SEAL: A Logic Programming Framework for Specifying and Verifying Access Control Models

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

Raghavendra K. R.
Indian Institute of Science
Bangalore, India
raghavendra.kaundinya@gmail.com

ABSTRACT

We present SEAL, a language for specification and analysis of safety properties for label-based access control systems. A SEAL program represents a possibly infinite-state non-deterministic transition system describing the dynamic behavior of entities and their relevant access control operations. The features of our language are derived directly from the need to model new access control features arising from state-of-the art models in Windows 7, Asbestos, HiStar and others. We show that the reachability problem for this class of models is undecidable even for simple SEAL programs, but a bounded model-checking algorithm is able to validate interesting properties and discover relevant attacks.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*access controls, verification*

General Terms

Security, Verification

Keywords

access control, label-based access, Windows 7, attacks, bounded model checking, logic programs

1. INTRODUCTION

The question of safety in access control was first studied in the 70s in the context of the HRU and Graham-Denning [13, 11] models, based on the access-control matrix abstraction due to Lampson [16]. The general safety question in this context, which formalizes the notion of authorized access was shown to be undecidable.

In practice however, there are restricted models for which safety is decidable [15, 5], and it can be shown that correct enforcement of authorized access requests preserves safety, using a reference monitor (RM) which mediates all accesses.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SACMAT'11, June 15–17, 2011, Innsbruck, Austria.
Copyright 2011 ACM 978-1-4503-0688-1/11/06 ...$10.00.

In most commercial and open-source operating systems, such as Windows 7 and different flavors of Linux, a modified discretionary access-control (DAC) model is implemented using this idea. The concept of ownership is used to define authorized access, which determines the ability to change permissions to resources. RMs mediate all requests from processes to resources and control access based on the instantaneous values in a virtual access matrix that captures the ownership relation. This automatically enforces safety, even though general safety properties require history for correctness. Since ownership implies authorization in these models, this notion is frequently exploited by attackers (e.g., using buffer overflow attacks to get "root-user" access), thereby making safety guarantees provided by enforcement using RMs meaningless.

In a bid to work around this weakness, in recent years, there is a growing interest in applying what we call label-based access control (LBAC) models to provide stronger confidentiality and integrity guarantees. Windows Vista and Windows 7 are the first commercial operating systems that use integrity labels to minimize the damage that can be caused by a compromised process running on behalf of an authenticated user. Windows' LBAC is called UAC (User Account Control). In UAC, applications can run only with low integrity levels by default, and cannot access trusted resources which are tagged with higher-integrity labels when compromised. Other examples of LBAC include SELinux[14] and IFEDAC[18], as well as Asbestos and Hi-Star [21, 22]. In all these models, customizable confidentiality or integrity labels are used to taint processes and resources that access sensitive data, thereby preventing them from being accessed or modified by processes with lower labels. Ownership-based discretionary access control is also allowed in these models.

LBAC models are inspired by the pioneering works on Multi-level Security (MLS) systems, exemplified by the Bell-LaPadula [3] and Biba [4] models, but differ from them in one crucial aspect. In traditional MLS models, labels assigned to processes and resources are immutable (fixed), and strong safety properties can be enforced by RMs that only need to compare the labels of processes and resources on access. A security lattice [12] of labels is defined, which imposes a partial-order on how information is allowed to flow between processes and resources, e.g., by disallowing write-ups in the lattice for integrity protection and preventing read-ups for confidentiality. However, immutable labels are not very useful in commercial operating systems [17]. Controlled and selective downgrading or upgrading of labels is required, to satisfy everyday information flow requirements,

such as installing web applications. The safety question in this context is whether a high process can access a low resource (confidentiality violation), or whether a low process can write to a high resource (integrity violation). Once the labels are allowed to change, safety cannot be enforced by instantaneous lookup using an RM, without maintaining auxiliary information about the value of the label. History, e.g., as taint information, needs to be stored along with a process or resource's current label.

We present SEAL, a language for modeling and analyzing safety properties in LBAC systems. Using SEAL we can model the system state as the set of relevant access control relations. Transitions are events that change labels and create new associations or entries in these relations. A SEAL program induces a possibly infinite state non-deterministic transition system over these sets of relations. The safety problem (or really its negation) can be viewed as a reachability property in this system. We show that analyzing this property is undecidable even in a very restricted fragment of SEAL. This is in contrast to other models of access control, where even though the general case turns out to undecidable, we admit decidable fragments that are expressive enough for practical systems.

The syntax of SEAL resembles Datalog [6] or logic programming closely. In terms of semantics however, typical Datalog programs are interpreted over a given (closed) set of relations. SEAL programs, on the other hand, can be thought of as specifying dynamic state-transition systems on sets of relations.

Access control problems have been specified using logic programs in the past [1, 2, 19]. In these works, the mechanism of access control is expressed using an appropriate language, i.e., constraint logic programming (CLP) or safe stratified Datalog respectively. Given an access request, and a database instance, logic specialization is used to answer the query correctly and efficiently. The declarative aspect of these programs also adds flexibility in terms of allowing one to change access control mechanisms with little overhead. Given the state of the system and an access query, one can use these frameworks to tell whether the access is authorized or not.

In contrast, we examine the problem of verifying if a dynamic access control model admits unsafe behavior. More precisely, a SEAL LBAC specification is declared secure when no unauthorized accesses are possible, assuming that the system starts from a valid state and evolves adhering to the behavior given in the specification. The difference between the two approaches can be expressed as examining the satisfiability aspect of safety in abstract models, in contrast to verifying the validity of queries in a model instance.

SEAL is related to EON [8], a language for expressing dynamic access control systems where the *base relations* of an EON program are unary. This restriction is not natural in generalized LBAC systems, since it does not capture concepts such as binding resources to their named entities (e.g., link files), or in expressing the exact semantics of label associations. We show that extending EON with even one binary predicate makes the query reachability question undecidable.

For verification in SEAL, we propose an algorithm that systematically explores all possible states in the model, for a given depth, as in bounded model checking [10, 9], and validates if the property can be proved in the expansion. If a counter-example is found, a finite state representation of this attack is automatically generated, and a corresponding RM can be built to track the history of states and transitions to enforce the property. The monitor will track changes to the state of the system and warn users before an unsafe state is imminent. If the property cannot be disproved in the bounded model, no safety guarantees can be asserted in general. However, bounded depth guarantees from the model can still carry over in natural use-cases in implementations appropriately.

We show how SEAL can be used to model state-of-the-art LBAC systems and models, including Windows 7, HiStar, and Asbestos. We present examples of vulnerabilities, including the UAC prompt elevation in the start menu, as well as the silent elevation-list vulnerability that were discovered by our bounded state-space exploration. These examples cannot be modeled by EON. For Asbestos, we show the absence of secrecy violations in bounded contexts for general safety properties. Finally we also discuss some limitations of SEAL, especially in the context of IFEDAC [18].

The rest of the paper is organized as follows: In Section 2 we introduce SEAL and present its syntax and operational semantics formally, and explain it informally with a relevant example. Section 3 explores the Windows 7 LBAC model and motivates the features of SEAL. Section 4 examines the query reachability problem in SEAL, and includes a brief description of our bounded state-space exploration algorithm. This is followed by our case studies in Section 5, where we highlight vulnerabilities found in Windows 7 as well as discuss its applicability to Asbestos and IFEDAC. Finally, we summarize our work in Section 6 and conclude with some pointers to future work.

2. SEAL LANGUAGE

In this section, we present the formal syntax and semantics of our logic programming language SEAL. SEAL uses a relational model to formulate the access control problem, i.e., every state of the modeled access control system is viewed as a set of relevant relations that are used to define the mechanism of access. To illustrate, the state $\{Process(a), File(b), Own(b, a)\}$, represents a model with one process a, and one file b whose owner is a. Note that at any instance (or snapshot), the relations have only a finite number of tuples.

2.1 SEAL Syntax

A SEAL program consists of three sections: a static part, a dynamic part and queries. The static part of a SEAL program encodes how the access relations can be constructed (or derived) from the base relations, capturing dependencies among the base relations appropriately. The dynamic part is the heart of SEAL and consists of a list of customized rules that specify how the base relations can be updated, under what conditions (if any).

The static part of a SEAL program P is identical in syntax to Datalog and denoted by $\widehat{P}$. We first give a brief description of the syntax of Datalog. For a full description of Datalog refer to [6]. A Datalog rule is of the form: $L_0 :- L_1, L_2, \ldots, L_n$, where L_i is a predicate with parameters. The parameters can either be constants (strings) or variables. L_0 is called the head of the rule. The head of the rule should not have constants as parameters. Datalog does not allow complex terms (such as functors) as arguments to

these predicates, and imposes certain stratification restrictions on the use of negation and recursion i.e., the variable appearing in the negated predicate should also appear in some positive predicate in the body of the same rule. Further each variable in the head of a rule must also appear in some positive clause in the body of the rule. There are two types of predicates: base and derived. Base predicates occur only in the body of the rules, and derived predicates occur in at least one rule as head.

The syntax of the dynamic rules in SEAL is as follows:

$$\begin{aligned}&\textsf{anext } \mathcal{B}(x_1,\ldots,x_m),\mathcal{B}'(y_1,\ldots,y_n) \ :-\\ &\qquad \mathcal{R}(u_1,\ldots,u_k),\mathcal{R}'(v_1,\ldots,v_l).\\ &\textsf{enext } \mathcal{B}(x_1,\ldots,x_m),\mathcal{B}'(y_1,\ldots,y_n) \ :-\\ &\qquad \mathcal{R}(u_1,\ldots,u_k),\mathcal{R}'(v_1,\ldots,v_l).\end{aligned}$$

In this rule, $\mathcal{R}(u_1,\ldots,u_k)$ denotes the conjunction of positive predicates with parameters from the variables $u_1,\ldots,u_k$, such that every $u_i, 1 \leq i \leq k$ occurs in some predicate. $\mathcal{R}'(v_1,\ldots,v_l)$ denotes the conjunction of negative predicates with parameters from the variables $v_1,\ldots,v_l$, and every $v_i, 1 \leq i \leq l$ also occurs in some positive predicate. Similarly, $\mathcal{B}(x_1,\ldots,x_m)$ denotes the conjunction of positive base predicates and $\mathcal{B}'(y_1,\ldots,y_n)$ denotes the conjunction of negative base predicates. All the variables of $\mathcal{R}'$ and $\mathcal{B}'$ occur in $\mathcal{R}$. Though the syntax of enext and anext are similar, the semantics is a little different. We explain this in detail in Section 2.2.

The dynamic rules can be normalized by restricting each rule to have only one positive guard (right-hand side) predicate, as it is equivalent to the above form. If the guard is not a single positive predicate, it has to be a conjunction of positive and negative predicates. This can be replaced with a single fresh predicate (containing all the variables appearing in the earlier guard predicates), without loss of generality. The original rule can now be replaced with a Datalog rule with the fresh predicate as the head and the earlier guard predicates as the body. Thus, a simplified (but equivalent) dynamic rule would look like:

$$\begin{aligned}&\textsf{anext} \quad \mathcal{B}(x_1,\ldots,x_m),\mathcal{B}'(y_1,\ldots,y_n) \ :-R(u_1,\ldots,u_k)\\ &\textsf{enext} \quad \mathcal{B}(x_1,\ldots,x_m),\mathcal{B}'(y_1,\ldots,y_n) \ :-R(u_1,\ldots,u_k)\end{aligned}$$

In these rules, all the variables of $\mathcal{B}'$ occur in R.

We allow two kinds of queries: simple and temporal. A simple query is written as $Q(x_1,\ldots,x_n)?$ and a temporal query as $Q_1(x_1,\ldots,x_m);Q_2(y_1,\ldots,y_n)?$. We disallow duplication of variables in queries in a given SEAL program. Note that if duplication of variables is needed, one can introduce a new Datalog rule with a fresh query predicate. Hence it is enough to consider just the name of the query predicate (along with its arity) and not the argument variables. In the rest of the paper, we assume that the all the dynamic rules have a single positive predicate as the guard without duplication of variables.

2.2 SEAL Semantics

Given an initial set of base predicates I, a SEAL program P induces a transition system $M_P = (Q, \Sigma, \longrightarrow, s_0)$ where Q is a (possibly infinite) set of states, Σ is a set of dynamic rules in the program, the transition relation is given by $\longrightarrow \subseteq Q \times \Sigma \times Q$, and $s_0 \in Q$ is the starting state constructed from I. A state is a set (or database) of relations. Note that this transition system may be non-deterministic. As mentioned earlier, we use $\widehat{P}(I)$ to denote the standard Datalog semantics for the Datalog portion of the SEAL program P against a given set of base predicates I. In other words, the Datalog rules of P are applied iteratively (and cumulatively) on the predicates of I and derived predicates are populated until they reach a fix-point. The conditions on the Datalog rules ensure the existence of a least fix-point [6]. We use s and t to denote such saturated sets of predicates. These saturated sets form the basic states in the induced state-transition system. The starting state $s_0 = \widehat{P}(I)$. We use $bp(s)$ and $bp(t)$ to denote only the set of base predicates in states s and t.

We now describe the semantics of the dynamic rules. Let $\alpha = \textsf{anext } \mathcal{B}(x_1,\ldots,x_m),\mathcal{B}'(y_1,\ldots,y_n) \ :-R(u_1,\ldots,u_k)$ in a SEAL program P, where all the variables of $\mathcal{B}'$ occur in R. Then we have a transition $s \xrightarrow{\alpha} s'$ where $s' = \widehat{P}(bp(s) \cup genan(\alpha, s) \setminus killan(\alpha, s))$ if the predicate R (with arity k) is non-empty in s, otherwise the a-transition is not enabled at s. The sets *genan* and *killan* are defined as:

$$\begin{array}{|l|}\hline genan(\alpha,s) = \{B(a_1,\ldots,a_r) \mid \\ B \in \mathcal{B} \wedge R(c_1,\ldots,c_k) \in s\} \\ \text{where, for every } 1 \leq i \leq r, \\ a_i = \begin{cases} c_j & \text{if } x_i = u_j 1 \leq j \leq k \\ \text{a fresh constant} & \text{otherwise} \end{cases} \\ \hline killan(\alpha,s) = \{B'(b_1,\ldots,b_t) \mid \\ B' \in \mathcal{B}' \wedge R(c_1,\ldots,c_k) \in s\} \\ \text{where, for every } 1 \leq i \leq t, \ b_i = c_j \\ \text{such that } y_i = u_j \text{ for some } 1 \leq j \leq k. \\ \hline \end{array}$$

Similarly, when $\alpha = \textsf{enext } \mathcal{B}(x_1,\ldots,x_m),\mathcal{B}'(y_1,\ldots,y_n) \ :-R(u_1,\ldots,u_k)$ where all the variables of $\mathcal{B}'$ occur in R, we have a transition $s \xrightarrow{\alpha} s'$ where $s' = \widehat{P}(bp(s) \cup genen(\alpha, s) \setminus killen(\alpha, s))$ if the predicate R with arity k is non-empty in s, otherwise the α-transition is not enabled at s. When the α-transition is enabled at s, pick one tuple $R(c_1,\ldots,c_k)$ from s and update the state as defined by *genen* and *killen* as follows:

$$\begin{array}{|l|}\hline genen(\alpha,s) = \{B(a_1,\ldots,a_r) \mid B \in \mathcal{B}\} \\ \text{where, for every } 1 \leq i \leq r, \\ a_i = \begin{cases} c_j & \text{if } x_i = u_j 1 \leq j \leq k \\ \text{a fresh constant} & \text{otherwise} \end{cases} \\ \hline killen(\alpha,s) = \{B'(b_1,\ldots,b_t) \mid B' \in \mathcal{B}'\} \\ \text{where, for every } 1 \leq i \leq t, \ b_i = c_j, \\ \text{such that } y_i = u_j \text{ for some } 1 \leq j \leq k. \\ \hline \end{array}$$

Note that in case of anext we consider all the tuples in R from s and in enext we non-deterministically pick one tuple of R in s. The semantics of enext is the source of non-determinism in the induced automaton. If there are multiple elements satisfying the guard of an enext rule e at a state s, the induced transition system has many transitions on s with the same label e accounting for every selection of the satisfying guard predicate in s. A dynamic rule without a guard is equivalent to having a zero-arity (constant) guard predicate true.

We now describe the semantics of SEAL query evaluation. A simple query $Q(x_1,\ldots,x_n)?$ holds from a set of a basic predicates I w.r.t. a SEAL program P if there exists a state s' and a sequence of dynamic rules $w = \alpha_1\alpha_2\ldots\alpha_m$ of P such that $\widehat{P}(I) \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} s'$ and the predicate Q (with arity n) is non-empty in s'. A temporal query

of the form $Q_1(x_1, \ldots, x_m)$; $Q_2(y_1, \ldots, y_n)$? holds from a state s w.r.t. a SEAL program P if there exist states s', s'' and two sequences of dynamic rules $u = \alpha_1\alpha_2 \ldots \alpha_k, v = \beta_1\beta_2 \ldots \beta_l$ of P such that $\widehat{P}(I) \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} s' \xrightarrow{\beta_1} t_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_l} s''$, the predicate Q_1 (with arity m) is non-empty in s' and Q_2 (with arity n) is non-empty in s''.

Temporal queries can equivalently be written as simple queries. Let $Q_1(x_1, \ldots, x_m)$; $Q_2(y_1, \ldots, y_n)$ be a temporal query. This query may be replaced with a new simple query $Q(x_1, \ldots, x_m, y_1 \ldots, y_n)$, a dynamic rule

$$\texttt{enext}\ \mathit{Done}(x_1, \ldots, x_m) :\!-Q_1(x_1, \ldots, x_m)$$

where *Done* is a fresh predicate, and a Datalog rule

$$Q(x_1, \ldots, x_m, y_1, \ldots, y_n) :\!-Q_2(y_1, \ldots, y_n), \mathit{Done}(x_1, \ldots, x_m)$$

We now state the problem of query-reachability formally. Given a snapshot of the system as a set of base relations I, a simple query predicate Q and a SEAL program P, does Q hold from $\widehat{P}(I)$ w.r.t. P. We also want to compute all such paths.

2.3 Example

We present a simple program written in SEAL, and explain our syntax and semantics informally with this example. Consider the following program that models the behavior of a user presented with a UAC prompt associated with a label change in Windows 7:

```
1. enext LowFile(x).
2. anext LinksTo(x,y):- LowFile(y), StdHighName(x).
3. AlwaysConsent(x) :- StdHighName(x).
4. StdHighName("regedit").
5. LinksTo(x,y),LowFile(y); AlwaysConsent(x)?
```

The first statement in our SEAL program is an `enext` rule, which specifies that a new low file (file with integrity label low) can be created by the user, or by a program running as low on behalf of the user at any point. Note that `anext` would not have made any difference here. The second statement is a guarded `anext` statement which specifies that we can create a link with a standard high name (say regedit, the registry editor) to a low file (a virus) and put it on the desktop. The third line is a regular Datalog rule that states that anything with standard high name always causes the user to accept prompt. The fourth statement is a database entry (or a database "fact"). The last statement is a query that says if we have a link to a low file on the desktop, can the user be fooled into giving consent?

This program induces a infinite state-transition system as shown in the Figure 1. We start with the initial state (say state 1) where `regedit` is the only entry in the `StdHighName` and `AlwaysConsent` relations, consistent with standard Datalog semantics. The first `enext` rule will cause the database to transition to a state where a new constant is added to the `Low` relation (state 2). Once this is added, no other Datalog rule can fire and this is the updated state. Now, the guard to the second `anext` rule is satisfied and a transition to a new state can occur, where the `LinksTo` relation can be updated appropriately. Note that the first `enext` rule is also enabled in state 2, and this transition creates another constant in the `Low` relation, and so on. Since there is only one constant in the `StdHighName` relation, we will end up eventually describing a world where there are many low links on the desktop to the high resource. It is easy to see that the query, about whether it is possible for a link to a low file (say a virus) to fool the user into accepting a UAC prompt(`AlwaysConsent`), is true in this model. We have intentionally kept this example simple, and hence a little contrived, attributing to user behavior semantics that is outside the scope of the specification given in the example.

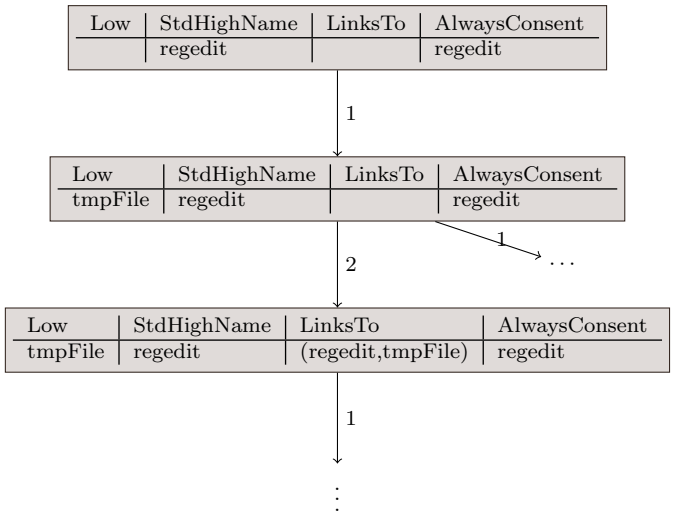

Figure 1: Induced transition system for the example

3. MODELING IN SEAL

In this section, we call out specific requirements in state-of-the art access control systems that motivate our choice of features for SEAL. In particular, we focus on the LBAC component of Windows 7, which is called User Account Control (UAC). With the help of examples we show how existing formalisms (specifically EON) cannot capture the required semantics adequately.

The intuition behind UAC is to enable authorized users (typically PC owners) to log in as standard users and perform common tasks that do not require administrative privileges. The idea is when a user process is compromised (say by a buffer-overflow attack) and control hijacked, the attacker will have no more privileges than the default (low) privileges associated with standard user accounts. Therefore, when UAC is enabled, even a local administrator will run as a standard user account with reduced privileges. This is the case until (s)he attempts to run an application or task that has an administrative token (i.e., requires special privileges). When such a member of the local administrators group attempts to start a privileged application or task, they are prompted to consent to running the application as elevated.

UAC is implemented using a lattice of integrity labels. The integrity labels for each process (or thread) and resource on the Windows 7 installation are stored in its security descriptor. Windows 7 has four integrity labels (a total order): System, High, Medium and Low. By default, processes (and threads) on behalf of standard user accounts

run as Medium and all resources created by these accounts can only be Medium or lower. Some applications however, such as Internet Explorer (IE), run with default low rights. Any resource downloaded from the Internet (say) is automatically assigned a Low label, and (explicit) consent from the user (who is Medium), is required by clicking on the UAC prompt in case of elevation. By default, for integrity protection, restrictions are placed on explicit information flows from lower levels to higher levels in the usual way. Write-up by a lower process to a higher resource is not allowed, and a read-down from a higher process to a lower user is not allowed, unless explicitly authorized by the user in response to a UAC prompt. Explicit authorization is allowed in UAC for functionality reasons. If a user is reasonably certain e.g., that they trust the source of the plugin (by checking the cryptographic hash of the binary with a trusted provider), the labels need to be upgraded naturally. Within IE it is possible to download and install plugins that need higher privileges. A separate process or thread is run on behalf of an existing process that requests additional privileges, and explicit user consent is required to make these label changes.

The formal model of UAC was presented by Chaudhuri et al in EON [7]. We present this model in SEAL, restricting our relations to unary base predicates to mimic the syntax of EON. This also sets the context to explain our extensions to this model in the next subsection.

The unary base relations used in the model example have the following informal meanings: `P` is a relation representing processes; `Obj` represents objects (including processes); and Low, Med, High, *etc.* represent processes and objects with those integrity labels (ILs).

```
enext Obj(x),Low(x).
enext Obj(x),Med(x).
enext Obj(x),High(x).

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

Guarded `enext` rules specify how ILs of processes and objects can be changed. For instance, a Medium process can raise the IL of an object from Low to Medium if that object is not a process; it can also lower the IL of an object from Medium to Low. A High process can lower its own IL to Medium (*e.g.*, to execute a Medium object), consistent with the safe upgrading and downgrading in the context of integrity properties.

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

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

Datalog rules 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).
...
```

Several interesting safety queries can be studied in this model. For instance, can a Medium object be read by a Medium 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 Medium?

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

When these queries were executed in EON, which has a sound and complete decision procedure, in addition to the obvious vulnerabilities that are introduced by explicit user consent, vulnerabilities were discovered that highlight the need to maintain the history of label-state of a process or a resource as the system evolves over time. This information is not explicitly available to the standard user when a UAC prompt is issued, and the attacks such as the one shown in our running-example can exist, due to additional native resource-access semantics that are not accounted for in the model.

In the next sub-section, we show how the EON language used to model UAC is not sufficient to analyze scenarios when UAC actions are composed with Windows native access semantics. While the results of modeling UAC with EON are useful, they do not specify interactions with components outside the direct scope of the access relations (such as file naming semantics etc.)

3.1 Windows 7 Folder Access

We motivate the need for a richer specification language, SEAL, by illustrating a behavior that captures the interaction between the UAC model and native Windows 7 folder-access semantics.

As mentioned earlier, Windows 7 attaches integrity labels: Low, Medium, High and System, to all resources and processes (threads) in an installation. While the UAC defines rules for safe upgrading and downgrading of integrity levels, we show with an example that the peculiar and no doubt useful semantics of the start Menu folder admits a behavior that cannot be modeled in EON.

In Windows, a user's start menu is populated with soft links (shortcuts to actual resources) from two separate folder locations. One is the user's local start menu folder and the other is a global folder, typically populated in enterprise networks with remotely sourced applications. These two locations are merged to create the start menu that the user sees. If the same shortcut exists in both the user's local folder and the global folder, upon access, the shortcut in the user's folder is given preference, and the application associated with the link is executed.

To model this behavior, we use the following base predicates to represent entities in our model:

`P(x)`	`x` is a process
`Low(x)`	`x` has a 'low' IL
`High(x)`	`x` has a 'high' IL
`Adm(x)`	`x` is an admin
`File(x)`	`x` is a file
`Link(x)`	`x` is a link
`Folder(x)`	`x` is a folder
`FreshName(x)`	`x` is a fresh name
`LinksTo(x,y)`	link name `x` points to resource y
`Name(x,y)`	Name of `x` appears as `y`
`InLocalFolder(x)`	`x` is in local start menu folder
`InGlobalFolder(x)`	`x` is in global folder

The modeling assumes that there are only two integrity levels: low and high (for simplicity), and that there is a high administrator process already running in the system.

The dynamic rules are from 1 to 7. The rules 1 and 2 model creation of a new low and a high process respectively.

```
1. enext P(y), Low(y) :- Adm(x).
2. enext P(y), High(y) :- Adm(x).
```

Rule 3 creates fresh names, which are used in naming the link files.

```
3. enext FreshName(x).
```

Rules 4 and 5 model the creation of a new low and high file respectively, where y is a fresh constant.

```
4. enext File(y), Low(y) :- P(x).
5. enext File(y), High(y):- P(x), High(x).
```

The rules 6 and 7 model the start menu semantics. Rule 6 creates a link in the global folder to a genuine executable which requires elevation. Rule 7 creates a link in the local folder to another executable with the same name as the link in the global folder, pointing to a low file.

```
6. enext LinksTo(y,z), High(y), Name(y, u),
   UsedName(u), !FreshName(u),
   InGlobalFolder(y) :-  P(x), High(x),
   File(z), High(z), FreshName(u).

7. enext LinksTo(y,z), Low(y), Name(y, u),
   InLocalFolder(y) :- P(x), File(z),
   UsedName(u).
```

The rest of the rules are pure Datalog rules, with easy-to-read meanings.

```
SameName(x,y) :- Name(x,z), Name(y,z).

StartMenu(x) :- SameName(x,y),
  InLocalFolder(x), InGlobalFolder(y).

StartMenu(x) :- InLocalFolder(x),
                !InGlobalFolder(x).
StartMenu(x) :- InGlobalFolder(x),
                !SameName(x,y).
AddPrivilege(x) :- StartMenu(x).
```

Below is the query modeling a safety property. The state satisfying this query represents the vulnerable state of the system.

```
Low(x); AddPrivilege(x)?
```

Note that both `LinksTo` and `SameName` are binary predicates and cannot be represented as multiple unary predicates in EON, as they both range over infinite domains. In Section 5, we explain how we can analyze this specification and examine if it admits unsafe states.

4. QUERY REACHABILITY IN SEAL

While the specification shown in Section 3.1 captures the semantics of native Windows 7 file access, we show that the existence of even one binary predicate in a SEAL program makes the query reachability problem undecidable. We show this with a reduction from Hilbert's tenth problem. This result adds further restrictions to the proof presented by Chaudhuri et al. [8] where they showed undecidability using two binary predicates.

THEOREM 1. *The query reachability problem for SEAL with one binary base predicate is undecidable.*

PROOF. We prove that the simple query reachability problem for SEAL with just one binary base predicate is undecidable by reducing Hilbert's tenth problem.

Hilbert's Tenth Problem. Given a diophantine equation (n degree polynomial with m unknowns: $x_1 \dots x_m$ and integer coefficients: $p_{11} \dots p_{mn}, p$) of the form

$$\begin{aligned} &p_{11}x_1^1 + p_{12}x_1^2 + \cdots + p_{1n}x_1^n + \\ &p_{21}x_2^1 + p_{22}x_2^2 + \cdots + p_{2n}x_2^n + \\ &\vdots \\ &p_{m1}x_m^1 + p_{m2}x_m^2 + \cdots + p_{mn}x_m^n + p = 0 \end{aligned}$$

does there exist natural number solution (zero included) for the unknowns? This problem is known to be undecidable. Hilbert initially defined the problem for an integer solution. However, the problem is equivalent for a natural number (zero included) solution. This follows from the fact that every natural number can be expressed as a sum of 4 squares, as proved by Lagrange.

We encode natural numbers using a single binary predicate and extend it to model Hilbert's Tenth problem (HTP). Given an instance of HTP: $p_{11} \dots p_{mn}$, p, we construct a SEAL program with `isZero` and `Succ` as base predicates, with their natural meanings (`Succ(x,x')` represents `x` being the successor of `x'`). The predicate `Succ` is the only binary base predicate in the program. From this, we can generate all natural numbers. We use the standard ordered pair notation for representing integers: (a, b) for the integer $a - b$. We give the sketch of the program here:

```
enext isZero(x).
anext Succ(x,x') :- NaturalNum(x').

NaturalNum(x) :- isZero(x).
NaturalNum(x) :- Succ(x,y), NaturalNum(y).

Integer(x,y) :- NaturalNum(x), NaturalNum(y).

Plus(x,y,x) :- isZero(y).
Plus(x,y,z) :- Succ(y,y'), Plus(x,y',z'),
                  Succ(z,z').

NMul(x,y,y) :- isZero(y).
NMul(x,y,z) :- Succ(y,y'), NMul(x,y',z'),
                   Plus(x,z',z).
Mul(a,b,x,c,d) :- NMul(a,x,c), NMul(b,x,d).

NExp(x,y,z) :- isZero(y), isZero(z'),
                   Succ(z,z').
NExp(x,y,z) :- Succ(y,y'), NExp(x,y',z'),
                   NMul(x,z',z).

One(x) :- Succ(x,x'), isZero(x').
Two(x) :- Succ(x,x'), One(x').
  .
  .
  .
N(x) :- ...

P11(x,y) :- ...
  .
```

```
  .
  .
PMN(x,y) :- ...
P(x,y) :- ...

Equal(x,y) :- isZero(x), isZero(y).
Equal(x,y) :- Succ(x,x'), Succ(y,y'),
                  Equal(x',y').

Query(x1...xm) :-
 One(1), Two(2), ... , N(n),  P(pa,pb),
 NExp(x1,1,x11'), P11(p11a,p11b),
  Mul(p11a,p11b,x11',x11a,x11b),
 NExp(x1,2,x12'), P12(p12a,p12b),
  Mul(p12a,p12b,x12',x12a,x11b),
  .
  .
  .
 NExp(x1,n,x1n'), P1N(p1na,p1nb),
  Mul(p1na,p1nb,x1n',x1na,x1nb),
  .
  .
  .
 NExp(xm,1,xm1'), PM1(pm1a,pm1b),
  Mul(pm1a,pm1b,xm1',xm1a,xm1b),
  .
  .
  .
 NExp(xm,n,xmn'), PMN(pmna,pmnb),
  Mul(pmna,pmnb,xmn,xmna,xmnb),
 Plus(pa,x11a, y11a), Plus(y11a,x12a, y12a), ... ,
 Plus(.,xmna, ymna),
 Plus(pb,x11b, y11b), Plus(y11b,x12b, y12b), ... ,
 Plus(.,xmnb, ymnb), Equal(ymna,ymnb).

Query(x1...xm) ?
```

We give a brief explanation of the above program. The first dynamic rule creates an element representing zero. The second rule with the help of Datalog rules for `NaturalNum` create natural numbers. Note that the type of the dynamic rule would not make any difference for this program. As mentioned earlier an integer is represented using an ordered pair of natural numbers. The predicate `Plus`, `NMul` and `NExp` describe addition, multiplication and exponentiation of natural numbers respectively. The predicate `Mul` denotes the multiplication of an integer with a natural number. The predicates `One` till `N` are used to describe exponentiation constants of the equation. A positive integer coefficient q is represented as $(q, 0)$ and a negative coefficient $-q$ as $(0, q)$. The predicates `P11` till `PMN` and `P` represent the coefficients. The predicate `Equal` is true when both the arguments (natural numbers) are same. The query predicate `Q` describes the given equation and checks if the result is zero.

Now, if the query reachability problem is decidable then we will have an algorithm to solve HTP. Hence proved. □

The undecidability result implies that it is not possible to find an algorithm (or a decision procedure) to compute reachability for a general SEAL program. We have looked at various abstractions and over-approximations and a general algorithm to guarantee soundness, by defining an appropriate equivalence relation or appropriate finite abstractions is still open.

Instead, we implement a bounded model-checking algorithm, which is complete for a given depth-bound. The procedure is to start with an initial state and explore all possible (non deterministic) transitions from each reachable state iteratively, until the bound is reached. If a counterexample or an unsafe state is found in the bounded model, then it is a true error. In practice, as we show in our Case Studies, a depth of 8 to 10 uncovers many vulnerabilities, previously known or otherwise. While we cannot assert that a model that does not admit an attack (or unsafe state) within this bound is safe, it may be unlikely that an attacker would use methods that involve many more state change operations.

In the next section, we present three case studies, where we model and analyze different aspects of safety in Windows 7, Asbestos, and IFEDAC.

5. CASE STUDIES

In this section, we present results from our modeling of three different access control systems, Windows 7, Asbestos (and HiStar), and IFEDAC. In each of the studies we explain our findings, in terms of true vulnerabilities discovered/not discovered for different depths. The SEAL tool to implement the bounded state-space exploration was written in F# and is available for download.

5.1 Windows 7 vulnerabilities

The first case study is the specification of the behavior of dynamically created links in the start menu folder. Through our analysis we discover that the specification admits a vulnerability at exploration depth 8.

To explain this vulnerability [20], we present an attack scenario as follows: A malicious user can write a proxy infection tool, which can be downloaded as a Trojan (as low), when the user clicks on an interesting third-party application. When this tool is run by the user, it can write to the user's start menu folder and read the contents of the global start menu folder without requesting elevated permissions. This malicious program searches the global start menu folder for all applications that require elevation, and creates duplicate links to malicious virus code (still labeled low) in the user's local folder, with the same name as the trusted program in the global folder.

When the user attempts to run a program that has been duplicated, they see a UAC prompt. Because the program already requires elevated permission, the user would not be alarmed, and would give consent. The malicious program, with elevated privileges, executes the intended program, fooling the user into thinking everything is normal. Meanwhile, the malicious program can clean up any traces, and install itself somewhere with permanently-elevated privileges.

The steps required to mount this attack, in the context of the SEAL program presented in Section 3.1, are shown in Figure 2. The model has to explore the states to depth level of 7 before the attack is discovered. We implemented a reduced version of the specification which had about 100000 states.

The next behavior we model is something called the silent elevation list. The silent elevation list is an option to allow the operating system developer to define a list of applications that can always bypass the UAC prompt, without user consent. The intent is to populate this list with only trusted applications, and improve the user experience.

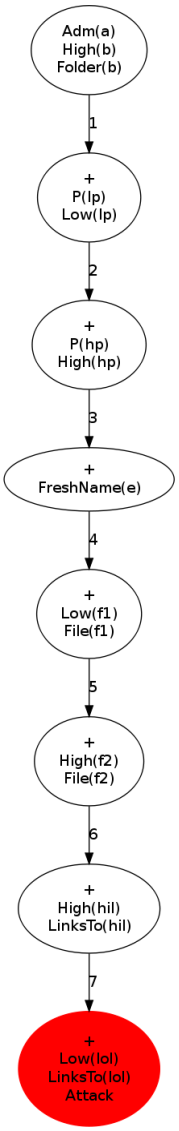

Figure 2: Start Menu Vulnerability

Popular third-party applications such as Adobe Reader and Flash Player occur in this list. These third-party applications are themselves untrusted and require explicit consent by the user to install as a plugin in the browser after they are downloaded. Once they are installed, they are automatically added to the silent elevation list. A SEAL program modeling this behavior is presented next:

```
1. enext Downloads(x,y), Low(y), App(y)
 :- Low(x), P(x).
2. High(y) :- Install(y), Low(y)
3. App(y), High(y) :- SEList(u), SameName(y,u).

4. Low(x); High(x)?
```

It is clear that a malicious plugin with the same name as a trusted binary on the Silent Elevation list can obtain administrative privileges, violating the safety requirement. A low process can download a low application, as specified in Step 1 and this can be installed as a plugin, assuming the source was trusted in Step 2. Now if this has the same name as an application in the Silent Elevation list, we have essentially bypassed the integrity guarantee as a High user can read a high application (this is not modeled in the example). This vulnerability was found in 5 steps in our tool. For the bounding depth of 5, our tool discovered over 6000 states.

Note that Windows 7 has an in-built tamper detection warning that automatically changes the color of the UAC prompt when an untrusted application is loaded. A trusted application whose integrity can be verified is presented with UAC window with a green border, and an untrusted application has a yellow border. This warning may prevent a vigilant user from falling victim to a start menu attack, but does not help much in the Silent Elevation attack, unless the browser validates the signature of the downloaded plugin outside the scope of the UAC prompt.

The vulnerabilities described were found by examining different components in the Windows 7 codebase that use UAC, and studying their behavior in the context of integrity guarantees (safety). In the future, we plan to examine all such interactions to systematically discover similar vulnerabilities (if any).

5.2 Asbestos

Asbestos is a Unix-based operating system developed by researchers that provides in-built support for creating confidentiality labels. The goal is to provide safety by implementing LBAC, by dynamically isolating trusted processes from untrusted ones using automatic tainting and taint propagation. Asbestos processes have both send and receive labels, corresponding to their clearance level and their taint level. Labels can also be set-valued, as in MLS compartments. The specification of Asbestos' LBAC provides a large number of choices. In EON, the authors have shown that it is possible to configure Asbestos labels in a manner that can cause unsafe behavior. However, safe defaults are suggested for send and receive labels. Even though these labels are set-valued, restricting the specifications to default values allows us to express it in EON and SEAL. For the example scenarios and configurations presented in the Asbestos work, EON was able to validate the safety properties effectively.

In terms of improving on EON, SEAL allows the use of multiary relations, reducing the size of the specifications, and potentially the complexity of verification for bounded models. For example, in EON, in order to create a user process and its associated port, with specific values for send and receive labels in the webserver example, we need the following relations:

```
enext Process(x), PortUW(x),
    LabelSendUcStar(x), LabelSendUtThree(x),
    LabelSendVcOne(x), LabelSendVtOne(x),
    LabelSendUwStar(x), LabelSendVwOne(x),
--
    LabelRecvUcStar(x), LabelRecvUtThree(x),
    LabelRecvVcTwo(x), LabelRecvVtTwo(x),
    LabelRecvUwStar(x), LabelRecvVwTwo(x),
--
    LabelPortUWUcTwo(x), LabelPortUWUtThree(x),
    LabelPortUWVcTwo(x), LabelPortUWVtTwo(x),
    LabelPortUWUwZero(x), LabelPortUWVwTwo(x) :- U.
```

In SEAL this can be specified using far fewer relations. In fact, this enext rule can be specified using only one relation, but we present an equivalent rule which uses four relations for readability:

```
enext Process(x, port),
ProcessSendLabels(x,s1, .. ,s6),
ProcessRecvLabels(x,r1, .. ,r6),
ProcessPortLabels(port,x,p1, .. ,p6).
```

Further, as in the case of Windows 7, exposing additional semantics to the access models may help in discovering new vulnerabilities. HiStar is an extension to Asbestos, and similar properties can be validated here.

5.3 IFEDAC

Information Flow enhanced DAC (IFEDAC [18]) is a proposed extension to the traditional DAC models that is specifically targeted at preventing Trojans. One of the problems with a DAC system is that a resource is associated with only one owner, whereas in real systems, the provenance of the resources, in terms of their past owners and their integrity plays an important part in arbitrating trust issues. To capture provenance, IFEDAC specifications extends the concept of ownership to include all principals that ever created or owned the file. This set of principals is used as a label to mark clearance or integrity level of a resource / process. These labels naturally form a lattice with set inclusion as the order. These labels may change as the system evolves (in accordance with the given IFEDAC specification).

The framework modeling the behaviors of IFEDAC specifications should account for these dynamic sets (labels). Let `X` and `Y` denote integrity label of a process p and *read protection class (rpc)* of an object o respectively. For the purpose of this illustration, it is enough to treat *rpc* of an object to be same as its integrity label. Let `dom(X,Y)` denote `X` dominates `Y`, i.e., `X` $\supseteq$ `Y`. A straight-forward translation of IFEDAC rule for a process p to read object o would read as follows.

```
reads(p,o) :- int(X,p), rpc(Y,o), dom(X,Y).
```

Note that here we are using an implicit quantification on set variables `X` and `Y`. SEAL does not allow quantification over set variables, and the IFEDAC model cannot be expressed in our framework. However, as in the case of Asbestos presented previously, it is possible to encode a particular instance of an IFEDAC system where these sets are fixed, and SEAL can be used in a limited way to verify safety when other relations can change dynamically. Verification with quantification over set-valued variables is outside the scope of SEAL.

6. CONCLUSIONS

We present SEAL, a language for specifying and analyzing dynamic LBAC systems using logic programs. SEAL improves on existing modeling languages such as EON by allowing for richer specifications, modeling behaviors that were not previously articulated. The safety problem in LBAC systems is reduced to query reachability in SEAL, and is shown to be undecidable even for a conservative program instance with one binary base predicate. This restriction poses an interesting open question about the existence of abstractions or over-approximations that can guarantee soundness in these models automatically.

Because of this restriction, we implement a bounded state-space exploration tool that is useful in analyzing the behavior of Windows 7, which is arguably the most widely used commercial LBAC system. While we cannot guarantee safety, any violation found by our tool is a true error. The vulnerabilities discovered by SEAL, including the Start Menu problem and the Silent Elevation list, highlight the need to define the interface between the access control model and modules that implement or require label-changes using UAC, more rigorously. We also study how SEAL can be used for modeling other LBAC systems, including Asbestos and IFEDAC. We recognize SEAL's limitations to handle IFEDAC models.

One natural direction to explore is to construct approximations. Ideally we would like to construct a finite state transition system containing all the behaviors (runs) of the induced possibly infinite state transtion system of a SEAL program. In this way, we can give guarantees on when a specification doesn't have a vulnerability.

Acknowledgements

The authors would like to thank their shepherd Steve Barker for his support, and thank the anonymous reviewers for their useful comments and suggestions that have improved the quality of this presentation greatly. We would also like to acknowledge the contributions of G. Ramalingam, Sriram Rajamani, and A. Baskar during our discussions about this work. Much of this work was done when the second author was an intern at Microsoft Research India.

7. REFERENCES

[1] S. Barker, M. Leuschel, and M. Varea. Efficient and flexible access control via jones-optimal logic program specialisation. *Higher Order Symbol. Comput.*, 21:5–35, June 2008.

[2] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6:501–546, November 2003.

[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] M. Bishop. Theft of information in the take-grant protection model. In *CSFW*, pages 194–218, 1988.

[6] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[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.

[8] A. Chaudhuri, P. Naldurg, S. Rajamani, G. Ramalingam, and L. Velaga. Eon: Modeling and analyzing dynamic access control systems. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 381–390. ACM, 2008.

[9] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. In *Formal Methods in System Design*, page 2001. Kluwer Academic Publishers, 2001.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[11] D. Denning. *Cryptography and Data Security*. Addison Wesley, 1982.

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

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

[14] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for selinux mls policy. *ACM Trans. Inf. Syst. Secur.*, 13:26:1–26:31, July 2010.

[15] A. K. Jones, R. J. Lipton, and L. Snyder. A linear time algorithm for deciding security. *Symposium on Foundations of Computer Science*, 0:33–41, 1976.

[16] B. W. Lampson. Protection. *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, 1971.

[17] 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, United Stated National Security Agency (NSA), 1995.

[18] Z. Mao, N. Li, H. Chen, and X. Jiang. Trojan horse resistant discretionary access control. In *SACMAT*, pages 237–246, 2009.

[19] P. Naldurg, S. Schwoon, S. Rajamani, and J. Lambert. Netra: seeing through access control. In *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security*, pages 55–66, 2006.

[20] R. Paveza. User-prompted elevation of unintended code in windows vista. World Wide Web electronic publication, 2009.

[21] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[22] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.