Application-Sensitive Access Control Evaluation using Parameterized Expressiveness (Extended Version)

Timothy L. Hinrichs*, Diego Martinoia*, William C. Garrison III†,
Adam J. Lee†, Alessandro Panebianco*, Lenore Zuck*
* Department of Computer Science, University of Illinois at Chicago
† Department of Computer Science, University of Pittsburgh

Abstract

Access control schemes come in all shapes and sizes, which makes choosing the right one for a particular application a challenge. Yet today's techniques for comparing access control schemes completely ignore the setting in which the scheme is to be deployed. In this paper, we present a formal framework for comparing access control schemes with respect to a particular application. The analyst's main task is to evaluate an access control scheme in terms of how well it implements a given access control *workload* (a formalism that we introduce to represent an application's access control needs). One implementation is better than another if it has stronger security guarantees, and in this paper we introduce several such guarantees: correctness, homomorphism, AC-preservation, safety, administration-preservation, and compatibility. The scheme that admits the implementation with the strongest guarantees is deemed the best fit for the application. We demonstrate the use of our framework by evaluating two workloads on ten different access control schemes.

Index Terms

access control; evaluation; state machine; parameterized expressiveness

I. Introduction

Access control, determining which actions are permitted in a system, is a fundamental issue in computer security and has been studied formally in numerous settings. Prior work has mainly focused on comparing the raw expressive power of two or more access control schemes, *e.g.*, [1]–[8]. While raw expressiveness is an interesting and meaningful basis for comparison, it fails to give a security analyst a methodology for choosing the access control scheme that will best serve the needs of a particular application (where by "application" we mean any kind of computer system, be it hardware, software, or cyberphysical system). The fact that scheme $\mathcal{S}$ is more expressive than scheme $\mathcal{T}$ only means that there are some applications for which $\mathcal{S}$ is adequate but $\mathcal{T}$ is not, a fact that fails to tell an analyst whether or not either scheme is adequate for her *particular* application.

We therefore advocate the development of an *application-sensitive* evaluation framework for access control schemes. Instead of comparing candidate access control schemes $\mathcal{S}$ and $\mathcal{T}$ with each other, we propose evaluating each candidate scheme against a specification of the application's access control *workload*, a formalism that we introduce to capture the access control demands of the application. The scheme that best meets the demands of that workload is the one deemed best-suited for the application, and it could be that the less expressive $\mathcal{T}$ better meets those demands than the more expressive $\mathcal{S}$.

While there are many ways to decide which access control scheme is best suited for a given application (*e.g.*, usability, maintenance overheads, development costs), in this paper we focus on two key issues: the basic functionality that the application requires and the security guarantees that are important for the application. We introduce ACEF, an application-sensitive access control evaluation framework, where the workload $\mathcal{W}$ describes the basic functionality the application requires of its underlying access control scheme, and candidate access control schemes are compared in terms of which application-relevant security guarantees that they can achieve. While ACEF is targeted at developers aiming to leverage hardened implementations of much-studied access control schemes by implementing $\mathcal{W}$ using an existing scheme, even developers implementing $\mathcal{W}$ from scratch can benefit from comparing potential implementations in terms of the security guarantees described by ACEF.

In ACEF, a workload $\mathcal{W}$ is based upon the concept of an idealized access control scheme for the application—a scheme that immediately meets the application's every access control need. Every operation the application would ever potentially execute that has access-control repercussions can be executed directly in $\mathcal{W}$; every bit of protection state the application ever needs to store is stored by $\mathcal{W}$; and every access control-relevant question the application would ever potentially need answered is one of the built-in queries of $\mathcal{W}$. Such an idealized access control scheme describes the basic functionality the application requires of any candidate scheme.

Each implementation of that basic functionality achieves different security guarantees. For example, the *safety* guarantee we introduce ensures that every right ever granted by an implementation must have been explicitly granted by the workload, even in transitory states. As another example, [5] describes the (strongly) security-preserving guarantee, which requires certain formulas in (infinitary) temporal logic to be preserved by the implementation. A central contribution of this work is the identification of several useful classes of desirable security guarantees that may hold over implementations of a workload. Which guarantees are important depends almost entirely on the application, and the scheme that can implement $\mathcal{W}$ while upholding the most application-relevant guarantees is deemed the best fit.

From the perspective of prior work, the key idea of this paper is that the same mathematical machinery used for years to compare access control schemes in absolute terms (the state machine and simulation relations), *e.g.*, [4], [5], [9], can also be used to develop an application-sensitive evaluation framework. This paper can be seen as an investigation into the validity of a simple but powerful thesis: that a state machine is an apt formalism for representing an application's basic access control functionality, and that the security properties achieved by the simulation relations between that state machine and each candidate scheme is a crucial aspect of comparing candidate schemes. ACEF can therefore be seen as a study of *parameterized expressiveness*: comparing the expressiveness of access control schemes relative to the workload and a set of application-relevant security guarantees. Change either the workload or the security guarantees, and the results of comparing two schemes $\mathcal{S}$ and $\mathcal{T}$ may change.

The main contributions of this work are as follows.

- We present the first framework for application-sensitive access control evaluation (ACEF). ACEF generalizes the application-insensitive frameworks studied in [4] and [5].
- We introduce several security guarantees for workload implementations (correctness, homomorphism, AC-preservation, safety, administration-preservation, and compatibility). For each guarantee, we develop useful proof techniques for negative results (*i.e.*, showing that a candidate scheme has no implementation with that guarantee), and introduce reductions between candidate schemes $\mathcal{S}$ and $\mathcal{T}$ that ensure every workload implementable in $\mathcal{S}$ with that guarantee is also implementable in $\mathcal{T}$ with that guarantee.
- We present an application-sensitive analysis of the workload for the dynamic coalition application described in [10] and another workload corresponding to a typical hospital administration application. For each workload, we analyze the suitability of four variations of the access matrix model, three variations of the RBAC model, and three variations of Bell-LaPadula.

In the remainder of the paper, we begin with an informal overview of ACEF (Section II) and then describe its three central formalisms: access control systems, workloads, and implementations (Section III). Then we introduce several novel security guarantees that we found to be important during our case studies, along with several theorems about those guarantees (Section IV). We then discuss the results of applying ACEF to two case studies (Section VI). Finally we describe related work (Section VII) and conclude (Section VIII).

II. Example and Informal Overview

The motivation for this work were the MITRE reports [10], [11] that conclude that the access control system currently used by the United States government is no longer adequate to secure the nation's information. The reports call for a re-design of the system to address a troublesome yet routine application of access control: dynamic coalitions. Dynamic coalitions arise whenever the U.S. joins forces with other countries to confront issues of global significance, *e.g.*, the military operations in Libya and the tsunami in Japan. Coalitions are problematic from the perspective of access control because each time a country joins a coalition, all participating governments must share large amounts of information with a large number of individuals, requiring massive changes in each country's access control policy. Similarly, each time a country leaves a coalition, a large number of rights must be revoked. Coalitions whose memberships change frequently put extraordinary demands on access control systems, and the MITRE reports cite anecdotes of how the current U.S. system has failed either to protect sensitive information or to release that information in a timely fashion. The key observation is not that the U.S. system has always been fundamentally flawed, but rather that it is a poor fit for the now-prevalent coalition operations. This dynamic coalition application serves as the running example throughout the paper.

ACEF is a rigorous mathematical framework that helps an analyst concerned about dynamic coalitions, for example, to design a new access control system for the U.S. government. Below we informally describe the methodology the analyst would follow and spend the remainder of the paper detailing ACEF's formal foundations.

To use ACEF, the analyst begins by describing the following idealized access control scheme to represent the workload for the dynamic coalitions application.

- **states**: Each state stores an access control policy and a record of which operative is a citizen of which country.
- **commands**: The *joinCoalition* command adds rights for all the joining country's users and records the country of each new user. The *leaveCoalition* command revokes all the rights granted any user of the country that is leaving the coalition.
- **queries**: The queries about the state that are relevant to the application include all the possible access control requests, and whether a given operative is a citizen of a given country.

Second, the analyst chooses a set of access control schemes that are viable candidates for the application. For example, the analyst might choose variants of the access-matrix (AM), role-based access control (RBAC), or Bell-LaPadula (BLP). In ACEF, each candidate scheme is formalized as a state machine: a set of states, commands for changing the state, and a set of queries for all states, just as in the workload described above.

Third, the analyst finds implementations of the workload for each of the candidate access control schemes. An implementation consists of three things:

- **state-mapping**: a mechanism that dictates how the access control scheme's states are used to represent the workload's states
- **command-mapping**: a prescription for how each workload command can be (weakly) simulated using the access control scheme's commands.
- **query-mapping**: a method of computing the workload's queries from the candidate scheme's queries

Next, the analyst chooses which security guarantees are important for the coalition workload's implementation, of which we introduce several in this paper. A *correct* implementation ensures that the access control scheme will faithfully simulate the end-to-end intent of each workload command. An *AC-preserving* implementation guarantees that the access control policy of the workload is represented the way the access control scheme was designed to represent access control policies. A *safe* implementation ensures that in even the intermediate states arising during a workload command's implementation, no right is ever granted or revoked unless the workload requires it. An *administration-preserving* implementation ensures that commands carried out by regular users in the workload never require an administrator to intervene in that command's implementation. A *homomorphic* implementation is one that is robust under constant substitutions, giving us confidence that the implementation is not blatantly abusing the scheme. A *compatible* implementation allows administrators to use the scheme the way that it was originally designed while simultaneously using it to meet the demands of the workload. This list of guarantees is by no means comprehensive (*e.g.*, [5] introduces the *(strong) security-preserving* guarantee), but is comprised of those we found useful when performing our evaluation.

Finally, the analyst compares candidate schemes in terms of parameterized expressiveness. For example, suppose the analyst decides that correctness, safety, and AC-preservation are the only important security guarantees for the dynamic coalition application. Then RBAC is better suited than BLP if RBAC can implement the coalition workload correctly and safely, but none of BLP's correct implementations are safe. If two schemes satisfy incomparable sets of security guarantees (*e.g.*, correctness and safety versus correctness and AC-preservation), the analyst must decide which set of guarantee is preferable.

III. Access Control, Workloads, Implementations

In this section we give formal definitions of access control, workloads, and a workload implementation in ACEF. At the heart of our formal framework is the *access control model*. Intuitively, an access control model is (i) a collection of data structures that store information pertinent to access control and (ii) a collection of queries that expose only certain kinds of information about those data structures to an external observer. Each snapshot of the data structures in the model is an *access control state*. Each method that exposes information about the state is a *query*. An access control model differs from an arbitrary data structure because every state supports a special set of queries that define the access control policy for that state. The access control policy for a state dictates which of all possible access control *requests* are granted and which ones are denied. In this paper we denote the access control queries with $auth(r)$, where r is one of the access control requests, *e.g.*, the typical combination of subject-object-right.

Definition 1 (Access Control Model): An access control model $\mathcal{M}$ has fields $\langle \mathcal{S}, \mathcal{R}, \mathcal{Q}, \models \rangle$

- $\mathcal{S}$: a set of states
- $\mathcal{R}$: a set of access control requests
- $\mathcal{Q}$: a set of queries including $auth(r)$ for every $r \in \mathcal{R}$
- $\models$: a subset of $\mathcal{S} \times \mathcal{Q}$ (the entailment relation)

If $\mathcal{M} = \langle \mathcal{S}, \mathcal{R}, \mathcal{Q}, \models \rangle$, we use *States*($\mathcal{M}$) to denote $\mathcal{S}$ and *Queries*($\mathcal{M}$) to denote $\mathcal{Q}$. We use the term *theory* to denote any truth assignment for all the queries in $\mathcal{Q}$. For state $s \in \mathcal{S}$, we use *Th*(s) (a subset of $\mathcal{Q}$) to denote the set of all $q \in \mathcal{Q}$ such that $s \models q$ (a convenient representation of the theory that holds at s). We use *Auth*(s) (a subset of *Th*(s)) to denote the set of all $auth(r) \in \mathcal{Q}$ such that $s \models auth(r)$. ♦

Example 1 (RBAC model): Traditionally each state in RBAC (specifically RBAC_0 [12]) includes a UR relation to record users and their roles and a PA relation to record roles and the object-right pairs assigned to them.[1] In our framework, each RBAC state is a finite collection of statements of the form $UR(a, b)$ or $PA(a, b, c)$. The permitted queries usually include all the possible $UR(a, b)$ and $PA(a, b, c)$. Additionally, the queries include all possible $auth(a, b, c)$, whose values are defined in terms of UR and PA: a subject is granted a right over an object exactly when there is some role to which the subject belongs and to which the right over that object is granted.

$$S \models auth(a, b, c) \iff \exists d.(UR(a, d) \in S \wedge PA(d, b, c) \in S)$$ ♦

While an access control model defines how to store and query information, an access control *system* adds methods for changing the state of an access control model. For example, when a user adds a document, the system changes state, perhaps

[1] Usually the set of all subjects, the set of all objects, and the set of all roles are also recorded in the state, but for brevity we ignore those here.

by adding the document's identifier to the set of known documents. Mathematically, an *access control system* adds labeled edges between the states of a model, where the labels record the command that caused the transition to occur.

Definition 2 (Access Control System): An access control system $\mathcal{Y}$ has fields $\langle \mathcal{M}, \mathcal{L}, next \rangle$

- $\mathcal{M}$: an access control model
- $\mathcal{L}$: a set of labels (also called commands)
- $next : \textit{States}(\mathcal{M}) \times \mathcal{L} \rightarrow \textit{States}(\mathcal{M})$ (the transition function)

If $\mathcal{Y} = \langle \mathcal{M}, \mathcal{L}, next \rangle$, we use *Labels*($\mathcal{Y}$) to denote *Labels*($\mathcal{M}$), *States*($\mathcal{Y}$) to denote *States*($\mathcal{M}$), and *Queries*($\mathcal{Y}$) to denote *Queries*($\mathcal{M}$). The *theories* of $\mathcal{Y}$ are all the theories of $\mathcal{M}$. For a finite sequence of labels $l_1 \circ \cdots \circ l_n$, we use $terminal(s, l_1 \circ \cdots \circ l_n)$ to denote the final state produced by repeatedly applying $next$ to the labels $l_1, \ldots, l_n$ starting from state s. ♦

Example 2 (RBAC system): The system commands for RBAC are given below.[1] All instances of those commands are the labels $\mathcal{L}$ in our framework, and the transition function $next$ is given implicitly by the changes the commands make to the state in which they are invoked.

- *assignUser(a,b)*: add $UR(a, b)$ to the state
- *revokeUser(a,b)*: remove $UR(a, b)$ from the state
- *assignPermission(a,b,c)*: add $PA(a, b, c)$ to the state
- *revokePermission(a,b,c)*: remove $PA(a, b, c)$ ♦

In ACEF, we use the concept of an access control system to define an *access control workload*—the mathematical construct intended to capture all the demands the application of interest places on its underlying access control system. A workload consists of two components: (i) an access control system defined to be ideal for the application of interest, and (ii) the set of all possible *traces* through that system that might arise depending on the environment in which the application is deployed.

The idealized access control system for a workload is one that immediately meets every access control need of the application. Every operation the application would ever potentially execute that has access-control repercussions can be executed directly in the workload. Every bit of protection state the application ever needs to store is stored by the workload. Every access control-relevant question the application would ever potentially need answered is one of the built-in queries of the workload.

The traces of the workload reflect the idea that the application could be deployed in many different settings (*e.g.*, an open-source web app is installed and run on many different systems) and hence the actual work that the application does varies from deployment to deployment. The traces describe how the environment in which the application is deployed will interact with that application by detailing all the possible sequences of commands the environment is permitted to invoke. In any single deployment, the environment will invoke only one of the workload's traces, but because the environment varies from deployment to deployment, the application must be able to properly cope with any one of the traces defined in the workload. Each of those traces is formalized as an initial state and the (possibly infinite) sequence of workload commands that are executed. This formalization assumes that if the environment executes commands concurrently, the workload traces include all possible linearizations of those concurrent executions.

Definition 3 (Workload): A workload $\mathcal{W}$ has fields $\langle \mathcal{A}, \mathcal{T} \rangle$.

- $\mathcal{A}$: an access control system.
- $\mathcal{T}$: a set of pairs $\langle s_0, \tau \rangle$ where $s_0 \in \textit{States}(\mathcal{A})$ and $\tau = l_1 \circ l_2 \circ \ldots$ is sequence where $l_i \in \textit{Labels}(\mathcal{A})$ for all i.

If $\mathcal{W} = \langle \mathcal{A}, \mathcal{T} \rangle$, we use *Labels*($\mathcal{W}$) to denote *Labels*($\mathcal{A}$), *States*($\mathcal{W}$) to denote *States*($\mathcal{A}$), and *Queries*($\mathcal{W}$) to denote *Queries*($\mathcal{A}$). ♦

From a formal perspective, an access control system is a special kind of workload: one where all possible traces are feasible. This similarity in formalism is useful because it helps keep the framework mathematically simple. But while formally similar, the intention of a workload differs appreciably from the intention of an access control system. An access control system is something that someone other than the analyst defines to represent a fixed piece of software. A workload is something the analyst defines to represent the high-level functionality an application requires of an access control system—functionality that would never be built directly into a general purpose access control system.

Example 3: In the coalition workload, one command involves an organization joining the coalition, and another command involves an organization leaving the coalition. The state includes statements of the form $auth(subject, object, right)$ to represent the authorization policy and $orgUser(orgID, subject)$ to track which subjects belong to which organizations.

The *joinCoalition* command takes as input an organization ID and a set of subject-object-right authorizations. For each authorization, it adds $auth(subject, object, right)$ and $orgUser(orgID, subject)$ to the state. The complementary command, *leaveCoalition* is applied to a given organization ID and revokes all the rights of subjects who are members of that organization. It also removes the record of those subjects belonging to that organization.

- *joinCoalition(orgID, newAuth)*: for each $\langle a, b, c \rangle \in newAuth$, add to the state (i) $auth(a, b, c)$ and (ii) $orgUser(orgID, a)$.

- *leaveCoalition(orgID)*: for each $orgUser(orgID, a)$ true in the state, remove from the state (i) all $auth(a, b, c)$ and (ii) $orgUser(orgID, a)$.

The assumption that *leaveCoalition* makes is that $orgUser$ is functional, *i.e.*, every subject belongs to at most one organization. This seems problematic because two invocations of *joinCoalition* could assign a single subject to two different organizations; however, for this application, *joinCoalition* is never used that way. To represent this within the workload, we say that the possible traces are all those that yield only states where $orgUser$ is functional; furthermore, we require that the start state of all traces is the empty state, and all subsequent states are finite. ♦

Once the analyst has formalized the application workload and the candidate access control systems, she searches for implementations of that workload for each of the candidate systems. The implementations we consider in this paper have three components. The first component is a specification for how each workload state can be represented by a state in the candidate access control system. The second component is a prescription for how each command in the workload is translated into a sequence of commands in the access control system—a prescription that can depend on the state in which the command is invoked. The third component describes how to compute the truth values for the workload queries given the truth values for the access control system queries. More precisely, this component consists of one function for each of the workload queries that maps each possible theory of the access control system to a truth-value for that workload query.

Definition 4 (Implementation): For a workload $\mathcal{W}$ and a system $\mathcal{Y}$, an implementation has fields $\langle \alpha, \sigma, \pi \rangle$

- $\sigma : \mathit{States}(\mathcal{W}) \rightarrow \mathit{States}(\mathcal{Y})$ (state-mapping)
- $\alpha : \mathit{States}(\mathcal{Y}) \times \mathit{Labels}(\mathcal{W}) \rightarrow \mathit{Labels}(\mathcal{Y})^*$ (command-mapping)
- π: for each $q \in \mathit{Queries}(W)$, a function π_q that maps each theory for $\mathcal{Y}$ to either true or false (query mapping)

With slight abuse of notation, for an access control theory T from system $\mathcal{Y}$, we write $\pi(T)$ to denote the set of all workload queries made true under π, *i.e.*,

$$\pi(T) \text{ denotes } \{q \in \mathit{Queries}(\mathcal{W}) \mid \pi_q(T) \text{ is true}\}$$

♦

Example 4 (RBAC and Coalitions): To implement the coalition workload in RBAC, the query mapping π might represent the workload's $auth(a, b, c)$ queries as RBAC usually does: there exists some role d such that $UR(a, d)$ and $PA(d, b, c)$ hold. To encode $orgUser$ we treat each $orgID$ as a role and represent $orgUser(orgID, a)$ as $UR(a, orgID)$ where the role $orgID$ is granted no rights for any object.

For the state mapping σ, the workload state w is mapped to an RBAC state s where the queries of w have the same values as when the query mapping is applied to s. In particular, σ chooses the minimal such RBAC state. For example, the initial workload state (wherein both $auth$ and $orgUser$ are empty), maps to the empty RBAC state (wherein both UR and PA are empty).

The command mapping α translates each workload command to a sequence of AC system commands (*assignUser*, *revokeUser*, *assignPermission*, and *revokePermission*). Each time *joinCoalition* adds $auth(a, b, c)$, a role d that occurs nowhere else in the state is created, and we invoke *assignUser(a,d)* and *assignPermission(d,b,c)*, thereby adding $UR(a, d)$ and $PA(d, b, c)$ to the state. For each $orgUser(orgID, a)$ that must be added, we invoke *assignUser(a,orgID)* to add $UR(a, orgID)$ to the state but first ensure that if $orgID$ is a legitimate role, that role (which our implementation invented) is first renamed to a fresh value. The implementation of *leaveCoalition* simply removes the appropriate UR and PA atoms using *revokeUser* and *revokePermission*. ♦

IV. Security Guarantees

This section formally introduces the security guarantees that we have developed and evaluated during our case study. Each guarantee is a property of the implementations from Definition (4). Typically, only some guarantees are relevant to a given application, and the access control system admitting an implementation with the largest number of application-relevant guarantees is the one best-suited for that application.

A. Correct Implementations

The most important guarantee is *correctness*. Intuitively, a correct implementation ensures that the environment cannot determine whether it is interacting with the workload state machine or with a candidate access control system at the basic level of inputs and outputs. More precisely, a correct implementation is one that for any of the workload's command traces, its execution produces a state sequence in the access control system that, except for intermediate states, is observationally equivalent to the workload's trace.

Definition 5 (Correctness): Consider a workload $\mathcal{W} = \langle \mathcal{A}, \mathcal{T} \rangle$, a system $\mathcal{Y}$, and an implementation $\langle \alpha, \sigma, \pi \rangle$. The implementation is correct if (i) the state-mapping preserves the query mapping: for every workload state w we have $\mathit{Th}(w) = \pi(\mathit{Th}(\sigma(w))$

and (ii) the command-mapping preserves the state mapping: for every workload trace $\langle w_0, \langle \beta_1, \beta_2, \dots \rangle \rangle \in \mathcal{T}$ where $s_0 = \sigma(w_0)$ and

$$\begin{array}{rclcrcl} w_1 & = & next(w_0, \beta_1) & & s_1 & = & terminal(s_0, \alpha(s_0, \beta_1)) \\ w_2 & = & next(w_1, \beta_2) & & s_2 & = & terminal(s_1, \alpha(s_1, \beta_2)) \\ \vdots & & & & \vdots & & \end{array}$$

we have that $s_i = \sigma(w_i)$ for all i. ♦

Notice that a single workload command can be implemented as a sequence of access control commands and that correctness places no limitations on what those intermediate states might be. Correctness only requires that the start and end state of every workload command's implementation is correct. For example, the implementation given in Example 4 is correct.

Our definition of correctness is, conceptually, a common one used in prior work on comparing access control systems in an application-insensitive manner (see [5] for a detailed treatment). While correctness is an intuitively necessary requirement for useful workload implementations, it is not a sufficient condition for guaranteeing the desirability of a workload implementation. For example, it has been shown that a simple variant of our notion of correctness can be used to simulate ATAM within RBAC [13], and to simulate RBAC within Strict DAC [5]. Thus, by transitivity, ATAM (in which the safety question is undeciable) can be simulated using Strict DAC (in which the safety question is decidable) [5]. Thus, while an implementation may be correct, it may not preserve all of the security guarantees that are important to an application. The remainder of this section describes additional restrictions on implementations that application developers can use to refine their evaluation of candidate access control systems.

B. AC-Preserving Implementations

An AC-preserving (access control-preserving) implementation is one that restricts how the authorization policy of the workload is represented by the access control system. It requires that the workload's authorization policy is represented in the system the way the system was designed to represent authorization policies. The intuition is that if an implementation violates this assumption, it has thrown out the central representational commitment of the access control system, and any application using the implementation is effectively using a custom access control solution. AC-preservation formalizes that intuition.

For example, in an AC-preserving RBAC implementation, the mapping for the workload's $auth(a, b, c)$ query is true exactly when there is some role d such that $UR(a, d)$ and $PA(d, b, c)$ are true in the RBAC state. An implementation that is not AC-preserving could choose to make $auth(a, b, c)$ true whenever $PA(a, b, c)$ is true in the RBAC state. The implementation given in Example 4 is AC-preserving.

Definition 6 (AC Preservation): An implementation with query-mapping π is called AC-preserving if for all workload states s and authorization requests r we have that $s \models auth(r)$ if and only if $\pi_{auth(r)}(Th(\sigma(s))) = true$. ♦

Notice that AC-preservation is different than correctness. AC-preservation puts a restriction on the query mapping that is not required either explicitly or implicitly by correctness. Notice also that for a system to achieve AC-preservation, it must support at least all those $auth$ queries in the workload.

C. Safe Implementations

Safety is a subject of much interest in the context of access control. It is often (though not always [14]) the name given to the following access control analysis problem: *given a system and an access control request, is that request ever permitted?* Instead of treating safety as an analysis problem pertaining to access control, here we treat it as a security guarantee that is tied to the original rights-leakage problem.

Whereas correctness restricts the start and end states of a workload command's implementation, a *safe* implementation restricts the states between the start and end states. Suppose the workload command β is executed from the workload state w, and an implementation causes a candidate access control system to transition from state s_0 through some number of intermediary states to end at s_n. Correctness only dictates that s_0 must represent w, and s_n must represent the workload state resulting from executing β in w. Safety requires that if the query $auth(r)$ changes to true anywhere between s_0 and s_n, then $auth(r)$ must true in s_n, and if $auth(r)$ changes to false then it must be false in s_n.

Notice that safety is not implied by correctness. Correctness requires that the $auth(r)$ queries be correct at s_1 and s_n, but it says nothing about the intermediate states. Similarly, correctness is not implied by safety. Safety puts restrictions on the $auth(r)$ queries, but it says nothing about the other queries. For an implementation to be both correct and safe, it must ensure that the access control system is observationally equivalent to the workload state machine for the non-intermediate states, and that the access control policy must monotonically change in the intermediate states.

For example, it is correct to implement the *joinCoalition* command by first adding 10 arbitrary rights to the access control policy, then adding the rights required by *joinCoalition*, and finally removing those 10 extraneous rights. However, such an implementation is unsafe because rights were changed that need not have been.

Definition 7 (Safety): An implementation is safe if the following holds for all i whenever the execution of a workload command yields the access control state sequence $\langle s_0, \ldots, s_n \rangle$.

$$\begin{array}{l} Auth(s_i) - Auth(s_0) \subseteq Auth(s_n) - Auth(s_0) \text{ (Grant)} \\ Auth(s_0) - Auth(s_i) \subseteq Auth(s_0) - Auth(s_n) \text{ (Revoke)} \end{array}$$

♦

For example, the RBAC implementation of the coalition workload in Example 4 fails to be safe. To represent the $orgUser$ component of the workload with the UR component of RBAC requires the implementation to sometimes rename roles used in representing the authorization policy to avoid conflicts. This renaming requires changes to the authorization policy not required by the workload commands.

D. Homomorphic Implementations

The goal of ACEF is to compare access control systems in terms of parameterized expressiveness: the access control system that is best-suited for a workload is the one with the implementation that has the strongest security guarantees. However, there is a style of implementation (the "string-packing implementation") that allows even the simplest access control system to implement the most complex workload while achieving some of the strongest guarantees possible, something that intuitively should not be possible. A string-packing implementation is one that represents the entire workload state with a single data element in the access control state (*e.g.*, a username or document identifier). That is, it encodes the entirety of a workload state as a string and then unpacks, manipulates, and re-packs that string as needed. For example, in the coalition workload, the entire $orgUser$ relation might be stored as a single RBAC username. When the $orgUser$ relation is queried, the implementation unpacks the that username to find the answer. When the $orgUser$ relation changes, the implementation unpacks, updates, and repacks that username.

The goal of the *homomorphic* security guarantee is to eliminate these implementations and in so doing capture our intuition that some workloads are too complex to be implemented by simple access control systems. Conceptually, it treats data elements as though they were opaque—as though they were not strings at all but rather indivisible entities. Said another way, if we were to replace all data elements with different data elements, the implementation's behavior would be the same under that substitution. The trouble with formalizing that intuition is that it requires knowing what the "data elements" for each system are—something that for exotic access control systems may not be straightforward. Thus, instead of attempting to eliminate string-packing implementations for all possible access control systems, we focus on a class of access control systems that are prevalent today and easy to define: the *extensional* access control systems.

An *extensional access control system* (*e.g.*, the access matrix, RBAC, Bell-La Padula) is one in which users enter atomic values (*e.g.*, roles, rights, classifications) into simple data structures (*e.g.*, a matrix or a pair of binary relations). We can represent each state of an extensional system as a set of relations over some universe of strings (*e.g.*, $\{UR(\text{“}alice\text{”}, \text{“}r\text{”}), UR(\text{“}bob\text{”}, \text{“}r\text{”}), PA(\text{“}r\text{”}, \text{“}doc\text{”}, \text{“}write\text{”})\}$) . Each query of an extensional system is the name of the query plus its arguments (*e.g.*, $auth(\text{“}alice\text{”}, \text{“}doc\text{”}, \text{“}write\text{”})$). Likewise, each command of an extensional system is the name of the command plus its arguments (*e.g.*, $assignUser(\text{“}alice\text{”}, \text{“}r\text{”})$).

Formally, an extensional access control model is a special kind of access control model where each of the states is a first-order interpretation from mathematical logic (sometimes called a first-order *model* or *structure*). Intuitively, a state represents a snapshot of a computer system's memory, and hence we are equating a first-order interpretation with such a snapshot, just as was done in a proof of the Church-Turing thesis [15].

While slightly non-standard, we represent a first-order interpretation as any set of *relational atoms* (sometimes called *ground facts* in logic programming). A relational atom is a statement of the form $p(a_1, \ldots, a_n)$ where p is called a relation constant and each a_i is called an object constant. Since in practice relation constants and object constants are strings, we assume they are drawn from $\mathcal{U}$, the set of finite-length strings over some finite character set.

Definition 8 (Extensional Access Control Model): An extensional access control model is $\langle \mathcal{S}, \mathcal{Q}, \models \rangle$

- $\mathcal{S}$: a set of sets of relational atoms (the states)
- $\mathcal{R}$: a set of access control requests (the requests)
- $\mathcal{Q}$: a set of relational atoms including $auth(a_1, \ldots, a_n)$ for every possible access control request $\langle a_1, \ldots, a_n \rangle$
- $\models$: a subset of $\mathcal{S} \times \mathcal{Q}$ (the entailment relation)

♦

An extensional access control *system* is a special kind of access control system where the labels are relations applied to first-order interpretations. Here we are equating a data structure that might be passed as an argument to a system command with a first-order interpretation. Thus a label is a relation constant applied to some number of sets of relational atoms.

Definition 9 (Extensional Access Control System): An extensional access control system $\mathcal{Y}$ has fields $\langle \mathcal{M}, \mathcal{L}, next \rangle$

- $\mathcal{M}$: an access control model
- $\mathcal{L}$: a set of $r(i_1, \ldots, i_n)$ where each i_j is a set of relational atoms (the labels)

- $next$: *States*$(\mathcal{M}) \times \mathcal{L} \rightarrow \mathcal{S}$ (the transition function) ♦

Extensional access control systems allow us to identify the data elements and therefore give a rigorous definition for the intuitive solution to the string-packing problem. A *homomorphic implementation* is one that is correct even when in the midst of a workload execution, every data element (in both the workload and the access control system) can be replaced consistently by any other data element.

The formal definition is based on a homomorphic function: a function that commutes with constant substitutions. Function f is homomorphic if for all constant substitutions v we have $f(x[v]) = f(x)[v]$[2]. Operationally, homomorphic functions can be understood as functions written in a special programming language that includes neither string constants nor string manipulation routines. See below for such a language.

Definition 10 (Homomorphisms): A constant substitution $v : \mathcal{U} \rightarrow \mathcal{U}$ is a bijection from strings to strings. The application of a substitution v to the mathematical structures important in this paper are given below.

- atom: $p(a_1, \ldots, a_n)[v] = p(v(a_1), \ldots, v(a_n))$
- set/state: $\{e_1, e_2, \ldots\} = \{e_1[v], e_2[v], \ldots\}$
- label: $p(S_1, \ldots, S_n)[v] = p(S_1[v], \ldots, S_n[v])$
- tuple: $\langle e_1, e_2, \ldots\rangle[v] = \langle e_1[v], e_2[v], \ldots\rangle$.
- function: if f' denotes $f[v]$ then for every $f(a_1, \ldots, a_n) = a$, $f'(a_1[v], \ldots, a_n[v]) = a[v]$.
- relation: if r' denotes $r[v]$ then we have $r'(a_1[v], \ldots, a_n[v])$ exactly when we have $r(a_1, \ldots, a_n)$.

The function f is homomorphic if for every constant substitution v, when $\gamma[v]$ is in f's domain then $f(\gamma[v]) = f(\gamma)[v]$. An implementation $\langle \alpha, \sigma, \pi\rangle$ is homomorphic with respect to permutation U on the set of strings $\mathcal{U}$ if for all workload states w and workload labels l such that some workload trace executes l in w, $\alpha(\sigma(w)[v], l[v], U[v]) = \alpha(\sigma(w), l, U)[v]$, $\sigma(w[v]) = \sigma(w)[v]$, and $\pi(Th(\sigma(w))[v]) = \pi(Th(\sigma(w)))[v]$. An access control system is homomorphic if $next$ and $\models$ are homomorphic. ♦

n our running example, suppose an implementation stores the $orgUser$ relation as a single user in the state, *e.g.*,

$$\{orgUser(\text{“}alice\text{”}, \text{“}USA\text{”}), orgUser(\text{“}bob\text{”}, \text{“}France\text{”})\}$$

is represented as the RBAC state

$$UR(\text{“}\langle alice, USA\rangle, \langle bob, France\rangle\text{”}, \text{“}r\text{”}).$$

Replacing “$alice$” with “eve” changes the workload state so that “eve” instead of “alice” belongs to “USA”. But that same substitution does not affect the RBAC state because it contains no single string “alice”; the only “alice” that appears is as a substring of the lone UR entry. Thus this implementation is not homomorphic.

$$\begin{aligned}
&\sigma(s[v])\\
&= \sigma(\{orgUser(\text{“}alice\text{”}, \text{“}USA\text{”}), orgUser(\text{“}bob\text{”}, \text{“}France\text{”})\}[\text{“}alice\text{”}/\text{“}eve\text{”}])\\
&= \sigma(\{orgUser(\text{“}eve\text{”}, \text{“}USA\text{”}), orgUser(\text{“}bob\text{”}, \text{“}France\text{”})\}\\
&= \{UR(\text{“}\langle eve, USA\rangle, \langle bob, France\rangle\text{”}, \text{“}r\text{”})\}\\
&\sigma(s)[v]\\
&= \{UR(\text{“}\langle alice, USA\rangle, \langle bob, France\rangle\text{”}, \text{“}r\text{”})\}[\text{“}alice\text{”}/\text{“}eve\text{”}]\\
&= \{UR(\text{“}\langle alice, USA\rangle, \langle bob, France\rangle\text{”}, \text{“”}r\text{”})\}
\end{aligned}$$

Thus σ fails to be homomorphic because $\sigma(s[\text{“}alice\text{”}/\text{“}eve\text{”}]) \neq \sigma(s)[\text{“}alice\text{”}/\text{“}eve\text{”}]$.

Unfortunately, one of the consequences of requiring a function to be homomorphic is that all the constants appearing in the output of the function must also appear in the input. This would preclude, for example, a homomorphic implementation that introduces new role names to a state in RBAC as in Example 4. To address this problem, we require every command-mapping α (but not the state-mapping σ or query-mapping π) to take an additional input: a total ordering of all possible constants $\mathcal{U}$. The presence of all possible constants in the input ensures that all constants in the output appear in the input, and the total ordering of those constants ensures that the application of a constant mapping is always reflected in the function's inputs.[3] In short, adding this extra argument enables the homomorphic restriction to eliminate string-packing implementations while enabling implementations that utilize an unlimited number of new constants.

[2]In the context of encryption, the term “homomorphic” is also used, but instead of commuting over constant substitutions as in this paper, homomorphic encryption is concerned with commuting over arithmetic. Naming our restriction “homomorphic” was intended to convey a conceptually similar but technically different requirement.

[3]Consider a function $g(a) = b$, which only intends b to be a new constant and thus should intuitively not be eliminated by the homomorphism requirement. But now consider the mapping $\{a \rightarrow a, b \rightarrow c, c \rightarrow b\}$. To be homomorphic, $f(a[v])$ must equal $b[v]$, but since $a[v] = a$ and $b[v] = c$, that requires $f(a) = c$, which is impossible since all constants are distinct. By adding $\langle a, b, c\rangle$ to the input of f, we start with $f(a, \langle a, b, c\rangle) = b$, and to be homomorphic, $f(a, \langle a, c, b\rangle)$ must equal c, which is permissible, thereby avoiding the elimination of this function.

The simplest style of string packing implementation we already discussed: encoding the entire $orgUser$ relation as a username that includes special characters to denote the beginning and ending of tuples the the separation of values within the tuple.

Second, suppose that instead of constructing a long username, the implementation converted the string encoding of the $orgUser$ relation to a number, *e.g.*, 2317, and then stored that number, and when necessary inverted the mapping from the number to reconstruct the $orgUser$ relation. That implementation too is eliminated by the homomorphism requirement because when 2317 is replaced by another constant, the implementation cannot decode the new constant correctly because the implementation is not told what the constant substitution was.

Third, suppose that instead encoding information as a number n and adding n to the system state that the implementation computed n and then added the nth string (according to the ordering on strings provided as input to g) to the state. This approach is especially attractive because, when the constants are substituted, the nth string before the substitution is replaced by the nth string after the substitution, and thus both before and after the substitution the implementation has the proper number n. This implementation is also eliminated by the homomorphic restriction because while the action-mapping α is given the ordered list of constants, neither σ nor π are given the ordered list of constants and hence they cannot properly decode the representation of $orgUser$.

Fourth, suppose that instead of encoding information as a number n and adding a single constant to the state to represent n that we add n distinct strings. This implementation is agnostic about what the n values are and hence loses no information when a constant substitution is applied. In fact there is an algorithm that encodes a relation as a number n using the ordered list of all strings so that when a constant substitution is applied, n represents that relation under the constant substitution (*i.e.*, the encoder is homomorphic). Nevertheless, the homomorphic restriction eliminates this option because neither the state mapping σ nor the query-mapping π are given access to the list of all strings and hence neither can decode n to the proper relation, despite the fact that α can perform the appropriate encoding.

In our experience (see Section VI), the homomorphism guarantee helped us prove some intuitively reasonable results: that several simple access control systems could not correctly implement the coalition workload. At the same time, we recognize that the homomorphism restriction is sometimes too strong and eliminates implementations that we would want to consider. The problem is that it assumes that no constants are meaningful to the system or workload. The strict DAC with change of ownership (SDCO) scheme violates this assumption. Specifically, the "own" right is handled differently from all other rights within the system.We believe that by parameterizing the definition of homomorphism by a finite set of reserved constants that are never substituted for, the homomorphic guarantee would handle SDCO properly while still eliminating string-packing implementations.

Since checking whether or not an implementation is homomorphic can be difficult, we developed a programming language in which all expressible programs are homomorphic: HPL (Homomorphic Programming Language). HPL is a simple language for manipulating sets and sequences of atoms and strings. Table I gives its semantics. The main restriction in the language is that it disallows both string manipulation and string literals. If the state-, command-, and query-mappings can all be written in HPL, the implementation is homomorphic.

$output[\![program(stmt), \sigma]\!] =$ if $exec[\![stmt, \langle \sigma, \epsilon \rangle]\!] = \langle \sigma', \gamma \rangle$ then γ
$exec[\![C_1; C_2, \kappa]\!] = exec[\![C_2, exec[\![C_1, \kappa]\!]]\!]$
$exec[\![v := E, \langle \sigma, \gamma \rangle]\!] = \langle \sigma[v \leftarrow eval[\![E, \langle \sigma, \gamma \rangle]\!]], \gamma \rangle$
$exec[\![output(x), \langle \sigma, \gamma \rangle]\!] = \langle \sigma, \gamma \circ eval[\![x, \langle \sigma, \gamma \rangle]\!] \rangle$
$exec[\![outputSet(\{x_1, \ldots, x_n\}), \langle \sigma, \gamma \rangle]\!] = \langle \sigma, \gamma \circ eval[\![x_1, \langle \sigma, \gamma \rangle]\!] \circ \cdots \circ eval[\![x_n, \langle \sigma, \gamma \rangle]\!] \rangle$, deterministically ordered
$exec[\![if(x, y, z), \kappa]\!] =$ if $eval[\![x, \kappa]\!] \neq \emptyset$ then $exec[\![y, \kappa]\!]$ else $exec[\![z, \kappa]\!]$
$exec[\![foreach(v_1 \in S_1, \ldots, v_n \in S_n, p_1(\bar{x}_1) \in T_1, \ldots, p_m(\bar{x}_m) \in T_m, x), \langle \sigma, \gamma \rangle]\!] =$ the sequential execution of
$exec[\![x, \langle \sigma \circ l, \gamma \rangle]\!]$ for a deterministic ordering of all variable bindings l such that all $v_i[l] \in S_i$ and all $p_i(\bar{x}_i)[l] \in T_i$
$eval[\![x, \langle \sigma, \gamma \rangle]\!] = \sigma(x)$ or $\{\}$ if $\sigma(x)$ is undefined, where x is a var
$eval[\![x \cup y, \kappa]\!] = eval[\![x, \kappa]\!] \cup eval[\![y, \kappa]\!]$
$eval[\![\overline{x}, \kappa]\!] =$ the set of all atoms not including $eval[\![x, \kappa]\!]$
$eval[\![\{p_1(\bar{v}_1), \ldots, p_n(\bar{v}_n) \mid q_1(\bar{u}_1) \in S_1, \ldots, q_m(\bar{u}_m) \in S_m\}, \kappa]\!] =$
the set of all $p_1(\bar{v}_1), \ldots, p_n(\bar{v}_n)[l]$ such that
$q_1(\bar{u}_1)[l] \in eval[\![S_1, \kappa]\!], \ldots, q_m(\bar{u}_m)[l] \in eval[\![S_m, \kappa]\!]$,
deterministically ordered, where all $\bar{v}_i, \bar{u}_i$ are variables.
$eval[\![nFreshConst(n, E, U), \kappa]\!] =$ the first $|eval[\![n, \kappa]\!]|$
strings in the sequence $eval[\![U, \kappa]\!]$ not appearing in the set
of atoms or strings $eval[\![E, \kappa]\!]$

TABLE I
HPL: A PROGRAMMING LANGUAGE (SEMANTICS) FOR HOMOMORPHIC IMPLEMENTATIONS.

The following theorem ensures that every program written in HPL is homomorphic.

Theorem 1: If σ is a variable assignment where every value is a set or sequence of atoms and/or strings, P is an HPL program (the semantics of which is defined in Table I), and P halts on input σ then $output[\![P, \sigma]\!][v] = output[\![P, \sigma[v]]\!]$.

Proof (sketch): We start with a two-level inductive proof about $eval[\![,]\!]$ and $exec[\![,]\!]$, which includes one inductive step for each of the programming language constructs of HPL.

We first show that if σ starts as a proper variable assignment (it assigns each variable to a set or sequence of atoms and/or strings) and $eval[\![e, \langle\sigma,\gamma\rangle]\!] = \langle\sigma',\gamma'\rangle$ then (i) σ' is a proper variable assignment and (ii) $\langle\sigma',\gamma'\rangle[v] = eval[\![e, \langle\sigma[v],\gamma[v]\rangle]\!]$.

Then we show for each $exec[\![s, \langle\sigma,\gamma\rangle]\!] = \langle\sigma',\gamma'\rangle$, that if σ is a proper variable assignment then (i) σ' is a proper variable assignment and (ii) $\langle\sigma',\gamma'\rangle[v] = exec[\![s, \langle\sigma[v],\gamma[v]\rangle]\!]$. Thus by induction we conclude that if σ is a proper variable assignment then for all those $stmt$ such that $exec[\![s, \langle\sigma,\epsilon\rangle]\!]$ halts, we know that $exec[\![s, \langle\sigma,\epsilon\rangle]\!]$ is homomorphic and that it returns a proper variable assignment.

Finally we know that if $exec[\![stmt, \langle\sigma,\epsilon\rangle]\!] = \langle\sigma',\gamma\rangle$ then $output[\![program(stmt), \sigma]\!] = \gamma$. And since by the above $exec[\![stmt, \langle\sigma[v],\epsilon[v]\rangle]\!] = \langle\sigma'[v],\gamma[v]\rangle$, we see that $output[\![program(stmt), \sigma[v]]\!] = \gamma[v]$, thus completing the proof.

Proof: We start with a two-level inductive proof about $eval[\![,]\!]$ and $exec[\![,]\!]$. We first show that if σ starts as a proper variable assignment (it assigns each variable to a set or sequence of atoms and/or strings) and $eval[\![e, \langle\sigma,\gamma\rangle]\!] = \langle\sigma',\gamma'\rangle$ then (i) σ' is a proper variable assignment and (ii) $\langle\sigma',\gamma'\rangle[v] = eval[\![e, \langle\sigma[v],\gamma[v]\rangle]\!]$.

Then we show for each $exec[\![s, \langle\sigma,\gamma\rangle]\!] = \langle\sigma',\gamma'\rangle$, that if σ is a proper variable assignment then (i) σ' is a proper variable assignment and (ii) $\langle\sigma',\gamma'\rangle[v] = exec[\![s, \langle\sigma[v],\gamma[v]\rangle]\!]$. Thus by induction we conclude that if σ is a proper variable assignment then for all those $stmt$ such that $exec[\![s, \langle\sigma,\epsilon\rangle]\!]$ halts, we know that $exec[\![s, \langle\sigma,\epsilon\rangle]\!]$ is homomorphic and that it returns a proper variable assignment.

Finally we know that if $exec[\![stmt, \langle\sigma,\epsilon\rangle]\!] = \langle\sigma',\gamma\rangle$ then $output[\![program(stmt), \sigma]\!] = \gamma$. And since by the above $exec[\![stmt, \langle\sigma[v],\epsilon[v]\rangle]\!] = \langle\sigma'[v],\gamma[v]\rangle$, we see that $output[\![program(stmt), \sigma[v]]\!] = \gamma[v]$, thus completing the proof.

Variables. By the inductive hypothesis, each assigned variable is assigned sets or sequences of atoms and/or strings; hence, the evaluation of an assigned variable is such a set or sequence. If the variable is unassigned, its evaluation is the empty set. Below we show the evaluation is homomorphic.

$$
\begin{aligned}
& eval[\![x, \langle\sigma,\gamma\rangle]\!][v] \\
& \qquad \text{by def of } eval[\![,]\!] \\
& = \sigma(x)[v] \\
& \qquad \text{since } \sigma \text{ is a function assigning each var to a} \\
& \qquad \text{set/sequence of atoms/strings} \\
& = \sigma[v](x) \\
& \qquad \text{by def of } eval[\![,]\!] \\
& = eval[\![x, \langle\sigma[v],\gamma[v]\rangle]\!]
\end{aligned}
$$

Union. By the inductive hypothesis, the two expressions x and y below represent sets of atoms and/or strings, or there is an error. The union of two such sets yields another set.

$$
\begin{aligned}
& eval[\![x \cup y, \kappa]\!][v] \\
& \qquad \text{By def of } eval[\![,]\!]. \\
& = (eval[\![x, \kappa]\!] \cup eval[\![y, \kappa]\!])[v] \\
& \qquad \text{Since each eval is an atom set and} \\
& \qquad (S_1 \cup S_2)[v] = S_1[v] \cup S_2[v] \\
& = eval[\![x, \kappa]\!][v] \cup eval[\![y, \kappa]\!][v] \\
& \qquad \text{By the inductive hypothesis} \\
& = eval[\![x, \kappa[v]]\!] \cup eval[\![y, \kappa[v]]\!] \\
& \qquad \text{By def of } eval[\![,]\!] \\
& = eval[\![x \cup y, \kappa[v]]\!]
\end{aligned}
$$

Complementation. By the inductive hypothesis, the expression x below evaluates to a set of atoms and/or strings, and the complement of such a set is well-defined; thus, $\overline{x}$ is also a set of atoms and/or strings.

$$
\begin{aligned}
& eval[\![\overline{x}, \kappa]\!][v] \\
& \qquad \text{By def of } eval[\![,]\!]. \\
& = \overline{eval[\![x, \kappa]\!]}[v] \\
& \qquad \text{Since eval is an atom/string set and } \overline{S}[v] = \overline{S[v]} \\
& = \overline{eval[\![x, \kappa]\!][v]} \\
& \qquad \text{By the inductive hypothesis} \\
& = \overline{eval[\![x, \kappa[v]]\!]} \\
& \qquad \text{By def of } eval[\![,]\!] \\
& = eval[\![\overline{x}, \kappa[v]]\!]
\end{aligned}
$$

nFreshConst. By definition nFreshConst returns a sequence of strings. Now suppose that $eval[\![nFreshConst(n, E, U), \kappa]\!]$ = $\langle a_1, \ldots, a_{|n|}\rangle$. By the definition of $eval[\![,]\!]$, we know that $eval[\![U, \kappa]\!]$ takes the form $\langle \bar{b}_1, a_1, \bar{b}_2, a_2, \ldots, \bar{b}_n, a_n\rangle$ where all of the strings in the tuple $\bar{b}_i$ appear in $eval[\![E, \kappa]\!]$, and none of the a_i appear in $eval[\![E, \kappa]\!]$.

Now consider $eval[\![nFreshConst(n, E, U), \kappa[v]]\!]$. By the inductive hypothesis, we know that $eval[\![U, \kappa[v]]\!]$ = $eval[\![U, \kappa]\!][v]$ and $eval[\![E, \kappa[v]]\!]$ = $eval[\![E, \kappa]\!][v]$. Therefore by the above, we see that $eval[\![U, \kappa[v]]\!]$ takes the form $\langle \bar{b}_1, a_1, \bar{b}_2, a_2, \ldots, \bar{b}_n, a_n\rangle[v]$ where all of the strings in the tuple $\bar{b}_i[v]$ appear in $eval[\![E, \kappa]\!][v]$, and none of the $a_i[v]$ appear in $eval[\![E, \kappa]\!][v]$. Since $eval[\![E, \kappa]\!]$ is a set/sequence of atoms/strings, and a string x appearing in such a set S holds if and only if $x[v]$ appears in $S[v]$, we see that $eval[\![U, \kappa[v]]\!]$ = $\langle a_1, \ldots, a_{|n|}\rangle[v]$, which completes the proof.

Set comprehension.. The result of set comprehension is by construction a set of atoms. It can be ordered deterministically with a homomorphic function as proven below. Now consider the evaluation of set comprehension.

$$\begin{aligned}
& eval[\![\{p_1(\bar{v}_1), \ldots, p_n(\bar{v}_n) \mid q_1(\bar{u}_1) \in S_1, \ldots, q_m(\bar{u}_m) \in S_m\}, \langle \sigma[v], \gamma[v]\rangle]\!] \\
& \qquad \text{by definition of } eval[\![,]\!] \\
& = \{p_1(\bar{v}_1)[l], \ldots, p_n(\bar{v}_n)[l] \mid q_1(\bar{u}_1)[l] \in eval[\![S_1, \langle \sigma[v], \gamma[v]\rangle]\!], \ldots, q_m(\bar{u}_m)[l] \in eval[\![S_m, \langle \sigma[v], \gamma[v]\rangle]\!]\} \\
& \qquad \text{by inductive hypothesis } eval[\![S_i, \langle \sigma[v], \gamma[v]\rangle]\!] = eval[\![S_1, \langle \sigma, \gamma\rangle]\!][v] \\
& = \{p_1(\bar{v}_1)[l], \ldots, p_n(\bar{v}_n)[l] \mid q_1(\bar{u}_1)[l] \in eval[\![S_1, \langle \sigma, \gamma\rangle]\!][v], \ldots, q_m(\bar{u}_m)[l] \in eval[\![S_m, \langle \sigma, \gamma\rangle[v]]\!]\} \\
& \qquad \text{Since all } \bar{u}_i \text{ are variables, we have } q(\bar{u}_i)[l] \in S[v] \text{ if and only if there is an } l' \text{ s.t. } q(\bar{u}_i)([l'][v]) \in S[v] \\
& = \{p_1(\bar{v}_1)([l'][v]), \ldots, p_n(\bar{v}_n)([l'][v]) \mid q_1(\bar{u}_1)([l'][v]) \in eval[\![S_1, \langle \sigma, \gamma\rangle]\!][v], \ldots, q_m(\bar{u}_m)([l'][v]) \in eval[\![S_m, \langle \sigma, \gamma\rangle]\!][v]\} \\
& \qquad \text{By associativity: } q(\bar{u}_i)([l'][v]) = q(\bar{u}_i)[l'][v] \\
& = \{p_1(\bar{v}_1)[l'][v], \ldots, p_n(\bar{v}_n)[l'][v] \mid q_1(\bar{u}_1)[l'][v] \in eval[\![S_1, \langle \sigma, \gamma\rangle]\!][v], \ldots, q_m(\bar{u}_m)[l'][v] \in eval[\![S_m, \langle \sigma, \gamma\rangle[v]]\!]\} \\
& \qquad \text{By definition of } [v] \\
& = \{p_1(\bar{v}_1)[l'], \ldots, p_n(\bar{v}_n)[l'] \mid q_1(\bar{u}_1)[l'] \in eval[\![S_1, \langle \sigma, \gamma\rangle]\!], \ldots, q_m(\bar{u}_m)[l'] \in eval[\![S_m, \langle \sigma, \gamma\rangle]\!]\}[v] \\
& \qquad \text{By definition of } eval[\![,]\!] \\
& = eval[\![\{p_1(\bar{v}_1), \ldots, p_n(\bar{v}_n) \mid q_1(\bar{u}_1) \in S_1, \ldots, q_m(\bar{u}_m) \in S_m\}, \langle \sigma, \gamma\rangle]\!][v]
\end{aligned}$$

Deterministic, homomorphic sorting. We can deterministically order a set of atoms or strings using a homomorphic function f that is constructed as follows. Build equivalence classes for the atoms/strings so that $a \sim b$ if and only if there is a constant substitution v such that $a = b[v]$. Then process each equivalence class independently. Assign $f(a)$ any ordering on the set a. Then for every v assign $f(a[v]) = f(a)[v]$. Note that this function is total since every set of atoms/strings belongs to some equivalence class. Moreover, it is a function since if $a[v_1] = a[v_2]$ then (i) $a[v_1]$ and $a[v_2]$ belong to the same equivalence class, and (ii) $f(a[v_1]) = f(a[v_2])$ since the constants in a are the same as the constants in $f(a)$.

Composition. $exec[\![C_1; C_2, \kappa]\!]$ = $exec[\![C_2, exec[\![C_1, \kappa]\!]]\!]$ by definition. By applying the inductive hypothesis to both instances of exec, we see that (i) the overall output is a set or sequence of atoms and/or strings and (ii) since the composition of homomorphic functions is homomorphic, the result is homomorphic.

Assignment. Consider $exec[\![x := E, \langle \sigma, \gamma\rangle]\!]$. Since $eval[\![E, \langle \sigma, \gamma\rangle]\!]$ is a set or sequence of atoms and/or strings, so too is the variable assignment that results from the assignment statement. Now we show assignment is homomorphic.

$$\begin{aligned}
& exec[\![x := E, \langle \sigma[v], \gamma[v]\rangle]\!] \\
& \qquad \text{By def of } exec[\![,]\!] \\
& = \langle \sigma[v][x \leftarrow eval[\![E, \langle \sigma[v], \gamma[v]\rangle]\!], \gamma[v]\rangle \\
& \qquad \text{by the homomorphism of } eval[\![,]\!] \\
& = \langle \sigma[v][x \leftarrow eval[\![E, \langle \sigma, \gamma\rangle]\!][v]], \gamma[v]\rangle \\
& \qquad \text{Since } \sigma[v][x \leftarrow y[v]] = \sigma[x \leftarrow y][v] \\
& = \langle \sigma[x \leftarrow eval[\![E, \langle \sigma, \gamma\rangle]\!]][v], \gamma[v]\rangle \\
& \qquad \text{By distribution of [v] over } \langle\rangle \\
& = \langle \sigma[x \leftarrow eval[\![E, \langle \sigma, \gamma\rangle]\!]], \gamma\rangle[v] \\
& \qquad \text{By def of } exec[\![,]\!] \\
& exec[\![x := E, \langle \sigma, \gamma\rangle]\!][v]
\end{aligned}$$

Output. $exec[\![outputSet(\{x_1, \ldots, x_n\}), \langle \sigma, \gamma\rangle]\!]$ does not change σ, and hence the resulting σ assigns all variables sets or

sequences of atoms and/or strings. Below we show it is homomorphic.

$$
\begin{aligned}
&exec[\![outputSet(\{x_1,\dots,x_n\}),\langle\sigma[v],\gamma[v]\rangle]\!]\\
&\qquad\text{By def of } exec[\![,]\!]\\
&=\langle\sigma[v],\gamma[v]\circ eval[\![x_1,\langle\sigma[v],\gamma[v]\rangle]\!]\circ\cdots\circ eval[\![x_1,\langle\sigma[v],\gamma[v]\rangle]\!]\rangle\\
&\qquad\text{Since } eval[\![,]\!] \text{ is homomorphic}\\
&=\langle\sigma[v],\gamma[v]\circ eval[\![x_1,\langle\sigma,\gamma\rangle]\!][v]\circ\cdots\circ eval[\![x_1,\langle\sigma,\gamma\rangle[v]]\!]\rangle\\
&\qquad\text{Since } a_1\circ a_2\circ\dots[v]=a_1[v]\circ a_2[v]\circ\dots\\
&=\langle\sigma[v],(\gamma\circ eval[\![x_1,\langle\sigma,\gamma\rangle]\!]\circ\cdots\circ eval[\![x_1,\langle\sigma,\gamma\rangle]\!])[v]\rangle\\
&\qquad\text{By distribution of } [v] \text{ over } \langle\rangle\\
&=\langle\sigma,(\gamma\circ eval[\![x_1,\langle\sigma,\gamma\rangle]\!]\circ\cdots\circ eval[\![x_1,\langle\sigma,\gamma\rangle]\!])\rangle[v]\\
&\qquad\text{By def of } exec[\![,]\!]\\
&exec[\![outputSet(\{x_1,\dots,x_n\}),\langle\sigma,\gamma\rangle]\!][v]
\end{aligned}
$$

Conditionals. $exec[\![if(x,y,z),\kappa]\!]$ is defined as if $eval[\![x,\kappa,\neq]\!]\emptyset$ then $exec[\![y,\kappa]\!]$ else $exec[\![z,\kappa]\!]$. By the inductive hypothesis, both the executions of y and z result in all variables assigned sets or sequences of atoms and/or strings; hence so does the execution of the if statement. To show the execution is homomorphic, we need only show that the condition is homomorphic, since the execution afterwards is homomorphic by the inductive hypothesis.

$$
\begin{aligned}
&exec[\![if(x,y,z),\kappa[v]]\!]\\
&\qquad\text{By def of } exec[\![,]\!]\\
&\text{if } eval[\![x,\kappa[v]]\!]\neq\emptyset \text{ then } exec[\![y,\kappa[v]]\!] \text{ else } exec[\![z,\kappa[v]]\!]\\
&\qquad\text{By homomorphism of eval and (inductively) exec}\\
&\text{if } eval[\![x,\kappa]\!][v]\neq\emptyset \text{ then } exec[\![y,\kappa]\!][v] \text{ else } exec[\![z,\kappa]\!][v]\\
&\qquad\text{Since } S[v]\neq\emptyset \text{ if and only if } S\neq\emptyset\\
&\text{if } eval[\![x,\kappa]\!]\neq\emptyset \text{ then } exec[\![y,\kappa]\!][v] \text{ else } exec[\![z,\kappa]\!][v]\\
&\qquad\text{Since } [v] \text{ is applied in either branch}\\
&(\text{if } eval[\![x,\kappa]\!]\neq\emptyset \text{ then } exec[\![y,\kappa]\!] \text{ else } exec[\![z,\kappa]\!])[v]\\
&\qquad\text{By def of } exec[\![,]\!]\\
&exec[\![if(x,y,z),\kappa]\!][v]
\end{aligned}
$$

Iteration. $exec[\![foreach(v_1\in S_1,\dots,v_n\in S_n,p_1(\bar{x}_1)\in T_1,\dots,p_m(\bar{x}_m)\in T_m,body),\langle\sigma,\gamma\rangle]\!]$ is by the definition of $exec[\![,]\!]$ equal to $exec[\![body,\langle\sigma\circ l_1,\gamma\rangle]\!]$; $exec[\![body,\langle\sigma\circ l_2,\gamma\rangle]\!]$; .... Since by the inductive hypothesis the result of each such exec results in all variables assigned to sets or sequences of atoms and/or strings, and the composition of such functions preserves that invariant, so too does $foreach$ result in a proper variable assignment. To see the result is homomorphic, we need only apply the inductive hypothesis to each exec and then note that the composition of homomorphic functions is homomorphic.

The only detail left is to show that the construction of the sequence $\langle l_1,l_2,\dots\rangle$ is homomorphic so that the sequence of execs is homomorphic. First we note that elsewhere we have shown how to construct a homomorphic function that sorts a set of atoms. So we can assume that all of the S_i and T_i have been sorted in this way. Then it is a simple matter to deterministically construct the sequence $\langle l_1,l_2,\dots\rangle$ in a depth-first manner by finding a binding for v_1 by walking over the sorted S_1, a binding for v_2 by walking over S_2, and so on. Now consider applying any constant substitution to the S_i and T_i. Since the sorting function is homomorphic, $sort(S_1)[v] = sort(S_1[v])$, and hence the deterministic algorithm will extract the same sequence of l_is under $[v]$. See the Set Comprehension case for more details.

Distributing $[v]$ over exec and eval. Notice that because there are no strings allowed in the programming language itself, all of the cases above implicitly begin with the following step where we apply $[v]$ to the arguments of $exec[\![,]\!]$ and $eval[\![,]\!]$.

$$
\begin{aligned}
&exec[\![S[v],\langle\sigma,\gamma\rangle[v]]\!]\\
&\qquad\text{Since } S \text{ contains no strings and } [v] \text{ distributes over } \langle\rangle\\
&=exec[\![S,\langle\sigma[v],\gamma[v]\rangle]\!]
\end{aligned}
$$

$$
\begin{aligned}
&eval[\![E[v],\langle\sigma,\gamma\rangle[v]]\!]\\
&\qquad\text{Since } E \text{ contains no strings and } [v] \text{ distributes over } \langle\rangle\\
&=eval[\![E,\langle\sigma[v],\gamma[v]\rangle]\!]
\end{aligned}
$$

□

The language above has two basic semantic types: relational atoms and strings, though there is no way to represent either in the language directly. Similarly, the language semantically allows for sets and sequences, but does not represent either in the language directly. Rather, in this language, the programmer applies the constructs above to variables whose values sets, sequences, atoms, and strings. Additionally, the language includes no string manipulation functionality, and it is deterministic.

E. Administration-Preserving Implementations

One important distinction in access control is that of the administrators versus regular users. Administrators can do everything regular users can do, but in addition they have special permissions to help deploy, maintain, and trouble-shoot the system. The important observation about administrators is that typically there are far fewer administrators than regular users, and good applications are designed to minimize administrator involvement. A good workload implementation then is one that minimizes the work for administrators. The *administration-preservation* security guarantee requires that any task executed by a regular user in the workload must not require administrative involvement in the candidate system.

To formalize this idea, we assume that the workload and the access control system each have designated some subset of their commands (labels) as "administrative". An administrative command is one that only an administrator is permitted to execute. In Bell-LaPadula administrators change the clearances and classifications of subjects and objects, whereas regular users change the access matrix; in the our hospital workload case study (see Section VI), administrators change the doctors and clerical staff employed by the hospital but regular users, like doctors, change patient charts. We say that an implementation is administration-preserving if the command mapping ensures that every non-administrative workload command maps to a sequence of non-administrative access control system commands.

Definition 11 (Administrative preservation): An implementation $\langle \alpha, \sigma, \pi \rangle$ is administration-preserving if for all workload labels l and system states s, if $\alpha(s, l) = l_1 \circ \cdots \circ l_n$ and l is not a workload administrative command then none of $\{l_1, \ldots, l_n\}$ are system administrative commands. ♦

F. Compatible Implementations

Part of the intuition behind an implementation is that it demonstrates how to augment an access control system to include commands for all of the workload's commands. That intuition brings with it the idea that in the resulting system we could ignore the new workload commands and use the system as it was originally intended. Or we could ignore the original commands and use just the new workload commands, or we could interleave the workload commands with the original commands. It turns out that some implementations are better suited to this kind of interleaving than others. We call such implementations *compatible* with the original system commands.

We can formalize this idea by comparing the implementations of workload $\mathcal{W}$ in system $\mathcal{Y}$ with implementations of a workload built by combining $\mathcal{W}$ and $\mathcal{Y}$ (which we denote $\mathcal{W} \cup \mathcal{Y}$). Intuitively, we say that if the implementation of $\mathcal{W}$ can be conservatively extended to an implementation of $\mathcal{W} \cup \mathcal{Y}$, then it is *compatible* with the original system. For example, for an implementation of the coalition workload in RBAC to be compatible, there must be an implementation of the coalition workload augmented with all the RBAC commands and queries (*e.g.*, *assignUser*, *assignPermission*) that conservatively extends the original implementation.

There are two kinds of compatibility that we have studied. To understand the difference, it is important to remember that an implementation treats a workload and an access control system as having different namespaces. So if a workload contains the *assignUser* command, an implementation in a system that also has a command named *assignUser* (such as RBAC) need not use the system's *assignUser* command at all. The implementation is free to use any sequence of system commands it likes; of course, it is free to implement the workload's *assignUser* with the system's *assignUser* command. This leads to two different kinds of compatibility. A *strongly compatible* implementation of $\mathcal{W} \cup \mathcal{Y}$ is one where the implementation of every $\mathcal{Y}$ command is that command itself. All other compatible implementations are *weakly compatible*.

The definition of compatibility relies on the definition of $\mathcal{W} \cup \mathcal{Y}$ for adding an access control system to a workload to produce a new workload. Conceptually, combining a workload $\mathcal{W}$ and an access control system $\mathcal{Y}$ requires two things: combining $\mathcal{Y}$ with the access control system embedded within $\mathcal{W}$ and choosing the traces permitted by that combined access control system. In this paper, we combine access control systems by building the state machine representing the cross product of those systems and by unioning the queries of the two systems. The traces for $\mathcal{W} \cup \mathcal{Y}$ are all the traces from $\mathcal{W}$ but where commands from $\mathcal{Y}$ are interleaved arbitrarily.

Definition 12 (Workload $\cup$ System): Consider a workload $\mathcal{W} = \langle \langle \langle \mathcal{S}_w, \mathcal{R}_w, \mathcal{Q}_w, \models_w \rangle, \mathcal{L}_w, next_w \rangle, \mathcal{T}_w \rangle$ and an access control system $\mathcal{Y} = \langle \langle \mathcal{S}_y, \mathcal{R}_y, \mathcal{Q}_y, \models_y \rangle, \mathcal{L}_y, next_y \rangle$. $\mathcal{W} \cup \mathcal{Y}$ is defined as:

model:	$\langle \mathcal{S}_w \times \mathcal{S}_y, \mathcal{R}_w \cup \mathcal{R}_y, \mathcal{Q}_w \cup \mathcal{Q}_y, \models \rangle$ $\langle s_w, s_y \rangle \models q$ iff $s_w \models_w q \vee s_y \models_y q$
system:	$\langle \mathcal{L}_w \cup \mathcal{L}_y, next \rangle$ $next(\langle s_w, s_y \rangle, l) = \begin{cases} \langle next_w(s_w, l), s_y \rangle, \text{if } l \in \mathcal{L}_w \\ \langle s_w, next_y(s_y, l) \rangle, \text{otherwise} \end{cases}$
traces:	the set of all $\langle \langle s_w, s_y \rangle, \tau \rangle$ where τ is a sequence of $\mathcal{L}_w \cup \mathcal{L}_y$ and $\langle s_w, \tau|_{\mathcal{L}_w} \rangle \in \mathcal{T}_w$ (where $\tau|_{\mathcal{L}_w}$ denotes τ's projection onto its $\mathcal{L}_w$ elements)

Definition 13 (Compatibility): The implementation $\langle \alpha, \sigma, \pi \rangle$ for workload $\mathcal{W}$ in system $\mathcal{Y}$ is compatible for implementation guarantees $\mathcal{G}$ if there is an implementation $\langle \alpha', \sigma', \pi' \rangle$ of $\mathcal{W} \cup \mathcal{Y}$ with guarantees $\mathcal{G}$ and the following properties.

- α' conservatively extends α: if $l \in \mathcal{L}_w$ then for all $s_w \in \mathit{States}(\mathcal{W})$ and $s_y \in \mathit{States}(\mathcal{Y})$, $\alpha'(\langle s_w, s_y \rangle, l) = \alpha(s_w, l)$.
- π' conservatively extends π: if $q \in Q_w$ then $\pi'_q = \pi_q$.

The implementation is *strongly compatible* when $\alpha'(x, l) = l$ for every state x and every label $l \in \mathcal{L}(Y)$; otherwise, the implementation is weakly compatible. ♦

For example, the combination of the RBAC system and the coalition workload would result in states whose fields are $auth$, $orgUser$, UR, and PA. The commands would be *joinCoalition*, *leaveCoalition*, *assignUser*, *revokeUser*, *assignPermission*, and *revokePermission*. If there were an implementation of this combined workload, the fragment of that implementation pertaining to just the coalition workload would be a compatible implementation.

Notice that compatibility is a different kind of guarantee than the others introduced so far because it is parameterized by a set of security guarantees. Furthermore, compatibility is concerned with the existence of another implementation, which also differentiates it qualitatively from the other guarantees.

V. Meta-theorems

In this section we discuss some of the consequences of the definitions of our analysis framework. In particular, we focus on theorems that guarantee the existence of an implementation with certain security guarantees. By applying these theorems, an analyst can forego the time-consuming process of writing the code comprising an implementation and proving it satisfies certain security guarantees. To this end, we show how *access control reductions* can be used to transfer implementability results from one system to another. Intuitively, a *reduction* from access control system A to access control system B is a mapping from A to B that ensures that every workload implementable in A is also implementable in B.

Mappings between access control systems have been studied in great detail in prior work on application-*in*sensitive evaluation of access control. For example, [4] studied the differences between strong mappings (where each command in system A is mapped to a single command in B) and weak mappings (where each command in A is mapped to a sequence of commands in B). [5] discussed the failings of previous mappings (pointing to their inability to differentiate access control systems in nontrivial ways) and introduces a new mapping that overcomes those failings: the strongly security-preserving mapping.

While prior works have introduced many different mappings from one access control system to another, they fail to tell us what those mappings mean in terms of workload implementations and security guarantees. For example, what kind of mapping from A to B ensures that whenever A admits an implementation for workload $\mathcal{W}$ that is both correct and homomorphic then B must also admit an implementation of $\mathcal{W}$ that is both correct and homomorphic? This question we address in this section generalizes the one usually asked about the expressiveness of two access control systems by adding a parameter: the set of security guarantees in question.

Definition 14 (Parameterized Expressiveness (Systems)): Suppose that for all workloads $\mathcal{W}$ if system $\mathcal{Y}_1$ correctly implements $\mathcal{W}$ with security guarantees $\mathcal{G}$ then system $\mathcal{Y}_2$ also correctly implements $\mathcal{W}$ with guarantees $\mathcal{G}$. Then we say that $\mathcal{Y}_1$ is no more expressive than $\mathcal{Y}_2$ with respect to $\mathcal{G}$, written $\mathcal{Y}_1 \leq^{\mathcal{G}} \mathcal{Y}_2$. ♦

Below we detail several mappings between systems that suffice to ensure that one system is no more expressive than another. These reductions are intended to be easy to construct. Our most basic reduction is a simplification of the workload implementation discussed in the previous section. Instead of a state-mapping, a query-mapping, and an command-mapping, our reduction is just a state-mapping combined with a query-mapping where the state-mapping preserves the query-mapping. In this section, we use the shorthand $t \equiv_Q s$ as shorthand for $\mathit{Th}(t) = \pi(\mathit{Th}(s))$ or $\mathit{Th}(t) = \mathit{Th}(s)$, when the precise meaning is clear from context.

Definition 15 (Reduction): A reduction from system $\mathcal{Y}_1$ to system $\mathcal{Y}_2$ is a state-mapping σ and a query-mapping π where the state-mapping preserves the query-mapping, *i.e.*, for all $\mathcal{Y}_1$ states s we have $s \equiv_Q \sigma(s)$. ♦

Theorem 2 (System Reductions for $\leq$): If there is a reduction $\langle \sigma, \pi \rangle$ from $\mathcal{Y}_1$ to $\mathcal{Y}_2$ where σ is one-to-one and preserves finite reachability (for all $s, s' \in \mathit{States}(\mathcal{Y}_1)$, if s' is reachable in a finite number of steps from s, then $\sigma(s')$ is reachable in a finite number of steps from $\sigma(s)$), then $\mathcal{Y}_1 \leq \mathcal{Y}_2$.

Proof: We demonstrate how to construct a correct implementation in system $\mathcal{Y}_2$ of any workload $\mathcal{W}$ that is correctly implementable by $\mathcal{Y}_1$. Suppose $\langle \alpha^{\mathcal{Y}_1}, \sigma^{\mathcal{Y}_1}, \pi^{\mathcal{Y}_1} \rangle$ is a correct implementation of W in $\mathcal{Y}_1$ and that $\langle \sigma, \pi \rangle$ is a reduction from $\mathcal{Y}_1$ to $\mathcal{Y}_2$ where σ preserves finite reachability and is 1-1. We first describe the state- and query- mappings for the implementation of $\mathcal{W}$ in $\mathcal{Y}_2$ and prove the the state-mapping preserves the query-mapping (the first property of a correct implementation). Then we describe the command-mapping and argue that it preserves the state-mapping (the second property of correctness).

The state- and query-mappings are given below.

$$\text{for all } x \in \mathit{States}(\mathcal{W}).\sigma^{\mathcal{Y}_2}(x) = \sigma(\sigma^{\mathcal{Y}_1}(x))$$
$$\text{for all } x \in \mathit{States}(\mathcal{Y}_2).\pi^{\mathcal{Y}_2}(\mathit{Th}(x)) = \pi^{\mathcal{Y}_1}(\pi(\mathit{Th}(x)))$$

We must show that the state-mapping preserves the query-mapping, *i.e.*, for all workload states w we have $w \equiv_Q \sigma^{\mathcal{Y}_2}(w)$. To do that, we must show that $w \models q$ if and only if $\pi_q^{\mathcal{Y}_2}(\mathit{Th}(\sigma^{\mathcal{Y}_2}(w))) = true$.

$$\begin{aligned}
&\pi_q^{\mathcal{Y}_2}(\mathit{Th}(\sigma^{\mathcal{Y}_2}(w))) = true\\
&\quad\text{By def of } \pi_q^{\mathcal{Y}_2}\\
\iff\ &\pi_q^{\mathcal{Y}_1}(\pi(\mathit{Th}(\sigma^{\mathcal{Y}_2}(w)))) = true\\
&\quad\text{By def of } \sigma^{\mathcal{Y}_2}\\
\iff\ &\pi_q^{\mathcal{Y}_1}(\pi(\mathit{Th}(\sigma(\sigma^{\mathcal{Y}_1}(w))))) = true\\
&\quad\text{By query-preservation of } \langle\sigma,\pi\rangle\\
\iff\ &\pi_q^{\mathcal{Y}_1}(\mathit{Th}(\sigma^{\mathcal{Y}_1}(w)))\\
&\quad\text{By correctness of } \langle\alpha^{\mathcal{Y}_1},\sigma^{\mathcal{Y}_1},\pi^{\mathcal{Y}_1}\rangle\\
\iff\ &w \models q
\end{aligned}$$

The command mapping is more difficult to construct. We must show there is some $\alpha^{\mathcal{Y}_2}$ that maps the $\mathcal{Y}_2$ states and a $\mathcal{W}$ label to a finite sequence of $\mathcal{Y}_2$ labels that preserves the state-mapping. More precisely, $\alpha^{\mathcal{Y}_2}$ must preserve the state-mapping for the traces in $\mathcal{W}$. For that, it suffices to assign values for $\alpha^{\mathcal{Y}_2}(y,l)$ where the $\mathcal{Y}_2$ state y represents some workload state w (*i.e.*, $\sigma^{\mathcal{Y}_2}(w) = y$) and there is some trace where workload label l is executed in w.

So consider any such y, w, and l. Suppose $\sigma^{\mathcal{Y}_1}(w) = s$ and that $\sigma(s) = y$. Building on the correctness of $\alpha^{\mathcal{Y}_1}$, we assign $\alpha^{\mathcal{Y}_2}(y,l)$ so that $terminal(y,\alpha^{\mathcal{Y}_2}(y,l))$ is query-equivalent to $terminal(s,\alpha^{\mathcal{Y}_1}(s,l))$. We know there is always a finite sequence of labels in $\mathcal{Y}_2$ that yield such a state because σ preserves the query-mapping and is known to preserve finite reachability.

The only potential problem with this construction is that there may be two workload states w_1 and w_2 that map to the same $\mathcal{Y}_2$ state y. This is potentially problematic because there may be two traces where the implementation of some workload label l must differ depending on whether executed from w_1 or w_2. This would mean the construction above is ill-defined because there would be two different values for $\alpha^{\mathcal{Y}_2}(y,l)$. But because the reduction from $\mathcal{Y}_1$ to $\mathcal{Y}_2$ is 1-1, the only way w_1 and w_2 can both map to y through $\sigma^{\mathcal{Y}_2}$ is if they both also map to a single $\mathcal{Y}_1$ state s through $\sigma^{\mathcal{Y}_1}$. If both w_1 and w_2 are both implemented using the same state s in $\mathcal{Y}_1$, then by the correctness of $\mathcal{Y}_1$, they need not be implemented differently (for $\alpha^{\mathcal{Y}_1}$ implements them the same).

We must show that for every workload trace $\langle w_0, l_1, w_1, \dots\rangle$ causing the correct $\mathcal{Y}_1$ implementation to induce the system sequence $\langle s_0, \alpha^{\mathcal{Y}_1}(s_0,l_1), s_1 \dots\rangle$ that the $\mathcal{Y}_2$ implementation induces the state sequence $\langle t_0, \alpha^{\mathcal{Y}_2}(t_0,l_1), t_1 \dots\rangle$ where $w_i \equiv_Q t_i$. But that is immediate by transfinite induction since by correctness $w_i \equiv_Q s_i$ and by construction $s_i \equiv_Q t_i$. □

In practice, the finite-reachability condition in the above theorem is often no real restriction at all. If both of the access control systems ensure that every state is finitely reachable from every other state (a condition we call *fully finitely connected*), then every reduction between them preserves finite-reachability. (Not all real systems have this property though. For example, incarnations of Bell-LaPadula define the set of security levels (*e.g.*, unclassified, classified, secret, top-secret) as a state variable L but provide no command for changing L. This results in a single "system" that is in reality a set of BLP systems, each of which is entirely disconnected from the others.) Thus when comparing the relative expressiveness of fully connected systems in terms of correctness, it is really the access control models of those systems that we are comparing.

Corollary 1: Suppose there is a reduction $\langle\sigma,\pi\rangle$ from access control model $\mathcal{M}_1$ to $\mathcal{M}_2$ where σ is 1-1. Then for any system $\mathcal{Y}_1$ of $\mathcal{M}_1$ and any fully finitely connected system $\mathcal{Y}_2$ of $\mathcal{M}_2$, we have $\mathcal{Y}_1 \leq \mathcal{Y}_2$.

Proof: Since $\mathcal{Y}_2$ is fully connected every state is finitely reachable from every other state; hence, it makes no difference which states σ maps to because they always connect to each other. Thus σ must preserve finite-reachability and Theorem 2 therefore guarantees the result. □

This corollary is an example of comparing two access control *models* instead of two access control *systems*, which gives rise to a version of relative expressiveness for comparing two access control *models*. It starts with the idea that we can lift the notion of "implementability" from a system to a model: workload $\mathcal{W}$ is implementable with security guarantees $\mathcal{G}$ in access control model M if there exists some system $\mathcal{Y}$ for $\mathcal{M}$ that admits an implementation of $\mathcal{W}$ with guarantees $\mathcal{G}$. Access control model $\mathcal{M}_1$ is less expressive[4] than model $\mathcal{M}_2$ with respect to guarantees $\mathcal{G}$ if for every workload $\mathcal{W}$ that $\mathcal{M}_1$ can implement

[4]Technically, "no more expressive"

with guarantees $\mathcal{G}$, $\mathcal{M}_2$ can implement with guarantees $\mathcal{G}$ as well, written $\mathcal{M}_1 \leq^{\mathcal{G}} \mathcal{M}_2$. That is, if $\mathcal{M}_1 \leq^{\mathcal{G}} \mathcal{M}_2$ then the systems for model $\mathcal{M}_1$ are collectively no more powerful than the systems of $\mathcal{M}_2$ for guarantees $\mathcal{G}$.

Definition 16 (Parameterized Expressiveness (Models)): Suppose that for all workloads $\mathcal{W}$ if model $\mathcal{M}_1$ has a system that correctly implements $\mathcal{W}$ with security guarantees $\mathcal{G}$ then model $\mathcal{M}_2$ also has a system that correctly implements $\mathcal{W}$ with guarantees $\mathcal{G}$. Then we say that $\mathcal{M}_1$ is no more expressive than $\mathcal{M}_2$ with respect to $\mathcal{G}$, written $\mathcal{M}_1 \leq^{\mathcal{G}} \mathcal{M}_2$. ♦

We now prove that if the only property we care about is correctness, showing a reduction between two models tells us that one is less expressive than another. The key difference between this corollary and the previous one is that we must exhibit a fully finitely connected system for the more expressive model, since the previous corollary is true whether such a system exists or not.

Corollary 2 (Model Reductions for $\leq$): If there is a reduction $\langle \sigma, \pi \rangle$ from access control model $\mathcal{M}_1$ to model $\mathcal{M}_2$ where σ is 1-1 then $\mathcal{M}_1 \leq \mathcal{M}_2$.

Proof: Corollary 1 guarantees that if there is a fully finitely connected system $\mathcal{Y}_2^*$ of $\mathcal{M}_2$ then for every system $\mathcal{Y}_1$ of $\mathcal{M}_2$, we have $\mathcal{Y}_1 \leq \mathcal{Y}_2^*$. This ensures that $\mathcal{Y}_2^*$ is capable of correctly implementing any workload that $\mathcal{M}_1$ can implement, giving us the conclusion we desire. It suffices, therefore, to argue that a fully finitely connected system of $\mathcal{M}_2$ exists. But this is easy because there are no restrictions on that system. So create one label for each state and from each state to every other state create an edge with one of those labels. Thus every state is reachable from every other state in a single step, ensuring fully finite connectivity. □

The following lemma tells us something about when state-mappings are not 1-1.

Definition 17 (Trace Equivalent States): If $\langle w_0, \beta_1, w_1, \dots \rangle$ is a workload trace, we say that $\langle w_i, \beta_{i+1}, w_{i+1}, \dots \rangle$ is a subtrace for w_i. Two workload states t_0 and u_0 are trace equivalent if for all subtraces for t_0 and all subtraces for u_0 that have the same command-sequence, *i.e.*, for all subtraces $\langle t_0, \beta_1, t_1, \dots \rangle$ and $\langle u_0, \beta_1, u_1, \dots \rangle$, we have that $t_i \equiv_Q u_i$. ♦

Lemma 1: Suppose $\langle \alpha, \sigma, \pi \rangle$ is a correct implementation of workload $\mathcal{W}$ in system $\mathcal{Y}$. Then if for two workload states w_1 and w_2, we have $\sigma(w_1) = \sigma(w_2)$, then w_1 and w_2 are trace equivalent.

Proof: Since the implementation is correct, we know that the sequence of system states induced by every workload trace is query-equivalent to the workload trace state-sequence: for all workload traces $\langle w_0, \beta_1, w_1, \dots \rangle$, the implementation induces $\langle \sigma(w_0), \alpha(\sigma(w_0), \beta_1), \sigma(w_1), \dots \rangle$ where $w_i \equiv_Q \sigma(w_i)$. Since the same holds for any subtrace, for any subtrace $\langle t_0, \beta_1, t_1, \dots \rangle$ for t_0, the system will induce $\langle \sigma(t_0), \alpha(\sigma(t_0), \beta_1), \sigma(t_1), \dots \rangle$. By transfinite induction, it is easy to see that for any subtrace $\langle u_0, \beta_1, u_1, \dots \rangle$ for u_0 that $\sigma(u_i) = \sigma(t_i)$ and hence that $t_i \equiv_Q u_i$. The base case is the condition of the lemma. Then, if $\sigma(t_k) = \sigma(u_k)$, we know $\alpha(\sigma(t_k), \beta_k) = \alpha(\sigma(u_k), \beta_k)$ and therefore that $\sigma(t_{k+1}) = \sigma(u_{k+1})$. □

The next security guarantee we consider is AC-preservation. If there is a correctness preserving reduction between two systems, and that reduction is itself AC-preserving, then any workload implementable under AC-preservation in the first system is also implementable under AC-preservation in the second system. The key to the proof is the simple observation that the composition of two AC-preserving query mappings is an AC-preserving query mapping.

Lemma 2 (AC Preservation Transitivity): If π_1 and π_2 are AC-preserving query-mappings then $\pi_1(\pi_2(x))$ is an AC-preserving query-mapping.

Proof: We must show that for all theories x that $auth(r) \in \pi_1(\pi_2(x))$ if and only if $auth(r) \in x$.

$$\begin{array}{l} auth(r) \in \pi_1(\pi_2(x)) \\ \qquad \text{(by AC-preservation of } \pi_1) \\ auth(r) \in \pi_2(x) \\ \qquad \text{(by AC-preservation of } \pi_2) \\ auth(r) \in x \end{array}$$

□

Theorem 3 (System Reduction for $\leq^A$): If there is a reduction $\langle \sigma, \pi \rangle$ from $\mathcal{Y}_1$ to $\mathcal{Y}_2$ where σ is one-to-one and preserves finite reachability and π is AC-preserving, then $\mathcal{Y}_1 \leq^A \mathcal{Y}_2$.

Proof: This proof is exactly the same as for Theorem 2, except at the end we apply Lemma (2) to conclude that the query mapping constructed in that proof is AC-preserving. □

Again, this tells us more about the underlying access control models than it does about the systems we are analyzing.

Corollary 3: Suppose there is a reduction $\langle \sigma, \pi \rangle$ from access control model $\mathcal{M}_1$ to model $\mathcal{M}_2$ where σ is 1-1 and π is AC-preserving. Then for any system $\mathcal{Y}_1$ of $\mathcal{M}_1$ and any fully finitely connected system $\mathcal{Y}_2$ of $\mathcal{M}_2$, we have $\mathcal{Y}_1 \leq^A \mathcal{Y}_2$.

Proof: Since $\mathcal{Y}_2$ is fully connected every state is finitely reachable from every other state; hence, it makes no difference which states σ maps to because they always connect to each other. Thus σ must preserve finite-reachability and Theorem 3 therefore guarantees the result. □

Just as with correctness, it is easy to construct a fully finitely connected system for any given model; hence, an AC-preserving reduction between two models guarantees that one model is less expressive than the other with respect to AC-preservation (and correctness).

Corollary 4 (Model Reductions for $\leq^A$): If there is a reduction $\langle \sigma, \pi \rangle$ from access control model $\mathcal{M}_1$ to model $\mathcal{M}_2$ where σ is 1-1 and π is AC-preserving then $\mathcal{M}_1 \leq^A \mathcal{M}_2$.

Proof (sketch): The proof is basically the same as for Corollary 2. □

We now turn out attention to homomorphisms. Unlike AC-preservation, where we only needed to require the reduction between two systems be AC-preserving to ensure $\leq^A$, requiring the reduction be homomorphic does not ensure $\leq^H$. To ensure $\leq^H$, we must also know that the systems themselves are homomorphic.

Theorem 4 (Reduction for $\leq^H$): Consider the case of extensional workloads and access control systems. If there is a reduction $\langle \sigma, \pi \rangle$ from $\mathcal{Y}_1$ to $\mathcal{Y}_2$ then $\mathcal{Y}_1 \leq^H \mathcal{Y}_2$ under the following conditions.

- σ is one-to-one, preserves finite reachability, and is homomorphic
- π is homomorphic
- $\mathcal{Y}_1$ and $\mathcal{Y}_2$ (*i.e.*, their transition functions and query computations) are homomorphic

Proof: In this proof we choose any fixed permutation U of the universe of strings $\mathcal{U}$. When given a correct implementation of $\mathcal{Y}_1$ utilizing U as an argument of its command-mapping, we demonstrate how to construct a correct implementation for $\mathcal{Y}_2$ also using U as the argument of its command-mapping. Thus, U is fixed, but unknown to the implementation, a detail that we need only be concerned with in the case of homomorphic implementations.

In this proof, we begin with the proof of Theorem 2, which demonstrates the existence of a correct implementation $\langle \alpha^{\mathcal{Y}_2}, \sigma^{\mathcal{Y}_2}, \pi^{\mathcal{Y}_2} \rangle$ for $\mathcal{Y}_2$ under weaker assumptions. Since in that construction $\sigma^{\mathcal{Y}_2}$ and $\pi^{\mathcal{Y}_2}$ are defined as the composition of two functions (see below) that in this proof are homomorphic, they are immediately homomorphic themselves. This is why the reduction and $\mathcal{Y}_1$ must be homomorphic. (Proof that the composition of homomorphic functions is a homomorphic function. If $f(\bar{x}[v]) = f(\bar{x})[v]$ and $g(\bar{x}[v]) = g(\bar{x})[v]$, then $f(g(\bar{x}))[v] = f(g(\bar{x})[v]) = f(g(\bar{x}[v]))$.)

$$\text{for all } x \in \mathit{States}(\mathcal{W}). \sigma^{\mathcal{Y}_2}(x) = \sigma(\sigma^{\mathcal{Y}_1}(x))$$
$$\text{for all } x \in \mathit{States}(\mathcal{Y}_2). \pi^{\mathcal{Y}_2}(\mathit{Th}(x)) = \pi^{\mathcal{Y}_1}(\pi(\mathit{Th}(x)))$$
□

Thus we need only demonstrate how to construct $\alpha^{\mathcal{Y}_2}$ that is both homomorphic and correct. To do that, recall how $\alpha^{\mathcal{Y}_2}$ was originally constructed. Consider a workload label l and a workload state w such that some trace executes l in w resulting in w'. If $\sigma^{\mathcal{Y}_1}(w) = s$ and the command-mapping for $\mathcal{Y}_1$ transitions from s to s', then ensure that the command-mapping for $\mathcal{Y}_2$ transitions from $\sigma(s)$ to $\sigma(s')$—a transition that always exists. In this proof, we build the command-mapping the same way except that we make the U argument explicit and carefully consider the impact of assigning values to $\alpha^{\mathcal{Y}_2}(x, y, z)$ when $z \neq U$. These careful assignments make $\alpha^{\mathcal{Y}_2}$ homomorphic and are only possible under the conditions above.

Consider a $\mathcal{Y}_2$ state y representing some workload state w and workload label l where some trace executes l in w. If $\mathcal{Y}_1$ transitions $\sigma(w) = s$ to s' and $\alpha^{\mathcal{Y}_1}(s, l) = \bar{m}$ and there is a finite path with labels $\bar{n}$ in $\mathcal{Y}_2$ from $\sigma(s)$ to $\sigma(s')$, then assign $\alpha^{\mathcal{Y}_2}(\sigma(s), l, U) = \bar{n}$ for one such $\bar{n}$. Moreover, for every constant substitution v, assign $\alpha^{\mathcal{Y}_2}(\sigma(s)[v], l[v], U[v]) = \bar{n}[v]$, as long as it is in the proper domain. Assuming well-definedness (*i.e.*, there is no $\alpha^{\mathcal{Y}_2}(x, y, z)$ assigned two distinct values) it is clear that $\alpha^{\mathcal{Y}_2}$ is correct as long as the third argument is U since those values are all defined the same way as in Theorem 2. Moreover, it is easy to see that by construction $\alpha^{\mathcal{Y}_2}$ is homomorphic with respect to U: for those w,l participating in a trace, if $\alpha^{\mathcal{Y}_2}(\sigma^{\mathcal{Y}_2}(w), l, U) = \bar{n}$ then for any v we have $\alpha^{\mathcal{Y}_2}(\sigma^{\mathcal{Y}_2}(w)[v], l[v], U[v]) = \bar{n}[v]$. Since these assignments suffice for demonstrating correctness and homomorphisms, we can freely choose any values for the unassigned entries in $\alpha^{\mathcal{Y}_2}$.

To complete the proof we must argue two things. First we must show that $\alpha^{\mathcal{Y}_2}(\sigma(s)[v], l[v], U[v]) = \bar{n}[v]$ is a proper assignment: that $\bar{n}[v]$ is a legitimate sequence of labels in $\mathcal{Y}_2$. Second we must show that there is no $\alpha^{\mathcal{Y}_2}(x, y, z)$ that is assigned two distinct values. This completes the proof.

To see that $\alpha^{\mathcal{Y}_2}(\sigma(s)[v], l[v], U[v]) = \bar{n}[v]$ is a proper assignment we need only show that $\bar{n}[v]$ is a legitimate sequence of labels in $\mathcal{Y}_2$. That only requires showing that each label in $\bar{n}[v]$ is a legitimate label in $\mathcal{Y}_2$. The key observation is that $next_2$ (the transition relation for $\mathcal{Y}_2$) is homomorphic: $next_2(y, l) = y'$ ensures that $next_2(y[v], l[v]) = y'[v]$ Since each label l in $\bar{n}$

is legitimate, we see that for $next_2$ to be homomorphic, $l[v]$ must also be a legitimate label. This ensures all the assignments we make are proper ones.

To see the algorithm above assigns at most one value to each $\alpha(y, l, z)$, we argue by contradiction. Suppose there are two values assigned. One possibility is that there are two workload states w_1 and w_2 such that $\sigma^{\mathcal{Y}_2}(w_1) = \sigma^{\mathcal{Y}_2}(w_2) = y$. As argued in Theorem 2, we can assign $\alpha(y, l, U)$ the same thing in both cases; hence, in the construction above we only assign a new value if one has not been assigned. Another possibility is that there are two distinct constant substitutions v_1 and v_2 such that $\langle y[v_1], \beta[v_1], U[v_1]\rangle = \langle y[v_2], \beta[v_2], U[v_2]$. But since a constant substitution is a function from $\mathcal{U} \to \mathcal{U}$, and U is an ordering on $\mathcal{U}$, every distinct pair of constant substitutions ensures that $U[v_1] \neq U[v_2]$. The last possibility is that there are two distinct y_1, l_1 and y_2, l_2 and two variable assignments v_1 and v_2 such that $\langle y_1[v_1], l_1[v_1], U[v_1]\rangle = \langle y_2[v_2], l_2[v_2], U[v_2]\rangle$. The only danger is if $U[v_1] = U[v_2]$, but this only happens when $v_1 = v_2$, which requires $y_1 = y_2$ and $l_1 = l_2$. Thus, there is no problem that arises from assigning multiple values to $\alpha(y, l, z)$.

We can rewrite this theorem so it focuses on the access control *models* instead of *systems*.

Corollary 5: Consider the case of extensional workloads and access control models and systems. Suppose there is a reduction $\langle \sigma, \pi \rangle$ from access control model $\mathcal{M}_1$ to model $\mathcal{M}_2$. Then under the following conditions, where $\mathcal{Y}_1$ is a system of $\mathcal{M}_1$ and $\mathcal{Y}_2$ is a system of $\mathcal{M}_2$, we have $\mathcal{Y}_1 \leq^H \mathcal{Y}_2$.

- σ is one-to-one and is homomorphic
- π is homomorphic
- $\mathcal{Y}_1$ and $\mathcal{Y}_2$ (*i.e.*, their transition functions and query computations) are homomorphic
- $\mathcal{Y}_2$ is fully finitely connected

Proof: Since $\mathcal{Y}_2$ is fully finitely connected every state is finitely reachable from every other state; hence, it makes no difference which states σ maps to because they always connect to each other. Thus σ must preserve finite-reachability and Theorem 4 therefore guarantees the result. □

Unlike for correctness and AC-preservation, a homomorphic reduction between models $\mathcal{M}_1$ and $\mathcal{M}_2$ does not imply $\mathcal{M}_1 \leq^H \mathcal{M}_2$. The reason is that in general it is difficult to construct a homomorphic, fully finitely connected system for any given model because the homomorphism requirement is a strong restriction on the candidate systems—in particular on the $next$ function of those systems. Recall that if label l causes a system to transition from state s_1 to state s_2 then $next(s_1, l) = s_2$. For $next$ to be homomorphic, every constant in s_2 must either appear in s_1 or in l. This means, for example, that we cannot in one step transition from a finite state to an infinite state (since both s_1 and l contain a finite number of constants but s_2 contains infinitely many constants). In fact, if an extensional access control model includes both finite and infinite states, it clearly admits no homomorphic, fully finitely connected system. Nevertheless, for a particular access control model, it is often easy to demonstrate a homomorphic, fully finitely connected system and therefore the parameterized expressiveness of access control models, which is what we do for our case study in Section VI.

Reductions for the three guarantees we have studied so far (correctness, AC-preservation, and homomorphisms) are transitive, ensuring that if we show a reduction between $\mathcal{Y}_1$ and $\mathcal{Y}_2$ and another reduction between $\mathcal{Y}_2$ and $\mathcal{Y}_3$, we know that there must also be a reduction from $\mathcal{Y}_1$ to $\mathcal{Y}_3$.

Proposition 1: Suppose ρ_1 is a reduction from $\mathcal{Y}_1$ to $\mathcal{Y}_2$ and ρ_2 is a reduction from $\mathcal{Y}_2$ to $\mathcal{Y}_3$, where ρ_1 and ρ_2 are both any subset of {1-1, preserve finite reachability, homomorphic, AC-preserving}. Then there is a reduction from $\mathcal{Y}_1$ to $\mathcal{Y}_2$ with the same properties.

Proof: Suppose $\rho_1 = \langle \sigma_1, \pi_1 \rangle$ and $\rho_2 = \langle \sigma_2, \pi_2 \rangle$. Then we define the reduction from $\mathcal{Y}_1$ to $\mathcal{Y}_2$ as follows.

$$\pi_3(x) = \pi_1(\pi_2(x))$$
$$\sigma_3(x) = \sigma_2(\sigma_1(x))$$

First we show this reduction has the properties required above, and then we show that it is a reduction (*i.e.*, that the state-mapping preserves the query-mapping). If σ_1 and σ_2 are 1-1, then so is σ_3, since composition preserves 1-1-ness. If π_1 and π_2 are AC-preserving, then so is π_3, by Lemma (2). If σ_1 and σ_2 are homomorphic, then so is σ_3, since composition preserves homomorphisms (as proven in Theorem 4). Finally, we show that if σ_1 preserves finite reachability and σ_2 preserves finite reachability, that σ_3 preserves finite reachability as well. If s and s' are finitely connected in $\mathcal{Y}_1$ then we know that $\sigma_1(s)$ and $\sigma_1(s')$ are finitely connected in $\mathcal{Y}_2$ and in turn that $\sigma_2(\sigma_1(s))$ and $\sigma_2(\sigma_1(s'))$ are finitely connected in $\mathcal{Y}_3$. Applying the definition of σ_3 therefore immediately gives $\sigma_3(s)$ and $\sigma_3(s')$ are finitely connected.

All that remains is to show that $\langle \sigma_3, \pi_3 \rangle$ is a proper reduction: that the state mapping preserves the query mapping. First, we know that the original two mappings are proper reductions, and hence that

$$\text{for all states } s \in \mathit{States}(\mathcal{Y}_1).\pi_1(\mathit{Th}(\sigma_1(s))) = \mathit{Th}(s)$$
$$\text{for all states } s \in \mathit{States}(\mathcal{Y}_2).\pi_2(\mathit{Th}(\sigma_2(s))) = \mathit{Th}(s).$$

We must show the same holds of σ_3 and π_3 over the states of $\mathcal{Y}_1$. We proceed as follows.

$$\begin{aligned}
&\pi_3(\mathit{Th}(\sigma_3(s))) \\
&\qquad \text{By defs of } \pi_3 \text{ and } \sigma_3 \\
&= \pi_1(\pi_2(\mathit{Th}(\sigma_2(\sigma_1(s))))) \\
&\qquad \text{By query-preservation of } \sigma_2, \pi_2 \\
&= \pi_1(\mathit{Th}(\sigma_1(s))) \\
&\qquad \text{By query-preservation of } \sigma_1, \pi_1 \\
&\mathit{Th}(s)
\end{aligned}$$

□

Parameterized expressiveness is a practical idea that helps analysts more quickly choose the right access control system for their application. As an added benefit, the conditions under which reductions preserve different security guarantees provide insight into the basic definitions of our framework and how they interact. The next question we answer is motivated by a desire to simply understand our framework: does our formal definition of and our intuitive understanding of an *implementation* coincide?

One of the crucial intuitions that we expect to be true of a proper definition of *implementation* is that it is monotonic: that the combination of any two workloads is harder to implement than either of the workloads individually. By "harder to implement" we mean that if a system $\mathcal{Y}$ implements the combination of two workloads then it must be able to implement each workload independently; however, just because $\mathcal{Y}$ implements each of two distinct workloads does not mean $\mathcal{Y}$ can implement their combination. This monotonicity property does in fact hold of our definitions, at least for a reasonable formalization of *workload combination*.

Combining workloads requires addressing two things: combining the access control systems of those workloads and combining the traces of the workloads. Combining the systems is a simple matter: we compute the cross-product of the machines, thereby making all the the commands from both workloads available at every state. A formal definition can be found in Definition (12) in Section IV-F. Combining the traces, however, can be performed in many different ways. Here we do not choose one particular scheme for combining traces; instead, we simply require that the combination of traces include the union of the traces from the two original workloads. We then show that monotonicity holds for any such workload combinator.

Definition 18 (Workload Combinator): A combinator for two workloads $\mathcal{W}_1 = \langle \mathcal{Y}_1, \mathcal{T}_1 \rangle$ and $\mathcal{W}_2 = \langle \mathcal{Y}_2, \mathcal{T}_2 \rangle$, yields a workload $\mathcal{W}_1 \cup \mathcal{W}_2 = \langle \mathcal{Y}_1 \times \mathcal{Y}_2, \mathcal{T} \rangle$ where

$$\begin{aligned}
\mathcal{T} \quad \supseteq \quad & \{\langle s \times s_2, \tau \rangle \mid \langle s, \tau \rangle \in \mathcal{T}_1, s_2 \in \mathit{States}(\mathcal{Y}_2)\} \\
\cup \quad & \{\langle s_1 \times s, \tau \rangle \mid \langle s, \tau \rangle \in \mathcal{T}_2, s_1 \in \mathit{States}(\mathcal{Y}_1)\}
\end{aligned}$$

♦

Proposition 2 (Monotonicity): For any workload $\mathcal{W}_1$, the set of systems correctly implementing $\mathcal{W}_1$ is a superset of the systems implementing $\mathcal{W}_1 \cup \mathcal{W}_2$ for any workload $\mathcal{W}_2$ whose queries are distinct from $\mathcal{W}_1$.

Proof: We demonstrate that every system $\mathcal{Y}$ that correctly implements $\mathcal{W}_1 \cup \mathcal{W}_2$ must also correctly implement $\mathcal{W}_1$. To do so, we construct an implementation $i_1 = \langle \alpha_1, \sigma_1, \pi_1 \rangle$ of $\mathcal{W}_1$ from an implementation $i = \langle \alpha, \sigma, \pi \rangle$ of $\mathcal{W}_1 \cup \mathcal{W}_2$. Define α_1 to be the restriction of α to $\mathit{Labels}(\mathcal{W}_1)$, *i.e.*, $\alpha_1(x, l)$ is only defined for $l \in \mathit{Labels}(\mathcal{W}_1)$ and is equal to $\alpha(x, l)$. Notice that since x ranges over the states of $\mathcal{Y}$, α_1 has the right type for an implementation of $\mathcal{W}_1$ in system $\mathcal{Y}$.

The construction of σ_1 is more difficult because the domain of σ is over states that combine the states from $\mathcal{W}_1$ and the states from $\mathcal{W}_2$; thus, we must define σ_1 over a domain that is not a subset of σ's domain. The states in the domain of σ all take the form $\langle s_1, s_2 \rangle$. Choose any arbitrary $s_2^* \in \mathit{States}(\mathcal{W}_2)$. Define $\sigma_1(s_1)$ to be equal to $\sigma(\langle s_1, s_2 \rangle)$. This construction ensures σ_1 has the right types because it maps states of workload $\mathcal{W}_1$ to states of system $\mathcal{Y}$.

For π_1, we need to dictate how to compute the queries of $\mathcal{W}_1$ from the states of $\mathcal{Y}$. Since the queries of $\mathcal{W}_1$ are distinct from those of $\mathcal{W}_2$, we can define π_1 to be the restriction of π to the queries of $\mathcal{W}_1$, *i.e.*, $\pi_1(x) = \pi(x) \cap \mathit{Queries}(()\mathcal{W}_1)$.

We must now show that this implementation is correct for all traces in $\mathcal{W}_1$. Consider any trace $\langle s_1, \tau \rangle$ from $\mathcal{W}_1$. By construction of $\mathcal{W}_1 \cup \mathcal{W}_2$, we know the trace $\langle s_1 \cup s_2^*, \tau \rangle$ is a trace in $\mathcal{W}_1 \cup \mathcal{W}_2$. Because the commands of $\mathcal{W}_1$ ignore the state of workload $\mathcal{W}_2$, the workload state sequence induced by the trace from $\mathcal{W}_1$ is exactly the same as the workload state sequence induced by the trace from $\mathcal{W}_1 \cup \mathcal{W}_2$, except each state in the latter has s_B^* added to it. Since $\sigma_1(s_1) = \sigma(\langle s_1, s_B^* \rangle)$ and $\alpha_1(x, l) = \alpha(x, l)$ for labels from our trace, by straightforward transfinite induction, we see that the two workload implementations yield the same state sequence in the system $\mathcal{Y}$. Since the queries of $\mathcal{W}_1$ and $\mathcal{W}_2$ are distinct, those state sequences yield the same query-sequence when restricted to $\mathcal{W}_1$'s queries. Since implementation i was correct, so too must implementation i_1 be correct.

□

VI. CASE STUDIES

During our case studies we evaluated several well-known access control systems against two workloads: the coalition workload used as a running example throughout the paper and a workload envisioned for a hospital management system. We analyzed four variants of the access matrix (AM) model, three variants of the basic role-based access control (RBAC) model, and three variants of the Bell-LaPadula (BLP) model. For lack of space, below we present only the four access matrix models AMa, AMb, AMc, and AMd and one BLP model, BLPb. (RBACa serves as the running example in the main body of the paper.) We report results for each workload on each candidate access control system for a broad spectrum of the security guarantees introduced in this paper.

A. Candidate Systems

1) Access Matrix: First we analyzed four variants of the Access Matrix model: AMa, AMb, AMc, and AMd. AMb is a common definition for the access matrix model. AMa simplifies it by not storing the types of subjects, objects, and rights. AMc differs from AMb in that the matrix is not required to obey the types (though it is permitted to); rather, the types are used in the definition of the authorization policy. AMd allows the matrix to be completely independent from the types, just as AMc, but does not even use the types for the authorization policy. AMb and AMd are quite similar except that AMd allows the matrix to be any triple of strings while AMb requires the matrix to be a subset of the cross product of the subjects, objects, and rights types.

Definition 19 (Access Matrix): $\mathcal{U}$ is the set of all strings.

- AMa: $\langle m \rangle$
  - $m \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{U}$

$$auth(x,y,z) \iff m(x,y,z)$$

- AMb: $\langle m, S, O, R \rangle$
  - $S \subseteq \mathcal{U}$, the set of legitimate users
  - $O \subseteq \mathcal{U}$, the set of legitimate documents
  - $R \subseteq \mathcal{U}$, the set of legitimate rights
  - $m \subseteq S \times O \times R$

$$auth(x,y,z) \iff m(x,y,z)$$

- AMc: $\langle m, S, O, R \rangle$
  - $S \subseteq \mathcal{U}$, the set of legitimate users
  - $O \subseteq \mathcal{U}$, the set of legitimate documents
  - $R \subseteq \mathcal{U}$, the set of legitimate rights
  - $m \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{U}$

$$auth(x,y,z) \iff s(x) \wedge o(y) \wedge r(z) \wedge m(x,y,z)$$

- AMd: $\langle m, S, O, R \rangle$
  - $S \subseteq \mathcal{U}$, the set of legitimate users
  - $O \subseteq \mathcal{U}$, the set of legitimate documents
  - $R \subseteq \mathcal{U}$, the set of legitimate rights
  - $m \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{U}$

$$auth(x,y,z) \iff m(x,y,z)$$

All states are finite. The queries of these models are all the possible instances of S, O, R, and m. ♦

Proposition 3: There are 1-1, AC-preserving, homomorphic reductions yielding $AMa \leq^{AH} AMb$, $AMb \leq^{AH} AMc$, and $AMb \leq^{AH} AMd$.

Proof: We demonstrate state- and query-mappings where the state-mapping is 1-1, the query-mapping is AC-preserving, and both are homomorphic.

(AMa $\rightarrow$ AMb). The AMa state $\langle m \rangle$ is represented by the AMb state $\langle m, S, O, R \rangle$ where S is the set of all x such that $m(x,y,z)$ holds, O is the set of all y where $m(x,y,z)$ holds, and R is the set of all z where $m(x,y,z)$ holds. This state-mapping is 1-1. The query-mapping, which is AC-preserving, is the identity.

$$m_{AMa}(x,y,z) \iff m_{AMb}(x,y,z)$$
$$auth_{AMa}(x,y,z) \iff auth_{AMb}(x,y,z)$$

Both mappings are homomorphic.

(AMb → AMc). The AMb state $\langle m, S, O, R\rangle$ is represented by the AMc state $\langle m, S, O, R\rangle$, *i.e.*, the AMb states are a subset of the AMc states and the state-mapping we demonstrate is the identity, which is 1-1. The query-mapping, which is AC-preserving, is the identity.

$$\begin{aligned}
&m_{AMb}(x, y, z) \iff m_{AMd}(x, y, z)\\
&auth_{AMb}(x, y, z) \iff auth_{AMc}(x, y, z)\\
&S_{AMb}(x) \iff S_{AMc}(x)\\
&O_{AMb}(x) \iff O_{AMc}(x)\\
&R_{AMb}(x) \iff R_{AMc}(x)
\end{aligned}$$

Both mappings are homomorphic.

(AMb → AMd). Same as for AMb → AMc. □

Below we give the commands for the access matrix.

Definition 20 (AM Commands): The following commands are used for all AM models, and we do not distinguish administrative from non-administrative commands.

- *addM(x,y,z)*: add $m(x, y, z)$ to the state
- *delM(x,y,z)*: delete $m(x, y, z)$ from the state

AMb and AMc systems also have the following commands.

- *addS(x)*: add $s(x)$ to the state
- *delS(x)*: delete $s(x)$ from the state
- *addO(x)*: add $o(x)$ to the state
- *delO(x)*: delete $o(x)$ from the state
- *addR(x)*: add $r(x)$ to the state
- *delR(x)*: delete $r(x)$ from the state ♦

2) Role-based Access Control: Now we analyze three variants of the RBAC access control model: RBACa, RBACb, and RBACc. RBACb is the most common definition, where the system keeps track of legitimate subjects, objects, rights, and roles and requires UR and PA to be constrained to those types. RBACa differs in that those types are not explicitly recorded but are implicit in UR and PA. RBACc differs in that the types are recorded but UR and PA are not constrained by those types—the types are only used in the definition of the authorization policy.

Definition 21: We consider three RBAC models: $\langle UR, PA\rangle$ and two versions of $\langle S, R, O, I, UR, PA\rangle$.

- RBACa: $\langle UR, PA\rangle$
  - $UR \subseteq \mathcal{U} \times \mathcal{U}$: user-role assignments
  - $PA \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{U}$: role-object-right assignments

$$auth_{RBACa}(x, y, z) \iff \exists w.ur(x, w) \wedge pa(w, y, z)$$

- RBACb: $\langle S, R, O, I, UR, PA\rangle$
  - $S \subseteq \mathcal{U}$: set of subjects
  - $R \subseteq \mathcal{U}$: set of roles
  - $O \subseteq \mathcal{U}$: set of objects
  - $I \subseteq \mathcal{U}$: set of rights
  - $UR \subseteq S \times R$: user-role assignments
  - $PA \subseteq R \times O \times I$: role-object-right assignments

$$auth_{RBACb}(x, y, z) \iff \exists w.ur(x, w) \wedge pa(w, y, z)$$

- RBACc: $\langle S, R, O, I, UR, PA\rangle$
  - $S \subseteq \mathcal{U}$: set of subjects
  - $R \subseteq \mathcal{U}$: set of roles
  - $O \subseteq \mathcal{U}$: set of objects
  - $I \subseteq \mathcal{U}$: set of rights
  - $UR \subseteq \mathcal{U} \times \mathcal{U}$: user-role assignments
  - $PA \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{U}$: role-object-right assignments

$$\begin{aligned}
&auth_{RBACa}(x, y, z) \iff\\
&s(x) \wedge o(y) \wedge i(z) \wedge \exists w.r(w) \wedge ur(x, w) \wedge pa(w, y, z)
\end{aligned}$$

All states are finite. Queries are all instances of UR, PA, S, R, O, I atoms. ♦

Proposition 4: There are 1-1, AC-preserving, homomorphic reductions yielding $RBACa \leq^{AH} RBACb$ and $RBACb \leq^{AH} RBACc$.

Proof: We demonstrate state- and query-mappings where the state-mapping is 1-1 and the query-mapping is AC-preserving.

(RBACa → RBACb) The RBACa state $\langle UR, PA \rangle$ is represented by the RBACb state $\langle S, R, O, I, UR, PA \rangle$, where S is the set of all x such that $UR(x, y)$ holds, R is the set of all y such that $UR(x, y)$ or $PA(y, z, w)$ holds, O is the set of all z such that $PA(y, z, w)$ holds, and I is the set of all w such that $PA(y, z, w)$ holds. This mapping is 1-1. The query-mapping is the identity.

$$UR_{RBACa}(x, y) \iff UR_{RBACb}(x, y)$$
$$PA_{RBACa}(x, y) \iff PA_{RBACb}(x, y)$$

Both the query-mapping and state-mapping are homomorphic.

(RBACb → RBACc) The RBACb state $\langle S, R, O, I, UR, PA \rangle$ is represented by the RBACc $\langle S, R, O, I, UR, PA \rangle$, *i.e.*, the RBACb states are a subset of the RBACc states. Thus both the state-mapping and query-mapping are the identity and hence AC-preserving, 1-1, and homomorphic. □

Below we give the commands for RBAC.

Definition 22 (RBAC Commands): RBACa, RBACb, and RBACc all include the following commands, all of which are administrative commands.

- *assignUser(u,r)*: add $ur(u, r)$ to the state
- *revokeUser(u,r)*: remove $ur(u, r)$ from the state
- *assignPermission(r,d,i)*: add $pa(r, d, i)$ to the state
- *revokePermission(r,d,i)*: remove $pa(r, d, i)$

RBACb and RBACc also include the following commands.

- *addS(x)*: add $s(x)$ to the state
- *delS(x)*: delete $s(x)$ from the state
- *addO(x)*: add $o(x)$ to the state
- *delO(x)*: delete $o(x)$ from the state
- *addR(x)*: add $r(x)$ to the state
- *delR(x)*: delete $r(x)$ from the state
- *addI(x)*: add $i(x)$ to the state
- *delI(x)*: delete $i(x)$ from the state ♦

3) Bell-LaPadula: Next we study three models based on the Bell-LaPadula (BLP) access control system. Since BLP was originally defined as a system, there are different ways of deriving the model the BLP system is based upon. Two of our variants, BLPb and BLPc, correspond to different derivations, and the third, BLPa, is a simplification of BLPb.

The important thing about BLPb is that it assigns subjects and objects to points on a lattice and then requires that the authorization policy be computed from the combination of that lattice and an access matrix. BLPa simplifies BLPb by removing the access matrix; thus, the authorization policy is only computed from the lattice. BLPc differs from BLPb in that the authorization policy is only computed from the access matrix.

Definition 23 (BLP): We consider three BLP models. The first two assume a fixed set of access modalities: read, write, execute, and append, and the third allows for arbitrary rights.

- BLPa has the following fields
  - C: a set of clearance levels
  - $<$: a total ordering on C
  - P: a set of compartments
  - S: a set of subjects
  - O: a set of objects
  - $clear$: $S \to C \times 2^P$ user clearances
  - $class$: $O \to C \times 2^P$ document classifications

$$\begin{aligned} auth(u,d,read) &\iff class(d) \sqsubseteq clear(u) \\ auth(u,d,append) &\iff class(d) \sqsupseteq clear(u) \\ auth(u,d,write) &\iff class(d) = clear(u) \\ auth(u,d,execute) &\iff true \end{aligned}$$

- BLPb has the following fields
  - C: a set of clearance levels
  - $<$: a total ordering on C
  - P: a set of compartments
  - S: a set of subjects
  - O: a set of objects
  - $clear$: $S \to C \times 2^P$ maximal user clearances
  - $class$: $O \to C \times 2^P$ document classifications
  - $clear_c$: $S \to C \times 2^P$ current user clearances
  - $b \subseteq S \times O \times \{read, write, execute, append\}$ records the set of accesses employed currently
  - $m \subseteq S \times O \times \{read, write, execute, append\}$ is a discretionary access matrix

  In all states, S, O, $clear$, $class$, $clear_c$, b, and m are finite, $clear_c(x) \sqsubseteq clear(x)$, and $b(x,y,z) \Rightarrow auth(x,y,z)$. The following have been slightly simplified from the original since they ignore the "is trusted" condition and by applying the first constraint below.

$$\begin{aligned} & auth(u,d,read) \iff \\ & \quad m(u,d,read) \wedge class(d) \sqsubseteq clear_c(u) \\ & auth(u,d,append) \iff \\ & \quad m(u,d,append) \wedge clear_c(u) \sqsubseteq class(d) \\ & auth(u,d,write) \iff \\ & \quad m(u,d,write) \wedge clear_c(u) = class(d) \\ & auth(u,d,execute) \iff \\ & \quad m(u,d,execute) \end{aligned}$$

- BLPc has the following fields
  - C: a set of clearance levels
  - $<$: a total ordering on C
  - P: a set of compartments
  - S: a set of subjects
  - O: a set of objects
  - R: a set of rights
  - $clear$: $S \to C \times 2^P$ maximal user clearances
  - $class$: $O \to C \times 2^P$ document classifications
  - $clear_c$: $S \to C \times 2^P$ current user clearances
  - $b \subseteq S \times O \times R$ records the set of accesses employed currently
  - $m \subseteq S \times O \times R$ is a discretionary access matrix

  In all states, S, O, R, $clear$, $class$, $clear_c$, b, and m are finite, $clear_c(x) \sqsubseteq clear(x)$, and $b(x,y,z) \Rightarrow auth(x,y,z)$.

$$auth(u,d,r) \iff m(u,d,r)$$

Above $\langle c_1, P_1 \rangle \sqsubseteq \langle c_2, P_2 \rangle$ is true if and only if $c_1 \leq c_2$ and $P_1 \subseteq P_2$. The queries are all the fields of each model. ♦

First we see the relationships between the various BLP models.

Proposition 5: There is a 1-1, AC-preserving, homomorphic reduction yielding $BLPa \leq^{AH} BLPb$.

Proof: The BLPa state $\langle C, <, P, S, O, R, clear, class \rangle$ is represented by the BLPb state that has the same values for the fields shared with BLPa. For the additional fields, we have (i) $clear_c$ is identical to $clear$: $clear_c(u) = clear(u)$ for all u, (ii) m grants all possible rights: $m = S \times O \times \{read, write, execute, append\}$, and (iii) b is empty. The query-mapping is the identity. Notice that the state mapping preserves the query-mapping because the $auth(x,y,z)$ are computed the same in both systems since $clear_c = clear$ and m is always true. The state-mapping is 1-1, the query-mapping is AC-preserving, and both are homomorphic. □

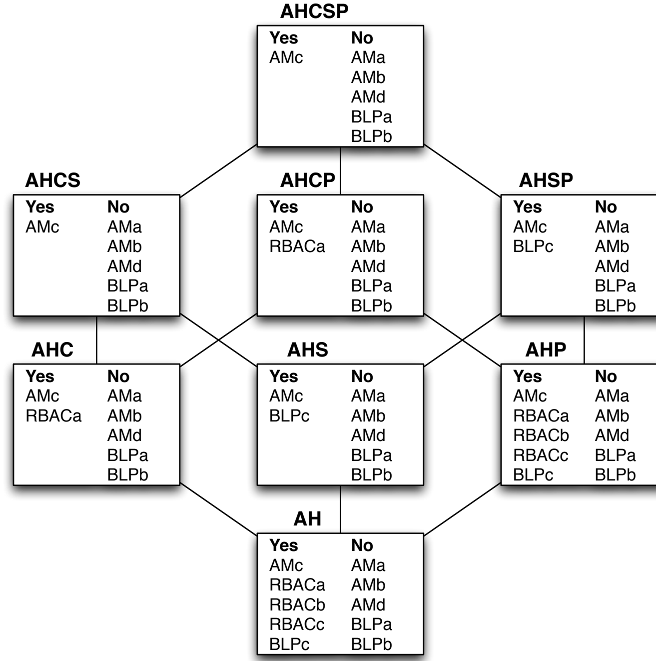

Fig. 1. Summary of the coalition workload results. AMx is an access matrix system; RBACx is a role-based access control system; BLPx is a Bell-LaPadula system. A stands for AC-preservation, H for homomorphism, C for weak compatibility, S for safety, P for administration-preservation. Correctness is required throughout.

Definition 24 (BLP Commands): Each BLP model has commands for changing each of its fields, with notable commands described below. Commands that change C, $<$, P, $class$, $clear$, S, O, and R are administrative commands; all others are not.

- *changeCurrentClearance(issuer, newc, newP)*: sets $clear_c(issuer) = \langle newc, newP \rangle$, unless that clearance is higher than $issuer$'s maximum clearance $clear(issuer)$.
- *addB(issuer, object, right)*: open $object$ with the given $right$ for subject $issuer$ by setting $b(issuer, object, right)$ to true. That is, only allow a user to modify her own slice of b.
- *delB(issuer, object, right)*: delete $b(issuer, object, right)$.
- *setM(issuer, object, right)*: set access rights for $issuer$'s $object$ to $right$ by setting $m(issuer, object, right)$ to true. That is, only allow a user to modify her own slice of m. ♦

B. Dynamic Coalitions Workload

Figure 1 summarizes the results for the coalition workload. Each box details the results for a different set of security guarantees, and the boxes are arranged so that higher boxes have more guarantees than lower boxes. Each box is labeled with letters describing the guarantees the box represents, and the caption details which letter corresponds to which guarantee. The contents of each box includes one column for access control systems that admit an implementation, and those systems that do not; the results for any system not listed in either column are unknown.

Proposition 6: All implementations of the coalition workload are administrator-preserving.

Proof: Every command in the coalition workload is an administrative command. Since the administrator-preserving guarantee is only effective for non-administrative commands, is vacuously true for all implementations of the coalition workload. □

1) Access Matrix: The crucial problem in implementing the coalition workload is representing the $orgUser$ relation. The main difference in the four models is how easy that representation is.

Theorem 5: AMb, AMc, and AMd admit correct, AC-preserving implementations of the coalition workload. AMa fails to admit a correct, AC-preserving implementation.

Proof (sketch): It is easy to represent the $orgUser$ relation of the coalition workload by string-packing it into a constant (*e.g.*, a username). The key, though, is that the constant cannot change the authorization policy, since otherwise we would not be representing the $auth$ atoms correctly. Because AMa has only the matrix m to store information, and m represents the

authorization policy directly, there is no space to store this long constant without violating AC-preservation. But AMb, AMc, and AMd can all store such a constant and therefore admit correct, AC-preserving implementations of the coalition workload. Proof for AMa is done by simple counting: there are fewer AMa states than workload states. Proof for AMb demonstrates (i) how to encode $orgUser$ as a constant and (ii) how to insert it into the state without changing the authorization policy of the state. For (ii), we simply add the constant to the subject type S without adding an entry to m. Proof for AMc and AMd are by reductions from AMb. □

Proof: (AMb) For AMb, include in every state a pseudo-subject that does not appear in m that encodes the entire $orgUser$ relation. To ensure that s does not appear in m, prefix it with the concatenation of all of the legitimate subjects. More precisely, the query mapping π ensures that $auth(r)$ in the workload is equivalent to $auth(r)$ in AMb, and $orgUser(x, y)$ is computed by searching for $orgUser(x, y)$ in the pseudo-subject. The state mapping σ chooses the minimal such AMb state (*e.g.*, the initial empty start state in the coalition workload is mapped to an AMb state that is empty except for the 0-length pseudo-subject string. The command-mapping α adjusts m and the pseudo-subject as required in the obvious way by each *joinCoalition* and *leaveCoalition*. This implementation is AC-preserving because the query $auth(r)$ is mapped to $auth(r)$. It is correct by inspection.

(AMc, AMd) Together with the proof above, the reductions from AMb to AMc and AMd guarantee the required implementations.

(AMa) We show there can be no query-mapping. Consider any two coalition states S and T that have the same authorization policies ($Th(S) = Th(T)$) but different $orgUser$ relations. To be AC-preserving, it must be the case that $auth(r)$ in the workload is given by exactly $auth(r)$ in AMa. Since there is only one AMa state for each theory, that state must be used to represent both S and T. Obviously, there is therefore no query mapping that will yield two different $orgUser$ relations from this state. Since the query mapping does not exist, neither does the implementation. □

When we impose the homomorphism restriction, we can no longer use a single constant to represent $orgUser$, and only AMc admits a correct, homomorphic implementation.

Theorem 6: AMc (but none of AMa, AMb, nor AMd) admits a correct, AC-preserving, homomorphic, safe implementation of the coalition workload.

Proof (sketch): AMa, AMb, and AMd fail (even without safety) because after storing the $auth$ relation, they have at most three unary relations (S, O, R) to store the $orgUser$ relation. Unary relations are inadequate to store a binary relation like $orgUser$, as long as the homomorphism requirement is in place. The proceeds by identifying a particularly troublesome class of traces: those where each organization has one subject. It goes on to apply the definition of homomorphism to demonstrate by counting that there is no way to represent a binary relation (like $orgUser$) with three unary relations (S, O, R).

AMc, on the other hand, can use a portion of the matrix m to store $orgUser$. It differs from the other models because its definition of $auth$ is not the matrix itself but rather a restriction of the matrix to the existing subjects, objects, and rights; any matrix entry mentioning a non-existent subject, object, or right can therefore be used to store $orgUser$. □

Proof: Failures. We show failure for AMd, which by the model reductions from AMa and AMb, ensures failure for AMa and AMb. (Also, the failure of AMa in the non-homomorphic case ensures failure in the homomorphic case.)

We demonstrate a particular kind of $orgUser$ relation that when large enough cannot be represented in AMd. Specifically, we show that there are more workload states than AMd states, ensuring that no state-mapping preserves queries (violating the first property of correctness). Consider any state mapping σ, which maps a $\langle auth, orgUser\rangle$ state from $\mathcal{W}$ to a state $\langle S, O, R, m, auth'\rangle$ from AMd. We then proceed as follows.

$$\begin{aligned}
&\sigma(\langle auth, orgUser\rangle)\\
&= \langle S, O, R, m, auth'\rangle\\
&\qquad \text{by AC-preservation}\\
&= \langle S, O, R, m, auth\rangle\\
&\qquad \text{since for AMd } auth(x, y, z) \iff m(x, y, z)\\
&= \langle S, O, R, auth, auth\rangle\\
&\qquad \text{Suppose the organizations occurring in } orgUser \text{ are}\\
&\qquad K. \text{ We treat } S, O, \text{ and } R \text{ as sets of constants. We}\\
&\qquad \text{partition } S, O, R \text{ into } S_K \cup S', O_K \cup O', \text{ and } R_K \cup R'\\
&\qquad \text{where } S_K = S \cap K, O_K = O \cap K, \text{ and } R_K = R \cap K.\\
&= \langle S' \cup S_K, O' \cup O_K, R' \cup R_K, auth, auth\rangle
\end{aligned}$$

Now consider a specific type of string mapping: a 1-1 mapping $v : K \to K$ (where all other strings are left unchanged). For

the implementation to be homomorphic, the following must apply for all such v.

$$\begin{array}{l}\sigma(\langle auth, orgUser\rangle[v]) \\ = \sigma(\langle auth[v], orgUser[v]\rangle) \\ \quad \text{By homomorphism and the equivalence from above} \\ = \langle S' \cup S_K, O' \cup O_K, R' \cup R_K, auth, auth\rangle[v] \\ = \langle S'[v] \cup S_K[v], O'[v] \cup O_K[v], R'[v] \cup R_K[v], auth[v], auth[v]\rangle \\ \quad \text{Since } v \text{ only affects constants of } K \\ = \langle S' \cup S_K[v], O' \cup O_K[v], R' \cup R_K[v], auth[v], auth[v]\rangle\end{array}$$

Furthermore, suppose the organizations K do not appear in $auth$, *i.e.*, that the subjects, objects, rights, and organizations are all mutually exclusive. Again, the implementation must be homomorphic for all traces, and the mutex assumption is one that can be realized in the workload; hence, we are iteratively refining the class of traces we are considering. Then we have

$$\begin{array}{l}\sigma(\langle auth, orgUser[v]\rangle) \\ = \langle S' \cup S_K[v], O' \cup O_K[v], R' \cup R_K[v], auth, auth\rangle.\end{array}$$

This ensures that the encoding of one variant of $orgUser$ differs from the encoding of every other $orgUser$ variant only in the S_K, O_K, and R_K sets of elements. Because each of S_K, O_K, and R_K are subsets of K, their maximum size is $|K|$. Thus, the total number of distinct $\langle S_K, O_K, R_K\rangle$ triples for this K is $(2^{|K|})^3$. The number of distinct $orgUser$ variants where we only permute the organizations is $|K|!$. The argument above ensures that each of the $|K|!$ distinct coalition states must therefore be mapped to the $(2^{|K|})^3$ AMd states, which for sufficiently large k ensures that two distinct coalition states are mapped to the same AMd state, ensuring there can be query-preserving state-mapping. □

Proof: Success. We give the implementation. First the state and query mappings. To satisfy AC-preservation, the coalition $auth(u, d, right)$ is represented as $s(u) \wedge o(d) \wedge r(right) \wedge m(u, d, right)$ in AMc (and therefore $auth(s, o, r)$ in the workload is computed as $auth(s, o, r)$ in AMc). To represent each $orgUser(u, orgID)$, we utilize a fresh constant sk that appears in none of S, O, or R (and therefore ensures that no entry in m that includes sk will make any $auth$ query true). $orgUser(u, orgID)$ is represented as two m entries: $m(u, sk, sk)$ and $m(sk, orgID, sk)$. The $orgUser(u, orgID)$ tuples are given by the following.

$$\{\langle u, y\rangle \mid m(u, v, x) \wedge m(x, y, z) \text{ where } x, v \text{ are skolems}\}$$ □

The command-mapping preserves the representation above. To implement *joinCoalition*, for all the $\langle u, d, right\rangle$ granted access, add $s(u)$,$o(d)$, $r(right)$, and $m(u, d, right)$. Then for every $u \in U$, add $m(u, sk, sk)$ and $m(sk, orgID, sk)$ for some fresh skolem constant sk. To implement *leaveCoalition(orgID)*, we first compute the orgUser relation. Then we remove the rights for all users belonging to that organization, remove the record that those users are users, and finally remove m entries encoding the $orgUser$ relation for those organizations.

The one technical issue is that a user/doc/right/org ID might arise during a joinCoalition that is the same as one of the skolems. Each time this happens it is simple enough to regenerate that skolem so that it is fresh. The pseudo-code below is expressed in a variant of HPL; hence, the implementation is homomorphic. Furthermore, the implementation is safe because the matrix entries it adds/deletes on a *joinCoalition* to address the uniqueness of skolem constants are those that represent $orgUser$ and do not contribute to $auth$. All of the $auth$ queries made true on *joinCoalition* must be made true. Similarly, *leaveCoalition* is safe.

```
joincoalition(M, newAuth, orgID, Univ) =
  // remove any matrix values encoding orgUser and store those matrix entries
  Let skolems = {x | m(x,y,z) ∈ M and s(x) ∉ M} // or could have found all y not in O
  Let toremove = {m(x,y,z) | z ∈ skolems}
  Let userOrg = {<u,y> | m(u,v,x) , m(x,y,z) where x,v in skolems}
  for each (m(x,y,z) ∈ toremove)
    output(delM(x,y,z))
  // add legit m/s/o/r  tuples
  addBasic = {addS(u), addO(d), addR(g), addM(u,d,g) | <u,d,g> ∈ newAuth }
  outputSet(addBasic)
  // add old+new orgusers
  userOrg = userOrg U {<u,orgID> | <u,d,g> ∈ newAuth}
  Let fs = nfreshConst(|userorg|,Consts(M U newAuth) − skolems,Univ)
  for each (<user,org> ∈ userOrg, f ∈ fs)
    output(addM(user,f,f))
    output(addM(f,org,f))

leavecoalition(M, orgid) =
  // just deleting here, so there's no need to rename skolems
```

```
  Let skolems = {x | m(x,y,z) ∈ M and s(x) ∉ M} // or could have found all y not in O
  Let toremove = {m(user,x,x), m(x,orgid,x)  |
          m(user,x,x) ∈ M, m(x,orgid,x) ∈ M, x ∈ skolems}
  Let userstoremove = {x | m(x,y,z) ∈ toremove}
  toremove = toremove ∪ {s(x), m(x,y,z) | m(x,y,z) ∈ M, x ∈ userstoremove}
  outputSet(toremove)
```

Theorem 7: Neither AMb, AMc, nor AMd admit a correct, strongly compatible implementation of the coalition workload.

Proof: For strong compatibility, the implementations of the AM system commands (for adding/deleting tuples from S, O, R, and m) are implemented as themselves; thus, executing any of the AM workload commands yields exactly the same changes to the underlying data structures as the original commands.

We will demonstrate two workload command sequences that end with query-distinct workload states and argue that any strongly compatible implementation will utilize an command-mapping that yields the same terminal system state for the two sequences. All such implementations are incorrect because if the final system states for the two sequences are the same, then the workload queries made true by any query-mapping are the same, ensuring that the queries in the system's terminal state are incorrect for at least one of the two workload terminal states.

For the first sequence of commands, invoke any number of *joinCoalition* commands resulting in the system state s that results in at least one $orgUser$ tuple being true in the final workload state. For the second sequence of commands, execute the AM workload commands to produce exactly the same state s, where notice no $orgUser$ tuples are true in the workload state. We know because of strong compatibility that the sequence of AM commands yields the state s both in the workload and in the system state spaces. This gives us the two command sequences with query-distinct terminal workload states (because one makes an $orgUser$ tuple true and the other does not) but the same system terminal state. This argument applies equally well to AMb, AMc, AMd. □

Theorem 8: AMc admits a correct, AC-preserving, safe, weakly compatible, homomorphic implementation of the coalition workload.

Proof (sketch): The implementation given in the proof of Theorem 6 can be extended so that the implementations of the AM system commands ensure no name clashes appear with the skolem constants utilized to represent $orgUser$. That is, if one of the commands for modifying m, S, O, or R include a constant that is intended to be a skolem, then that skolem is renamed before the command is executed. That implementation can be written in HPL and changes no $auth$ queries except those required by the workload. □

Proof: We demonstrate the implementation that conservatively extends the correct, AC-preserving, homomorphic implementation given in Theorem 6. The new implementation builds upon the original by adding implementations of the AM system commands that ensure the skolem constants constructed in that implementation are always unique. That is, if one of the commands for modifying m, S, O, or R include a constant that is intended to be a skolem, then that skolem is renamed before the command is executed. This command-mapping, given below for one add and one delete operation, conservatively extends the original implementation.

As for the query mapping, the S, O, and R workload queries are just the contents of the S, O, and R system fields, respectively. The m queries of the workload are true of the m tuples in the system that have no $orgUser$ entry for the subject. The $orgUser$ workload queries are computed just as they were before (which gives the same answer because all the commands preserve our notion of a skolem). The $auth$ queries of the workload are mapped to the $auth$ queries of AMa (to ensure AC-preservation). $auth(s, o, r)$ is therefore true either because $auth(s, o, r)$ is true in the coalition workload or because $auth(s, o, r)$ was made true by the AM system commands; all the extraneous m entries are not part of $auth$ because they include a skolem not in S, O, or R. Thus, this new query mapping is a conservative extension of the original.

Notice also that none of the commands either make $auth$ queries true that ought not be true nor make $auth$ queries false that ought not be false. The only m entries that are changed besides the ones to make $auth$ queries true/false are those for representing $orgUser$ and therefore, by necessity, do not impact $auth$ queries.

```
addM(M,u,d,g) =
  // remove any matrix values encoding userOrg and store those matrix entries
  Let skolems = {x | m(x,y,z) ∈ M and s(x) ∉ M} // or could have found all y not in O
  Let toremove = {m(x,y,z) | z ∈ skolems}
  Let userOrg = {<u,y>\suchthat m(u,v,x) ∧ m(x,y,z) where x,v ∈ skolems}
  for each (m(x,y,z) ∈ toremove)
    output(delM(x,y,z))
  // add new m  tuple
  output(addM(u,d,g))
```

```
  // add back orgusers
  Let fs = nfreshConst(|userorg|,Consts(M ∪ {u,d,g}) − skolems,Univ)
  for each (<user,org> ∈ userOrg, f ∈ fs)
    output(addM(user,f,f))
    output(addM(f,org,f))

delM(M, u,d,g) =
  output(delM(u,d,g))
```

2) RBAC: It turns out that all three RBAC variants admit correct and homomorphic implementations of the coalition workload. To represent each of the workload's $auth(x, y, z)$ atoms, we invent a role r and then make $ur(x, r)$ and $pa(r, y, z)$ true. To represent each of the workload's $orgUser(x, y)$ atoms, we use $UR(x, y)$ and ensure that there is no $PA(y, t, u)$ that is true. The only time the existence of $PA(y, t, u)$ is a problem is when the orgID y happens to have been chosen for a role name. But if that happens, we simply rename the role to something else.

Theorem 9: RBACa, RBACb, and RBACc admit correct, AC-preserving homomorphic implementations of the coalition workload.

Proof: Below we give the command-mapping for RBACa in HPL. The homomorphic reductions from RBACa to RBACb and RBACc, RBACb and RBACc and Proposition 1 ensure that RBACb and RBACc must admit correct, homomorphic implementations.

```
joincoalition(M,orgID,U,D,UDdesc,tag,Univ) =
  // rename any role that happened to be orgID
  Let nameconf = { <u,d,y> | ur(u,orgID) ∈ M and pa(orgID,d,y) ∈ M }
    If nameconf != {} then
      output(delUR(u,orgID))
      output(delPA(orgID,d,y))
      f = freshConst(Consts(M U {orgID}), Univ)
      output(addUR(u,f))
      output(addPA(f,d,y))
    endif
  // add new tuples
  Let toadd = { <u,d,g> | u ∈ U, d ∈ D and there is a <usertag,doctag,g> ∈ UDdesc where
        tag(u) ∈ usertag and tag(d) ∈ doctag}
    addOrgUser = {addUR(u,orgID) | Ed. <u,d> ∈ toadd }
    outputSet(addOrgUser)
    f = nfreshConst(|toadd|, Consts(M U addOrgUser),Univ)
    for each (f ∈ F, <u,d,g> ∈ toadd)
      output(addUR(u,f))
      output(addPA(f,d,g))

leavecoalition(M, orgid) =
  for each ( ur(u,orgid) ∈ M ) )
    // delete orgUser tuple
    output(delUR(u,orgID))
    // delete all ur and rd
    for each ( <u,r,d,g> | ur(u,r) ∈ M and pa(r,d,g) ∈ M )
      output(delUR(u,r))
      output(delPA(r,d,g))
```

The query-mapping is implicit in the above code: $auth$ is computed in the usual RBAC way, and $orgUser$ is computable from the state, as described earlier. The state-mapping chooses the minimal RBAC state needed to represent the workload state in this way. Correctness holds because both the state-mapping and command-mapping preserve the queries (here assumed to be both $auth$ and $orgUser$). □

Theorem 10: RBACa fails to admit a correct, strongly compatible implementation of the coalition workload.

Proof: This is a simple variation on the proof that the AM systems fail to admit a correct, strongly compatible implementation (Theorem 7). The only difference is that the system commands we use are the RBAC commands, instead of the AM commands. Consider any sequence of *joinCoalition* commands that yields a workload state where at least $orgUser$ tuple is true. Suppose the command-mapping for this sequence yields the RBAC system state s. It is easy to see that there is also a sequence of RBAC commands that yields s both in the workload state space and because the implementation is *strongly* compatible, in the system state space. Since these two sequences of workload commands end in query-distinct workload states but in the same system state, the implementation must not be correct as it violates the requirement that the command-mapping must

preserve the query-mapping. This proof applies to all of our RBAC systems because all of the fields in the RBAC models are queryable. □

Theorem 11: RBACa admits a correct, AC-preserving, weakly compatible, homomorphic implementation of the coalition workload.

Proof (sketch): The most important insight is how to represent the workload state ($auth$, $orgUser$, UR and PA) with the system state. Because AC-preservation holds, every coalition $auth$ tuple must be represented as a pair of UR and PA tuples, which we do using fresh roles for each coalition $auth$ tuple. Thus it suffices to show how to represent $orgUser$, UR, and PA in the system state. To represent UR and PA, we first find the set of UR and PA tuples that contribute to the $auth$ policy (which we denote with E), *i.e.*, the set of all $UR(x, y)$ tuples such that there is some $PA(y, z, w)$, together with the set of all $PA(y, z, w)$ such that there is some $ur(x, y)$. The set E is part of the system state. We now have the remaining ur and pa atoms (which we call the dangling atoms) together with the $orgUser$ atoms to represent. Additionally, we store the set of all fresh roles introduced by the implementation for encoding coalition $auth$ atoms (which we call the pseudo-roles). To accomplish this, we encode all 4 sets of atoms as ur atoms so that there is no corresponding pa atom to change the authorization policy from what it should be. This gives the query- and state- mappings and hints at the command-mapping. Since all the mappings are expressible in HPL, the implementation is homomorphic, and by construction it is AC-preserving, as noted above. □

Proof: Here we show how to represent UR, PA, and $orgUser$ tuples all with the ur and pa tuples. Then we show the command-mappings for *joinCoalition*, *leaveCoalition*, and the RBAC system commands in HPL. That code implicitly includes the query-mapping. The state-mapping maps each workload state to the minimal one preserving the query-mapping, as usual.

First we explain how to represent the workload state, which consists of atoms of the form $ur(x, y)$, $pa(x, y, z)$, and $orgUser(x, y)$. To represent ur and pa, we first find the set of ur and pa atoms that contribute to the $auth$ policy (which we denote with E), *i.e.*, the set of all $ur(x, y)$ tuples such that there is some $pa(y, z, w)$, together with the set of all $pa(y, z, w)$ such that there is some $ur(x, y)$. The set E is part of the RBAC state. We now have the remaining ur and pa atoms (which we call the dangling atoms) together with the $orgUser$ atoms to represent. Additionally, we will store the set of all roles appearing in non-dangling ur/pa tuples that were not explicitly added by system commands, which we call the pseudo-roles. To accomplish this, we will encode all 4 sets of atoms as ur atoms so that there is no corresponding pa atom to change the authorization policy from what it should be. Lemma (3) demonstrates how to encode those relations as the binary relation ur; moreover, by inspecting the proof the second argument to every ur tuple is fresh and hence never matches a pa atom. It is easy to see how to extract the true state from the RBAC state.

Now we give the implementation in HPL.

```
// compute entire true state before generating new constants.
// Set is implemented as list and union copies the first list and adds missing things from 2nd
joincoalition(M, orgID, newAuth, Univ) =
  // translate state to set of ur, pa, dangleur, danglepa, orgUser, pseudo atoms
  // and empty the state
  Let truestate = decodeState(M)
  emptyState(M)

  // existing <u,d,g> pairs granted access using fresh roles
  Let toadd = {<x,z,w> | ur(x,y), pa(y,z,w) ∈ truestate and pseudo(y) ∈ truestate}

  // remove toadd ur/pa from truestate as they will be added anew
  for each (<u,d,w> ∈ toadd)
    for each { ur(u,y), pa(y,d,w) ∈ truestate }
      truestate = truestate − {ur(u,y), pa(y,d,w)}

  // add orgUser atoms to truestate
  truestate = truestate U {orgUser(u,orgID) | ∃ d. <u,d,i> ∈ newAuth }

  // encode truestate and add additional auth tuples (that require skolems)
  encodeState(truestate, newAuth U toadd, Univ)

encodeState(M, toadd, Univ) =
  // add nondangling ur/pa
  for each (ur(x,y) ∈ M) output(addUR(x,y))
  for each (pa(x,y) ∈ M) output(addPA(x,y))

  // remaining require fresh constants.
  // Start with non-dangling ur/pa tuples; remember to update pseudo
```

```
  Let C = Consts(M)
  F = nfreshConst(|toadd|, C, Univ)
  M = M U {pseudo(x) | x ∈ F}
  for each (f ∈ F, <u,d,w> ∈ toadd) // iterate both at same time -- zip
    output(addUR(u,f))
    output(addPA(f,d,w))
  C = C U F  // since used constants now larger

  // encode M with uniquification
  map = {}
  for each (r(t1,...,tn) ∈ M)
    Let tuple = <r,t1,...,tn>
    F = nfreshConst(n,C,Univ)
    for each (f ∈ F, a ∈ tuple) // zip
      map = map U {f/a}
    broken = tobinaryRel (totuple(F))  // break <1,2,3,4> into <1,2><2,3><3,4>
    for each (<x,y> ∈ broken)
      output(addUR(x,y))
    C = C U F
  // encode mapping in RBAC state
  for each {f/a ∈ mapping}
    output(addUR(a,f))

decodeState(M) =
  Let nondangling = { ur(x,y), pa(y,z,w) | ur(x,y) ∈ M and pa(y,z,w) ∈ M}
  Let dangling = { ur(x,y) | ur(x,y) ∈ M and there is no z,w s.t. pa(y,z,w) ∈ M}
  Let userOrg,dangUR,dangPA,pseudoRoles = decodeDangling(dangling)
  return nondangling U userOrg U dangUR U dangPA U pseudoRoles

leavecoalition(N, orgid, Univ) =
  Let M = decodeState(N)
  emptyState(N)
  toadd = {}
  for each ( userOrg(u,orgid) ∈ M )
    M = M - {userOrg(u,orgid)}
    for all {ur(u,x), pa(x,y,z) | ur(x,y),pa(y,z,w) ∈ M}
      if pseudo(x) ∈ M
        M = M - {pseudo(x)}
        M = M - {ur(u,x),pa(x,y,z)}
      else
        // need to revoke rights by either removing ur(u,x) or pa(x,y,z)
        // We choose to remove u from role x but then for all other pa(x,y,z)
        //     create new role ur(u,role) and pa(role,y,z)
        M = M - {ur(u,x)}
        for all {pa(x,y,z) ∈ M}
          toadd = toadd U {<u,y,z>}
      endif

  // encode truestate and add additional auth tuples (that require skolems)
  encodeState(M,toadd, Univ)

// can only change true ur/pa by adding a single ur
assignUser(N, u, r, Univ) =
  Let M = decodeState(N)
  emptyState(N)
  if pseudo(r) M = M - {pseudo(r)}
  // non-dangling
  if ∃ x,y. pa(r,x,y) ∈ M or dangpa(r,x,y) ∈ M
    M = M U {ur(u,r)}
    for each dangpa(r,x,y) ∈ M
      M = M - dangpa(r,x,y)
      M = M U pa(r,x,y)
  else  //dangling
    M = M U {dangur(u,r)}
  endif
  encodeState(M,{},Univ)
```

```
revokeUser(N, u, r, Univ) =
  Let M = decodeState(N)
  emptyState(N)
  if dangur(u,r) ∈ M
    M = M − {dangur(u,r)}
  else if ur(u,r) ∈ M
    M = M − {ur(u,r)}
     // if removing ur changes nondangling pas to dangling, update
     if ∄ x. ur(x,r) ∈ M
       for each pa(r,x,y) ∈ M
         M = M − {pa(r,x,y)}
         M = M U {dangpa(r,x,y)}
  endif
  if pseudo(r) ^ !exists dangur(x,r) ∈ M or ur(x,r) ∈ M or dangpa(r,x,y) ∈ M or pa(r,x,y) ∈ M
    M = M − pseudo(r)
  endif
  encodeState(M,{},Univ)

// same as assignUser except operating on pa instead of ur
assignPermission(N, r, d, i, Univ) =
  Let M = decodeState(N)
  emptyState(N)
  if pseudo(r) M = M − {pseudo(r)}
  // non−dangling
  if ∃ x,y. ur(x,r) ∈ M or dangur(x,r) ∈ M
    M = M U {pa(r,d,i)}
     for each dangur(x,r) ∈ M
      M = M − dangur(x,r)
      M = M U ur(x,r))
  else   //dangling
    M = M U {dangur(u,r)}
  endif
  encodeState(M,{},Univ)

// same as revokeUser except operating on pa instead of ur
revokePermission(N, r, d, i, Univ) =
  Let M = decodeState(N)
  emptyState(N)
  if dangpa(r,d,i) ∈ M
    M = M − {dangpa(r,d,i)}
  else if pa(r,d,i) ∈ M
    M = M − {pa(r,d,i)}
     // if removing ur changes nondangling urs to dangling, update
     if ∄ x. pa(r,x,y) ∈ M
       for each ur(x,r) ∈ M
         M = M − {ur(x,r)}
         M = M U {dangur(x,r)}
  endif
  if pseudo(r) ∧ ∄ dangur(x,r) ∈ M or ur(x,r) ∈ M or dangpa(r,x,y) ∈ M or pa(r,x,y) ∈ M
    M = M − pseudo(r)
  endif
  encodeState(M,{},Univ)
```

Lemma 3 (Relation Encoding): Every finite set of finite relations can be encoded in a finite binary relation while obeying the homomorphism requirement. Moreover, as long as the output of the encoding is produced using a series of commands that always contribute at least one tuple, that series of commands is homomorphic.

Proof: We are going to use the following notations:

- A: set of arities of the relations. //BASE 1
- $N(a)$: number of relations with arity a that we are considering. //BASE 0
- R^a_n: nth relation of arity a
- $R^a_n[r]$: rth record of relation R^a_n
- $R^a_n[r][i]$: ith element of the record $R^a_n[r]$
- $Add(x, y)$ = adds to our final binary relation the tuple $\langle x, y\rangle$.

The information contained in a relation is given by two elements: the arity of the relation and the elements of the records. We are going to store the order of the elements using chains of skolems, and we are going to store the arity of the relation using the cardinality of particular subsets of the final relation.

For example, suppose that we wanted to encode three unary relations and one binary relation:

$$R(x), Q(x), Z(w), R2(x, y), R2(x, z)$$

What we store is the following. We use the prefix s to indicate a skolem—a constant not appearing in the relations we are encoding.

$$\text{1-ary: } \langle x, s1\rangle\langle x, s2\rangle\langle x, s3\rangle\langle w, s4\rangle\langle w, s5\rangle\langle w, s6\rangle\langle w, s7\rangle$$
$$\text{2-ary: } \langle x, s8\rangle\langle y, s9\rangle\langle x, s10\rangle\langle z, s11\rangle\langle s8, s9\rangle\langle s10, s11\rangle$$

Note that its important to store the information using cardinalities that are powers of 2 in order to remove ambiguity between, in this case, R and Q. To avoid ambiguity within the same arity, we want to store cardinalities that obey the following property (we refer to the number of elements into which a record of the relation is mapped):

$$R_n^a > \sum_{0 \le 0 \le n-1} R_i^a$$

We use powers of 2 like this:

$$R_n^a = 2^n * (2a - 1)$$
$$2^n * (2a - 1) > \sum_{0 \le i \le n-1} (2^i * (2a - 1))$$
$$2^n > \sum_{0 \le i \le n-1} 2^i$$
$$2^n > (2^n - 1)$$

Note that its impossible to have ambiguity over different arities as the linking structure conveys that information. Lets see how the actual encoding happens. In this section, the a : A construct is a java-like for-all construct, while the b: //B// represent the numerals from the base of B to its value.

```
For a: A
  for n : N(a)
    let numBlocks = 2^n
    for r: R_n^a
      for b: //numBlocks//
        let newconsts = n—new—consts(a)
        for c: //newconsts −1//
          Add(newConsts[c], newconsts[c+1])
          Add(r[c], newconsts[c])
        add(r[last], newconsts[last])
```

The decode, on the other hand, happens rather easily. We assume REL is the relation containing the encoding given above.

$R_n^a(X_1, X_2, , X_a) \iff$
there exist 2^n disjoint sets of skolems s such that
For-each s such that there exists a bijective function enum: $\{1, 2, ..., a\} \rightarrow s$ such that
For-each i such that $1 \le i <$ //s//
$REL(enum(i), enum(i + 1)) \wedge REL(X_i, enum(i)) \wedge REL(X_a, enum(a))$

The above pseudo-code assumes we can determine which values are constants and which are actual data. Skolems are all those values that appear in the right-hand side of some tuple. Actual data are all those values appearing on the left side of some tuple and are not skolems. □

3) BLP: It turns out that neither BLPa nor BLPb admit correct implementations of the coalition workload, whereas BLPc admits a correct, homomorphic, safe implementation. The lattice-model of BLPa cannot represent any arbitrary access control policy; hence, it is simply too inexpressive to admit even a correct, AC-preserving implementation. Similarly BLPb, which is outfitted with an access matrix in addition to the lattice, can only represent access control policies with four rights, and so again admits no correct, AC-preserving implementation. For BLPc, the key insight to the proof for the implementation is that we can arrange it so the model is effectively the access matrix but with additional data structures that can be used to store the $orgUser$ relation.

Theorem 12: Neither BLPa nor BLPb admit correct, AC-preserving implementations of the coalition workload.

Proof: In both BLPa and BLPb there are only 4 rights (read, append, write, and execute). Any authorization policy that has more than four rights therefore cannot be expressed correctly so that for every state the coalition query $auth(x, y, z)$ is equivalent to the BLPa/BLPb query $auth(x, y, z)$. □

Theorem 13: BLPc admits a correct, AC-preserving, homomorphic, safe implementation of the coalition workload.

Proof: Here we describe the implementation, which is simple. We use m to represent $auth$, as required by AC-preservation. Doing so means ensuring S, O, and R contain a superset of all subjects, objects, and rights that appear in m. We represent each $orgUser(org, u)$, as $clear(u) = \langle u, \{org\}\rangle$. To do this we ensure both C and P are the set of all strings, and $<$ is any ordering on C. Furthermore, we let b be empty; we let $clear_c = clear$ so that $clear_c(x) \sqsubseteq clear(x)$ holds as required. For $class$, we assign $class(o) = \langle o, \{o\}\rangle$. This gives the query- and state- mapping.

When *joinCoalition* is executed, for each added $auth(a, b, c)$ we add $S(a)$, $O(b)$, $R(c)$, and $m(a, b, c)$ to the state with *addS(a)*, *addO(b)*, *addR(c)*, *addM(a,b,c)*. For each added $orgUser(d, e)$ we add $S(e)$ and $clear(e) = clear_c(e) = \langle e, \{d\}$ to the state with *addS(e)*, *setClear(e,e,*$\{d\}$*)*, and *setClearc(e,e,*$\{d\}$*)*. When *leaveCoalition* is executed with orgID o, we find the set of all subjects e such that $clear(e) = \langle e, \{o\}\rangle$, and for each such e delete all m atoms for e, remove e from S, and set $clear(e)$ to $\perp$.

This implementation is correct by inspection, and it is homomorphic since it can easily be expressed in HPL (it is deterministic, uses no string manipulation, and includes no string constants). It is also safe since no unnecessary changes are made to $auth$.

Theorem 14: BLPa, BLPb, BLPc fail to admit a correct, strongly compatible implementation of the coalition workload.

Proof: This is a simple variation on the proof that the AM/RBAC systems fail to admit a correct, strongly compatible implementation (Theorem 7). The only difference is that the system commands we use are the RBAC commands, instead of the AM commands. Consider any sequence of *joinCoalition* commands that yields a workload state where at least $orgUser$ tuple is true. Suppose the command-mapping for this sequence yields the BLP system state s. It is easy to see that there is also a sequence of BLP commands that yields s both in the workload state space and, because the implementation is *strongly* compatible, in the system state space. Since these two sequences of workload commands end in query-distinct workload states but in the same system state, the implementation must not be correct as it violates the requirement that the command-mapping must preserve the query-mapping. This proof applies to all of our BLP systems because all of the fields in the BLP models are queryable. □

C. Hospital Workload

In the hospital workload, we consider some of the normal operations that the typical workers at a hospital carry out. Clerical staff admit patients to wards (*e.g.*, the Oncology ward or the Emergency ward) and assign each patient a primary doctor. A patient's primary doctor can examine the patient's chart of treatments and modifies that chart by prescribing new treatments and modifying old treatments. Once a patient's treatments have finished, the patient's primary doctor discharges her.

Formally, the workload state is comprised of the following fields.

- D: the set of doctors
- P: the set of patients
- C: the set of clerical staff
- W: the set of wards
- T: the set of treatments
- $belongs(s) = w$ indicates the patient, doctor, or administrator s belongs to ward w.
- $primary(p) = d$ indicates that the primary doctor of patient p is d.
- $chart(p, t)$ means that patient p has been assigned treatment t.
- R_{pri} is the set of the rights only the primary doctor has.
- R_{med} is the set of the rights for all doctors in the Ward.

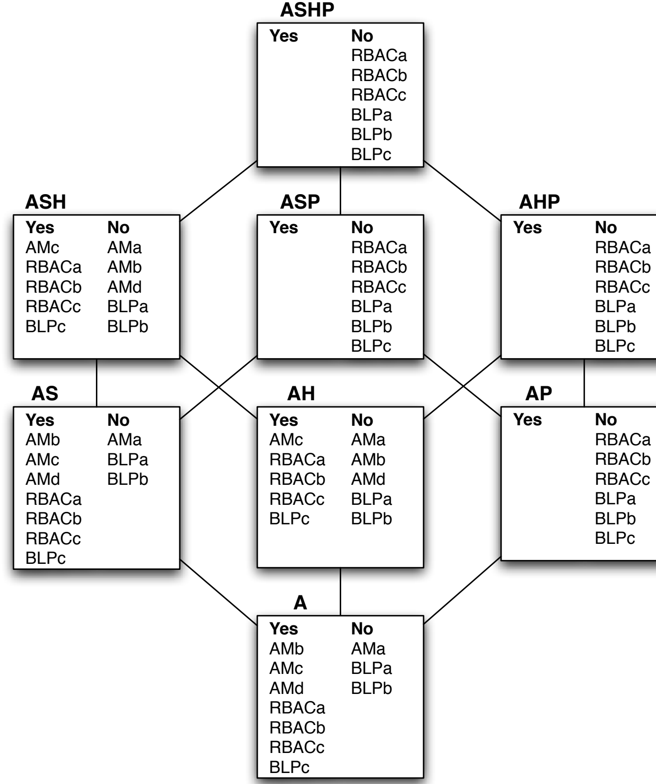

Fig. 2. Summary of the hospital workload results. AMx is an access matrix system; RBACx is a role-based access control system; BLPx is a Bell-LaPadula system. A stands for AC-preservation, H for homomorphism, C for weak compatibility, S for safety, P for administration-preservation. Correctness is required throughout.

- R_{cler} is the set of the rights for clerical staff.

To change the state, the workload has the following commands, none of which are administrative. In addition, the workload has administrative commands for changing D, C, W, R_{pri}, R_{med}, R_{cler}.

- *modifyMedicalData(issuer,p)*: edits the medical data of patient p. Belongs to R_{pri}.
- *viewMedicalData(issuer,p)*: view the medical data of patient p. Belongs to R_{med}.
- *admitPatient(issuer,p,d)*: admit patient p with primary doctor d (add p to P, $primary(p)$:= d and $belongs(p)$:= $belongs(d)$). Belongs to R_{cler}.
- *dischargePatient(issuer,p)*: delete patient's occupancy records (set $belongs(p)$:=$\perp$ and $primary(p)$:=$\perp$, remove p from P). Belongs to R_{pri}.

The access control policy of this workload dictates who is allowed to perform each of the commands listed above. Authorization policy.

$$auth(s,o,r) \iff \bigvee \begin{pmatrix} r \in R_{pri} \land s \in D \land o \in P \land primary(o) = s \\ r \in R_{med} \land s \in D \land o \in P \land belongs(s) = belongs(o) \\ r \in R_{cler} \land s \in A \land o \in O \land belongs(s) = belongs(o) \end{pmatrix}$$

We analyzed this workload using the same candidate access control systems as for the coalition workload. Below we describe results for just the BLP systems, but Figure 2 details the full results.

1) Access Matrix: We begin with the strongest positive result.

Proposition 7: AMc admits a correct, AC-preserving, homomorphic, safe implementation of the hospital workload.

Proof: First of all, we must define how we are going to encode the various relations and sets within the matrix. Note that in AMc, the records of the matrix affect the auth policy only if the first element of the tuple is a subject, the second element is an object and the third element is a right. We can therefore use additional skolems that are not belonging to any of the S, O, R sets as "indexes" (in the Database sense) to distinguish the records that specify auth from those that specify the extra info. First we show how to encode the basic sets using m.

$$
\begin{array}{rcl}
R_{pri}(r) & \iff & \exists S_a, S_b.m(r, S_a, S_a) \wedge \nexists S_b \neq S_a \wedge m(r, S_b, S_b) \wedge S_a, S_b \notin S \cup O \cup R \\
R_{med}(r) & \iff & \exists S_a, S_b.m(S_a, r, S_a) \wedge \nexists S_b \neq S_a \wedge m(S_b, r, S_b) \wedge S_a, S_b \notin S \cup O \cup R \\
R_{adm}(r) & \iff & \exists S_a, S_b.m(S_a, S_a, r) \wedge \nexists S_b \neq S_a \wedge m(S_b, S_b, r) \wedge S_a, S_b \notin S \cup O \cup R \\
P(u) & \iff & \exists S_a, S_b.m(u, S_a, S_a) \wedge m(u, S_b, S_b) \wedge S_a \neq S_b \wedge \nexists S_c \wedge S_c \neq S_a \wedge S_c \neq S_b \wedge m(u, S_c, S_c) \wedge S_a, S_b, S_c \notin S \cup O \cup R \\
D(u) & \iff & \exists S_a, S_b.m(S_a, u, S_a) \wedge m(S_b, u, S_b) \wedge S_a \neq S_b \wedge \nexists S_c \wedge S_c \neq S_a \wedge S_c \neq S_b \wedge m(S_c, u, S_c) \wedge S_a, S_b, S_c \notin S \cup O \cup R \\
A(u) & \iff & \exists S_a, S_b.m(S_a, S_a, u) \wedge m(S_b, S_b, u) \wedge S_a \neq S_b \wedge \nexists S_c \wedge S_c \neq S_a \wedge S_c \neq S_b \wedge m(S_c, S_c, u) \wedge S_a, S_b, S_c \notin S \cup O \cup R \\
W(u) & \iff & \exists S_a, S_b.m(u, S_a, S_a) \wedge m(u, S_b, S_b) \wedge m(u, S_c, S_c) \wedge S_a \neq S_b \wedge S_c \neq S_a \wedge S_c \neq S_b \wedge S_a, S_b, S_c \notin S \cup O \cup R \\
T(u) & \iff & \exists S_a, S_b.m(S_a, u, S_a) \wedge m(S_b, u, S_b) \wedge m(S_c, u, S_c) \wedge S_a \neq S_b \wedge S_c \neq S_a \wedge S_c \neq S_b \wedge S_a, S_b, S_c \notin S \cup O \cup R
\end{array}
$$

Next we give the non-unary relations encoding within m.

$$
\begin{array}{rcl}
primary(p) = d & \iff & \exists S_a.m(d, S_a, S_a) \wedge m(p, S_a, S_a) \wedge D(d) \wedge S_a \notin S \cup O \cup R \\
belongs(s) = w & \iff & \exists S_a.m(s, S_a, S_a) \wedge m(w, S_a, S_a) \wedge W(w) \wedge S_a \notin S \cup O \cup R \\
chart(p, t) & \iff & \exists S_a.m(p, S_a, S_a) \wedge m(t, S_a, S_a) \wedge T(t) \wedge S_a \notin S \cup O \cup R
\end{array}
$$

□

We can now look at how we can specify the command mapping for this encoding using HPL. The decoding of the state is trivial given the specification above.

```
admitPatient(M, p, d, Univ) =
  // save and then remove the representations of primary and belongs, as well as the other information
  Let (primary, belongs, Rp, Ra, Rm, P, D, A, W, auth) = decodeState(M)
  for each (m(x,y,z) ∈ M | ¬s(x) or ¬o(y) or ¬r(z))
    output(delM(x,y,z))
  // lookup d's ward
  Let ward = x | belongs(d,x) ∈ belongs
  // add primary/belongs for p,d
  P = P(p) ∪ {P}
  primary = primary ∪ {primary(p,d)}
  belongs = primary ∪ {belongs(p,ward)}
  // compute auth tuples that must be added (none must be deleted)
  Let addauth = computeNewAuth(primary,belongs,Rp, Rm, Ra, P, D, A, W, auth)
  // add legit m  tuples (while ensuring proper S,O,R)
  addBasic = { addS(s), addO(o), addR(r), addM(s,o,r) | <s,o,r> ∈ addAuth }
  outputSet(addBasic)
  // encode informative values
  Let pseudom = encodeState(primary, belongs, Rp, Rm, Ra, P, D, A, W, auth ∪ addauth,Univ)
  for each (m(s,o,r) ∈ pseudom)
    output(addM(s,o,r))

dischargePatient(M, p, Univ) =
  // save and then remove the representations of primary, belongs, and other information
  Let (primary, belongs, Rp, Ra, Rm, P, D, A, W, auth) = decodeState(M)
  // See which matrix records we must remove. Primary, Belongs and P of the patient.
  // Belongs and Primary are conveniently groupable in a single query, as we only remove the part
  //that belongs to the patient
  Let toremove = {m(p, Sk1, Sk2), m(p, Sk3, Sk3) | Sk1 ≠ Sk2 ∧ Sk1, Sk2, Sk3 ∉ S ∪ O ∪ R}
  // finally, we delete
  output(removeM(toremove))

encodeState(primary, belongs, Rp, Rm, Ra, P, D, A, W, auth,Univ)
  Let (SkPri1, SkPri2, SkBel1, SkBel2, SkPri, SkMed, SkAdm, SkP1,
            SkP2, SkD1, SkD2, SkA1, SkA2, SkW1, SkW2, SkW3) = nfreshConsts(16,Consts(auth),Univ)
  matrixPri = {m(d, SkPri1, SkPri2), m(p, SkPri1, SkPri2) | primary(p)=d}
  matrixBel = {m(s, SkBel1, SkBel2), m(w, SkBel2, SkBel1) | belongs(s)=w}
  matrixRp = {m(r, SkPri, SkPri) | Rp(r)}
  matrixRm = {m(SkMed, r, SkMed) | Rm(r)}
  matrixRa = {m(SkAdm, SkAdm, r) | Ra(r)}
```

```
matrixP = {m(u, SkP1, SkP1), m(u, SkP2, SkP2) | P(u)}
matrixD = {m(SkD1, u, SkD1), m(SkD2, u, SkD2) | D(u)}
matrixA = {m(SkA1, SkA1, u), m(SkA2, SkA2, u) | A(u)}
matrixW = {m(u, SkW1, SkW1), m(u, SkW2, SkW2), m(u, SkW3, SkW3) | W(u)}
return {matrixPri ∪ matrixBel ∪ matrixRp ∪ matrixRm ∪ matrixRa ∪
        matrixP ∪ matrixD ∪ matrixA ∪ matrixW}
```

Proposition 8: AMa fails to admit a correct, AC-preserving implementation of the hospital workload. AMb and AMd admit correct, AC-preserving implementations.

Proof: (AMa) Since AMa only has the matrix m to store information, and m must be identical to the hospital workload's authorization policy, there is no way to store the additional state in the hospital workload.

(AMb) AMb has additional storage besides the matrix m: any subject in S not appearing in m does not affect the authorization policy. Hence, we can use such a subject to store the additional information of the hospital workload while achieving correctness and AC-preservation.

(AMd) By reduction from AMb. □

It is noteworthy that while AMa does not allow us to store all the information required by the hospital workload, we can extract information about the wards structure simply by inspecting the access matrix (assuming each ward has at least one member). We can recover some of the ward structure as follows: if a Patient p is followed by Doctor d (i.e. d has primary rights over p), then it means that p and d belongs to the same ward. The same follows for the admin staff.

Proposition 9: AMb and AMd fail to admit a correct, AC-preserving, homomorphic implementation of the hospital workload.

Proof (sketch): Both AMb and AMd have only a collection of 3 unary sets worth of storage space outside of the access matrix. Since the hospital workload requires storing non-unary data, neither AMb nor AMd can store that data under the homomorphism guarantee. □

Proof: The workload is composed of 2 binary functions (belongs and primary), three unary relations that represent a tri-partition of the rights set ($R_{pri}, R_{med}, R_{adm}$) and 4 sets D, P, A, W. Therefore the number of possible states of the workload is given by (we drop the || notation for ease of reading)

$$|Workload| = D^P \times W^O \times R^3$$

while AMb only has the M matrix, which is at most

$$|AMb| = 2^{S\times O\times R=(D+A)\times(D+A+P)\times R} = 2^{(D^2+DP+2DA+AP+A^2)R}$$ □

Note that technically the S, O, R spaces of AMb could be larger than the S, O, R spaces of the Workload, but since the auth policy is affected directly by the matrix, and since we are assuming AC-preservation, those records of M where s, o or r are not present in the Workload S, O, R sets MUST be false, and therefore, being constant, do not affect the state space size of AMb. Albeit AMb's space grows exponentially while the one of the Workload grows polynomially, we see that if the number of Wards tends to infinity, the size of AMb stays constant, while the size of the Workload space grows. This means that for scenarios with a huge number of wards, AMb has less states than the workload and therefore cannot, under the AC-preservation and homomorphic assumptions, represent the workload. Note that this implies that in these scenarios most of the wards must be empty, because adding doctors, administrative or patients would make the size of AMb skyrocket.

2) Role-Based Access Control:

Proposition 10: None of RBACa, RBACb, or RBACc admit administrator-preserving implementations of the hospital workload.

Proof: The hospital workload commands include non-administrative commands, but all of the RBAC commands are administrative. Thus no implementation of the hospital workload in any of the RBAC systems can be administrator-preserving. □

Proposition 11: RBACa, RBACb and RBACc all admit a correct, AC-preserving, safe, homomorphic implementation of the hospital workload.

Proof: In RBACa, the representation of all the sets and relations can be achieved via the introduction of appropriate roles. More specifically, we are going to use skolem constants and cardinality artifacts, similar to those used in AM, to store the extra information. We are going to use UR records that do not have a correspondent in PA, and PA records that do not have a correspondent UR. In this way, we do not affect the auth. We will need first to compute the P, D, A, W sets, because we'll

be using them in the belongs and primary definition. We will use S_b as a skolem different from S_a. Below we give the query mapping.

$$\begin{aligned}
P(u) &\iff PA(S_a, u, S_a) \wedge \neg PA(S_a, S_a, u) \wedge \neg PA(S_a, S_b, u) \wedge \neg UR(x, S_a) \\
D(u) &\iff PA(S_a, u, S_a) \wedge PA(S_a, S_a, u) \wedge \neg PA(S_a, S_b, u) \wedge \neg UR(x, S_a) \\
A(u) &\iff PA(S_a, u, S_a) \wedge \wedge PA(S_a, S_b, u) \neg PA(S_a, S_a, u) \wedge \neg UR(x, S_a) \\
W(u) &\iff PA(S_a, u, S_a) \wedge PA(S_a, S_a, u) \wedge PA(S_a, S_b, u) \wedge \neg UR(x, S_a) \\
T(u) &\iff PA(S_a, S_b, u) \wedge PA(S_a, u, S_b) \wedge \neg PA(S_a, S_a, u) \wedge \neg UR(x, S_a) \\
R_{pri}(u) &\iff \neg PA(S_a, u, S_a) \wedge PA(S_a, u, S_b) \wedge \neg PA(S_a, S_a, u) \wedge \neg PA(S_a, S_b, u) \wedge \neg UR(x, S_a) \\
R_{med}(u) &\iff \neg PA(S_a, u, S_a) \wedge \neg PA(S_a, u, S_b) \wedge PA(S_a, S_a, u) \wedge \neg PA(S_a, S_b, u) \wedge \neg UR(x, S_a) \\
R_{adm}(u) &\iff \neg PA(S_a, u, S_a) \wedge \neg PA(S_a, u, S_b) \wedge \neg PA(S_a, S_a, u) \wedge PA(S_a, S_b, u) \wedge \neg UR(x, S_a)
\end{aligned}$$

Similarly, we are going to use UR to store the primary and belong information (S(u) is true iff u belongs to S):

$$\begin{aligned}
primary(p) = d &\iff \exists S_a.UR(p, S_a) \wedge UR(d, S_a) \wedge P(p) \wedge D(d) \wedge \forall xy.\neg PA(S_a, x, y) \\
belongs(u) = w &\iff \exists S_a.UR(u, S_a) \wedge UR(w, S_a) \wedge S(u) \wedge W(w) \wedge \forall xy.\neg PA(S_a, x, y) \\
chart(p, t) &\iff \exists S_a.UR(p, S_a) \wedge UR(t, S_a) \wedge P(p) \wedge T(t) \wedge \forall xy.\neg PA(S_a, x, y)
\end{aligned}$$

In this way, we are sure to be able to retrieve the information even if the various skolem indexes change name under homomorphic situations.
RBACb and RBACc are at least as expressive as RBACa, and can therefore store the same degree of information (possibly even easier).

Let's now look at how we can write down the command mapping in HPL; the state-mapping is implicit in the following code and in the queries above.

```
admitPatient(UR, PA, p, d, Univ) =
  // save and then remove the representations of primary, belongs, and others
  Let (primary, belongs, Rp, Ra, Rm, P, D, A, W, auth) = decodeState(UR, PA)
  for each (UR(x,y) ∈ PA | ¬ PA(y,w,z))
    output(delUR(x,y))
  for each (PA(y,w,z) ∈ PA | ¬ UR(x,y))
    output(delPA(x,y,z))
  Let ward = x | belongs(d,x) ∈ belongs
  // add primary/belongs for p,d
  P = P(p) ∪ {P}
  primary = primary ∪ {primary(p,d)}
  belongs = primary ∪ {belongs(p,ward)}
  // compute auth tuples that must be added (none must be deleted)
  Let addauth = computeNewAuth(primary, belongs, Rp, Rm, Ra, P, D, A, W, auth)
  // add legit m  tuples (while ensuring proper S,O,R)
  addBasic = { addPA(x,y), addUR(y,w,z) | <x,y,w,z> ∈ addAuth }
  outputSet(addBasic)
  // encode informative values
  Let pseudoUR, pseudoPA = encodeState(primary, belongs, Rp, Rm, Ra, P, D, A, W, auth ∪ addauth,Univ)
  for each (UR(x,y) ∈ pseudoUR)
    output(addUR(x,y))
  for each (PA(y,w,z) ∈ pseudoPA)
    output(addPA(y,w,z))

dischargePatient(UR, PA, p, Univ) =
  //Since the encoding of P is also part of UR and PA, it's easier to
  // save and then remove the representations of primary, belongs, and others
  Let (primary, belongs, Rp, Ra, Rm, P, D, A, W, auth) = decodeState(UR, PA)
  //See which records we must remove. Primary, Belongs and P of the patient.
  Let toRemoveP = {PA(Sk1, p, Sk1) | ¬ UR(x, Sk1) ∧ ¬ PA(Sk1, Sk1, p) ∧
                  ¬ PA(Sk1, Sk2, p) ∧ Sk1 ≠ Sk2}
  Let toRemovePri = {UR(p,Sk1), UR(d, Sk1) | ¬ PA(Sk1, x, y) ∧ P(p) ∧ D(d)}
  Let toRemoveBel = {UR(p,Sk1), UR(w, Sk1) | ¬ PA(Sk1, x, y) ∧ P(p) ∧ W(w)}
```

```
  //finally, we delete
  output(removePA(toremoveP))
  output(removeUR(toremovePri))
  output(removeUR(toremoveBel))

encodeState(primary, belongs, Rp, Rm, Ra, P, D, A, W, auth,Univ)
  Let (SkPri, SkBel, SkPri1, SkPri2, SkMed1, SkAdm1, SkAdm2, SkP1, SkD1,
      SkA1, SkA2, SkW1, SkW2) = nfreshConsts(13,Consts(auth),Univ)
  PA-P = {PA(SkP1, p, SkP1) | P(p)}
  PA-D = {PA(SkD1, d, SkD1), PA{SkP1, SkP1, d} | D(d)}
  PA-A = {PA(SkA1, a, SkA1), PA(SKA1, SkA2, a) | A(a)}
  PA-W = {PA(SkW1, w, SkW1), PA(SkW1, SkW1, w), PA(SkW1, SkW2, w) | W(w)}
  PA-R_{Pri}= {PA(SkPri1, r, SkPri2)) | Rp(r)}
  PA-R_{Med} = {PA(SkMed1, SkMed1, r) | Rm(r)}
  PA-R_{Adm} = {PA(SkAdm1, SkAdm2,r) | Ra(r)}
  UR-Pri = {UR(p, SkPri), UR(d, SkPri) | primary(p)=d}
  UR-Bel = {UR(u, SkBel), UR(w, Bel) | belongs(u)=w}
  //Return the two parts of the encoding
  return {PA-P ∪ PA-D ∪ PA-A ∪ PA-W ∪ PA-R_{Pri} ∪ PA-R_{Med} ∪ PA-R_{Adm} , UR-Pri ∪ UR-Bel}
```

The implementation is homomorphic because it can be expressed in HPL (*i.e.*, uses no string literals or string manipulation). It is AC-preserving by construction. It can be made safe because even if an existing skolem becomes part of the legitimate authorization policy, it is a simple matter to rename just that skolem before carrying out the usual decoding/manipulation/decoding of the remainder of the state. □

3) Bell-Lapadula:

Proposition 12: No implementation of the hospital workload using BLPa is administrator-preserving.

Proof: All of the hospital workload commands are non-administrative, but all of the BLPa commands are administrative; thus, every implementation in BLPa must map a non-administrative workload command to a command sequence with an administrative command. □

Proposition 13: BLPa does not admit an implementation that is both correct and AC-preserving.

Proof: Suppose we have two patients, p_1 and p_2, and two doctors, d_1 and d_2, such that $primary(p_1) = d_1$ and $primary(p_2) = d_2$. What we want to achieve is that d_1 has read and write rights over p_1 and d_2 has read and write rights over p_2, but also that both d_1 and d_2 have read access over both p_1 and p_2. According to the axioms of BLP's lattice, we have that":

- Since d_1 has read AND write rights over p_1, this implies Clearance(d_1)=Class(p_1)
- Since d_2 has read access over p_1, this implies that Class(p_1)$\sqsubseteq$ Clearance(d_2)
- Since d_2 has read AND write rights over p_2, this implies Clearance(d_2)=Class(p_2)
- Since d_1 has read access over p_2, this implies that Class(p_2)$\sqsubseteq$ Clearance(d_1)
- Hence we have: $Clearance(d_1) = Class(p_1) \sqsubseteq Clearance(d_2) = Class(p_2) \sqsubseteq Clearance(d_1)$
- The only possible way for this to hold is to have: $Clearance(d_1) = Class(p_1) = Clearance(d_2) = Class(p_2)$

But this would violate the fact that the authorizations of the two doctors are different, as they cannot both write on both the patients, therefore there cannot be a correct implementation.
The only way to obtain an usable result, would be to separate the medical data of the patient into two distinct objects, one for reading and one for writing: in this way, we would break the loop over the lattice. Such solution is not uncommon: we can easily imagine an interface to the data that allows actual manipulation (the writable object) and a "print" command that creates a readable report for the other Doctors.
But doing so would mean mapping the single Patient object of the workload to two distinct objects in the ACS, a solution that directly conflicts with the AC-preservation of the requirement. □

Proposition 14: BLPb does not admit a correct, AC-preserving implementation of the Hospital workload.

Proof: Differently from BLPa, BLPb also has a matrix m for storing information. Therefore it can withstand the logical attack that crushed BLPa, as we get to a point where we do have $Clearance(d_1) = Class(p_1) = Clearance(d_2) = Class(p_2)$ indeed, but we only get $m(d_1, p_1, write)$ and $m(d_2, p_2, write)$, but not $m(d_1, p_2, write)$ or $m(d_2, p_1, write)$, and therefore the auth policy for the two Doctors is different. We can represent the various sets as labels of the clearances. $primary(p) = d \iff m(d, p, write) \wedge Clearance(d) = Class(p)$
$belongs(s) = w \iff "ward_w" \sqsubseteq Class(s)$

Nevertheless, BLPb only has a reduced set of possible rights, that do not map directly to the custom rights of the workload, and therefore cannot be used to implement it in an AC-preserving way. □

Proposition 15: BLPc admits a correct, AC-preserving homomorphic, safe implementation of the Hospital workload.

Proof: We must give the state-mapping, query-mapping, and command-mapping. We start with the state-mapping. We use the $class$ relation of BLPc to represent the different sets of the hospital workload: D, P, C, W, T, R_{pri}, R_{med}, and R_{cler}. To do this, we create a different clearance level in the BLPc state for each of the set names (which we denote with the name of the set itself), thereby defining BLPc's C, and set $<$ to give $D < P < C < W < T < R_{pri} < R_{med} < R_{cler}$. Then we assign $class(d) = \langle D, \{\}$ for every $d \in D$; likewise for the other sets. To ensure this encoding is homomorphic, the query mapping is defined so that a doctor D is anyone whose classification is in the first position of $<$; likewise, patients are all those people where their classification is in the second position of $<$; and so on.

To represent the remaining elements of the hospital workload state ($belongs$, $primary$, and $chart$), we can treat $belongs$ and $primary$ as relations and union them together with $chart$ and store the result in the $clear$ relation. So for patients $belongs(p) = w$ and $primary(p) = d$ $chart(p, t_1)$ and $chart(p, t_2)$ is represented as $clear(p) = \langle c, \{w, d, t_1, t_2\}$. Similarly we can encode the $belongs$ relation for doctors and clerical workers.

Now consider the command mapping. Each time the $auth$ policy changes, compute which subject-object-right tuples must be added and which must be deleted in the workload; in BLPc, add and delete exactly those entries from m, modifying S, O, and R as appropriate to ensure m has the right type. Similarly, when the state elements of the hospital workload are modified, extract the workload state from the BLPc state and modify the BLPc state as required.

The implementation is homomorphic since it can be encoded without string manipulation or string literals. It is AC-preserving and safe by construction.

Proposition 16: BLPc admits no correct, AC-preserving, administration-preserving implementation of the hospital workload.

Proof: In the hospital workload, the authorization policy treats patients as objects. The non-administrative workload command $admitPatient$ can enlarge the set of patients and assign rights pertaining to those patients in the authorization policy. For there to be an administration-preserving implementation, the implementation of $admitPatient$ would need to be executed by non-administrative BLPc commands; however, that is impossible under AC-preservation since when $admitPatient$ adds rights to the access control policy it must also add the patient to the BLPc set O, which can only be accomplished using an administrative command. □

VII. Related Work and Future Work

Evaluation Frameworks. We know of three fundamentally different, prior frameworks for evaluating access control systems [4], [5], [9]. None address application-sensitive evaluation directly, but there are close connections nevertheless.

Chander et al. [4] analyzes several access control system features (capability passing, trust management delegation, and access control lists) by constructing simple systems that exhibit those features and investigating the existence of one-to-one and one-to-many command-mappings between them. What they fail to discuss is why a one-to-one simulation might be preferred to a many-to-one simulation. In ACEF, a one-to-one simulation between access control systems ensures that every workload implementable with the safety guarantee in one is also safely implementable in the second, whereas the same cannot be said for the one-to-many simulation.

Tripunitara and Li [5] aim to compare access control systems in terms of their ability to simulate one another while preserving a particularly strong security guarantee: (strong) security-preservation. Their framework was the inspiration for ours, though one technical difference is noteworthy. Their query-mappings of [5] require each workload query to be computed from exactly one candidate system query, whereas our query-mappings allow each workload query to be computed by any function over the candidate system queries. In our framework, the one-to-one query mapping of [5] could be defined as another security guarantee, and doing so yields far more negative results for the coalition workload. Our more general framework allows an analyst to decide which type of security mapping is more appropriate for each application.

Tripunitara and Li [5] also analyze a security guarantee that requires certain formulas in (infinitary) temporal logic be preserved across the implementation. They also provide a reduction between systems that, in our language, ensures that any workload implementation in one system that achieves (strong) security-preservation is also implementable in the other system with (strong) security-preservation. The security guarantees introduced in this paper are much simpler than (strong) security-preservation yet are important for ACEF for two reasons. First, the workload state machine may not have been designed to ensure that all the temporal properties preserved by (strong) security-preservation are actually important to the application. For example, the workload might include a swap operation, and if the candidate system does not include a one-step swap, the (strong) security-preservation guarantee may not be possible but that system might otherwise be a good candidate. Second, when a candidate system fails to admit an implementation with (strong) security-preservation, it is useful to have a variety

of weaker guarantees that allow the analyst to identify the root cause. If the underlying system simply fails to have a swap operation, that is a very different failure than if the underlying system admits no AC-preserving implementation. Thus the spectrum of guarantees provided by ACEF helps an analyst understand failures and their severity.

Bertino et al. [9] aim to compare systems by axiomatizing each in a variant of Datalog and then comparing the resulting logic programs. They assume that each system has components with particular semantics (*e.g.*, user, group, role, process) and compare systems assuming those components are used according to those semantics. Unlike our framework and the two frameworks discussed above, which require the analyst to formalize the candidate systems and compare those systems as two distinct steps, in this work the analyst performs both steps simultaneously by virtue of formalizing each system using the basic building blocks of the framework. This building-block approach has the drawback that formalizing the system presupposes that each component of a system will always be used only in the way its designers intended, and hence our analysis of that system may not reflect what happens in the real world. In contrast, in our framework, there are no restrictions about how a system's components are used and hence we can formalize abuses of that system as specific kinds of workload implementations, *e.g.*, the string-packing implementations.

Authorization logics. An authorization logic is an access control system where the state of the system is represented by a set of logical formulae, *e.g.*, [16]. Typically the primitive command for changing the state of one of these systems replaces the existing policy with a new one; thus, any state can be reached from any other state in exactly one step. Such access control systems are hard to distinguish from one another using our current list of security guarantees. In the future we plan to investigate security guarantees that leverage the common technique for comparing logical languages *e.g.*, [16]: check if each policy idiom expressible in one system is also expressible in the other (without having to modify the entire policy). Furthermore, with just our current set of guarantees logical systems are always superior to non-logical systems (because every workload command can be implemented with a single command). In the future we plan to investigate how to compare logical and non-logical systems in a meaningful way.

Modal Logic and Decidability. ACEF—our logic where workloads are axiom sets and access control systems are models—is semantically similar to modal logic (*e.g.*, temporal logic), where modal formulas are the axiom sets and Kripke structures are the models. The main difference is that in temporal logic, all axiom sets are finite or at least recursively enumerable, whereas in ACEF there are no such restrictions placed on workloads. We have intentionally avoided any analysis based on axiomatizability, computability, or complexity so as to focus on the semantic issues of application-sensitive evaluation. In the future we plan to incorporate such issues into ACEF, giving the analyst additional dimensions of control.

Bisimulations. Here we review several well-known versions of implementation/simulation/bisimulation (see [17, Ch. 1] for details) and point out which variants we and the other most closely related works [4] and [5] have used. In the future we plan to investigate the relationships between these variants of simulation to understand how they fit into our framework.

A *mapping* between two state transition systems $\langle W_1, R_1, \vdash_1\rangle$ and $\langle W_2, R_2, \vdash_2\rangle$ (where W_i is a set of states, R_i is an adjacency relation, and $\vdash_i$ dictates which queries are true in which states) is a relation on the two sets of states $h(x, y) \subseteq W_1 \times W_2$ such that $h(x, y)$ implies $x \vdash_1 q$ if and only if $y \vdash_2 q$ for all q.

A mapping is a *weak simulation* if for every $w, w' \in W_1$ such that w' is reachable from w and for every $v \in W_2$ where $h(w, v)$ holds there exists a v' reachable from v such that $h(w', v')$ holds. We use a weak simulation for reductions between access control systems, where for AC-preservation, for example, $\vdash$ is only concerned with $auth$ queries. [4] uses a variation on this definition where each system has labels on transitions. Their version of weak simulation requires (i) w' is reachable from w in one step, (ii) the sequence of labels from v to v' be computable from just the label between w and w', and (iii) if an $auth$ atom is true in w' then it is true in v' (but not the converse).

A mapping is a *simulation* if it is a *weak simulation* but where "reachable" is defined as "reachable in one step". We posit that such simulations in our system reductions could be used to preserve safety. [4] uses the term *strong simulation* to mean their version of weak simulation but where "reachable" means "reachable in one step".

A mapping is a *(weak) bi-simulation* if it is a (weak) simulation plus for every $v, v' \in W_2$ such that v' is reachable from v and for every $w \in W_1$ such that $h(w, v')$ holds there is a $w' \in W_1$ such that $h(w', v')$ holds. [5] uses weak bi-simulation but where queries belong to different namespaces.

VIII. CONCLUSION

To enable security analysts to determine which access control system is best-suited for a new or existing application, we developed a formal framework ACEF for application-sensitive access control evaluation—a way of comparing access control systems in terms of parameterized expressiveness. The analyst's main task is checking if an access control system can implement an application's workload in a way that meets a set of application-relevant security guarantees. An analyst can exhibit an implementation either by writing a computer program or by applying one of ACEF's theorems to an existing implementation in another system. An analyst unable to find such an implementation can attempt to prove that such an implementation does not exist using several of the proof techniques developed in this paper. We applied ACEF to perform two case studies: one based on the dynamic coalitions described in [10] and one on a hospital management system.

We envision researchers in access control and security analysts attempting to build real-world applications utilizing ACEF in different ways. Researchers will produce rigorous proofs within ACEF to gain deep understanding of the applications and access control systems they have chosen to investigate. They will be concerned with the precise definitions of systems, workloads, implementations, and security guarantees. In contrast, security analysts will take the main concepts of ACEF and use them to guide how they integrate access control into their applications. They will focus mainly on the idea that different implementations of the access control component of an application have qualitatively different security guarantees, and one must weigh the tradeoffs of those guarantees when choosing an implementation.

Currently, we have begun to explore ACEF within the proof assistant PVS, in particular formally proving the correctness of an implementation of the coalition workload within the access matrix. While the proofs can be complex, preliminary results are encouraging. In the future we plan to extend ACEF to enable proper evaluation of access control systems based on formal logic. Such systems differ from those addressed in our case studies (the access matrix, RBAC, and Bell-LaPadula) because each state can transition to any other state in a single step by changing the logical formulae in the state. To properly evaluate such systems, ACEF must include security guarantees that account for the expressiveness and modularity of logical policy languages.

Acknowledgments

Thanks to John Mitchell for early conversations about application-sensitive access control evaluation. This work was supported in part by the National Science Foundation under awards CCF-0916438, CNS-0964295, CNS-1131863, and CNS-1228697.

References

[1] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, no. 8, pp. 461–471, 1976.

[2] R. J. Lipton and L. Snyder, "A linear time algorithm for deciding subject security," *Journal of the ACM*, vol. 24, no. 3, pp. 455–464, 1977.

[3] P. Ammann, R. J. Lipton, and R. S. Sandhu, "The expressive power of multi-parent creation in monotonic access control models," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 149–166, 1996.

[4] A. Chander, J. C. Mitchell, and D. Dean, "A state-transition model of trust management and access control," in *CSFW*. IEEE Computer Society, 2001, pp. 27–43.

[5] M. V. Tripunitara and N. Li, "A theory for comparing the expressive power of access control models," *Journal of Computer Security*, vol. 15, no. 2, pp. 231–272, 2007.

[6] S. Osborne, R. Sandhu, and Q. Munawer, "Configuring role-based access control to enforce mandatory and discretionary access control policies," *ACM Transactions on Information and System Security*, vol. 3, no. 2, pp. 85–106, May 2000.

[7] R. Sandhu, "Expressive power of the schematic protection model," *Journal of Computer Security*, vol. 1, no. 1, pp. 59–98, 1992.

[8] R. Sandhu and S. Ganta, "On testing for absence of rights in access control models," in *Proceedings of the Sixth Computer Security Foundations Workshop (CSFW)*, Jun. 1993, pp. 109–118.

[9] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca, "A logical framework for reasoning about access control models," *ACM Transactions on Information and System Security*, vol. 6, no. 1, pp. 71–127, 2003.

[10] U.S. Air Force Scientific Advisory Board, "Networking to enable coalition operations," MITRE Corporation, Tech. Rep., 2004.

[11] "Horizontal integration: Broader access models for realizing information dominance," MITRE Corporation JASON Program Office, Tech. Rep. JSR-04-13, Dec. 2004.

[12] J. Crampton, "Understanding and developing role-based administrative models," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2005, pp. 158–167.

[13] Q. Munawer and R. S. Sandhu, "Simulation of the augmented typed access matrix model (ATAM) using roles," in *Proceedings of INFOSECU99 International Conference on Information and Security*, 1999.

[14] N. Li, J. C. Mitchell, and W. H. Winsborough, "Beyond proof-of-compliance: security analysis in trust management," *Journal of the ACM*, vol. 52, no. 3, pp. 474–514, 2005.

[15] N. Dershowitz and Y. Gurevich, "A natural axiomatization of computability and proof of Church's thesis," *Bulletin of Symbolic Logic*, vol. 14, no. 3, pp. 299–350, 2008.

[16] M. Y. Becker, C. Y. Fournet, and A. D. Gordon, "SecPAL: Design and semantics of a decentralized authorization language," *Journal of Computer Security*, 2009.

[17] P. Blackburn, J. v. Benthem, and F. Wolter, *Handbook of Modal Logic*. Elsevier Science, 2006.