EON: Modeling and Analyzing Dynamic Access Control Systems with Logic Programs

Avik Chaudhuri
University of California
Santa Cruz, USA
avik@cs.ucsc.edu

Prasad Naldurg
Microsoft Research
Bangalore, India
prasadn@microsoft.com

Sriram K. Rajamani
Microsoft Research
Bangalore, India
sriram@microsoft.com

G. Ramalingam
Microsoft Research
Bangalore, India
grama@microsoft.com

Lakshmisubrahmanyam Velaga
Indian Institute of Management
Bangalore, India
lakshmis07@iimb.ernet.in

ABSTRACT

We present EON, a logic-programming language and tool that can be used to model and analyze dynamic access control systems. Our language extends Datalog with some carefully designed constructs that allow the introduction and transformation of new relations. For example, these constructs can model the creation of processes and objects, and the modification of their security labels at runtime. The information-flow properties of such systems can be analyzed by asking queries in this language. We show that query evaluation in EON can be reduced to decidable query satisfiability in a fragment of Datalog, and further, under some restrictions, to efficient query evaluation in Datalog. We implement these reductions in our tool, and demonstrate its scope through several case studies. In particular, we study in detail the dynamic access control models of the Windows Vista and Asbestos operating systems. We also automatically prove the security of a webserver running on Asbestos.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, Information flow controls, Verification*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques, Mechanical verification*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematic Logic—*Logic and constraint programming*

General Terms

Security, Verification, Languages, Algorithms

Keywords

dynamic access control, logic programming, automatic verification

1. INTRODUCTION

Most modern operating systems implement access control models that try to strike a reasonable balance between security and practice. Unfortunately, finding such a balance can be quite delicate: security concerns often lead to inflexible restrictions, which do not always seem practical. To tackle this conflict, these systems typically admit various ways of controlling access at runtime.

This paper is about verifying such access control systems automatically. We focus on systems in which processes and objects are labeled with security levels, and processes are prevented from accessing objects based on their labels. Such access control systems represent the state of the art in both the commercial world and the academic world, exemplified by Windows Vista and Asbestos [11]. They are typically weaker than the pioneering models of this approach, due to Bell-LaPadula [3] and Biba [4], which have strong secrecy and integrity properties, but turn out to be too restrictive in practice [15]. In particular, some facility to control labels at runtime often seems to be necessary in these systems.

We illustrate this point with an example. Consider a model in which objects downloaded from the Internet are labeled `Low`, and `High` processes are prevented from executing `Low` objects. In this model, suppose that a `High` process needs to run an executable f downloaded from the Internet (say, to install a new application), and the integrity of f can be established (say, by verifying a digital certificate). Then, the `High` process should be able to run f by upgrading it to `High`. On the other hand, if the integrity of f cannot be established, the `High` process should still be able to run f by downgrading itself to `Low` (following the principle of least privilege [14]).

Windows Vista implements an access control model along these lines. In particular, Windows Vista's access control model aims to prevent privilege escalation, data tampering, and code tampering by viruses by enforcing a system-wide integrity policy based on labels. However, to admit the above scenarios, the model allows labels to be lowered or raised at runtime. This requires explicit authorization by the user. While an informed user may be able to decide whether such authorization is safe, there is a real danger that an uninformed user may inadvertently authorize unsafe information flows. For instance, a `High` process can run a `Low` executable f, as above, by downgrading itself to `Low`. As such, running f cannot do much damage—in particular, f cannot write `High` objects, since `Low` processes are prevented from writing `High` objects in the model. However, another `High` process may upgrade f to `High` and run it, without verifying its integrity. Unfortunately, f may be a virus that can then write `High` objects.

The Asbestos operating system implements a related access control model. In this model, process labels are dynamically tainted on communication with other processes, and such taints are propagated to isolate processes based on the secrets they carry. The

CCS'08, October 27–31, 2008, Alexandria, Virginia, USA.

model aims to prevent leaking of those secrets. At the same time, the model allows a customized form of declassification, and thereby admits some information-flow vulnerabilities.

Although Windows Vista and Asbestos differ in their details, both systems implement dynamic access control models, based on labels, that try to balance concerns of security and practice. The information-flow properties of these systems have not been fully studied. In this paper, we develop a technique to model and analyze such systems, and to automatically find information-flow attacks in those systems, or conversely prove their security.

At the heart of our technique is a new logic-programming language called EON, that extends Datalog with dynamic operators for creating and modifying simple objects. We show how we can code information-flow violations as queries in this language, and use query evaluation to find possible vulnerabilities. EON has some carefully designed restrictions—new names can be introduced only through unary relations, only unary relations can be transformed, and some monotonicity conditions must be satisfied. These restrictions are obeyed naturally by our specifications of Windows Vista and Asbestos. We show that with these restrictions, query evaluation for EON is decidable. Our crucial insight is that with these restrictions, it is possible to reduce query evaluation in EON to query satisfiability in a fragment of Datalog. Then, we adapt an algorithm due to Halevy *et al*. [12] (with minor corrections), to decide this satisfiability problem. Further, if the EON program does not have negations over derived relations, we show a simpler reduction to query evaluation in Datalog, which allows us to solve the program and generate attacks or proofs very efficiently.

We implement these reductions in our tool, and evaluate in detail the security of Windows Vista and Asbestos with EON. Our experiments highlight EON's programmability. For instance, we study the impact of various design choices, by making small, local changes in specific models and observing their influence on the attacks or proofs generated. We also model specific usage disciplines, and prove that some attacks are not possible if those disciplines are enforced (either statically or at runtime). In sum, EON seems to be an effective tool to specify, understand, and verify access control models. We expect that this approach can be used to study other dynamic systems just as well.

The rest of the paper is organized as follows. The first half (Sections 2–4) is devoted to theory. The next half is devoted to applications. In Section 2, we describe the syntax and semantics of the EON language. In Section 3, we show how query evaluation in EON can be reduced to query satisfiability in a fragment of Datalog. (A satisfiability algorithm for this fragment is reviewed in [7].) In Section 4, we show how query evaluation in a fragment of EON can be reduced to efficient query evaluation in Datalog. In Section 5, we show applications of our technique through several experiments with the EON tool. In Section 6, we discuss related work. Finally, in Section 7, we discuss our contributions and conclude.

2. THE EON LANGUAGE

In this section, we introduce the EON language, and describe its syntax and semantics. We begin by providing a brief review of Datalog. We then extend Datalog with some carefully designed dynamic operators[1] (Section 2.1), and present the semantics of these operators (Section 2.2). Finally, we define the syntax and semantics of queries in the language (Section 2.3).

Datalog is a convenient logic-programming language to express relational access control models [18, 16, 10, 2]. In Datalog, a *positive literal* $\mathcal{S}$ is of the form $R(t_1, \ldots, t_m)$, where R is a relation, $m \geq 0$, and each t_i is a variable or a constant. A *negative literal* is of the form $!\mathcal{S}$. A *clause* is of the form

$$\mathcal{S} \ :- \ \mathcal{L}_1, \ldots, \mathcal{L}_n.$$

where each $\mathcal{L}_i$ is a positive or negative literal. We refer to the left hand side of $:-$ as the *head*, and the right hand side of $:-$ as the *body*. A Datalog program is a collection of clauses.

A clause without a body is a *fact*. A clause is *safe* if every variable in the clause appears in some positive literal in the body. A program is safe if all clauses in the program are safe.

A relation *depends* on another if there is a clause in the program that has the former relation in the head and the latter in the body; the dependency is *negative* if the literal that contains the latter relation is negative. A *base relation* does not depend on any other relation. A *base fact* is a fact on a base relation. A program is *stratified* if there is no negative dependency in any dependency cycle between relations in the program.

In a safe stratified program, a clause "$\mathcal{S} \ :- \ \mathcal{L}_1, \ldots, \mathcal{L}_n.$" with variables $\widetilde{x}$ is interpreted as the first-order logic formula $\forall \widetilde{x}.\ \mathcal{L}_1 \wedge \cdots \wedge \mathcal{L}_n \Longrightarrow \mathcal{S}$. A program is interpreted as the conjunction of the interpretations of its clauses.

A *database* is a set of base facts. Given a program $\mathbb{F}$ and a database $\mathbb{DB}$, let $\mathcal{I}(\mathbb{F}, \mathbb{DB})$ be the set of facts that are implied by the interpretation of $\mathbb{F} \cup \mathbb{DB}$. This set can be computed efficiently [20].

2.1 Syntax

In EON, we extend Datalog with two dynamic operators: new and next. Before we formally describe their syntax and semantics, we present a simple example to illustrate the language. This example describes a dynamic system where new administrators and users can be added to the system over time, any user x can be promoted to an administrator by another administrator y, and any administrator can control the system. The new operator models the creation of fresh constants, and the next operator models the transformation of relations over those constants:

```
new Admin.
new User.
next Admin(x) :- User(x), Admin(y).
Control(x) :- Admin(x).
```

Can a user that is not an administrator control the system?

```
? User(x), !Admin(x), Control(x).
```

This query evaluates to false according to the operational semantics, described in Sections 2.2 and 2.3. Intuitively, the program does not reach a state where $\texttt{User}(x)$ and $\texttt{Control}(x)$ are both true but $\texttt{Admin}(x)$ is not. In contrast, the following query asks if a user that is not an administrator can *eventually* control the system:

```
? User(x), !Admin(x) ; Control(x).
```

Here ; denotes sequencing of queries. This query evaluates to true; intuitively, the program can reach a state where $\texttt{User}(x)$ is true but $\texttt{Admin}(x)$ is not, then reach a state where $\texttt{Control}(x)$ is true. (In the latter state, we expect that $\texttt{Admin}(x)$ is also true.)

Formally, an EON program is a set of EON clauses. Let $\mathcal{E}$ be a set of unary base relations, called *dynamic relations*, and $\mathcal{B}$ range over subsets of $\mathcal{E}$. Intuitively, these relations can be introduced or transformed by the dynamic operators. The grammar of EON clauses is shown below.

$\mathcal{C}$ ::=	EON clause
$\mathcal{S} \ :- \ \mathcal{L}_1, \ldots, \mathcal{L}_n.$	clause
new $\mathcal{B} \ :- \ R.$	create object
next $\mathcal{B}(x), !\mathcal{B}'(x) \ :- \ R(x).$	modify object

[1] In [7], we show that even a slight generalization of these operators leads to undecidability of query evaluation in the language.

For our convenience, we require that the body of a new or next clause contains exactly one positive literal. In examples, we sometimes omit that literal, or write several literals instead, since the required literal can be equivalently defined by a Datalog clause.

The Datalog fragment of an EON program $\mathbb{P}$ is written as $\widehat{\mathbb{P}}$. We say that $\mathbb{P}$ is safe if $\widehat{\mathbb{P}}$ is safe, and $\mathbb{P}$ is stratified if $\widehat{\mathbb{P}}$ is stratified. In the sequel, we consider only safe stratified programs.

2.2 Semantics

We now give an operational semantics for EON programs. Specifically, we describe the reduction of an EON program $\mathbb{P}$ by a binary relation $\xrightarrow{\mathbb{P}}$ over databases; thus, an EON program defines a (possibly nondeterministic) transition system over databases.

We first describe the semantics of the new operator. The clause "new $\mathcal{B}$:– R." is *enabled* if R evaluates to true in the current database. Execution of the clause creates a fresh constant c and adds $B(c)$ to the database, for every B in $\mathcal{B}$.

$$\frac{\textsf{new}\ \mathcal{B}\ \text{:–}\ R.\ \in \mathbb{P} \qquad R \in \mathcal{I}(\widehat{\mathbb{P}}, \mathbb{DB}) \qquad c \text{ is a fresh constant} \qquad \mathbb{DB}^{+} = \{B(c) \mid B \in \mathcal{B}\}}{\mathbb{DB} \xrightarrow{\mathbb{P}} \mathbb{DB} \cup \mathbb{DB}^{+}}$$

Next, we describe the semantics of the next operator. The clause "next $\mathcal{B}(x), !\mathcal{B}'(x)$:– $R(x)$." is *enabled* if there is some constant c such that $R(c)$ evaluates to true in the current database. Execution of the clause modifies the interpretation of some relations in $\mathcal{E}$ for c. Specifically, it adds $B(c)$ to the database for every B in $\mathcal{B}$ and removes $B(c)$ from the database for every B in $\mathcal{B}'$. Note that if there are several constants c such that $R(c)$ evaluates to true in the current database, then execution of the clause non-deterministically chooses *one* such c for the update.

$$\frac{\textsf{next}\ \mathcal{B}(x), !\mathcal{B}'(x)\ \text{:–}\ R(x).\ \in \mathbb{P} \qquad R(c) \in \mathcal{I}(\widehat{\mathbb{P}}, \mathbb{DB}) \qquad \mathbb{DB}^{+} = \{B(c) \mid B \in \mathcal{B}\} \qquad \mathbb{DB}^{-} = \{B(c) \mid B \in \mathcal{B}'\}}{\mathbb{DB} \xrightarrow{\mathbb{P}} \mathbb{DB} \cup \mathbb{DB}^{+} \backslash \mathbb{DB}^{-}}$$

The reflexive transitive closure of $\xrightarrow{\mathbb{P}}$ is written as $\xrightarrow{\mathbb{P}}{}^{\star}$.

2.3 Queries

Queries in EON can include basic (Datalog-style) queries; they can further use the operator ; to sequence such queries.

$\mathcal{Q} ::=$	EON query
$\mathcal{S}$	basic query
$\mathcal{S}\ ;\ \mathcal{Q}$	sequencing

As usual, for our convenience we require that a basic query contains exactly one positive literal; elsewhere, we often write several literals instead. Let σ range over substitutions of variables by constants. The judgment $\mathbb{DB}, \mathbb{DB}', \sigma \vdash_{\mathbb{P}} \mathcal{Q}$ means that:

> "Starting from a database $\mathbb{DB}$, the program $\mathbb{P}$ eventually reaches a database $\mathbb{DB}'$, satisfying the query $\mathcal{Q}$ with substitution σ".

We first describe the semantics of basic queries. If the initial database $\mathbb{DB}$ evolves to a database $\mathbb{DB}'$ such that $\mathcal{S}\sigma$ evaluates to true in $\mathbb{DB}'$, then the program satisfies the basic query $\mathcal{S}$ with substitution σ.

$$\frac{\mathbb{DB} \xrightarrow{\mathbb{P}}{}^{\star} \mathbb{DB}' \qquad \mathcal{S}\sigma \in \mathcal{I}(\widehat{\mathbb{P}}, \mathbb{DB}')}{\mathbb{DB}, \mathbb{DB}', \sigma \vdash_{\mathbb{P}} \mathcal{S}}$$

Next, we describe the semantics of sequencing. If the initial database $\mathbb{DB}$ evolves to a database $\mathbb{DB}'$ such that the basic query $\mathcal{S}$ is satisfied with substitution σ, and $\mathbb{DB}'$ evolves to a database $\mathbb{DB}''$ such that the query $\mathcal{Q}$ is satisfied with substitution σ, then the program satisfies the query $\mathcal{S}\ ;\ \mathcal{Q}$ with substitution σ.

$$\frac{\mathbb{DB}, \mathbb{DB}', \sigma \vdash_{\mathbb{P}} \mathcal{S} \qquad \mathbb{DB}', \mathbb{DB}'', \sigma \vdash_{\mathbb{P}} \mathcal{Q}}{\mathbb{DB}, \mathbb{DB}'', \sigma \vdash_{\mathbb{P}} \mathcal{S}\ ;\ \mathcal{Q}}$$

3. QUERY EVALUATION IN EON

We now describe how EON queries can be evaluated. Formally, the query evaluation problem for EON is:

> Given an EON program $\mathbb{P}$ and an EON query $\mathcal{Q}$, are there some database $\mathbb{DB}$ and substitution σ such that $\varnothing, \mathbb{DB}, \sigma \vdash_{\mathbb{P}} \mathcal{Q}$?

We show that this problem is decidable under some suitable assumptions of monotonicity (see below). The essence of our algorithm is to reduce the EON query evaluation problem to a decidable satisfiability problem over Datalog.

Recall that, given a Datalog program $\mathbb{F}$ and a database $\mathbb{DB}$, $\mathcal{I}(\mathbb{F}, \mathbb{DB})$ denotes the result of evaluating $\mathbb{F}$ over $\mathbb{DB}$. Given a positive literal $\mathcal{S}$, we use the notation $\mathbb{DB} \vdash_{\mathbb{F}} \mathcal{S}$ to indicate that there is some substitution σ such that $\mathcal{I}(\mathbb{F}, \mathbb{DB})$ contains $\mathcal{S}\sigma$. Now, $\mathcal{S}$ is *satisfiable* in $\mathbb{F}$ if there exists a database $\mathbb{DB}$ such that $\mathbb{DB} \vdash_{\mathbb{F}} \mathcal{S}$. The following satisfiability problem over Datalog is decidable.

THEOREM 1 (A DECIDABLE FRAGMENT OF DATALOG [12]). *Satisfiability is decidable for safe stratified Datalog programs with unary base relations.*

Recall that a database is a set of base facts. Given an EON program $\mathbb{P}$, we say that a database is *reachable* in $\mathbb{P}$ if it can be reached from the initial database $\varnothing$ by a sequence of transitions defined by $\mathbb{P}$. Now, the only base facts in any reachable database are over relations in $\mathcal{E}$. In the sequel, we focus on such databases. In particular, we view a database $\mathbb{DB}$ as a pair (U, I), where U is a set of constants and $I : \mathcal{E} \to 2^{U}$.

Given a database $\mathbb{DB} = (U, I)$ and a subset of constants $X \subseteq U$, we define the restriction of $\mathbb{DB}$ to X, denoted $\mathbb{DB}|_X$, to be (X, I_X), where $I_X(B) \triangleq I(B) \cap X$. We say that $\mathbb{DB}_1 \leq \mathbb{DB}_2$ if there exists an X such that $\mathbb{DB}_1 = \mathbb{DB}_2|_X$.

Now, a positive literal S is *monotonic* in $\mathbb{P}$ if for all $\mathbb{DB}_1$ and $\mathbb{DB}_2$, if $\mathbb{DB}_1 \vdash_{\widehat{\mathbb{P}}} S$ and $\mathbb{DB}_1 \leq \mathbb{DB}_2$, then $\mathbb{DB}_2 \vdash_{\widehat{\mathbb{P}}} S$.

3.1 Basic queries, unguarded transitions

Suppose that we are given a basic query $\mathcal{S}$ to evaluate on an EON program $\mathbb{P}$. We assume that $\mathcal{S}$ is monotonic in $\mathbb{P}$. Further, suppose that all dynamic clauses in $\mathbb{P}$ are *unguarded*. A new clause is unguarded if its body is a fact (*e.g.*, "`True`.") in the program. A next clause is unguarded if the relation in its body is a *pure relation*. The concept of a pure relation is defined inductively as follows: a (unary) relation R is pure if either $R \in \mathcal{E}$, or every clause in the program with R in its head is of the form "$R(x)$:– $\mathcal{L}_1, \ldots, \mathcal{L}_n$.", where each $\mathcal{L}_i$ is either $R_i(x)$ or $!R_i(x)$ for some pure R_i.

Note that an unguarded new clause is always enabled. Whether an unguarded next clause is enabled for a constant c depends only on the value of the relations in $\mathcal{E}$ for c.

Now, we evaluate $\mathcal{S}$ on $\mathbb{P}$ by translating $\mathbb{P}$ to a Datalog program $\lfloor \mathbb{P} \rfloor$, and deciding if there exists a database $\mathbb{DB}$ such that $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \mathcal{S}$ and $\mathbb{DB}$ is reachable in $\mathbb{P}$. The latter problem is reduced to a basic satisfiability problem of the form $\mathbb{DB} \vdash_{\lfloor \mathbb{P} \rfloor} \lfloor \mathcal{S} \rfloor$, by encoding the reachability condition into $\lfloor \mathbb{P} \rfloor$ and defining $\lfloor \mathcal{S} \rfloor$ to be $\mathcal{S}$ augmented with the reachability condition.

Given a constant c that belongs to a database $\mathbb{DB} = (U, I)$, we define its *atomic state* to be the set $\{B \in \mathcal{E} \mid c \in I(B)\}$. We say that an atomic state $X \subseteq \mathcal{E}$ is reachable if there exists a reachable database $\mathbb{DB}$ that contains a constant whose atomic state is X.

LEMMA 1. *Given an EON program $\mathbb{P}$ in which all dynamic clauses are unguarded, a database $\mathbb{DB}$ is reachable iff all constants in the database have a reachable atomic state.*

3.1.1 From EON to Datalog

We now show how reachable atomic states can be encoded in Datalog. Specifically, given a EON program $\mathbb{P}$, we define a set of Datalog clauses $\mathcal{T}(\mathbb{P})$ for a unary relation `Reachable`, such that every constant in `Reachable` has a reachable atomic state, and every constant that has a reachable atomic state is in `Reachable`. Some of these clauses are not safe. Later, we present a clause transformation that uniformly transforms all clauses to ensure safety.

We begin by defining some auxiliary relations. Let $\mathcal{E} = \{B_1, \ldots, B_k\}$. For each B_i ($i \in \{1, \ldots, k\}$), we include the following Datalog clauses, that check whether a pair of constants have the same value at B_i:

$$\texttt{Same}B_i(x,y) \ :- \ B_i(x),\ B_i(y).$$
$$\texttt{Same}B_i(x,y) \ :- \ !B_i(x),\ !B_i(y).$$

Now, consider an unguarded new clause of the form:

$$\texttt{new}\ \ B_{i_1}, \ldots, B_{i_m}.$$

Let $\{B_{j_1}, \ldots, B_{j_n}\} = \mathcal{E} \backslash \{B_{i_1}, \ldots, B_{i_m}\}$. We replace this clause with the following reachability clause in Datalog:

$$\texttt{Reachable}(x) \ :- \ B_{i_1}(x), \ldots, B_{i_m}(x), \\ !B_{j_1}(x), \ldots, !B_{j_n}(x).$$

This clause may be read as follows: a satisfying database for the transformed Datalog program may contain a constant x whose atomic state is $\{B_{i_1}, \ldots, B_{i_m}\}$. Intuitively, new constants in EON are represented by existentially quantified variables in Datalog.

Now, consider an unguarded next clause of the form:

$$\texttt{next}\ \ B_{i_1}(x), \ldots, B_{i_m}(x), \\ !B_{j_1}(x), \ldots, !B_{j_n}(x) \ :- \ R(x).$$

Let $\{B_{k_1}, \ldots, B_{k_r}\} = \mathcal{E} \backslash \{B_{i_1}, \ldots, B_{i_m}, B_{j_1}, \ldots, B_{j_n}\}$. R is pure; so we replace this clause with the following reachability clause in Datalog:

$$\texttt{Reachable}(x) \ :- \\ \texttt{Reachable}(y),\ R(y), \\ B_{i_1}(x), \ldots, B_{i_m}(x), \\ !B_{j_1}(x), \ldots, !B_{j_n}(x), \\ \texttt{Same}B_{k_1}(x,y), \ldots, \texttt{Same}B_{k_r}(x,y).$$

This clause may be read as follows: a satisfying database for the transformed Datalog program may contain a constant x whose atomic state is $\mathcal{B} \cup \{B_{i_1}, \ldots, B_{i_m}\} \setminus \{B_{j_1}, \ldots, B_{j_n}\}$, if that database also contains a constant y that satisfies $R(y)$ and has some atomic state $\mathcal{B}$. Intuitively, the Datalog variables x and y represent the same EON constant, in possibly different "states", one of which can be reached from the other.

Finally, the following clause checks whether there is any constant in a satisfying database for the transformed Datalog program whose atomic state is unreachable:

$$\texttt{BadState} \ :- \ !\texttt{Reachable}(x).$$

The set of clauses $\mathcal{T}(\mathbb{P})$ contains all of the clauses above. Now, let $\mathsf{U} \in \mathcal{E}$ be a fresh relation, which models the range of substitutions. For any clause $\mathcal{C} \in \mathcal{T}(\mathbb{P})$, we obtain a transformed clause $\lfloor \mathcal{C} \rfloor$ by augmenting the body of $\mathcal{C}$ with an additional condition $\mathsf{U}(x)$ for every variable x in $\mathcal{C}$. The clause $\lfloor \mathcal{C} \rfloor$ is guaranteed to be safe.

Now, let $\lfloor \mathbb{P} \rfloor = \{\lfloor \mathcal{C} \rfloor \mid \mathcal{C} \in \widehat{\mathbb{P}} \cup \mathcal{T}(\mathbb{P})\}$. Let $\lfloor \mathcal{S} \rfloor$ be the query $\mathcal{S}, !\texttt{BadState}$ augmented with an additional condition $\mathsf{U}(x)$ for every variable x in $\mathcal{S}$. We then have the following result.

THEOREM 2. *Given an EON program $\mathbb{P}$ in which all dynamic clauses are unguarded, a monotonic basic query $\mathcal{S}$ is true in $\mathbb{P}$ iff the query $\lfloor \mathcal{S} \rfloor$ is satisfiable in the Datalog program $\lfloor \mathbb{P} \rfloor$.*

3.1.2 An optimization

The use of (double) negation to define the transformed query $\lfloor \mathcal{S} \rfloor$ can lead to potential inefficiencies in the satisfiability algorithm (described in [7]). We can eliminate the use of this negation by transforming every Datalog clause $\mathcal{C}$ in the given program $\mathbb{P}$ as follows: we augment the body of the clause with the condition $\texttt{Reachable}(x)$ for every variable x in the body. (It is possible to further optimize this transformation, by adding the condition only for variables that do not occur in the head of the clause, as long as we add a similar condition for all variables occurring in $\mathcal{S}$.)

3.2 Basic queries, monotonic transitions

Guarded dynamic clauses do not significantly complicate the transformation. The reachability clause generated for a guarded dynamic clause now includes the guard (*i.e.*, the literal in the body of the dynamic clause) in the body of the reachability clause. The correctness proofs require the guards to be monotonic. Specifically, a generalization of Lemma 1 holds true even for programs with guarded dynamic clauses, as long as the guards are monotonic.

Recall that in the case of unguarded dynamic clauses, the `Reachable` relation depends only on the relations in $\mathcal{E}$, the auxiliary relations $\texttt{Same}B_i$, and itself. However, the encoding of guards in reachability clauses makes the `Reachable` relation dependent on other relations mentioned in those guards. If we now do the optimization of Section 3.1.2, which adds reachability conditions to the clauses of the given program, we may introduce cyclic dependences between `Reachable` and other relations. Thus, we must verify that the transformed program is stratified before checking satisfiability on the transformed program. Interestingly, it turns out that *the transformed program is stratified if and only if the guards are monotonic*. This result yields a simple method to test for the monotonicity of guards.

3.3 Queries with sequencing

Finally, we show how we can handle queries with sequencing. We assume that every basic query in such queries is monotonic. Consider the query $\mathcal{S} \ ; \ \mathcal{Q}$. We first assume that $\mathcal{S}$ and $\mathcal{Q}$ share exactly one variable x. Let $\texttt{Done} \in \mathcal{E}$ be a fresh relation, and $\mathcal{Q}$ be of the form $\mathcal{S}_1 \ ; \ \ldots \ ; \ \mathcal{S}_n$, for some $n \geq 1$. We augment the original EON program with the following dynamic clause:

$$\texttt{next}\ \texttt{Done}(x) \ :- \ \mathcal{S}.$$

We then evaluate the query $\texttt{Done}(x), \mathcal{S}_1 \ ; \ \ldots \ ; \ \mathcal{S}_n$ on the augmented EON program.

More generally, we add a next clause with a fresh `Done` relation for each variable shared by $\mathcal{S}$ and $\mathcal{Q}$, and augment $\mathcal{Q}$ accordingly to account for those variables. On the other hand, if $\mathcal{S}$ and $\mathcal{Q}$ do not share any variable, we add a new clause with a fresh `Done` relation, and augment $\mathcal{Q}$ with $\texttt{Done}(z)$, where z is a fresh variable.

4. EFFICIENT QUERY EVALUATION IN A FRAGMENT OF EON

Under further assumptions, we now show that query evaluation in EON can be reduced to simple query evaluation in Datalog. This result is independent of what we present above. The main advantage of this transformation is efficiency—while checking satisfiability of Datalog programs may take exponential time in the worst case, evaluating Datalog programs takes only polynomial time.

The requirements for this transformation are as follows. There should be no (in)equality constraints over variables. In particular, variables cannot be repeated in the head of a clause. Next, there should be no negations on non-base (derived) relations, although there may be negations on base relations. These conditions turn out to be quite reasonable in practice. In particular, our models of of Vista and Asbestos in Section 5 satisfy these conditions, and most of our queries on these models satisfy these conditions as well.

We assume that sequencing is compiled away as in our original reduction, and consider only basic queries. Further, we assume that no constants appear in the EON program itself. (The transformation can be extended in a straightforward way to allow constants.) The intuition behind the transformation is as follows. Let $\mathcal{E} = \{B_1, \ldots, B_k\}$. We can represent the atomic state of a constant c as the vector $(v_1, \ldots, v_k)$ where v_i is 1 if $B_i(c)$ is true and 0 otherwise. We say that two constants c and c' are *similar* if they have the same atomic state. Now in our case, a Datalog program cannot distinguish between similar constants, *i.e.*, it is not possible to define a query $R(x)$ that is satisfied by c and not c'. (More generally, if c_i is similar to c'_i for $1 \leq i \leq r$, then $R(c_1, \ldots, c_r)$ is true iff $R(c'_1, \ldots, c'_r)$ is true in the program.) Thus we can define a query $\lfloor R \rfloor(x_1, \ldots, x_k)$ which is true iff $R(x)$ is true for any x with atomic state $(x_1, \ldots, x_k)$ that is generated by the EON program.

For every non-base relation R of arity r, we define a new relation $\lfloor R \rfloor$ of arity rk. Given any Datalog clause $\mathcal{C}$, we replace it with a transformed clause $\lfloor \mathcal{C} \rfloor$ as follows. For every variable x in the clause, we introduce k new variables $x_1, \ldots, x_k$. Then, every literal of the form $R(y_1, \ldots, y_r)$, where R is a non-base relation, is transformed into a corresponding literal $\lfloor R \rfloor(y_{11}, \ldots, y_{1k}, \ldots, y_{r1}, \ldots, y_{rk})$ by replacing every occurrence of a variable y_j by the corresponding vector of variables $y_{j1}, \ldots, y_{jk}$. Further, every literal of the form $B_i(x)$ is transformed into the literal $\texttt{True}(x_i)$ and every literal of the form $!B_i(x)$ is transformed into $\texttt{False}(x_i)$. (The auxiliary predicates `True` and `False` are defined by the facts $\texttt{True}(1)$ and $\texttt{False}(0)$.) Finally, for every variable x in the head of the clause, we add the condition $\texttt{Reachable}(x_1, \ldots, x_k)$ to the body of the transformed clause. (As an optimization, we may consider adding this reachability condition only if no non-base relation is applied to x in the body of the clause.) For example, the clause

$$R(x, y) \ :- \ R'(x), !B_1(x), B_2(y).$$

yields the transformed clause:

$$\lfloor R \rfloor(x_1, x_2, y_1, y_2) \ :- \\ \lfloor R' \rfloor(x_1, x_2), \texttt{False}(x_1), \texttt{True}(y_2), \texttt{Reachable}(y_1, y_2).$$

Now, every clause of the form "new $\mathcal{B}$:- R." is transformed to

$$\texttt{Reachable}(z_1, \ldots, z_k) \ :- \ \lfloor R \rfloor.$$

where z_i is 1 if $B_i \in \mathcal{B}$ and 0 otherwise.

Further, every clause of the form "next $\mathcal{B}(x), !\mathcal{B}'(x)$:- $R(x)$." is transformed to

$$\texttt{Reachable}(z_1, \ldots, z_k) \ :- \\ \lfloor R \rfloor(x_1, \ldots, x_k), \\ \texttt{Reachable}(x_1, \ldots, x_k), \\ \texttt{Update}(x_1, z_1), \ldots, \texttt{Update}(x_k, z_k).$$

where $\texttt{Update}(x_i, z_i)$ is $\texttt{True}(z_i)$ if x_i is in $\mathcal{B}$, $\texttt{False}(z_i)$ if x_i is in $\mathcal{B}'$, and $z_i = x_i$ otherwise. (The literal $z_i = x_i$ is implemented by replacing z_i with x_i in the clause.)

We then have the following result.

THEOREM 3. *Given an EON program $\mathbb{P}$ with the above restrictions, a query $\mathcal{Q}$ is true in $\mathbb{P}$ iff the query $\lfloor \mathcal{Q} \rfloor$ is true in the Datalog program $\lfloor \mathbb{P} \rfloor$.*

Proof details for all our results appear separately in [7], and are omitted here due to lack of space.

5. EXPERIMENTS WITH THE EON TOOL

The transformations described in Sections 3 and 4 are implemented in the EON tool [7]. Further, the back end includes implementations of satisfiability and evaluation algorithms over Datalog, and the front end supports some syntax extensions over EON, such as *embedded scripts* [7].

We now present a series of examples that illustrate how the EON tool can be used to model and analyze dynamic access control systems. We begin with Windows Vista's access control model (Section 5.1). We automatically find some integrity vulnerabilities in this model. Then, we automatically prove that exploits for those vulnerabilities can be eliminated by enforcing a certain usage discipline on the model (via static analysis or runtime monitoring). Roughly, this means that a user can be informed about potentially unsafe authorization decisions in this model. Next, we consider Asbestos's access control model (Section 5.2). We automatically validate some secrecy properties of that model. Finally, we model a full-fledged webserver that runs on Asbestos (as described in [11]), and automatically prove a secrecy guarantee for the webserver.

5.1 Windows Vista

The goal of Windows Vista's access control model is to maintain boundaries around trusted objects, in order to protect them from less trusted processes. Trust levels are denoted by *integrity labels* (ILs), such as `High`, `Med`, and `Low`. Every object has an IL. Further, every process is itself an object, and has an IL. A process can spawn new processes, create new objects, and change their ILs, based on its own IL. In particular, a process with IL P_L can[2]:

- lower an object's IL from O_L only if $O_L \sqsubseteq P_L$;
- raise an object's IL to O_L only if $O_L \sqsubseteq P_L$ and the object is not a process;
- read an object;
- write an object with IL O_L only if $O_L \sqsubseteq P_L$;
- execute an object with IL O_L by lowering its own IL to $P_L \sqcap O_L$.

Below, we present an excerpt of a model of such a system in EON. (The full model appears in [7].) The unary base relations in the model have the following informal meanings: `P` contains processes; `Obj` contains objects (including processes); and `Low`, `Med`, `High`, *etc.* contain processes and objects with those ILs.

With `new` and `next` clauses, we specify how an unbounded number of processes and objects, of various kinds, can be created.

```
new Obj,Low.
new Obj,Med.
new Obj,High.

next P(x) :- Obj(x).
...
```

Further, with `next` clauses, we specify how ILs of processes and objects can be changed. For instance, a `Med` process can raise the IL of an object from `Low` to `Med` if that object is not a process; it can also lower the IL of an object from `Med` to `Low`. A `High` process can lower its own IL to `Med` (*e.g.*, to execute a `Med` object).

[2]In fact, the capabilities of a process may be further constrained by Windows Vista's discretionary access control model. However, we ignore this model because it is rather weak; see [7] for a detailed discussion.

```
next Med(y),!Low(y) :- Low(y),!P(y),Med(x),P(x).
next Low(y),!Med(y) :- Med(y),Med(x),P(x).

next Med(x),!High(x) :- High(x),P(x).
...
```

The full model contains several other rules that are implemented by the system. Specifying these rules manually can be tedious and error-prone; instead, EON allows us to embed scripts in our model (as syntax extensions) that generate these rules automatically [7].

Finally, with Datalog clauses, we specify how processes can `Read`, `Write`, and `Execute` objects. A process x can `Read` an object y without any constraints. In contrast, x can `Write` y only if the IL of x is `Geq` (greater than or equal to) the IL of y. Conversely, x can `Execute` y only if the IL of y is `Geq` the IL of x.

```
Read(x,y) :- P(x),Obj(y).

Write(x,y) :- P(x),Geq(x,y).

Execute(x,y) :- P(x),Geq(y,x).

Geq(x,y) :- Med(x),Med(y).
Geq(x,y) :- Med(x),Low(y).
Geq(x,y) :- Low(x),Low(y).
...
```

5.1.1 Integrity vulnerabilities on Windows Vista

We now ask some queries on the model above. For instance, can a `Med` object be read by a `Med` process after it is written by a `Low` process? Can an object that is written by a `Low` process be eventually executed by a `High` process by downgrading to `Med`?

```
? Med(y); Low(x),Write(x,y); Med(z),Read(z,y).
? Low(x),Write(x,y); High(z); Med(z),Execute(z,y).
```

The former encodes a simple data-flow integrity vulnerability; the latter encodes a simple privilege-escalation vulnerability. (In the full model, we study some more complicated vulnerabilities.) When we run these queries, we obtain several exploits for those vulnerabilities. (See, *e.g.*, [6, 8], for more details on such vulnerabilities.) For each exploit, our tool shows a derivation tree; from that tree, we find a sequence of `new`, `next`, and other clauses that lead the system to an insecure state and derive the query. For instance, the former query is derived as follows: first, a `Med` object y is created; next, y is downgraded to `Low` by a `Med` process; next, y is written by a `Low` process x; finally, y is read by a `Med` process z. The latter query is derived as follows: first, a `Low` object y is created; next, y is written by a `Low` process x; next, y is upgraded to `Med` by a `Med` process; next, a `High` process z is downgraded to `Med`; finally, y is executed by z.

Thus, EON can be quite effective as a *debugging* tool—if there is a bug, EON is guaranteed to find it. But recall that if there are no bugs, EON is also guaranteed to terminate without finding any! That is, EON can be just as effective as a theorem-proving tool. In particular, we now prove that the vulnerabilities above cannot be exploited if suitable constraints are imposed on the model. In practice, these constraints may be implemented either by static analysis or by runtime monitoring on programs running in the system.

5.1.2 An usage discipline to recover integrity

Basically, we attach to each object a label `SHigh`, `SMed`, or `SLow`, which indicates a static lower bound on the integrity of the contents of that object; further, we attach to each process a label `DHigh`, `DMed`, or `DLow`, which indicates a dynamic lower bound on the integrity of the values known to that process. The semantics of these labels are maintained as invariants by the model. In particular, these labels are initialized as follows.

```
new Obj,Low,SLow.
new Obj,Med,SMed.
new Obj,High,SHigh.

next P(x),DHigh(x) :- Obj(x).
...
```

Now, whenever an object's IL is lowered, the IL should not fall below the static label of the object.

```
next Low(y),!Med(y) :- Med(y),SLow(y),Med(x),P(x).
...
```

A process's dynamic label may be lowered to reflect that it may know the contents of an object with a lower static label.

```
next DLow(x),!DHigh(x) :- DHigh(x),SLow(y).
...
```

Now, a process x can `Read` an object y only if the dynamic label of x is `DSLeq` (less than or equal to) the static label of y. Conversely, x can `Write` y only if the dynamic label of x is `DSGeq` (greater than or equal to) the static label of y. In contrast, x can `Execute` y only if its own IL is lowered to or below the static label of y. This condition, `SGeq`(y, x), subsumes the earlier condition `Geq`(y, x).

```
Read(x,y) :- P(x),Obj(y),DSLeq(x,y).

DSLeq(x,y) :- DLow(x),SLow(y).
DSLeq(x,y) :- DLow(x),SMed(y).
DSLeq(x,y) :- DMed(x),SMed(y).
...

Write(x,y) :- P(x),Obj(y),Geq(x,y),DSGeq(x,y).

DSGeq(x,y) :- DLow(x),SLow(y).
DSGeq(x,y) :- DMed(x),SMed(y).
DSGeq(x,y) :- DMed(x),SLow(y).
...

Execute(x,y) :- P(x),Obj(y),SGeq(y,x).

SGeq(y,x) :- SLow(y),Low(x).
SGeq(y,x) :- SMed(y),Low(x).
SGeq(y,x) :- SMed(y),Med(x).
...
```

Finally, recall the temporal queries that we ask above. We reformulate the former query for this model—instead of constraining the IL of z, we now constrain its dynamic label, which is a better approximation of its runtime taint.

```
? Med(y) ; Write(x,y),Low(x) ; Read(z,y),DMed(z).
```

This query evaluates to false, showing that the encoded data-flow integrity vulnerability cannot be exploited. The latter query also evaluates to false, showing that the encoded privilege-escalation vulnerability cannot be exploited. The full constrained model appears in [7]. There, we show that exploits for some other complicated vulnerabilities are also eliminated under these constraints.

Thus, with EON, we not only find vulnerabilities in Windows Vista's access control model, but also prove that they can be eliminated by imposing suitable constraints on the model. We conclude that these constraints encode a formal "discipline" that is required to safely exploit the flexibilities provided by the model. In fact, a similar discipline already appears in [6], with manual proofs.

5.2 Asbestos

The goal of Asbestos's access control model is to dynamically isolate trusted processes that require protection from less trusted processes. This isolation is achieved by *taint propagation*. Specifically, in Asbestos each process P has two labels: a *send label* P_S, which is a lower bound on the security of messages that can be sent by P, and a receive label P_R, which is an upper bound on the security of messages that can be received by P. Further, each communication port C has a *port label* C_L, which is an upper bound on the security of messages that can be carried by c. Sending a message from process P to process Q on port C requires that:

$$P_S \sqsubseteq Q_R \sqcap C_L$$

Further, on communication, Q is tainted by P:

$$Q_S \leftarrow Q_S \sqcup P_S$$

In fact, this situation is slightly more complicated with *declassification*. Specifically, there are a small number of *security levels* ($\star$, 0, 1, 2, 3), with minimum $\star$ and maximum 3. A *label* is a finite record of security levels. Labels form a lattice $(\sqsubseteq, \sqcup, \sqcap)$, as follows. (Here L_1, L_2 range over labels, and ℓ over label fields.)

$$\begin{array}{rcl} L_1 \sqsubseteq L_2 & \Leftrightarrow & \forall \ell.\, L_1.\ell \leq L_2.\ell \\ (L_1 \sqcup L_2).\ell & = & \max(L_1.\ell, L_2.\ell) \\ (L_1 \sqcap L_2).\ell & = & \min(L_1.\ell, L_2.\ell) \end{array}$$

Now, an operation $\_^\star$ is defined as follows.

$$L^\star.\ell = \begin{cases} \star & \text{if } L.\ell = \star \\ 3 & \text{otherwise} \end{cases}$$

On communication, Q is tainted by P only in fields that are not $\star$.

$$Q_S \leftarrow Q_S \sqcup (P_S \sqcap Q_S^\star)$$

5.2.1 Secrecy on Asbestos

To understand some security consequences of this model, let us focus on a single field ℓ, and the security levels $\star$, 1, 2, and 3. Below, we present an excerpt of a model of such a system in EON. Let `STAR` denote $\star$, and `i`,`j` range over $\{1, 2, 3\}$. The unary base relations in the model have the following informal meanings: `P` contains processes; `LRi` and `LSj` contain processes x such that $x_R.\ell =$ `i` and $x_S.\ell =$ `j`, respectively; `LSTAR` contains processes x such that $x_S.\ell = \star$ and $x_R.\ell = 3$; and `Mj` contains processes x that carry messages generated by processes y such that $y_R.\ell =$ `j`, respectively. We boot our system with the following clauses; these clauses create an unbounded number of processes of various kinds, and let them generate messages accordingly.

```
new P,LSTAR.
new P,LR1,LS1.
new P,LR2,LS1.
new P,LR3,LS1.

next M2(x),LS2(x),!LS1(x) :- LS1(x),LR2(x).
next M3(x),LS3(x),!LS1(x) :- LS1(x),LR3(x).
...
```

Next, we specify clauses for communication on unrestricted ports (*i.e.*, ports C such that $C_L.\ell = 3$). The requirements and effects of such communication appear in the bodies and heads of these clauses, respectively. Note, in particular, how the relations `Mj` are augmented on such communication. (The full model contains several other, similar rules, generated automatically by scripts.)

```
next M2(x) :- P(x),LSTAR(y),M2(y).
next M3(x) :- P(x),LSTAR(y),M3(y).

next M2(x) :- LSTAR(x),P(y),M2(y).
next M3(x) :- LSTAR(x),P(y),M3(y).

next M2(x),LS2(x),!LS1(x) :-
                        M2(y),LS2(y),LS1(x),LR2(x).
next M3(x),LS2(x),!LS1(x) :-
                        M3(y),LS2(y),LS1(x),LR2(x).
...
```

Finally, we ask some queries. According to [11], in Asbestos the default security level in any field of a receive label is 2. Thus, having 3 in some field of the receive label gives higher read privileges than default; processes with such labels should be able to share messages that default processes cannot know. On the other hand, having 1 in some field of the receive label gives lower read privileges than default; processes with such labels should not be able to know messages shared by default processes. Let `ReadWithout3` denote the existence of a process x for which `M3`(x) is true despite `LRi`(x) for some $i < 3$. On the other hand, let `ReadWith1` denote the existence of a process x for which `Mj`(x) is true for some $j > 1$ despite `LR1`(x). These queries encode secrecy vulnerabilities.

```
ReadWithout3 :- M3(x),LR2(x).
ReadWithout3 :- M3(x),LR1(x).

ReadWith1 :- M2(x),LR1(x).
ReadWith1 :- M3(x),LR1(x).

? ReadWithout3.
? ReadWith1.
```

We find exploits for both queries with EON. The derivations for these queries may be anticipated—messages can be declassified, that is, forwarded by processes z for which `LSTAR`(z) is true, without any constraints or effects.

Now, let `BlameReadWithout3` denote the existence of a process z for which `M3`(z) and `LSTAR`(z) are true. On the other hand, let `BlameReadWith1` denote the existence of a process z for which `Mj`(z) and `LSTAR`(z) are true for some $j > 1$. We now ask the following, revised queries that account for declassification. (These queries encode violations of *robust declassification* [21].)

```
BlameReadWithout3 :- M3(y),LSTAR(y).
BadReadWithout3 :- ReadWithout3,!BlameReadWithout3.

BlameReadWith1 :- M2(y),LSTAR(y).
BlameReadWith1 :- M3(y),LSTAR(y).
BadReadWith1 :- ReadWith1,!BlameReadWith1.

? BadReadWithout3.
? BadReadWith1.
```

Now EON does not find exploits for either query. Note that the revised queries use negation on non-base relations, and thus take a long time to run. We can simulate these queries without using negation, simply by removing the following clauses and asking the same queries as before.

```
next M2(x) :- LSTAR(x),M2(y).
next M3(x) :- LSTAR(x),M3(y).
```

Once again, EON does not find exploits for either query; however, the queries now run much faster. Thus, we have the following secrecy theorem, proved automatically by EON.

THEOREM 4 (SECRECY). *Assume that X is either $\{P \mid P_R.\ell = 3\}$ or $\{P \mid P_R.\ell \neq 1\}$. If $Q \notin X$, then Q can never carry a message generated by a process in X, unless some declassifying process carries that message as well.*

5.2.2 Secrecy in a webserver running on Asbestos

We now present a significantly more ambitious example to demonstrate the scope of our techniques. Specifically, we apply EON to verify a webserver running on Asbestos. This webserver is described in detail in [11]; below, we briefly review its architecture. We then present an excerpt of a model of this webserver in EON, and study its key security guarantee. The full model appears in [7].

The relevant principals include a net daemon, a database proxy, and the users of the webserver. When a user connects, the net daemon spawns a dedicated worker process for that user. The worker process can communicate back and forth with that user over the net; further, it can access a database that is common to all users. The webserver relies on sophisticated protocols for connection handling and database interaction; the aim of these protocols is to isolate processes that run on behalf of different users, so that no user can see a different user's data.

In our model, we focus on two users u and v; processes that run on behalf of these users are tagged as such on creation. We focus on label fields that are relevant for secrecy—these include uc and ut (used for communication and taint propagation by u), and vc and vt (used for communication and taint propagation by v). We model labels with unary base relations that specify the security levels in each field: *e.g.*, for processes x, LSuc1(x) denotes x_S.uc $= 1$; LRut2(x) denotes x_R.ut $= 2$; and LSvcSTAR(x) denotes x_S.vc $=$ STAR; similarly, *e.g.*, for communication ports y, Lvt2(y) denotes y_L.vt $= 2$.

The other unary base relations in the model have the following informal meanings. Useru and Userv contain processes run by u and v, respectively; NETd contains processes run by the net daemon; and Wu and Wv contain worker processes that are spawned by the net daemon for u and v, respectively. All of these processes participate in a connection handling protocol. Further, Ready contains any such process that is ready for communication, after that protocol is executed. Other processes are run by the database proxy. In particular, DBproxyRu and DBproxyRv contain processes that receive database records for u and v, respectively; and DBproxySu and DBproxySv contain processes that send database records for u and v, respectively.

The processes above communicate on well-defined ports. Portu and Portv contain ports on which data is sent over the net by processes running on behalf of u and v, respectively. PortDBu and PortDBv contain ports on which data is received by the database proxy from processes running on behalf of u and v, respectively. PortAny contains unrestricted ports that are used for all other communication.

Finally, to verify secrecy, we let Mu and Mv contain processes that carry u's data and v's data, respectively. We require that no process that runs on behalf of v is eventually in Mu (and vice versa).

We now outline our model. We describe only clauses that involve u; the clauses that involve v are symmetrical. Most processes in the system are created with default send and receive labels. (Any security level in a default send label is 1, and any security level in a default receive label is 2.) For instance, user processes are created as follows.

```
new Useru,Ready,Mu,
      LSuc1,LSut1,LSvc1,LSvt1,
      LRuc2,LRut2,LRvc2,LRvt2.
...
```

Next, we model the connection handling protocol in [11]. When a user u initiates a connection, the net daemon creates a new process, as follows.

```
new NETd,
      LSuc1,LSut1,LSvc1,LSvt1,
      LRuc2,LRut2,LRvc2,LRvt2.
```

This process creates a new port on which data can be sent over the net to u. The security level in the relevant communication field uc of the port's label is 0; thus, processes with default send labels cannot send messages on this port.

```
new Portu,
      Luc0,Lut2,Lvc2,Lvt2.
...
```

The net daemon now lowers the security level in the field uc of its send label to STAR.

```
next LSucSTAR(x),!LSuc1(x) :-
      NETd(x),LSuc1(x),Portu(y).
...
```

Next, the net daemon lowers the security level in the relevant taint propagation field ut of its send label to STAR, and becomes ready for communication.

```
next LSutSTAR(x),!LSut1(x),Ready(x) :-
      NETd(x),LSut1(x),LSucSTAR(x).
...
```

Eventually, the net daemon can raise the security level in the field ut of its receive label to 3, to receive tainted data from u. It can similarly raise the security level in the field ut of the port's label, to allow it to carry tainted data back to u.

```
next LRut3(x),!LRut2(x) :-
      NETd(x),LRut2(x),LSutSTAR(x).

next Lut3(x),!Lut2(x) :-
      Portu(x),Lut2(x),NETd(y),LucSTAR(y).
...
```

Further, the net daemon can spawn a new worker process for u.

```
new Wu,
      LSuc1,LSut1,LSvc1,LSvt1,
      LRuc2,LRut2,LRvc2,LRvt2.
...
```

The security levels in the fields uc and ut of the worker process are lowered and raised to STAR and 3, respectively, before the worker process becomes ready for communication.

```
next LSucSTAR(x),LSut3(x),!LSuc1(x),!LSut1(x),
   Ready(x) :-
      Wu(x),LSuc1(x),LSut1(x),
      NETd(y),LSucSTAR(y),LSutSTAR(y).
...
```

Elsewhere, the database proxy creates the following processes and ports for receiving and sending records for u.

```
new DBproxyRu,
       LSuc1,LSutSTAR,LSvc1,LSvtSTAR,
       LRuc2,LRut3,LRvc2,LRvt3.

new PortDBu,Luc2,Lut3,Lvc2,Lvt2.

new DBproxySu,
       LSuc1,LSut3,LSvc1,LSvt3,
       LRuc2,LRut3,LRvc2,LRvt3.
...
```

Unrestricted ports can also be created.

```
new PortAny,Luc3,Lut3,Lvc3,Lvt3.
```

We model all valid communication links between the above processes, following the implementation described in [11]. Specifically, let $\texttt{Send}(x, z)$ denote that process x may send a message to process z. This condition is constrained by the auxiliary conditions $\texttt{Link}(x, y, z)$ and $\texttt{Comm}(x, y, z)$ for some port y, as follows. $\texttt{Link}(x, y, z)$ requires that x and z are ready for communication, and y is actually available for communication between x and z (see below). $\texttt{Comm}(x, y, z)$ is an encoding of the requirement $x_S \sqsubseteq z_R \sqcap y_L$ for communication, as described in the beginning of Section 5.2; the rules are generated automatically by scripts. Note that some of the communication links that we model below turn out to be redundant, because of taint propagation. (Indeed, some links allow communication that is dangerous for secrecy.)

```
Link(x,y,z) :-
   Useru(x),PortAny(y),Wu(z),Ready(z).
...
Link(x,y,z) :-
   Wu(x),Ready(x),Portu(y),Ready(z).
...
Link(x,y,z) :-
   Wu(x),Ready(x),PortAny(y),Wv(z),Ready(z).
...

Link(x,y,z) :-
   NETd(x),Ready(x),PortAny(y),Ready(z).

Link(x,y,z) :-
   Ready(x),PortDBu(y),DBproxyRu(z).
Link(x,y,z) :-
   DBproxyRu(x),PortAny(y),DBproxySu(z).
Link(x,y,z) :-
   DBproxySu(x),PortAny(y),Wu(z),Ready(z).
Link(x,y,z) :-
   DBproxySu(x),PortAny(y),Wv(z),Ready(z).
...

Send(x,y,z) :- Link(x,y,z), Comm(x,y,z).
```

Finally, we model the effects of communication. Specifically, the clauses below encode the effects of sending a message from process x to process z, as described in the beginning of Section 5.2: the label z_S is transformed to $z_S \sqcup (x_S \sqcap z_S^\star)$. For any field ℓ, the security level $z_S.\ell$ does not need to be raised if $\min(z_S^\star.\ell, x_S.\ell) \leq z_S.\ell$, that is, if $z_S.\ell = \star$ or $x_S.\ell \leq z_S.\ell$. This condition is denoted by $\texttt{LeqSTAR}\ell(x, z)$. Further, the relation Mu is augmented on such communication. (The rules are generated automatically by scripts.)

```
next Mu(z) :-
             Send(x,z),Mu(x),
             LeqSTARut(x,z),LeqSTARvt(x,z).
             LeqSTARuc(x,z),LeqSTARvc(x,z).
next Mu(z),
     LSvt3(z),!LSvt1(z) :-
             Send(x,z),Mu(x),
             LeqSTARut(x,z),LSvt1(z),LSvt3(x).
             LeqSTARuc(x,z),LeqSTARvc(x,z).
next Mu(z),
     LSut3(z),!LSut1(z) :-
             Send(x,z),Mu(x),
             LSut1(z),LSut3(x),LeqSTARvt(x,z).
             LeqSTARuc(x,z),LeqSTARvc(x,z).
next Mu(z),
     LSvt3(z),!LSvt1(z),LSut3(z),!LSut1(z) :-
             Send(x,z),Mu(x),
             LSut1(z),LSut3(x),LSvt1(z),LSvt3(x).
             LeqSTARuc(x,z),LeqSTARvc(x,z).
...
```

We now ask the query SecrecyViolation, which denotes the existence of a process x that runs on behalf of v, *i.e.*, $\texttt{Userv}(x)$ or $\texttt{Wv}(x)$, but carries u's data, *i.e.*, $\texttt{Mu}(x)$.

```
SecrecyViolation :- Userv(x),Mu(x).
SecrecyViolation :- Wv(x),Mu(x).

? SecrecyViolation.
```

EON does not find any exploits for this query. In other words, we have the following theorem, automatically proved by EON.

THEOREM 5 (DATA ISOLATION). *A user* u*'s data is never leaked to any process running on behalf of a different user* v.

We conclude by mentioning some statistics that indicate the scale of this experiment. The whole specification of the webserver is around 250 lines of EON. The translated Datalog program contains 152 recursive clauses over a 46-ary Reachable relation (that is, over 46-bit atomic states). Our query takes around 90 minutes to evaluate on a Pentium IV 2.8GHz machine with 2 GB memory—in contrast, the queries for the other examples take less than a second.

Scripts for all the examples in this section are available in [7].

6. RELATED WORK

It is well-known that the "safety" problem for access control models (*i.e.*, whether a given access is allowed by a given access control model) is undecidable in general [13, 9]. Nevertheless, there are restricted classes of access control models for which this problem is decidable. Our work may be viewed as a step towards identifying such classes of models: we design an expressive language for dynamic access control systems, in which information-flow properties remain decidable.

Analyzing access control models with logic programs has a fairly long history. We focus here only on more closely related work. Recently Dougherty *et al.* [10] propose a technique to study the security properties of access control policies under dynamic environments. There, a policy is specified in a fragment of Datalog without negation and recursion, and an environment is specified as a finite state machine. The composition of the policy and the environment is then analyzed by reduction to first-order logic formulae. While the authors identify some decidable problems in this framework, the lack of recursion and negation limits the expressivity of both models and queries, and it is not always possible to specify accurate finite state machines for environments. Indeed, none of the dynamic access control models studied in this paper can be analyzed in their framework.

Sarna-Starosta and Stoller [18] study the Security-Enhanced Linux (SELinux) system in Prolog. The SELinux system enforces access control policies written in SELinux's policy language. The authors describe a tool called PAL that translates such policies into logic programs, and analyzes them by query evaluation. Prasad *et al.* [16] study both SELinux and Windows XP configurations in Datalog in a tool called Netra. Unlike PAL, Netra is both sound and complete, since query evaluation is decidable in Datalog (while in Prolog is not). However, neither tool can find vulnerabilities that are exploited dynamically. Some of these concerns are addressed by Stoller *et al.*'s recent work on policy analysis for administrative role-based access control [19], which is similar in spirit to ours.

More recently, Becker *et al.* [2] propose a language called SecPAL that can express authorization policies and fine-grained delegation control in decentralized systems. Their specifications are compiled down to programs in Datalog, much as in our work. Since Datalog is a subset of EON, it follows that EON is at least as expressive as SecPAL. On the other hand, it is not clear whether Sec-

PAL is as expressive as EON; the former is tailored to express authorization and delegation policies, while the latter remains largely agnostic in that respect. An interesting aspect of SecPAL is that it allows negations within queries. While EON allows such negations, the fragment discussed in Section 4 does not. However, we have checked that this restriction can be lifted from that fragment without compromising correctness or efficiency.

Other relevant work includes Blanchet's ProVerif [5], which is a powerful tool that can analyze security protocols written in the applied pi-calculus. The underlying engine rewrites the protocols and associated equational theories into Prolog-like rules, and uses customized resolution procedures to answer queries about secrecy and authenticity properties. ProVerif is sound but not complete; it may not terminate on queries, and it may also fail to prove or disprove queries. Indeed, while ProVerif can handle Windows Vista's access control model, it does not terminate on our model of Asbestos's webserver. In sum, EON is less expressive than ProVerif; but for models that satisfy our restrictions, EON guarantees sound and complete results.

Finally, we are not the first to propose a dynamic language based on Datalog. Related languages have been studied, for instance, by Abadi and Manna [1] and Orgun [17]. However, we seem to be the first to introduce a new operator to Datalog, and show that it can be reduced to existential quantification in Datalog. Such an operator allows us to express specifications that quantify over an unbounded number of processes and objects.

7. CONCLUSIONS

In this paper, we present EON, a logic-programming language and tool that can be used to model and analyze dynamic access control systems. Security violations can be modeled as temporal queries in this language, and query evaluation can be used to find attacks. We show that query evaluation in EON can be reduced to decidable query satisfiability in a fragment of Datalog, and under further restrictions, to efficient query evaluation in Datalog.

Our design of EON requires much care to keep query evaluation decidable. In particular, we require that any base relation that is introduced or transformed be unary—allowing dynamic binary base relations easily leads to undecidability [7]. Moreover, we require transitions to have monotonic guards, and queries to be monotonic.

These restrictions do not prevent us from modeling state-of-the-art access control models, such as those implemented by Windows Vista and Asbestos. With unary base relations and new clauses, we can create and label processes and objects. Further, with next clauses, we can model runtime effects such as dynamic access control, communication, and taint propagation. Thus, EON turns out to be a good fit for modeling dynamic access control systems.

Further, we demonstrate that EON can verify various security properties of interest. Since our query evaluation strategy is both sound and complete, EON either finds bugs or decisively proves the absence of bugs. We expect that there are other classes of systems that can be modeled and analyzed using this approach.

8. REFERENCES

[1] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computing*, 8(3):277–295, 1989.

[2] M. Becker, C. Fournet, and A. Gordon. Design and semantics of a decentralized authorization language. In *CSF'07: Computer Security Foundations Symposium*. IEEE, 2007.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp., 1975.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp., 1977.

[5] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW'01: Computer Security Foundations Workshop*, page 82. IEEE, 2001.

[6] A. Chaudhuri, P. Naldurg, and S. Rajamani. A type system for data-flow integrity on Windows Vista. In *PLAS'08: Programming Languages and Analysis for Security*, pages 89–100. ACM, 2008.

[7] A. Chaudhuri, P. Naldurg, S. Rajamani, G. Ramalingam, and L. Velaga. EON: Modeling and analyzing dynamic access control systems with logic programs. Technical Report MSR-TR-2008-21, Microsoft Research, 2008. See `http://www.soe.ucsc.edu/~avik/projects/EON/`.

[8] M. Conover. Analysis of the windows vista security model. Symantec Report. Available at `www.symantec.com/avcenter/reference/Windows_Vista_Security_Model_Analysis.pdf`.

[9] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[10] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR'06: International Joint Conference on Automated Reasoning*, 2006.

[11] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP'05: Symposium on Operating Systems Principles*, pages 17–30. ACM, 2005.

[12] A. Y. Halevy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Static analysis in datalog extensions. *Journal of the ACM*, 48(5):971–1012, 2001.

[13] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. On protection in operating systems. In *SOSP'75: Symposium on Operating systems Principles*, pages 14–24. ACM, 1975.

[14] B. W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, Jan 1974.

[15] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, J. Turner, and J. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. Technical report, NSA, 1995.

[16] P. Naldurg, S. Schwoon, S. Rajamani, and J. Lambert. Netra: seeing through access control. In *FMSE'06: Formal Methods in Security Engineering*, pages 55–66. ACM, 2006.

[17] M. A. Orgun. On temporal deductive databases. *Computational Intelligence*, 12:235–259, 1996.

[18] B. Sarna-Starosta and S. D. Stoller. Policy analysis for security-enhanced linux. In *WITS'04: Workshop on Issues in the Theory of Security*, 2004. Available at `http://www.cs.sunysb.edu/~stoller/WITS2004.html`.

[19] S. D. Stoller, P. Yang, C. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS'07: Conference on Computer and Communications Security*. ACM, 2007.

[20] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Volume II: The New Technologies*. Computer Science Press, New York, 1989.

[21] S. Zdancewic and A. C. Myers. Robust declassification. In *CSFW'01: Computer Security Foundations Workshop*, pages 5–16. IEEE, 2001.