SLAT: Information Flow Analysis in Security Enhanced Linux*

Joshua D. Guttman   Amy L. Herzog   John D. Ramsdell

July 5, 2005

This paper describes the theoretical foundation of the Security Enhanced Linux policy analysis tool Slat [2]. The goal of this paper is an accurate and complete treatment of the subject, and is not intended as an introduction to this topic.

1 Information Flow Policy

The SELinux security server makes decisions about system calls, for instance whether a process should be allowed to write to a particular file, or whether a process should be allowed to overlay its memory with the binary image contained at a particular pathname, and continue executing the result. For each system call, SELinux specifies one or more checks that must be satisfied in order for the call to be allowed. Each check is labeled by a pair consisting of a *class* and a *permission*. The class describes a kind of resource that the access involves, such as `file`, `process`, or `filesystem`. The permission describes the action itself, such as `read`, `write`, `mount`, or `execute`. We will use the term *resource* to cover any object in an SELinux system; processes, files, sockets, etc. are all regarded as resources. Each resource has a *security context* which summarizes its security relevant status.

In making a check, the security server receives as input two facts, the *security contexts* of the process and of another resource involved in the system call. A security context is a tuple consisting of three components,[1] called a *type*, a *role*, and a *user*. The user is similar in intent to the normal Unix notion of user, and represents the person on behalf of whom the system is executing a process or maintaining a resource. The role, derived from the literature on role-based access control, is an intermediate notion intended to specify that collections of users should be permitted to execute corresponding collections of programs. The main purpose of the *user* component is to specify what roles that user is permitted; the main purpose of the *role* is to specify what types of processes those users are permitted to execute. The type specifications do the remainder of the work.

A labeled-transition system is used to model the information flow policy specified by SELinux policy file. A security context is a state in the transition system, and an event labels each transition. We now formally define an information flow policy. The derivation of an information flow policy from an SELinux policy file is described in Section 2.

As stated above, security context is a type-role-user triple. For a given information flow policy, let T be the set of names for types, R be the set of names for roles, and U be the set of names for users, so that the set of security contexts S is $T \times R \times U$.

An event is a class-permission pair. Let C be the set of names for classes, and P be the set of names for permissions, so that the set of events G is $C \times P$.

Formally, an information flow policy is a 4-tuple (S, G, Δ, S_0), where S is a finite set of security contexts, G is a finite set of events, $S_0 \subseteq S$ is the set of initial security contexts, and $\Delta \subseteq S \times G \times S$ is the information flow transition relation. When

*This work was funded by the United States National Security Agency. Authors' address: The MITRE Corporation, 202 Burlington Road, Bedford MA, 01730-1420.

[1] or four components, if the system is compiled with support for multi-level security as it can be, but normally is not. For definiteness, we will assume MLS support is not compiled into the kernel in the remainder of this paper, although the approach we describe is equally applicable if it is.

$(s, g, s') \in \Delta$, the information flow policy allows information to flow from an object labeled by s to one labeled by s' as a result of event g.

The set of information flow paths of length n, P^n, is the subset of $(S \times G)^n \times S$, such that

$$(s_0, g_0, s_1, g_1, \ldots, s_{n-1}, g_{n-1}, s_n)$$

is in P^n if and only if $s_0 \in S_0$ and $(s_i, g_i, s_{i+1}) \in \Delta$ for all $i < n$. Note that $P^0 = S_0$.

A path is in the set of infinite information flow paths, P^∞, if and only if $s_0 \in S_0$ and $(s_i, g_i, s_{i+1}) \in \Delta$ for all i.

1.1 Logic of the Transition System

We use many-sorted first order logic with equality. The sorts are T, R, U, C, and P. We assume that the language has a constant for each defined value of these sorts, and include axioms stating that every value of a particular sort is equal to one of the constants of that sort. No predicates other than equality are needed. We are interested in formulas in this language involving at most the free variables $t, r, u, c, p, t', r', u'$, which we call *information flow transition formulas*. Here a lowercase variable ranges over the sort whose name is the same letter in uppercase. Primed variables refer to the value in the next state, and unprimed variables refer to the value in the current state.

A *state formula* σ is a formula containing at most the free variables t, r, u. A *event formula* γ is a formula containing at most the free variables c, p. We write δ for general information flow transition formulas.

For our purposes, we need not distinguish between a set of tuples and a formula true of exactly the same tuples. Furthermore, within formulas, we will write $t \in \{\tau_1, \ldots, \tau_n\}$ as a shorthand for $t = \tau_1 \vee \cdots \vee t = \tau_n$.

2 Deriving a Policy

An SELinux configuration file defines the names for classes, permissions, types, and users, and all but one name for the roles. The role `object_r` is an implicit part of a configuration.

The main component in a configuration file is the type, accounting for at least 22,000 out of the 22,500 access control statements in the example policy file contained in the distribution. The type is used to specify the detailed interactions permitted between processes and other resources. Each type specification determines some actions that are allowed; in the SELinux configuration file they are introduced by the keyword `allow`. For a request to succeed, some `allow` statement in the configuration file must authorize it.

The syntax of a type allow statement follows.

$$\texttt{allow}\ T_s\ T_t : C_a\ P_a;$$

Each `allow` statement specifies a set of process (source) types T_s, a set of resource (target) types T_t, a set of classes C_a, and a set of permissions P_a. If a process whose type is in T_s requests an action with a class-permission pair in $C_a \times P_a$ against a resource with type in T_t, then that request is authorized. As an information flow transition relation formula, an allow statement means:

$$\begin{aligned}&\alpha_t[\![\texttt{allow}\ T_s\ T_t : C_a\ P_a;]\!] =\\ &\quad (t \in T_s \wedge t' \in T_t \setminus \{\texttt{self}\} \wedge c \in C_a \wedge p \in P_a) \vee\\ &\quad (\texttt{self} \in T_t \wedge t \in T_s \wedge t = t' \wedge c \in C_a \wedge p \in P_a).\end{aligned}$$

The special identifier `self` is not the name of a type, but is instead used like a target type to indicate that the statement should be applied between each source type and itself. Thus, the formula generated for an allow statement must be simplified so as to eliminate references to `self`.

When a process changes security context, the role may change, but only when permitted by another form of the allow statement.

$$\texttt{allow}\ R_c\ R_n;$$

The role allow statement permits a process whose current role is in R_c to transition to one with a new role in R_n. As a formula, the statement means:

$$\begin{aligned}&\alpha_r[\![\texttt{allow}\ R_c\ R_n;]\!] =\\ &\qquad c = \texttt{process} \wedge p = \texttt{transition} \Rightarrow\\ &\qquad\quad r \in R_c \wedge r' \in R_n.\end{aligned}$$

A role is declared with the following syntax.

$$\texttt{role}\ \rho\ \texttt{types}\ T_r;$$

In addition to declaring the role name ρ, the statement defines the set of types with which the role is permitted to be associated. As a formula, the statement means:

$$\beta_r[\![\texttt{role}\ \rho\ \texttt{types}\ T_r;]\!] = r = \rho \wedge t \in T_r.$$

A user is declared with the following syntax.

$$\texttt{user}\ \mu\ \texttt{roles}\ R_u;$$

In addition to declaring the user name μ, the statement defines the set of roles the user is permitted to assume. As a formula, the statement means:

$$\beta_u[\![\texttt{user}\ \mu\ \texttt{roles}\ R_u;]\!] = u = \mu \wedge r \in R_u.$$

Constraint definitions specify additional limits on transitions with the following syntax.

$$\texttt{constraint}\ C_c\ P_c\ \delta;$$

The constraint expression δ has a natural translation as a formula as long as role operations are limited to equality testing, the case we observe. As a formula, the statement means:

$$\chi[\![\texttt{constraint}\ C_c\ P_c\ \delta;]\!] = c \in C_c \wedge p \in P_c \Rightarrow \delta.$$

Since we abstract from auditing and other issues that do not affect information-flow security goals, the configuration file defines five relations of interest. Each relation is built up by statements contained in the same configuration file.

- α_t is the formula built up by disjoining the meaning of every type `allow` statement.
- α_r is the formula built up by disjoining the meaning of every role `allow` statement.
- β_r is the formula built up by disjoining the meaning of every `role` statement to the formula $r = \texttt{object\_r} \wedge t \notin T_p$, where T_p is the set of process types.
- β_u is the formula built up by disjoining the meaning of every `user` statement to the atomic proposition $r = \texttt{object\_r}$. As a result, $\beta_u \wedge r = \texttt{object\_r}$ holds for all user names.
- χ is the formula built up by conjoining the meaning of every `constraint` statement.

In what follows, we write $\tilde{\delta}$ to mean the formula derived from δ by interchanging primed and unprimed variables.

Some events (file write, for instance) transfer information from process to resource, while others (file read, for instance) transfer it from resource to process. Let ψ be a class name, and π be a permission name. SELinux has a file that describes how each (ψ, π) transfers information, whether like a read, like a write, in both directions, or in neither. From it, we extract descriptions of two sets, γ_w and γ_r. When event $(\psi, \pi) \in \gamma_w$, it has write-like flow, while $(\psi, \pi) \in \gamma_r$ means it has read-like flow. The derivation of the two sets of flow related events follows.

For class ψ, the set of compatible permissions is declared in the policy file with the following two syntactic forms.

$$\begin{array}{c}\texttt{common}\ \phi\ \{\ \pi^+\ \}\\ \texttt{class}\ \psi\ [\ \texttt{inherits}\ \phi\ ]^?\ \{\ \pi^+\ \}\end{array}$$

If the `class` statement does not include an `inherits` phrase, the set of compatible permissions is just the ones listed between the curly braces, otherwise, the permissions in the `common` statement named ϕ are also included.

In the SELinux file that describes the direction of information flow, the relevant statements appear in the following two syntactic forms.

$$\begin{array}{c}\texttt{common}\ \phi\ \{\ F^+\ \}\\ \texttt{class}\ \psi\ \{\ F^+\ \}\end{array}$$

where F associates a flow direction with a permission π.

$$\begin{array}{lcl}F & ::= & \pi : E.\\ E & ::= & \texttt{none} \mid D \mid \{D\} \mid \{D, D\}.\\ D & ::= & \texttt{read} \mid \texttt{write}.\end{array}$$

As a write-like flow formula, the `class` statement of a class that was declared without an `inherits` phrase in the policy file means:

$$\begin{array}{c}\gamma_w[\![\texttt{class}\ \psi\ \{\ F^+\ \}]\!] =\\ c = \psi \wedge p \in \{\pi_1, \ldots, \pi_n\}.\end{array}$$

where $\{\pi_1, \ldots, \pi_n\}$ is the set of permissions declared to be write-like in the body of the `class` statement. For a class declared with an `inherits` phrase, the write-like permissions in its associated `common` statement are also included. The formula γ_w is the disjunction of the write-like meaning of every `class` statement in the file.

The direction-flow file contains only a simple approximation. It does not take into account indirect flows caused by error conditions or variations in timing, and it does not consider flow into other system resources besides the process requesting the event and the resource against which the event is requested. This is why our analysis avoids the subtleties of covert channels.

Information flows from an entity with security context (t, r, u) to (t', r', u') if an event (c, p) has write-like flow and

$$\alpha_t \wedge \alpha_r \wedge \beta_r \wedge \beta_u \wedge \tilde{\beta}_r \wedge \tilde{\beta}_u \wedge \chi.$$

When event (c, p) has read-like flow, information flows from an entity with security context (t, r, u) to (t', r', u') if

$$\tilde{\alpha}_t \wedge \tilde{\alpha}_r \wedge \tilde{\beta}_r \wedge \tilde{\beta}_u \wedge \beta_r \wedge \beta_u \wedge \tilde{\chi}.$$

The information flow transition relation formula δ is the disjunction of the previous two formulas.

$$\begin{aligned} &\beta_r \wedge \beta_u \wedge \tilde{\beta}_r \wedge \tilde{\beta}_u \\ &\quad \wedge ((\gamma_w \wedge \alpha_t \wedge \alpha_r \wedge \chi) \\ &\qquad \vee (\gamma_r \wedge \tilde{\alpha}_t \wedge \tilde{\alpha}_r \wedge \tilde{\chi})) \end{aligned}$$

The initial states of the information flow policy are the ones that are compatible with β_r and β_u so that

$$\sigma = \beta_r \wedge \beta_u.$$

Inspecting δ and σ leads to the conclusion that every reachable state is an initial state.

3 Information Flow Diagrams

Some information flow policy goals stipulate which sequences of causal interactions are permissible. It is easy to visualize these causal chains using something we call information flow diagrams.

There are four kinds of freedom in constructing the chains. First, we can define what security contexts appear at a stage in the process; we refer to these sets by symbols such as σ_i. Second, we may characterize what events may occur at a particular stage; we refer to these sets by symbols such as γ_i. Third, we may be interested in the consequence of a single event, or a sequence of iterated events. We indicate these by decorating γ_i by a superscript 1 or $+$, respectively. Let λ_i be a label of one of the forms γ_i^1 or γ_i^+. Finally, we may specify exceptional security contexts, σ_e, or exceptional events, γ_e, that should be ignored by the assertion–a concept to be made precise later. Then we can notate an information flow policy goal in the form:

$$\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{n-2}} \sigma_{n-1} \xrightarrow{\lambda_{n-1}} \sigma_n \quad [\sigma_e; \gamma_e] \qquad (1)$$

An assertion of an information flow policy in the above form is called an *information flow diagram.*

We interpret this information flow policy as an assertion about all paths from σ_0 to σ_n. It asserts that this path must encounter the σ_i in the order given, executing events from λ_i in each stage, and that there must be just one such event if the decoration is 1 and may be more events if the decoration is $+$. Additionally, any path that visits any exceptional security contexts in σ_e before getting to σ_n, or gets there via the exceptional events in γ_e, does not violate the assertion.

An information flow diagram prohibits paths that contain certain prefixes. An information flow policy satisfies the information flow policy goal expressed as an information flow diagram if the information flow policy allows no prohibited paths. The paths prohibited by an information flow diagram of length n all must meet at least the following assertions:

$$\begin{array}{ll} s_m \in \sigma_n, & \text{for some } m, \\ s_j \in \sigma_0, & \text{for some } j < m, \\ s_k \notin \sigma_e, & \text{for all } j \le k < m, \text{ and} \\ g_k \notin \gamma_e, & \text{for all } j \le k < m. \end{array} \qquad (2)$$

3.1 Order Assertions

One form of path prohibited by an information flow diagram is one that visits states in the wrong order. A path satisfying Eqn. 2 is prohibited if there exists an $i < n$ such that the path visits a state in σ_{i+1} without having previously visited a state in σ_i. In other words, let $s_m \in \sigma_n$ and $s_j \in \sigma_0$. A path is prohibited if there is some k with $j \leq k \leq m$, that satisfies:

$$s_k \in \sigma_{i+1} \wedge s_\ell \notin \sigma_i \text{ for all } j \leq \ell < k. \tag{3}$$

3.2 Event Assertions

The other form of path prohibited by an information flow diagram is one that wanders from its bounds. A path satisfying Eqn. 2 is prohibited if having reached a state in σ_i, the path fails to reach a state in σ_{i+1} using events in γ_i.

Let $\Theta_{i,j}$ be the assertion that a path satisfying Eqn. 2 properly reaches σ_i at step j. The assertion $\Theta_{i,j}$ is false whenever $j > m$. The assertion $\Theta_{0,j}$ is true when $s_j \in \sigma_0$. Assertion $\Theta_{i+1,j'}$ is true when

$$\Theta_{i,j} \wedge g_j \in \gamma_i \wedge g_k \in \gamma_i \wedge s_k \notin \sigma_{i+1} \wedge s_{j'} \in \sigma_{i+1},$$

for all k with $j < k < j'$. When $\lambda_i = \gamma_i^1$, i.e, the single event case, $j' = j + 1$.

A path at step j wanders at σ_i in a single step event ($\lambda_i = \gamma_i^1$) if

$$\Theta_{i,j} \wedge (j \geq m \vee g_j \notin \gamma_i \vee s_{j+1} \notin \sigma_{i+i}). \tag{4}$$

A path at step j wanders at σ_i in a multiple step event ($\lambda_i = \gamma_i^+$) if $\Theta_{i,j}$ and

$$\begin{array}{ll} s_\ell \notin \sigma_{i+1} & \text{for all } j < \ell \leq m, \text{ or} \\ g_k \notin \gamma_i & \text{while } s_\ell \notin \sigma_{i+1}, \end{array} \tag{5}$$

for some k and all ℓ such that $j < \ell \leq k + 1 \leq m$.

4 Model Checking

Model checking is used to automatically show that an information flow policy satisfies an information flow policy goal stated as an information flow diagram. This section explains how an information flow policy is stated in the formalisms used by model checkers.

Following [1], we derive a Kripke structure from the first order formulas $\mathcal{S}_0$ and $\mathcal{R}$ that represent an information flow policy (S, G, δ, σ).

There are six forms of atomic propositions–one form for each of the six system variables, t, r, u, c, p, and k. The first five forms are identical to the ones defined at the end of Section 1. The system variable k ranges over boolean values. When k is asserted to be false, the atomic proposition is $\neg k$, and when true, the atomic proposition is written simply as k. The boolean system variable k is used to construct a Kripke structure with a transition relation that is total. It asserts that a state is okay.

The formula $\mathcal{S}_0$ is $k \Rightarrow \sigma$. The formula $\mathcal{R}$ is $k' \Leftrightarrow k \wedge \delta$.

A Kripke structure can be derived from $\mathcal{S}_0$ and $\mathcal{R}$ as follows. The set of states is $T \times R \times U \times C \times P \times B$, where B is the set of boolean values.

The formula $\mathcal{S}_0$ implies that the initial state of the Kripke structure contains members of σ when k is true. It has all members of S when k is false.

The formula $\mathcal{R}$ implies that the transition relation of the Kripke structure is total. Whenever k is false, any transition is allowed as long as k' is false. Whenever k is true, the relation allows a transition when $(s, g, s') \in \delta$ and k' is true, and allows a transition when $(s, g, s') \notin \delta$ and k' is false. Clearly, this transition relation is total. Additionally, $k' \Rightarrow k$, that is, k is true in the next state only if it is in the current state.

The labeling function of the Kripke structure is obvious from this construction.

Inspecting $\mathcal{S}_0$ and $\mathcal{R}$ leads to the conclusion that every reachable state is an initial state.

5 Diagrams as CTL Assertions

We translate the semantics of an instance of an information flow diagram into assertions in Computational Tree Logic (CTL) [1, Ch. 3]. Because all reachable states are initial states, assertions of interest assume the initial state of a path is in σ_0. Had the Kripke structure not had this property, assertions would have had to be asserted globally using the **AG** operator.

5.1 Order Assertions

The first set of assertions states that if we move to σ_n from σ_0, we visit all intermediate σ_i in the appropriate order:

$$\sigma_0 \Rightarrow \mathbf{A}[\hat{\sigma}_i \,\mathbf{R}\, (\sigma_{i+1} \Rightarrow \mathbf{A}[\sigma_e \vee \gamma_e \,\mathbf{R}\, \neg(\sigma_n \wedge k)])], \quad (6)$$

where $1 \leq i < n$, and $\hat{\sigma}_i = \sigma_i \vee \sigma_e \vee \gamma_e$.

To understand these assertions, consider what happens when one fails. When assertion i fails, the following is true:

$$\sigma_0 \wedge \mathbf{E}[\neg\hat{\sigma}_i \,\mathbf{U}\, (\sigma_{i+1} \wedge \mathbf{E}[\neg(\sigma_e \vee \gamma_e) \,\mathbf{U}\, \sigma_n \wedge k])].$$

In words, the assertion says that there is some path from an initial state in σ_0, that reaches a part of σ_{i+1} that eventually leads to σ_n without visiting an exception, and between σ_0 and σ_{i+1} avoids security contents in $\sigma_i \vee \sigma_e$, and events in γ_e. The properties of the transition relation ensure that the prefix of the path that ends in a state denoted by σ_n contains only states in which k is true.

5.2 Event Assertions

The second set of assertions states that a causal chain from σ_0 to σ_n uses acceptable events at each step. It uses events in γ_0 until a state in σ_1 is reached, and then uses events in γ_1 until a state in σ_2 is reached, and so forth.

The set of assertions is given by recurrence relations.

$$\neg f_0^i, \qquad 0 \leq i < n, \quad (7)$$

where for $j < i$, let $\hat{\gamma}_i = \gamma_i \wedge \neg\sigma_e \wedge \neg\gamma_e$ in

$$\begin{array}{ll} f_j^i = \sigma_j \wedge \hat{\gamma}_j \wedge \mathbf{EX}\, f_{j+1}^i, & \text{if } \lambda_j = \gamma_j^1, \\ f_j^i = \sigma_j \wedge \hat{\gamma}_j \wedge \mathbf{EX}\,\mathbf{E}[\hat{\gamma}_j \,\mathbf{U}\, f_{j+1}^i], & \text{if } \lambda_j = \gamma_j^+, \end{array}$$

and

$$f_i^i = \sigma_i \wedge (\neg\gamma_i \wedge g \vee \hat{\gamma}_i \wedge \mathbf{EX}\, h_i),$$

where

$$g = \mathbf{E}[\neg(\sigma_e \vee \gamma_e) \,\mathbf{U}\, \sigma_n \wedge k],$$

and when $\lambda_i = \gamma_i^1$,

$$h_i = \neg\sigma_{i+i} \wedge g,$$

and when $\lambda_i = \gamma_i^+$,

$$h_i = \mathbf{E}[\neg\hat{\sigma}_{i+1} \,\mathbf{U}\, \neg\gamma_i \wedge \neg\sigma_{i+1} \wedge g].$$

To understand the assertions defined by Eqn. 7, consider what happens when one fails. When assertion i fails, f_0^i is true. In words, this assertion says that there is some path that goes through $\sigma_0, \ldots, \sigma_i$ as prescribed by the information flow diagram, however, something goes wrong after this point. Assertion f_i^i states that after reaching σ_i, and while avoiding σ_{i+1}, the path finds a state that leads to σ_n without visiting an exception. This state does not have a security context in σ_{i+1}.

If the CTL assertions in Eqns. 6 and 7 hold, then the information flow policy expressed in a diagram of the form shown in Eqn. 1 is true. These general forms make security goal statements simple to produce: appropriate contexts, classes, and permissions can simply be substituted for the variables in Eqn. 1 as appropriate.

6 Diagrams as LTL Assertions

We translate the semantics of an instance of an information flow diagram into assertions in Linear Temporal Logic (LTL) [1, Ch. 3].

6.1 Order Assertions

The first assertion states that if we move to σ_n from σ_0 along a non-exceptional path, a security context in σ_i occurs before the first occurrence of a security context in σ_{i+1}:

$$\sigma_0 \wedge (\neg\sigma_e \wedge \neg\gamma_e) \,\mathbf{U}\, (\sigma_n \wedge k) \Rightarrow \bigwedge_{i=1}^{n-1} \sigma_i \,\mathbf{R}\, \neg\sigma_{i+1}. \quad (8)$$

The operator **R** ("releases") asserts that its right hand operand is true and remains true until its left hand operator has been true at least once. Thus, this formula asserts that each set σ_{i+1} is not encountered until after σ_i has been encountered, along paths from σ_0 to σ_n.

Here is another way to state the same thing.

$$\sigma_0 \Rightarrow \hat{\sigma}_i \,\mathbf{R}\, (\sigma_{i+1} \Rightarrow (\sigma_e \vee \gamma_e) \,\mathbf{R}\, \neg(\sigma_n \wedge k))$$

where $1 \leq i < n$, and $\hat{\sigma}_i = \sigma_i \vee \sigma_e \vee \gamma_e$.

6.2 Event Assertions

The second assertion states that a causal chain from σ_0 to σ_n uses acceptable events at each step. It uses events in γ_0 until a state in σ_1 is reached, and then uses events in γ_1 until a state in σ_2 is reached, and so forth:

$$\begin{array}{l} \sigma_0 \wedge (\neg\sigma_e \wedge \neg\gamma_e) \,\mathbf{U}\, (\sigma_n \wedge k) \\ \qquad \Rightarrow \gamma_0 \,\mathcal{O}_0\, (\sigma_1 \wedge \gamma_1 \,\mathcal{O}_1\, (\sigma_2 \ldots)). \end{array} \tag{9}$$

where $\gamma_i \,\mathcal{O}_i\, f = \gamma_i \wedge \mathbf{X}\, f$ for a single step event ($\lambda_i = \gamma_i^1$), and $\gamma_i \,\mathcal{O}_i\, f = \gamma_i \wedge \mathbf{X}(\gamma_i \,\mathbf{U}\, f)$ for a multiple step event ($\lambda_i = \gamma_i^+$).

References

[1] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2001.

[2] Joshua D. Guttman, Amy L. Herzog, and John D. Ramsdell. Information flow in operating systems: Eager formal methods. IFIP WG 1.7 Workshop on Issues in the Theory of Security, April 2003. `http://www.dsi.unive.it/IFIPWG1_7/wits2003.html`.