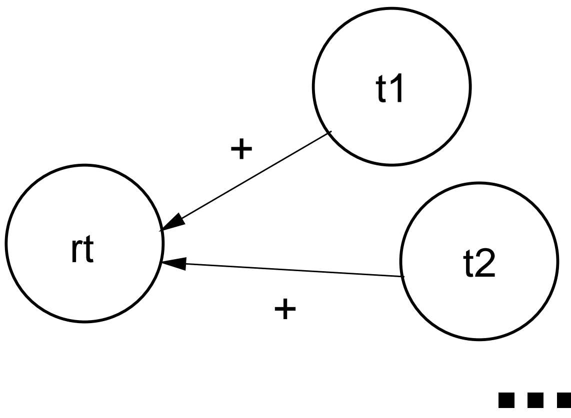
It is frequently convenient to denote the application of the same function to both sides of a constraint operator by subscripting the operator with the function name. Thus instead of $P(A) \perp P(B)$ we may write $A \perp_P B$. The most common usage of this is apply to constraints to the objects assigned to a node (subscripted O), the permissions held by a node (subscripted P) or the types assigned to a node (subscripted T).

4 Complexity Examples

In order to get an intuitive sense of complexity such that we can evaluate the various metrics that can be proposed, we define a series of constraint examples representative of various incarnations of user-user conflict separation of duty constraints [17, 14]. In general, such a constraint restricts subjects from sharing a common authorization type. Since the assignment of authorization types implies the assignment of permissions, a user-user conflict may also be stated in terms of the permissions that a user can be assigned. Below, we provide the examples and discuss the intuitive complexity of each.

Example 1 In a user-user conflict separation of duty constraint, it is forbidden for two users to both be assigned to any common authorization type. This constraint is enforced by requiring that the authorization type sets of the two users be disjoint as shown in Figure 3, $T(u1) \perp T(u2)$. Also, using our compressed notation we write $u1 \perp_T u2$.

In the second example, the set of authorization types to which common assignment is prohibited is limited. The complexity of this constraint is similar



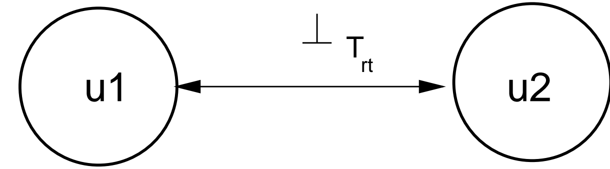


Figure 4: The graphical representation of yet another alternative interpretation of the *user-user conflict separation of duty* constraint. In this constraint, $u1$ and $u2$ must be restricted from sharing an authorization type in the set of restricted types rt. That is, the set of authorization types in rt that are shared between $u1$ and $u2$ must be null.

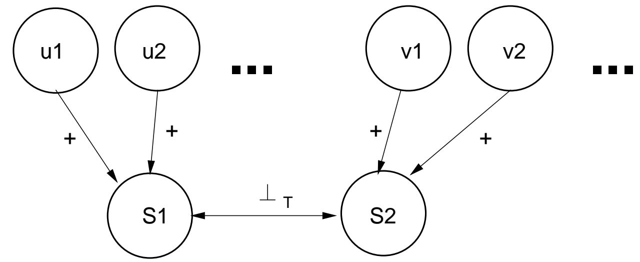


Figure 5: The graphical representation of an alternative interpretation of the *user-user conflict separation of duty* constraint. In this constraint, no subject in $S1$ may assigned to an authorization type to which a member of the $S2$ is assigned. That is, the set of shared authorization types between these groups of subjects must be null.

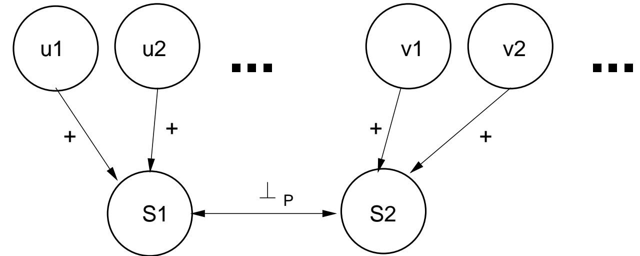


Figure 6: The graphical representation of yet another alternative interpretation of the *user-user conflict separation of duty* constraint. In this constraint, $S1$ and $S2$ must be restricted from sharing any permission. That is, the set of permissions that are shared between $S1$ and $S2$ must be null.

to Example 1, but we claim that the need to understand the restricted set of authorization types makes this constraint slightly more complex.

Example 2 In another interpretation, we want to restrict two users from a particular authorization type or from sharing one authorization type from a set of types. In this constraint the restricted authorization types are grouped in an aggregate named restricted types rt. Then, a constraint is made between the two users (or subject aggregates as above), $u1 \perp_{T_{rt}} u2$ as shown in Figure 4. This constraint checks for a null intersection between the types of the two users that are within the restricted types.

In a third example, the number of users to which a conflict is possible may be greater than 1. Clearly, this is more complex than either of the first two constraints.

Example 3 We consider an alternative interpretation of the user-user conflict constraint expressed in Example 1 in which the two sets of users are restricted from being assigned to any common authorization type. This constraint is shown graphically in Figure 5. The users $u1, u2, ...$ and $v1, v2, ...$ are assigned to two separate subject sets $S1$ and $S2$. There is a disjoint relationship on the types that these two sets may be assigned: $S1 \perp_T S2$. In this case, the constraint is between each of the members of the sets $S1$ and $S2$ since $T(S_i)$ is the union of the values of T for each element in S_i.

In the last example, the user-user conflict is taken to the permission level. That is, rather than restricting the set of authorization types, we restrict the set of permissions that each user set may obtain. Since the additional issue of how permissions are assigned to authorization types must be understood this constraint is more complex than any of the first three.

Example 4 In another interpretation, we want to restrict the two sets of users from sharing any common permission. Like Example 3, the users are grouped into two sets $S1$ and $S2$ and the permission sets of the users must be disjoint: $S1 \perp_P S2$ as shown in Figure 6.

Intuitively , we state the progression of complexity of these constraints is: $E1 << E2 < E3 << E4$. Example 1 is a very simple constraint. Examples 2 and 3 represent two uses of aggregations, so they should be approximately the same. We give Example 3 a more complex rating because it uses two aggregates.

Example 4 expands the constraint from types to the permissions of types. We expect that this will significantly increase the constraint's complexity.

While this is nowhere near a sufficiently exhaustive sample to validate a complexity metric, our aim in this initial work is to identify promising complexity metrics, not the optimal. We envision that the order and magnitude of complexity differences are significant measures toward this cause.

5 Approach

Previous research has found that people think effectively in terms of *chunks* of information [13]. For example, in programs the use of structured code where short, simple code blocks form programs enable programmers to convert programs into higher level concepts such as reference counters that represent their semantic intent. Also and more importantly, such 'chunked' concepts can be stored in long-term memory for reuse. As the complexity of a particular program increases, the harder it is for the programmers to create useful chunks. Thus, they must re-learn complex each time that they need to use them.

In the context of an access control model, the smallest chunks of information are the nodes in the access control graph. These represent the basic concepts of the access control model: subjects, permissions, and authorization types. However, we imagine that the complexity of chunks is not always equal. Given that different information is stored in each type of access control node, we surmise that their complexity depends on how many types of information must be understood from the node to complete an operation.

5.1 Complexity in Authorization

In order to compute authorization, we need to compute whether a particular subject has a particular permission. The relevant functions are the permission assignment to authorization types $P(t)$ for all $t \in T$ and the assignment of subjects to authorization types $S(t)$ for all $t \in T$. From these functions, we can determine whether one of the authorization types of a subject have the desired permission. Therefore, for authorization, two functions must be understood to perform authorization. Therefore, we can think of the complexity of the node chunks in authorization as the number of functions that must be computed. This has the result that primitive entities, subjects and permissions, have a complexity of zero.

The complexity of authorization is increased by the use of other relationships, such as aggregation and inheritance. For aggregation, the functions authorization types and hence permissions are complicated by the fact that the subject may belong to some aggregates with additional permissions assigned to them. Similarly, inheritance of authorization types also has the potential to increase the number of ways that a permission may be assigned to a subject. Since these relationships have a similar effect to assignment, the complexity of these relationships is approximately the same as one function for individual relationship. Since these relationships introduce new nodes (e.g., aggregates) with additional relationships, the overall increase in complexity is higher.

Note that inheritance along multiple relationships is often perceived to be much more complex than single inheritance. In a graphical model, we can see that limiting a model to single inheritance limits the chunk that must be evaluated to compute permissions to a branch of an inheritance tree. However, the use of multiple inheritance creates chunks that are graphs, so, in worst case, the entire inheritance hierarchy may need to be examined. Thus, the complexity measure for multiple aggregation and/or inheritance relationships may not simply be the number of relationships, but rather it may depend on the number of additional nodes that must be examined or be even greater.

5.2 Complexity in Safety

The other important computation in an access control configuration is that of safety. Safety is computed by verifying that each constraint is satisfied for that configuration. Therefore, understanding safety requires understanding the complexity of computing each constraint. Interestingly, common constraints, such as separation of duty, are comparisons between nodes for which we desire that no common relationship exist. Therefore, constraints often have much greater chunks than other concepts. Also, constraints, in general, are comparisons between two or more sets, so much more computation is necessary to verify a constraint. Lastly, constraints may require the computation of functions that are not necessary for authorization, so additional computations not typically in the model must be made.

Given this intuition, we can see why the use of constraints to enforce safety creates a problem for access control model designers. The desire to provide a means for ensuring safety in an access control model cannot be provided by a tool that is even more error-prone. Unfortunately, constraints, by their very nature, create a significant amount of complexity in the model. Therefore, the use of constraints must be managed to keep the complexity from overtaking the potential value.

Complexity in constraints derives from the arity of the constraint relationship (i.e., the number of sets being compared), the number of concepts involved in the computation, and the number of new functions that must be computed.

The number of nodes upon whom a constraint is applied defines its arity. This is indicated in the graphical access control model by the number of nodes connected by a constraint edge. We consider constraint arity to be a major source of constraint complexity. A data structure in which edges can be between more than two nodes is no longer a graph. In this case, we would have to consider access control models with more complex data structures, such as hypergraphs.

To prevent the need for such data structures, we convert references to other nodes into new concept functions. For example, in Example 2, we compare the authorization types of subjects $u1$ and $u2$ to ensure that they are disjoint to the type set T_{rt}. To verify this constraint, we must compute the functions $T_{rt}(u1)$ and $T_{rt}(u2)$ and determine whether the intersection is null.

This clearly is more complex than a constraint using existing nodes and concept functions, so we must determine how to account for this additional complexity. We again revert back to identifying the chunk. First, understanding of a constraint requires understanding of each of the nodes involved in computing the constraint, including aggregation and inheritance. Therefore, we count the number of concept functions that are included in a computation of the two sets that comprise the constraint comparison.

5.3 Complexity Metrics

For each aggregation and authorization type, we count the following classes of complexity metrics.

- **Authorization Types**: The base complexity for each authorization type is the two functions necessary to compute authorization.
- **Aggregation**: Each aggregation counts as one function that is computed to return the set in the aggregation.
- **Inheritance**: Each inheritance counts as two functions as each authorization type has a new permission assignment and a new subject assignment.

For each class, we apply a function to compute the complexity value by applying a *combination function* to the metric value. The combination function can either be the identity function, a polynomial function of the metric value, or an exponential function of the metric value. Similarly, another combination function can be used to combine the values of all three metrics. The metrics may be combined by addition, multiplication, or exponentiation to give the overall complexity metric per node. Finally, the node metric values are collected. Initially, we will sum the node metrics.

For each constraint, we count the following classes of complexity metrics:

- **Arity**: The number of nodes involved in the comparison (one or two)
- **Functions**: The number of functions that must be computed to verify the constraint
- **New Functions**: The number of functions that are added for the constraint

Again, the metric values for each metric type are collected, and combination functions are applied to these metrics. Then, another combination function is used to combine the combination metrics. Finally, the constraint metric values are summed.

To compute the overall complexity of an access control graph, the constraint metric values are added to the authorization metric values.

Thus, the experiment is to compute the authorization and constraint metric values for the different types, and try different combination functions to see which have the closest relationship to our intuitive expectation.

6 Evaluation

We now measure the complexity of our examples using different combination functions to determine: (1) whether the choice of metrics correspond to our intuition of the relative complexities of the examples and (2) which of the combination functions best corresponds to our intuition of the relative complexities.

In order to perform the experiment, we need to add some further details regarding the examples. We assume a configuration that contains 11 authorization types. We assume a three-level authorization hierarchy with a root type, two second-level types, two sets of four third-level types.

Given this inheritance hierarchy, we have the following measure for the base authorization complexity metrics: (1) 2 points for each of the 11 authorization types; (2) 0 points for aggregations; (3) 2 points for the inheritance functions for two second-level types and 1 point for the root and the third level types. Using the combination function choice of

addition per node, we get 3 points for the first and third-level nodes and 4 for the second-level nodes. Using addition over all nodes, we get 35 points total for authorization complexity. Using the combination choice of multiplication of product by 2, we get $(2(2pts * 2pts * 2nodes) + 2(2pts * 1pt * 9nodes) = 52$. If we use exponentiation combination (i.e., 2^n), then we get $(2^4 * 2nodes) + (2^3 * 9nodes) = 104$.

We now examine the complexity of the configurations that include the constraint examples.

Example 1 In Example 1, we have a simple binary constraint of two functions, $T(u1)$ and $T(u2)$. Therefore, we have an arity of 2 and a function count of 2. No new functions were necessary. Using additive combination, we get a complexity of 4. Using multiplicative combination, we get a complexity of 8 (2 * 2 arity * 2 functions). Using exponential combination, we get a complexity of 16 ($2^{(2+2)}$).

Example 2 In Example 2, we add the notion that the users' authorization types must be disjoint with respect to a set of types T_{rt}. Therefore, we have an arity of 2. Two new functions must be computed: $T_{rt}(u1)$ and $T_{rt}(u2)$. In order to compute these functions, we need to know $T(u1)$ and $T(u2)$, so the new of functions that need to be computed is 4. Using additive combination, we get a complexity of 8. Using multiplicative combination, we get a complexity of 32 (2 * 2 arity * 4 functions * 2 new). Using exponential complexity, we get a complexity of 256 ($2^{(2+4+2)}$). The additional complexity of testing a new function is high. We also penalize the constraint because T_{rt} is limited by T, and this makes some sense as the need to understand two functions to compute the constraint does increase the complexity significantly.

Plus, must add the aggregation of T_{rt} to the configuration which creates an aggregate cost of 1.

Example 3 Example 3 is like Example 1, except the constraint is between two aggregates $S1$ and $S2$ rather than two individual nodes. Therefore, we increase the aggregate complexity by two, but the constraint is otherwise the same complexity as Example 1. Since the chunking is done in the aggregate, the additional complexity is minimal.

Example 4 Example 4 is like Example 3, except that the constraint is over permissions. This means that we must compute the permissions for all the types to which all the subjects in $S1$ and $S2$ belong ($P(t)$). If we presume that the subjects in $S1$ belong to one or more of the types in one of the second-level branches and $S2$ to one or more of the other, then we get one inheritance function plus the five type functions for each. These functions would be necessary for computing authorizations, so no new functions are added.

Therefore, we get the following complexities. Using additive combination, we get a complexity of 14 (2 for arity and 6 functions for each of the two aggregates). Using multiplicative combination, we get 48 (2 * 2 arity * 12 functions). Using exponential combination, we get 16K (2^14). As we can see, a severe penalty is incurred if the constraint refers to concepts within an assignment. In this case, a larger chunk of the configuration had to be assessed to determine the result.

In summary, our intuitive ordering of the Examples is close to that which the metrics indicated. Example 1 is indeed the simplest. However, it was only slightly simpler than Example 3. The addition of a single independent aggregate had little effect on the complexity. In hindsight this makes some sense.

However, our intuition differed from the metric's ordering of Examples 2 and 3. The addition of a new function for computing the number of restricted types held by a subject T_{rt} had a significant effect on complexity. This was primarily for two reasons: (1) the function had to added especially for computing the constraint, thus requiring additional understanding, and (2) the value of the function is limited by the value of the function T. Recall that this functional approach has been created to avoid needed to express this as a ternary constraint, so while some graph simplicity was maintained the complexity of this approach is still apparent in the metric.

As expected, Example 4 was the most complex. The reason for this is that it required the consideration of a greater portion of the graph (i.e., the chunk became bigger). We expect that constraints on objects which comprise permissions will add a similar increase in complexity to the constraint.

The choice of which combination function best displayed the complexity of the graph is hard to discern from four examples, but we would tend to prefer the multiplicative combination at present. The additive combination means that the complexity of the constraint is only a small percentage of the complexity of our sample configuration (10%). Since we expect that constraints are a significant percentage of the overall complexity, the multiplicative combination came closer to our expectation. While we think that there is a exponential facet to the increasing complexity of a constraint, exponential combination seemed excessive as it currently stands. We do not believe that Example 4 is 64 times more complex than Example 2.

Lastly, we noticed in the execution of the exper-

iment that the metrics as specified are both too informally specified and too complex to compute. In the cases, of Examples 2 and 4, further definition of the formal semantics of the metrics was necessary to compute them. Thus, more work in defining a formal semantics for these metrics is necessary. However, we found that the metrics themselves were fairly complex, so a search for a somewhat simpler set of metrics would be beneficial.

7 Conclusions

This goal of this paper is two-fold: (1) to motivate the complexity of safety expression as the key factor preventing the use of general-purpose access control models for highly safety-critical applications and (2) to propose a mechanism for measuring the complexity of access control configurations in hopes of managing complexity, such that such models may be used in safety-critical applications with confidence. The design of complexity metrics is based on notion that understanding the complexity of the configuration is key to making correct configuration changes. Understanding is based on the notion of *chunks*. That is, the simpler the independent chunks in a configuration are, the easier it is to understand the model. We use a graphical access control model to define the system which aids us in seeing the effect of the addition of concepts on the complexity of the graph. For example, since constraints connect graphs that are typically intended to be independent (e.g., in separation of duty), they add links that significantly impact the size of the chunks that need to be considered.

We identified metrics for computing both authorization and safety. Derivation of authorization metrics was useful in understanding how to generate safety metrics and also in verifying that the magnitude of the safety metric calculations were reasonable. We applied the metrics to four constraint examples in a single configuration. The results indicate that such metrics might give an indication of relative complexity. The constraints that require new functions or the examination of more concepts were rated as significantly more complex. However, this is just an initial investigation of this issue, and the difficulty of computing the metrics given these definitions indicate that simpler metrics may be necessary. Since we only considered a subset of a graphical access control model, the difficulty will only increase.

Acknowledgements

The author thanks Peter Gutmann for his insight into cognition that motivated the development of complexity metrics.

References

[1] G. Ahn and R. Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the 4th Workshop on Role-Based Access Control*, 1999.

[2] P. Ammann and R. Sandhu. One-representative safety analysis in the non-monotonic transform model. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, pages 138–149, 1994.

[3] P. E. Ammann and R. S. Sandhu. Safety analysis for the extended schematic protection model. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1991.

[4] D. Bell and L. La Padula. Secure computer systems: Mathematical foundations (Volume 1). Technical Report ESD-TR-73-278, Mitre Corporation, 1973.

[5] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information System Security*, 1(2), Feb. 1999.

[6] M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *Proceedings of the 7th ACM Symposium on Operating System Principles*, pages 45–54, 1979.

[7] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[8] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8), August 1976.

[9] T. Jaeger, A. Prakash, J. Liedtke, N. Islam. Flexible control of downloaded executable content. *ACM Transactions on Information and System Security (TISSEC)*, 2(2), May 1999.

[10] T. Jaeger and J. Tidswell. Practical safety for flexible access control models. Submitted for publication.

[11] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[12] B. W. Lampson. Protection. In *Proceedings Fifth Princeton Symposium on Information Sciences and Systems*, March 1971. reprinted in Operating Systems Review, 8, 1, January 1974, pages 18 – 24.

[13] G. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2), March 1956.

[14] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security (TISSEC)*, 2(1), Feb 1999.

[15] S. Osborn and Y. Guo. Modelling users in role-based access control. In *Proceedings of the* 5th *ACM Role-Based Access Control Workshop*, July 2000.

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

[17] R. Simon and M. E. Zurko. Mutual exclusion of roles as a means of implementing separation of duty in a role-based access control system. In *Proceeding of the* 10th *IEEE Computer Security Foundations Workshop*, June 1997.

[18] J. E. Tidswell and T. Jaeger. Integrated constraints and inheritance in DTAC. In *Proceedings of the* 5th *ACM Role-Based Access Control Workshop*, July 2000.

[19] J. E. Tidswell and T. Jaeger. An access control model for simplifying constraint expression. In *Proceedings of the* 7th *ACM Conference on Computer and Communication Security*, November 2000.

[20] J. Tidswell and J. Potter. A dynamically typed access control model. In *Proceedings of the Third Australasian Conference on Information Security and Privacy*, July 1998.