Application-Sensitive Access Control Evaluation using Parameterized Expressiveness

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 V). Finally we describe related work (Section VI) and conclude (Section VII). Material omitted for lack of space can either be found in the Appendix or in the extended version [11].

II. Example and Informal Overview

The motivation for this work were the MITRE reports [10], [12] 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 redesign 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$_0$ [13]) 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 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)
- $next$: *States*($\mathcal{M}$) $\times \mathcal{L} \rightarrow$ *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

[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.

- *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 $s_0 \in \mathit{States}(\mathcal{A})$ and $\tau = l_1 \circ l_2 \circ \ldots$ is sequence where $l_i \in \mathit{Labels}(\mathcal{A})$ 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. ♦

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 $\langle \alpha, \sigma, \pi \rangle$

- σ : *States*($\mathcal{W}$) $\rightarrow$ *States*($\mathcal{Y}$) (state-mapping)
- α : *States*($\mathcal{Y}$) $\times$ *Labels*($\mathcal{W}$) $\rightarrow$ *Labels*($\mathcal{Y}$)* (command-mapping)
- π: for each $q \in$ *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

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

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 [14], 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 [15]) 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.

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 $\langle s_0, \ldots, s_n \rangle$.

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

♦

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("alice", "r"), UR("bob", "r"), PA("r","doc","write") }). 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.*, $assignUser(\text{"}alice\text{"}, \text{"}r\text{"})$). A formal definition of extensional systems can be found in Appendix A.

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 without forfeiting correctness.

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 Appendix A for such a programming language as well as a more complete definition for the homomorphic security guarantee.

Definition 8 (Homomorphisms): A constant substitution $v : \mathcal{U} \to \mathcal{U}$ is a bijection from strings to strings. The application of a substitution v to the common mathematical structures is the usual one, *e.g.*, function f is homomorphic if $f(\bar{x})[v] = f(\bar{x}[v])$. An implementation $\langle \alpha, \sigma, \pi \rangle$ is homomorphic if α, σ, and π are all homomorphic. ♦

In 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.

In our experience (see Section V), 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 data elements 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 data elements (strings) that are never substituted for, the homomorphic guarantee would handle SDCO properly while still eliminating string-packing implementations.

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 V), 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 9 (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.

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

[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.

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 10 (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:	$\langle\mathcal{L}_w \cup \mathcal{L}_y, next\rangle$ $next(\langle s_w, s_y\rangle, l) = \begin{array}{l}\langle next_w(s_w, l), s_y\rangle, \text{if } l \in \mathcal{L}_w \\ \langle s_w, next_y(s_y, l)\rangle, \text{otherwise}\end{array}$
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 11 (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 $s_w \in$ *States*$(\mathcal{W})$ and $s_y \in$ *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 $\pi'_q = \pi_q$. ♦

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.

G. Meta-theorems

Instead of introducing another security guarantee, in this section we focus on theorems that simplify an analyst's most time-consuming tasks: (i) exhibiting implementations with a given set of security guarantees and (ii) showing that no implementation exists. By showing that system $\mathcal{Y}_1$ can simulate $\mathcal{Y}_2$ in a precise way and applying the theorems in this section, the analyst can conclude that every workload implementable with some particular set of security guarantees $\mathcal{G}$ by $\mathcal{Y}_2$ can also be implemented with guarantees $\mathcal{G}$ by $\mathcal{Y}_1$. Then if the analyst shows an implementation in $\mathcal{Y}_2$, she knows there must also be an implementation in $\mathcal{Y}_1$, and if she shows there is no implementation in $\mathcal{Y}_1$, then there can be no implementation in $\mathcal{Y}_2$. That is, the theorems in this section help the analyst to compare systems in terms of parameterized expressiveness: the ability to implement workloads under security guarantees $\mathcal{G}$.

Definition 12 (Parameterized Expressiveness): Suppose every workload that can be correctly implemented by $\mathcal{Y}_1$ with security guarantees $\mathcal{G}$ can be correctly implemented by $\mathcal{Y}_2$ with security guarantees $\mathcal{G}$. Then we say that $\mathcal{Y}_1$ is no more expressive than $\mathcal{Y}_2$ with respect to $\mathcal{G}$, written $\mathcal{Y}_1 \leq^{\mathcal{G}} \mathcal{Y}_2$. ♦

Demonstrating $\mathcal{Y}_1 \leq^{\mathcal{G}} \mathcal{Y}_2$ directly can be difficult; hence, a *reduction* between two systems is a sufficient condition that ensures $\mathcal{Y}_1 \leq^{\mathcal{G}} \mathcal{Y}_2$ and is easier to demonstrate. The notion of reduction we introduce here is a simplification of the implementation definition given earlier. Instead of a state-mapping, a query-mapping, and a command-mapping, a reduction includes just a state-mapping and a query-mapping and requires the state-mapping preserve the query-mapping.

Definition 13 (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 *Th*$(s) = \pi($*Th*$(\sigma(s)))$. ♦

Different kinds of reductions yield different parameterized complexity results. Our first theorem says that a reduction where the state-mapping is one-to-one and preserves finite reachability ensures that each workload correctly implementable by one system can be correctly implemented by the other. Finite reachability says that if two states s_1 and s_2 are connected by finitely many steps in $\mathcal{Y}_1$ then $\sigma(s_1)$ and $\sigma(s_2)$ must be connected by finitely many steps in $\mathcal{Y}_2$.

Theorem 1 (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 then $\mathcal{Y}_1 \leq \mathcal{Y}_2$. Finite reachability requires that for all $s, s' \in$ *States*$(\mathcal{Y}_1)$, if s' is reachable in a finite number of commands from s, then $\sigma(s')$ is reachable in a finite number of commands from $\sigma(s)$.

Our second theorem uses a reduction that is a special case of the one in the previous theorem. The reduction for AC-preservation must be one-to-one, preserve finite reachability, and ensure that the query-mapping π answers each workload query $auth(r)$ with the access control system query $auth(r)$.

Theorem 2 (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 $\mathcal{Y}_1 \leq^A \mathcal{Y}_2$.

Our third theorem also refines the reduction defined in Theorem 1. Unlike AC-preservation, where we only needed to add the requirement that the query-mapping was AC-preserving to ensure $\leq^A$, adding the requirement that the reduction be homomorphic does not alone ensure $\leq^H$. To ensure $\leq^H$, we must also know that the systems themselves are homomorphic.

Theorem 3 (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
- the transition functions and entailment relations of $\mathcal{Y}_1$ and $\mathcal{Y}_2$ are homomorphic

For safety, we posit that a reduction for which there is a command mapping that is one-to-one suffices to preserve safety between systems. For compatibility, it is likely to suffice that each command of system $\mathcal{Y}_1$ is a special case of some command in $\mathcal{Y}_2$. For administration preservation, it should suffice that each non-administrative command in $\mathcal{Y}_1$ be mapped to a sequence of non-administrative commands n $\mathcal{Y}_2$. Proofs of some of the theorems above can be found in Appendix B, and further details can be found in the full version of the paper [11].

V. 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. Our candidate access control systems consisted of four variants of the access matrix, three variants of role-based access control, and three variants of Bell-LaPadula. We report results for each workload on each candidate access control system for a broad spectrum of the security guarantees introduced in this paper. After describing the candidate systems, we discuss the results for each of the workloads.

A. Candidate Access Control Systems

We analyzed four variants of the access matrix (AM) system, three variants of the basic role-based access control (RBAC) system, and three variants of the Bell-LaPadula (BLP) system. For lack of space, below we present only the four access matrix systems (AMa, AMb, AMc, and AMd) and one BLP system (BLPc). One of the RBAC systems (RBACa) serves as the running example in the main body of the paper. For each system, we investigated several different combinations of implementation properties. All of the systems we analyzed are extensional and hence the homomorphism guarantee is applicable.

The main difference between the four variants of the access matrix is (i) whether the subjects, objects, and rights types are stored outside the matrix as separate relations and (ii) when the matrix is restricted to the separately stored types, *e.g.*, when the matrix is stored or when the matrix is queried.

Definition 14 (Access Matrix): $\mathcal{U}$ is the set of all strings over some alphabet.

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

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

- AMb has fields $\langle m, S, O, R \rangle$
  - $S \subseteq \mathcal{U}$, the set of legitimate subjects
  - $O \subseteq \mathcal{U}$, the set of legitimate objects
  - $R \subseteq \mathcal{U}$, the set of legitimate rights
  - $m \subseteq S \times O \times R$

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

- AMc has fields $\langle m, S, O, R \rangle$
  - $S \subseteq \mathcal{U}$, the set of legitimate subjects
  - $O \subseteq \mathcal{U}$, the set of legitimate objects
  - $R \subseteq \mathcal{U}$, 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 has fields $\langle m, S, O, R \rangle$
  - $S \subseteq \mathcal{U}$, the set of legitimate subjects
  - $O \subseteq \mathcal{U}$, the set of legitimate objects
  - $R \subseteq \mathcal{U}$, the set of legitimate rights
  - $m \subseteq \mathcal{U} \times \mathcal{U} \times \mathcal{U}$

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

The commands for the AM models add and delete an element from each of the fields. Commands that modify S and R are the only administrative commands. ♦

The three variations of role-based access control (RBAC) we studied differ in the same way as the access matrix: (i) whether or not the subjects, objects, rights, and roles types are stored separately from the UR and PA relations and (ii) when the UR and PA relations are restricted to those separately stored types. RBACa, for example, does not store the types, whereas RBACb and RBACc do. RBACb applies the type restrictions at query-time, similar to AMc, whereas RBACc applies the type restrictions at the time the UR and PA relations are stored. The commands for the RBAC systems add or delete elements from each of the fields of the state; all commands are administrative commands.

The next three systems are based on the Bell-LaPadula (BLP) access control system. 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 15 (BLPc): 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
- $clear$: $S \rightarrow C \times 2^P$ maximal user clearances
- $class$: $O \rightarrow C \times 2^P$ document classifications
- $clear_c$: $S \rightarrow C \times 2^P$ current user clearances
- $m \subseteq S \times O \times R$ is a discretionary access matrix
- $b \subseteq m$ records the set of accesses employed currently

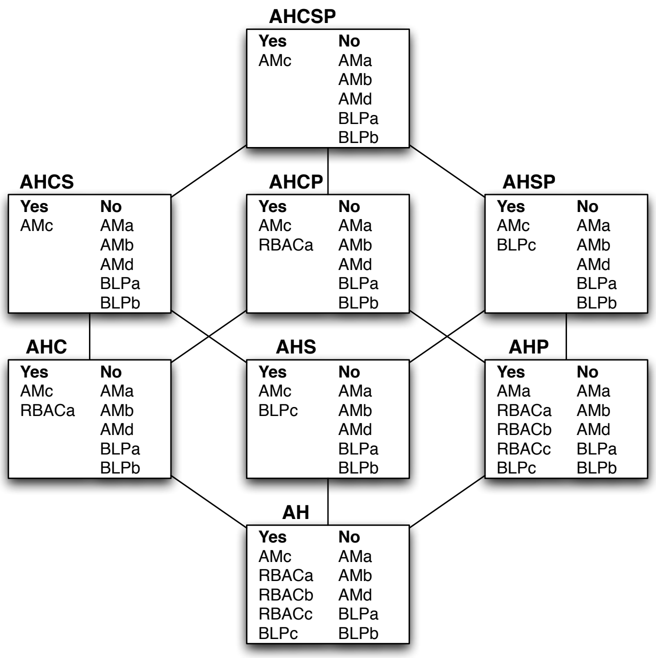

Fig. 1. Summary of the coalition workload results. AMx is an access matrix system; RBACx is a role-based access control system; BLPx is a Bell-LaPadula system. A stands for AC-preservation, H for homomorphism, C for compatibility, S for safety, P for administration-preservation. Correctness is required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

All fields of the model can be queried. The query $auth(s, o, r)$ is true exactly when $m(s, o, r)$ is true.

Each BLP variant has commands for changing each of its fields. The command that changes the clearance $clear_c$ of a user ensures that the current clearance is set no higher than the maximum clearance defined by $clear$. The commands that change C, $<$, P, $class$, $clear$, S, O, and R are the only administrative commands. ♦

B. Dynamic Coalition Workload Results

Recall that the coalition workload contains two commands: *joinCoalition*, which makes a number of $auth$ atoms true and records which subject belongs to which organization in $orgUser$, and *leaveCoalition*, which revokes all the $auth$ atoms for any subject belonging to the organization leaving the coalition and removes all those subjects from $orgUser$. The crucial problem in implementing the coalition workload is representing the $orgUser$ relation while achieving the desired security guarantees. While our results cover all our candidate systems (see Figure 1), for lack of space we only discuss the access matrix and RBAC systems. See [11] for further details.

We begin with the AC-preservation guarantee, which requires every one of the workload's $auth(x, y, z)$ queries to be equivalent to the system's $auth(x, y, z)$. (Correctness is required throughout.) The main difference in the models we studied is how easy it is to represent the $orgUser$ relation. AMa has no space to store anything besides the coalition's authorization policy, so it fails to admit an AC-preserving implementation. The rest of the AM models as well as all the RBAC models can all store at least one string in addition to the coalition's authorization policy; hence, they admit AC-preserving implementations by string-packing $orgUser$ into that one string.

Theorem 4: AMb, AMc, AMd, and RBACa 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 encoding it in a single, long string and adding that to the state. The key, though, is that the string cannot change the authorization policy, since otherwise we would not satisfy AC-preservation. 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 string, but AMb, AMc, and AMd can all store such a string and therefore admit correct 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 string and (ii) how to insert it into the state without changing the authorization policy of the state. For (ii), we simply add the string to the subject type S without adding an entry to m. Proof for AMc and AMd are by reductions from AMb.

For RBACa, the authorization policy is given as the database join of the UR and PA relations on their respective role columns. Any UR entry with a role that is assigned no permissions (a "hanging" UR) is an entry that does not impact the authorization policy of the system; likewise, for hanging PA entries. Hence for each $auth(a, b, c)$ in the coalition policy enter $UR(a, r)$ and $PA(r, b, c)$ into the RBAC state, and for some role r not already in the state, enter $UR(stringpack, r)$ where $stringpack$ is the string packing of $orgUser$. □

When we impose the homomorphism restriction, we can no longer use a single string to represent $orgUser$, and only AMc and RBACa admit an AC-preserving, homomorphic implementation. The key insight is that while AMb and AMd have storage space besides the matrix, all of that storage space is a unary relation, and under the homomorphism guarantee a unary relation does not suffice to store the binary relation $orgUser$; in contrast, AMc and RBACa have at least binary relations for storing $orgUser$.

Theorem 5: Neither AMa, AMb, nor AMd admit correct, AC-preserving, homomorphic implementations of the coalition workload. AMc and RBACa admit a correct, AC-preserving homomorphic implementation.

Proof (sketch): AMa, AMb, and AMd fail 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 proof begins by showing that if there is a homomorphic implementation, there is a homomorphic function that encodes the workload state as a system state and another homomorphic function that decodes the system state. The proof goes on to show that there is no pair of homomorphic encoder/decoder functions that can 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 the $auth$ query 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 $orgUser$.

RBACa has hanging UR and hanging PA entries to store additional state, both of which are binary. For each $orgUser(a, b)$, it suffices to ensure that no roles named b already exist and to enter $UR(a, b)$ into the RBAC state. □

Finally, we show that AMc admits a correct, AC-preserving, homomorphic, compatible implementation but requires each AMc command to be implemented as something other than itself.

Theorem 6: AMc admits a compatible implementation but cannot implement each workload AMc command with the corresponding system AMc command.

Proof (sketch): The existence of a compatible implementation is proven by demonstrating the implementation in a homomorphic programming language. For the negative result, the proof is by contradiction. It assumes the implementation exists and then demonstrates a command trace from the workload for which there is a command trace from the system commands that lead to the same state. The contradiction arises because the two traces have distinct $orgUser$ relations but nevertheless end at the same system state. □

C. Hospital Workload Results

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

To change the state, the workload has administrative commands for altering D, C, W, R_{pri}, R_{med}, R_{cler} and the following non-administrative commands.

- *modifyMedicalData(issuer,p)*: edits the medical data of patient p. Permission to issue belongs to R_{pri}.
- *viewMedicalData(issuer,p)*: view the medical data of patient p. Permission to issue 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.

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

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.

Recall that BLPa and BLPb both fix the set of rights to $\{read, write, append, execute\}$. Since the hospital workload has more rights, neither BLPa nor BLPb can implement the hospital workload correctly with AC-preservation. BLPc, however, allows for arbitrary rights and in fact has a simple access matrix to represent its access control policy. BLPc also has additional storage that can be used to represent the extra fields in the hospital workload state; thus, there is a correct, AC-preserving implementation. Since that extra state includes a binary relation, there is also a homomorphic implementation. Since that extra state can be modified independently of the access matrix, there is never any need to add or delete entries from the access matrix unless required to by the workload; thus, there is a safe implementation. Administration-preservation, however, cannot be achieved with AC-preservation and correctness because regular hospital staff are free to add new patients, which requires adding new objects to achieve AC-preservation, but adding new objects in BLPc requires administrative rights.

Proposition 1: BLPc admits a safe, homomorphic, AC-preserving, correct implementation of the hospital workload. BLPc admits no administration-preserving, AC-preserving, correct implementations.

Proof (sketch): The hospital access control policy can be represented as the access matrix m of BLPc. The remainder of the hospital state is a finite collection of finite relations, which can be homomorphically represented using the binary relation $clear$. □

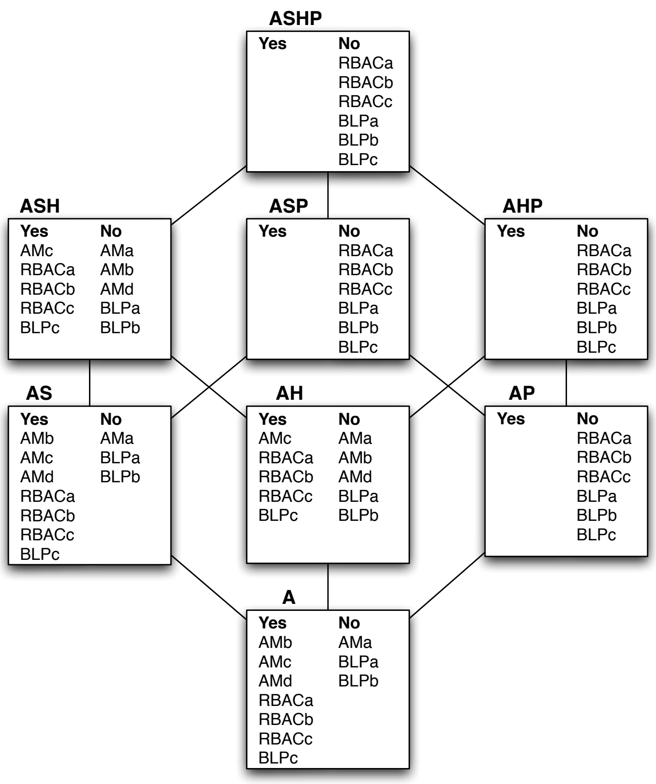

Fig. 2. Summary of the hospital workload results. AMx is an access matrix system; RBACx is a role-based access control system; BLPx is a Bell-LaPadula system. A stands for AC-preservation, H for homomorphism, S for safety, P for administration-preservation. Correctness is required throughout.

VI. Related Work and Future Work

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.

For a discussion of work on authorization logics, modal logics, and the bisimulations related to the frameworks discussed here, see the extended version of this paper [11].

VII. 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] T. L. Hinrichs, D. Martinoia, W. C. Garrison III, A. J. Lee, A. Panebianco, and L. Zuck, "Application-Sensitive Access Control Evaluation using Parameterized Expressiveness (Extended Version)," http://people.cs.pitt.edu/~adamlee/pubs/2013/hinrichs2013applicationsensitive-long.pdf, 2013.

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

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

[14] 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.

[15] 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.

[16] 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.

Appendix

A. *Extensional Access Control Systems and Homomorphisms*

Here we include formal definitions for *extensional* access control systems and workloads, as well as a full definition for *homomorphic implementations*. We also describe a programming language in which all expressible implementations are homomorphic.

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 [16].

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 16 (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 17 (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)
- $next : States(\mathcal{M}) \times \mathcal{L} \to \mathcal{S}$ (the transition function) ♦

With these definitions in hand, we can give a more thorough definition of the homomorphism security guarantee.

Definition 18 (Homomorphisms): A constant substitution $v : \mathcal{U} \to \mathcal{U}$ 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. ♦

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 similar to Algol-like programming languages. An HPL program is a sequence of statements that (i) manipulates a variable assignment by changing existing values for variables and binding new variables to values and (ii) prints values to an output buffer. HPL differs from traditional programming languages in that (i) it disallows string manipulation and string literals, and (ii) the only data structures it supports are sets and sequences of relational atoms and strings. We can think of HPL as a language for writing a compiler that takes as input a command from a workload and outputs a (sequence of) commands in the implementing access control system.

To write a state-mapping for a workload implementation in HPL, we write a program that takes as input a variable assigned to the current workload state and creates a new variable whose value is the system state that corresponds to that workload state. To write an command-mapping for a workload implementation, we write one HPL program for each workload command. Each program takes as input the current state plus all the inputs of the corresponding workload command and outputs the sequence of system commands that ought to be executed. To write a query-mapping in HPL, we write one program for each workload query that takes as input the set of system queries that are true and creates a new variable whose value dictates whether the workload query is true or false.

Table I gives HPL's semantics. μ is a variable assignment (*i.e.*, at each step of program execution, the mapping from program variables to values); γ is the output buffer (*i.e.*, at each step of execution, the sequence of relational atoms that have been output by the program). The commands $output$ and $outputSet$ append relational atoms to the output buffer. The semi-colon sequences two program statements; := is variable assignment; set comprehension is expressed using traditional set-notation; and the command $nFreshConst$ returns object constants that are not present in the current state.

The following theorem ensures that the sequence of commands output by every program written in HPL is homomorphic. In the course of the proof, we also show that the process of manipulating the variable assignment given to every HPL program is homomorphic. See [11] for full details.

Theorem 7: If μ 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 μ then $output[\![P, \mu]\!][v] = output[\![P, \mu[v]]\!]$.

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

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

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

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

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

B. Meta-theorem Proofs

Theorem 8 (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.

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

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

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

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 $\mathit{terminal}(y,\alpha^{\mathcal{Y}_2}(y,l))$ is query-equivalent to $\mathit{terminal}(s,\alpha^{\mathcal{Y}_1}(s,l))$. 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 $\mathit{Th}(w_i) = \pi^{\mathcal{Y}_2}(\mathit{Th}(t_i))$. But that is immediate by transfinite induction since by correctness $\mathit{Th}(w_i) = \pi^{\mathcal{Y}_1}(\mathit{Th}(s_i))$ and by construction $\mathit{Th}(s_i) = \pi(\mathit{Th}(t_i))$. □

Theorem 9 (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 1, except at the end we apply the following claim. If π_1 and π_2 are AC-preserving query-mappings then $\pi_1(\pi_2(x))$ is an

AC-preserving query-mapping. To prove the claim we must show that for all theories x that $auth(r) \in \pi_1(\pi_2(x))$ if and only if $auth(r) \in x$.

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

Theorem 10 (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 1, 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 $f(g(\bar{x}))[v] = f(g(\bar{x})[v]) = f(g(\bar{x}[v]))$.)

$$\begin{aligned} & \text{for all } x \in \textit{States}(\mathcal{W}).\sigma^{\mathcal{Y}_2}(x) = \sigma(\sigma^{\mathcal{Y}_1}(x)) \\ & \text{for all } x \in \textit{States}(\mathcal{Y}_2).\pi^{\mathcal{Y}_2}(\textit{Th}(x)) = \pi^{\mathcal{Y}_1}(\pi(\textit{Th}(x))) \end{aligned}$$

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 1. 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 1, 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 $\alpha(y, l, z)$. □