FLEXFLOW: A FLEXIBLE FLOW CONTROL POLICY SPECIFICATION FRAMEWORK

Shiping Chen, Duminda Wijesekera and Sushil Jajodia
Center for Secure Information Systems,
George Mason University, Fairfax, VA 22030

Abstract We propose FlexFlow, a logic based flexible flow control framework to specify data-flow, work-flow and transaction systems policies that go beyond point-to-point flows. Both permissions and prohibitions are specifiable in FlexFlow and meta-policies such as *permissions take precedence* themselves can be specified over the meta-policy neutral policy specification environment of FlexFlow. We show the expressibility of FlexFlow by expressing three existing flow control models which were proposed for different applications and used different mechanisms.

Keywords: Flow control policy, Data flow, Security policy

1. Introduction

Information flow policies govern the exchange of information at various levels in systems. At the lowest levels, information is copied in and out of registers and memory locations inside processors. At a higher level, information is exchanged among variables in programs, methods in object oriented systems and transactions in database systems. In networked systems, messages are copied across system boundaries in order to exchange information. In all of these examples the levels at which information flows, the units of transfer and the number of destinations vary, but the central issue of information flow remains the same. Thus, it is interesting to investigate the commonalities among policies that govern information flow at an abstract level. This paper proposes FlexFlow, a framework to do so.

Being designed to capture properties common across flows, FlexFlow is formulated using abstractions of *nodes* and *trees* of flows among them. Because of this abstractness, FlexFlow has two advantages. Firstly, FlexFlow is not limited to high or low level information exchange policies. Thereby it can be used to reason about and derive consequence of mixing flow control polices at different levels. For example, higher level information exchange policies may

govern flows between method calls, and lower level policies may govern data flows inside method calls. Because our framework can model information flow at both levels, it is able to combine policies across both of them.

The second advantage is that our policy specification framework does not depend on any meta policies. Therefore, policies using different meta policies can be modelled and their total effect can be compared using our framework. This is similar to the advantage gained by the *Flexible Authorization Framework* (FAF) of Jajodia et al. [9] over its predecessors in specifying access control policies.

FlexFlow specifies flow control policies as a set of stratified Horn clauses, which is sound and complete, in the sense that every requested flow is either granted or denied, but not both. FlexFlow is based on unique stable model semantics, and therefore FlexFlow rules can be interpreted unambiguously.

The rest of the paper is organized as follows. Section 2 informally introduces abstract concepts of nodes, flow trees and their representations. Section 3 formally describes the FlexFlow framework. Section 4 shows that FlexFlow can be used to express various existing flow control models. Section 5 describes related work and Section 6 concludes the paper.

2. An Informal Description of FlexFlow

This section informally describes the FlexFlow framework and the reasons that went into making our design choices. FlexFlow has trees referred to as *flow trees* build up from *nodes* and *branches*. Nodes represent sources and sinks of information and branches represent pathways taken by information flowing between nodes. Thus, information flows from the leaves of a tree via intermediate nodes to its root. Given that the same node can either send information or refuse to do so under different circumstances, FlexFlow uses *node environments* to capture sufficient data to enforce such *local* decisions. Similarly, a flow tree itself may be acceptable or rejectable due to policies under different global circumstances, despite the nodes enforcing their local policies. Such circumstances are captured by *tree environments* of flow trees. The exact relationship between the environment of a flow tree and those of its nodes is not rigidly fixed by FlexFlow and is therefore application specifiable. Consequently, the flow trees with their environments, consisting of nodes and node environments constitute the basic entities of focus in FlexFlow. In order to specify how to construct acceptable or rejectable flow trees, FlexFlow has rules written in the form of Horn clauses about trees and their structural properties. Our position - one that we put forth in this paper - is that such rules suffice to enforce existing flow control policies in a uniform and meta-policy independent manner.

As an example, consider an abstract syntax tree of the expression $x+(y+z)$. As shown in the left hand side of Figure 1 (the right hand side is its Prolog representation - to be explained shortly), it consists of three leaves with variables (variables are considered named locations that can hold values) x, y and z, one intermediate node R_{temp} and a root node R_{final}. Assume that each variable in this example is accessible by a specified set of subjects. For example, x is accessible by Alice and Bob, y accessible by Bob and Cindy, and z accessible by Alice, Bob and Cindy. We model this situation by making the environment of each node be the access control list. We view each binary addition operation as a computation tree in which the root stores the sum of the values stored in the leaves. Thus, as shown in Figure 1 there is an *intermediate* node R_{temp} holding the value of $x + y$ and the root R_{final} holding $x + (y + z)$. As it is shown in Figure 1, R_{final} holds the value of $x + R_{temp}$. The tree rooted at R_{temp} is said to resemble a *one step* flow, as it has depth one. By piecing together two trees with depth one, (namely the depth-one binary tree rooted at R_{final} and the one rooted at R_{temp}) we get a depth-two flow tree, namely the tree rooted at R_{final} with x, y and z as leaves. Having

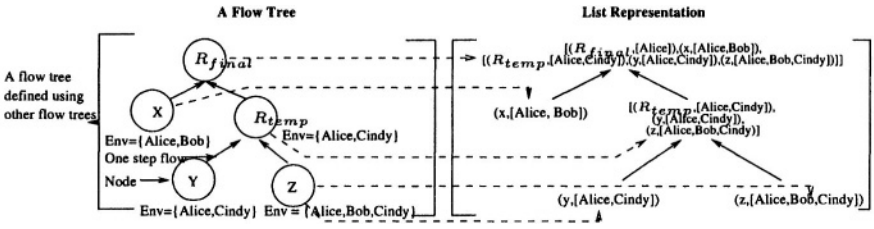

Figure 1. An Example Flow Tree

to reason with tree structures and lists in Horn clauses, FlexFlow uses Prolog's list notation to represent trees, and uses ordered pairs of (node names, environments) as our nodes. For example, (z, [*Alice, Bob, Cindy*]) represents the rightmost bottom (i.e. the last in the in-order representation of the tree) node in Figure 1. And, [(R_{temp}, [*Alice, Cindy*]), [(y, [*Alice,Cindy*]), (z, [*Alice, Bob, Cindy*])]] represents the subtree rooted at R_{temp} with an access control list *[Alice, Cindy]* and two children y and z.

FlexFlow states flow control policies as Horn clauses using flow trees as individual elements. In order to create these rules we used some predicates. Both rules and predicates used in FlexFlow belong to a five level stratification. FlexFlow use these stratified rules to specify flows that are permitted or prohibited without assuming any meta policies such as the open or closed policy. Formal details are given in Section 3.

2.1. Architecture of FlexFlow

As shown in Figure 2, FlexFlow rules have five stages, numbered zero through four in the figure. The first of them is the structural module (stage 0) containing data structures and functions needed to define flows. These include subject and object hierarchies, predicates necessary to model the environments and list manipulation operations such as adding or removing elements, taking the union of two lists etc.

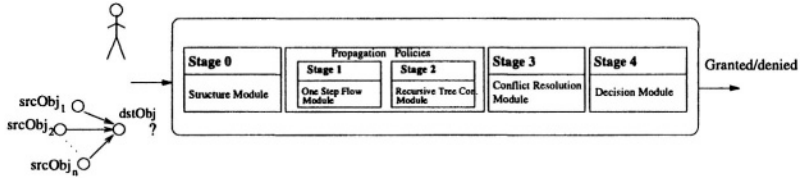

Figure 2. FlexFlow System Architecture

Stage 1 is used to specify one-step flows that are permitted or prohibited based on the structural properties specified at the previous stage. Stage 2 is used to build permitted or prohibited flow trees that may use already defined one-step flows. Recursion is allowed in this step. For example if flows are transitive, they can be specified at this stage, as transitive closure can be defined recursively. Similarly, permissions can be propagated up and down subject, object and role hierarchies using recursive rules.

Although propagation policies are flexible and expressive, they may result in *over specification.* That is, rules could be used to derive both negative and positive flows that may be contradictory. This possible conflict is due to the fact that positive and negative permissions is an application level inconsistency. This kind of inconsistency is not recognized by the underlying stable model semantics of locally stratified logic programs. Similar encodings have been used in the *Flexible Authorization Framework* by Jajodia et al. [9]. In order to weed out contradictive specifications, FlexFlow uses *conflict resolution policies.* They are stated in stage 3.

At stage 4, *decision policies* are applied in order to ensure the completeness of flow specification. That is because every flow request made to FlexFlow must be either granted or denied. This is necessary because, as otherwise the framework makes no assumption about flows that are not derivable only using stated rules.

3. Syntax and Semantics of FlexFlow

3.1. The Language of FlexFlow

Terms of the language: The language of FlexFlow consists of terms that are made up from constants and variables for nodes, environments and actions. It also has constants and variables over lists of (node, environment) pairs. Such lists are considered as flow trees, where the first element is the root and the rest are the children. In addition, FlexFlow allows environments to have application defined structures and predicates. An example of such a structure is an access control list, as used in Figure 1.

Predicates of the language: Predicates in FlexFlow belong to five strata, as summarized in Table 1 and explained below.

Stratum 0: Consists of list manipulation and application-specific predicates. An example of an application specific predicate is $\mathsf{role}(x_s, x_r)$ stating that subject x_s plays role x_r. An example list manipulation predicate is $\mathsf{isMember}((x_n, x_e), X_L)$ stating that the (node, environment) pair (x_n, x_e) is in the list X_L. More examples are given in section 4.

Stratum 1: Consists of a four-ary predicate safeFlow. The formal parameters x_n, x_e, X_L and x_{action} of $\mathsf{safeFlow}(x_n, x_e, X_L, \pm x_{action})$ are respectively the destination node, destination environment, a finite list of source (node, environment) pairs and a name for the one step flow. $\mathsf{safeFlow}(x_n, x_e, X_L, \pm x_{action})$ holds if the one step flow consisting of the source list X_L and the destination (x_n, x_e) named x_{action} is either permitted or prohibited depending upon the sign (+ or -) appearing in front of x_{action}.

Stratum 2: Consists of a ternary predicate safe Flow*. The formal parameters x_{flowT}, x_{flowE} and x_{action} in $\mathsf{safeFlow}^*(x_{flowT}, x_{flowE}, \pm x_{action})$ are respectively a flow tree, its environment x_{flowE} and its name x_{action}. $\mathsf{safeFlow}^*(x_{flowT}, x_{flowE}, \pm x_{action})$ represents a permitted or prohibited flow tree depending upon the sign (+ or -) appearing in front of x_{action}.

Strata 3 and 4: Consist of a ternary predicate finalsafeFlow, with the same arguments as safeFlow*, representing the flow control decisions finally made by FlexFlow. It is used to express conflict resolution policies. $\mathsf{finalSafeFlow}(x_{flowT}, x_{flowE}, +x_{action})$ holds when FlexFlow permits the flow tree x_{flowT} and is included in stratum 3. $\mathsf{finalSafeFlow}(x_{flowT}, x_{flowE}, -x_{action})$ holds when FlexFlow prohibits the flow tree x_{flowT} and is included in stratum 4.

Table 1. **Stratification of FlexFlow and Predicates**

Stratum	Predicate	Rules defining predicate
0	application-specific predicates List manipulation predicates	base application-specific relations. recursive list processing rules.
1	safeFlow	Body may contain literals from $\mathbf{SR}_0$.
2	safeFlow*	Body may contain safeFlow*and other literals from $\mathbf{SR}_0$ and $\mathbf{SR}_1$. Occurrences of safeFlow* literal must be positive.
3	finalSafeFlow	The head is of the form finalSafeFlow(flowT,_,+_), the body may contain safeFlow, safeFlow*, and literals from $\mathbf{SR}_i$ for $i \leq 2$.
4	finalSafeFlow	The head is of the form safeFlow*(flowT,_,−_), the body contains just one literal ¬safeFlow*(flowT,_,+_).

3.2. Rules used in FlexFlow

A FlexFlow specification consists of a finite set of Horn clauses (rules) constructed using above mentioned predicates. These rules are constructed so that they form a locally stratified logic program. Obtaining a locally stratified logic program requires that the predicates used in the rules belong to a finite set of *strata,* and the rules follow some syntactic constraints. Generally, predicates in the body of any Horn clause are from the lower strata than that of the head of the clause. The only exception to this rule occurs in stratum 2, where safeFlow* is allowed to appear in the head and the body of a rule. Rules that permit the head predicate to appear in the body have to satisfy further syntactic restrictions that they form a *local stratification.* Namely, the occurrence of safeFlow* in the body cannot be negative. Logically, this restriction implies that every instance of the recursive rule can be *unravelled* in a finite number of steps, where negation at any such *unravelling* is interpreted as failure over *previous unravelling.* FlexFlow is stratified by assigning levels to predicates as shown in Table 1, and the level of a rule is the level of its head predicate. Now we explain rules at each stratum with some examples.

Stratum 0: Rules in this stratum consists of basic facts related to application specific predicates and list processing functions. Following facts relate to the syntax tree example in Figure 1.

$$\begin{aligned} \mathsf{isEnv}(x, [Alice, Bob]) &\leftarrow \\ \mathsf{isEnv}(y, [Alice, Cindy]) &\leftarrow \\ \mathsf{isEnv}(z, [Alice, Bob, Cindy]) &\leftarrow \end{aligned}$$

These rules state that the access control lists of variables x, y and z are [*Alice, Bob*], [*Alice, Cindy*] and [*Alice, Bob, Cindy]* respectively.

Stratum 1: Rules in this stratum have literals from Stratum 0 in their bodies and heads that are instances of safeFlow. Example rules for the syntax tree example of Figure 1 is as follows.

$$\begin{aligned} \mathsf{safeFlow}(x_n, x_e, X_L, +binaryAdd) \leftarrow\ & \mathsf{union}([(u, u_e)], [(v, v_e)], X_L), \mathsf{isEnv}(u, u_e), \\ & \mathsf{isEnv}(v, v_e), \mathsf{intersection}(u_e, v_e, x_e) \end{aligned}$$

The rule says that it is safe for the source list X_L to *binaryAdd* into the root (x_n, x_e) provided that X_L has two elements (u, u_e) and (v, v_e), and the environment of x_e is the intersection of access lists of the sources. This rule also uses the list processing predicates union(A,B,C), intersection(A,B,C) that hold when C is the union/intersection of lists A and B respectively.

Stratum 2: Rules in this stratum have literals from Strata 0 and 1 in their bodies and heads that are instances of safeFlow*. Example rules for the syntax tree of Figure 1 are as follows.

$$\begin{aligned}
\mathsf{safeFlow*}(x_t, x_e, +bTree) \leftarrow\ & \mathsf{safeFlow}(x_n, x_e, [(u, u_e), (v, v_e)], +binaryAdd), \\
& \mathsf{append}((x_n, x_e), [(u, u_e), (v, v_e)], x_t), \\
& \mathsf{union}(u_e, v_e, x_e) \\
\mathsf{safeFlow*}(x_t, x_e, +bTree) \leftarrow\ & \mathsf{safeFlow*}(x_1, x_{e_1}, +bTree), \mathsf{mkBinTree}(x_1, x_2, x_t), \\
& \mathsf{safeFlow*}(x_2, x_{e_2}, +bTree), \mathsf{union}(x_{e_1}, x_{e_2}, x_e)
\end{aligned}$$

The first rule says that a one step flow tree is a safe flow tree. The second rule constructs larger flow trees from smaller ones. The larger tree is made by using the predicate mkBinTree. $\mathsf{mkBinTree}(x_1, x_2, x_t)$ makes a binary tree x_t with first and second children x_1 and x_2 respectively. In addition, the environment of the new tree is the union of environments of the two trees used to make up the larger tree.

Stratum 3: Rules in this stratum may contain literals from stratum 0 through 2 in their bodies but only finalsafeFlow heads that have (+) action terms. They are used to specify conflicts that are resolved in favor of permissions. Example rules for the syntax tree of Figure 1 is as follows.

$$\begin{aligned}
\mathsf{finalSafeFlow}(x_t, x_e, +bTree) \leftarrow\ & \mathsf{isMember}(Alice, x_e), \mathsf{isMember}(Bob, x_e), \\
& \mathsf{safeFlow*}(x_t, x_e, +bTree)
\end{aligned}$$

The rule says that a flow tree is safe provided that Alice and Bob are included in its environment.

Stratum 4: This stratum has one rule only. It is inserted by the FlexFlow system automatically to ensure the completeness . It reads as follows.

$$\mathsf{finalSafeFlow}(x_t, x_e, -x_{action}) \leftarrow \neg\mathsf{finalSafeFlow}(x_t, x_e, +x_{action})$$

This rule says that permissions not derivable using given rules are prohibited by the system.

3.3. Semantics of FlexFlow

The semantics of FlexFlow is given by the well known stable model semantics [8] and well founded model semantics [7] of logic programs. In fact, as we showed in Section 3.2, FlexFlow specifications are locally stratified. This property guarantees that their stable model semantics is equivalent to their well founded semantics, thus ensuring that they have exactly one stable model (this follows from a result of Baral and Subrahmanian [1]).

4. Using FlexFlow to Express Existing Flow Control Models

This section shows how FlexFlow can express existing flow control models and their flow control policies. We choose three different models from previous literatures. We refer the reader to [3] for the reasons of our choice and the difference between the models and FlexFlow.

4.1. The Lattice Based Flow Control Model of Denning [4]

In [4], an information flow model FM is defined as $< N, P, SC, \oplus, \rightarrow >$, where N is a set of *objects, P* is a set of *processes* and *SC* is a set of disjoint *security* classes. $\oplus$ is a binary operator on SC and $\rightarrow$ specifies permissible flows among security classes. That is, for security classes A and B, $A \rightarrow B$ iff information in class A is permitted to flow into class B. Under some assumptions, referred to as *Denning's Axioms,* $< SC, \rightarrow, \oplus >$ form a universally bounded lattice where $< SC, \rightarrow >$ forms a partially ordered set.

Suppose f is an n-ary computable function, and a_i are object belonging to security classes $\underline{a_i}$ for all $i \leq n$. Then the flow control policy enforced by FM is as follows. If a value $f(a_1, \ldots, a_n)$ flows to an object b that is bound to a security class $\underline{b}$, then $\underline{a_1} \oplus \ldots \oplus \underline{a_n} \rightarrow \underline{b}$ must hold.

To specify this flow control policy, we use the following sets. C is a set of classes, Obj is a set of objects, P is a set of processes. In this example, we use *objects* as *nodes* and *classes* as *environments of nodes* and the root's environment as the tree's environment. In addition, we define application specific predicates shown in Table 2. For instance, **dominate**(c, c') says that the class c dominates class c' in the class set C. The FM policy is now formulated as follows.

Table 2. **Predicates used to express Lattice Based Flow Control Model**

Predicate	Argument Types	Intended Meaning
dominate(c, c')	(class, class)	Class c dominate class c' in the class set C.
class(o, c)	(object,class)	Object o belongs to class c.
leastUB$(cList, c)$	(class list,class)	The least upper bound of the elements in the class list $cList$ is c.
setClass(X, Y)	((object,class) pair list, class list)	The classes of objects in X constitute class list Y.

$$\begin{aligned}
\textbf{class}(o_1, c_1) &\leftarrow \\
\textbf{class}(o_2, c_2) &\leftarrow \\
\textbf{dominate}(c_1, c_2) &\leftarrow \\
\textbf{dominate}(x_c, z_c) &\leftarrow \textbf{dominate}(x_c, y_c), \textbf{dominate}(y_c, z_c) \\
\textbf{safeFlow}(x_o, x_c, X, +x_{action}) &\leftarrow \textbf{class}(x_o, x_c), \textbf{setClass}(X, Y), \textbf{leastUB}(Y, x), \\
&\quad\ \textbf{dominate}(x_c, x) \\
\textbf{safeFlow*}([(x_o, x_c) \mid X], x_c, +x_{action}) &\leftarrow \textbf{safeFlow}(x_o, x_c, X, +x_{action})
\end{aligned}$$

$$\mathsf{finalSafeFlow}(X, x_c, +x_{action}) \leftarrow \mathsf{safeFlow}*(X, x_c + x_{action})$$

The first two rules specify that o_1, o_2 are associated with c_1 and c_2 respectively. The third rule says that c_1 dominate c_2 in C. The fourth rule is the transitivity of the dominance relation. The fifth rule says that information from objects in the list X is allowed to flow into x_o iff the least upper bound of the classes of objects of X is dominated by the class of x_o. The sixth rule constructs the depth-one permissible flow tree $[(x_o, x_c) \mid X]$ with environment x_c. Because the flow control policy in this lattice based flow control model has only one-step flows, we don't need to construct flow trees of depth greater that one. Furthermore, the model doesn't allow negative permission, so the conflict resolution policy is simple and given by the last rule.

4.2. The Decentralized Label Model of Mayer and Liskov [11]

The *Decentralized Label Model* [11] of Mayer and Liskov is applied at the runtime data flow analysis level. This model has variables storing values and updating them during a computation. A label contains a list of owners whose data was observed in order to construct the data value in question. Each owner declares a set of principals that may read the value, referred to as the reader set of that owner. Each (owner, reader set) pair is said to constitute a per-principal flow control policy. When a value is read from a variable or an input channel, the value acquires the label of the variable or the input channel. When a value is written to a variable, a new copy of the value is generated with the label of the variable.

The flow control policy used in this model is that when information flows from one object (here, object refers variable or channel) to another, the label of the destination must be more restrictive than the label of the source. A label L_1 is said to be more restrictive than label L_2 iff L_1 contains all the owners of label L_2, and the same or fewer readers for each owner.

To express the policies of [11], we define *Obj, P* and *L* as the sets of variables, principals and labels respectively. We use variables (input and output channels included) as nodes of the flow trees and labels of variables as environments of the nodes. Before specifying flow control policies as rules, we define some application specific predicates as shown in Table 3.

$$\begin{aligned}
\mathsf{safeFlow}(x_o, x_l, Y, +x_{action}) &\leftarrow \mathsf{label}(x_o, x_l), \mathsf{listOfRdSet}(Y, Y'), \mathsf{eRdSet}(x_l, X), \\
&\quad \mathsf{allIntersec}(Y', W), \mathsf{cover}(X, W) \\
\mathsf{safeFlow}*([(x_o, x_l) \mid Y], x_l, +x_{action}) &\leftarrow \mathsf{safeFlow}(x_o, x_l, Y, +x_{action}) \\
\mathsf{finalSafeFlow}(X, x_l, +x_{action}) &\leftarrow \mathsf{safeFlow}*(X, x_l, +x_{action})
\end{aligned}$$

The first rule says that information can flow from objects in the list Y to x_o provided that each of principals in the effective reader set of label x_l can act

Table 3. **Predicates used to express Decentralized Label Model**

Predicate	Parameter Types	Intended Meaning
label(o, l)	(object,label)	l is the label of the object o.
eRdSet(l, X)	(label, principal list)	The effective reader set of label l is X.
cover(X, Y)	(principal list, principal list)	Every principal in X can act for some principals in Y.
listOfRdSet(X, Y)	((obj,label) pair list,RdSet list)	Reader sets of all the labels in X constitute the list Y.
allIntersec(X, Y)	(list of principal list, principal)	The intersection of all the principal lists in X is Y.

for some principals in the intersections of the effective reader sets of labels in *Y*. The second rule constructs the depth one permissible flow tree. The third rule specifies the conflict resolution policy.

4.3. The Flexible Information Flow Control Model of Ferrari et al. [5]

This model provides an approach to control information flow in object-oriented systems that takes into account, beside authorizations on object, also how the information has been obtained and/or transmitted. They coined a notation of *transaction execution tree* to represent the invocation relations of the methods calls initiated by a user. Figure 3 gives an example *transaction execution tree* simplified from the example in [5], where each node is a execution notated by execution id with the method name and the object on which it executed on, and each directed branch is caller-callee relation between the methods.

Based on the transaction execution tree, they define that there is an *information flow* from o_h to o_k by e_h and e_k, written (o_h, e_h, e_k, o_k) if the method of e_h is *read,* the method of e_k is *write,* and there is a transmission channel between e_h and e_k. For example, there are four information flows in the transaction execution tree example we given, which are (o_3, e_3, e_{17}, o_6), (o_4, e_9, e_{17}, o_6), $(o_5, e_{13}, e_{17}, o_6)$, and $(o_5, e_{16}, e_{17}, o_6)$.

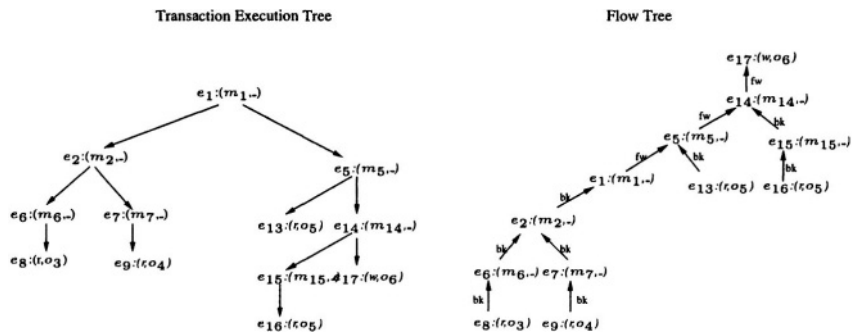

Figure 3. A Transaction Execution Tree and its Flow Tree

In their model, every object has an access control list (ACL) specifying the users allowed to read and write the object. Some methods are associated with waivers. Two kinds of waivers are supported: invoke-waiver (IW), specifying exceptions applicable during a method's execution, and reply (RW), specifying exceptions applicable to the information returned by a method.

Based on the concepts of waivers, they consider each flow under the following policy, referred to as *flexible policy*. A flow (o_h, e_h, e_k, o_k) is safe iff the union of the reader list and the waiver lists for o_h along the transmission channel from o_h up to but excluding o_k subsume the readers list of o_k. The execution of a write method is said to be safe iff all flows ending at it are safe.

In order to express the flexible policy using FlexFlow, we first extract all flow trees from the transaction execution trees using an algorithm called *flow tree generator*[3]. Figure 3 shows a flow tree extracted from the example transaction execution tree. We define four sets, *Obj, U, M, E* which are sets of objects, users, methods and executions respectively. We also define some application predicates which are shown in Table 4. Some of these are not standard list manipulation predicates. Due to the space limitation we refer the reader to [3] for their definitions. Using these predicates some sample rules that fit in stratum 0 of FlexFlew are as follows.

Table 4. Predicates used to express the Flexible Flow Control Model

Predicate	Attribute Types	Intended Meaning
obj(e, o)	(exec., object)	Execution e is executed on object o.
mtd(e, m)	(exec., method)	The method of execution e is method m.
rAcl(o, X)	(object, list of users)	The access control list of object o is X.
rW(m, X)	(method,waiver list)	X is the reply waiver list associated with method m.
iW(m, X)	(method,waiver list)	X is the invoke waiver list associated with method m.
releaseBK(X, m, X')	(flowE, method, flowE)	The result of applying reply waiver of method m to flow tree environment X is X'.
releaseFW(X, m, X')	(flowE, method, flowE)	The result of applying invoke waiver of method m to flow tree environment X is X'.
unionAdd(X, Z)	(list of flowE, flowE)	The union-add of flow environments in X is Z.
eRdSet(X, Y)	(flowE, user list)	The effective readers set of flow tree environment is Y.
subset(X, Y)	(user list, user list)	User list X is subset of user list Y.
bkFlowCH(e_i, e_j)	(exec., exec.)	There is a backward edge from e_j to e_i.
fwFlowCH(e_i, e_j)	(exec., exec.)	There is a forward edge from e_j to e_i.
listOfSafeFlow$((x, e), Y, a)$	((exec.,env.),list, action)	For all $y_i \in Y$, safeFlow$((x, e), y_i, +a)$ holds.
listOfSafeFlow*(X, Y, E, a)	(list, list, list,action)	For all $(x_i, e_i) \in X, y_i \in Y$, and $e_i \in E$, safeFlow*$(y_i, e_i, +a)$ and isHead$((x_i, e_i), y_i)$ hold.
listOfMtd(X, M)	(list, list)	For all $x_i \in X, m_i \in M$, mtd(x_i, m_i) holds.
listOfReleaseBK(X, M, Y)	(list, list, list)	For all $x_i \in X, m_i \in M$, and $y_i \in Y$, releaseBK(x_i, m_i, y_i) holds.

mtd(e_8, r) ←

obj(e_8, o_3) ←

$$\begin{aligned}
\mathsf{rAcl}(o_3, [Ann, Bob, Carol, David]) &\leftarrow \\
\mathsf{rW}(m_3, [[o_1, Frank], [o_3, Frank]]) &\leftarrow \\
\mathsf{iW}(m_3, [[o_4, David], [o_5, David]]) &\leftarrow \\
\mathsf{fwFlowCH}(e_{14}, e_5) &\leftarrow
\end{aligned}$$

The first two rule specify that e_8 is a read method of o_3. The third rule says that [Ann,Bob,Carol,David] is the read access control list of o_3. The fourth and fifth rules say that m_3's reply and invoke waivers are $[[o_1, Frank], [o_3, Frank]]$ and $[[o_4, David], [o_5, David]]$ respectively. The last rule says that there is a one-step forward flow channel from e_5 to e_{14}. One-step information flows are specified using the following two rules.

$$\begin{aligned}
\mathsf{safeFlow}(x_e, v, [(y_e, v)], +bkChannel) &\leftarrow \mathsf{bkFlowCH}(x_e, y_e) \\
\mathsf{safeFlow}(x_e, v, [(y_e, v)], +fwChannel) &\leftarrow \mathsf{fwFlowCH}(x_e, y_e)
\end{aligned}$$

These two rules say that in order to have one-step forward/backward flow, there must be a one-step forward/backward channel from the source to the sink. Permissible information flow trees are constructed using the following recursive rules.

$$\begin{aligned}
\mathsf{safeFlow}*([x_e], [x_o|X], +u) &\leftarrow \mathsf{obj}(x_e, x_o), \mathsf{mtd}(x_e, r), \mathsf{rAcl}(x_o, X) \\
\mathsf{safeFlow}*(X_{flowT}, X_{flowE}, +n) &\leftarrow \mathsf{listOfSafeFlow}((x_e, v), Y, +bkChannel), \\
&\quad \mathsf{listOfSafeFlow}*(Y, Y_{flowT}, Y_{flowE}), \\
&\quad \mathsf{listOfReleaseBK}(Y_{flowE}, Y_m, Y'_{flowE}), \\
&\quad \neg\mathsf{mtd}(x_m, w), \mathsf{listOfMtd}(Y, Y_m), \\
&\quad \mathsf{unionAdd}(Y'_{flowE}, X_{flowE}), \mathsf{mtd}(x_e, x_m), \\
&\quad \mathsf{append}((x_e, v), Y_{flowT}, X_{flowT}) \\
\mathsf{safeFlow}*(X_{flowT}, X_{flowE}, +n) &\leftarrow \mathsf{safeFlow}(x_e, v, [(z_e, v)], +fwChannel), \\
&\quad \mathsf{safeFlow}*(Z_{flowT}, Z_{flowE}, +n), \\
&\quad \mathsf{listOfSafeFlow}((x_e, v), Y, +bkChannel), \\
&\quad \mathsf{listOfSafeFlow}*(Y, Y_{flowT}, Y_{flowE}), \\
&\quad \mathsf{listOfReleaseBK}(Y_{flowE}, Y_m, Y'_{flowE}), \\
&\quad \mathsf{releaseFW}(Z_{flowE}, z_m, Z'_{flowE}), \\
&\quad \mathsf{append}(Y'_{flowE}, z'_{flowE}, X'_{flowE}), \\
&\quad \mathsf{unionAdd}(X'_{flowE}, X_{flowE}), \\
&\quad \mathsf{append}((x_e, v), Z'_{flowT}, X'_{flowT}), \\
&\quad \mathsf{append}(X'_{flowT}, Y_{flowT}, X_{flowT}), \\
&\quad \mathsf{mtd}(z_e, z_m), \mathsf{isHead}((z_e, v), Z_{flowT}), \\
&\quad \mathsf{listOfMtd}(Y, Y_m) \\
\mathsf{safeFlow}*([x_e, [y_e|Tail], X, +n) &\leftarrow \mathsf{safeFlow}(x_e, v, [(y_e, v)], +fwChannel), \\
&\quad \mathsf{mtd}(x_e, w), \mathsf{safeFlow}*([y_e|Tail], Y, +n), \\
&\quad \mathsf{mtd}(y_e, y_m), \mathsf{releaseFW}(Y, y_m, Y'), \\
&\quad \mathsf{eRdSet}(Y', Y''), \mathsf{obj}(x_e, x_o), \\
&\quad \mathsf{rAcl}(x_o, X), \mathsf{subset}(X, Y'') \\
\mathsf{finalSafeFlow}(X_{flowT}, X_{flowE}, +x_{action}) &\leftarrow \mathsf{safeFlow}*(X_{flowT}, X_{flowE}, +x_{action})
\end{aligned}$$

The first rule says that executing a read method m constructs a one node flow tree that has $[o, X]$ as its environment where X is the ACL of o. Second and third rules recursively constructs flow trees that have non-write roots. Note that for any node in any flow tree extracted from a given transaction execution tree, there is at most one forward channel feeding into it. The fourth rule constructs the whole permissible flow tree whose root is a write execution. The last rule resolves conflicts.

5. Related Work

Flow control is a heavily researched area in the context of multi-level security. In the early days, Bell and LaPadula [2] and Denning [4] proposed the lattice based models of information flow. Some flexibility to Denning's lattice based model was later added by Foley [6], where each entity was associated with a label pair instead of a single label.

McCollum et al. [10] presented an Owner Retained Access Control model (ORAC) to control information dissemination. Based on the same idea, Myers and Liskov [11] provided a language-based information flow control model. Samarati et al. [12] presents an information flow control model for object-oriented system. Ferrari et al. [5] extend that model by allowing waivers associated with methods.

Our work has been influenced by the *Flexible Authorization Framework* (FAF) of Jajodia et al. [9]. FAF is proposed for specification of access control policies, but not flow control policies. To our best knowledge, FlexFlow is the first flexible logic framework for flow control policies specification.

6. Conclusions

The FlexFlow framework specifies flow control policies as stratified logic programs consisting of five levels. Both permissions and prohibitions are specifiable in FlexFlow. By expressing flow control policies at the programming language and transaction specification we have shown the generality of FlexFlow.

Acknowledgments

This work was partially supported by the National Science Foundation under grant CCR-0113515.

References

[1] C. Baral and V.S. Subrahmanian. Stable and extension class theory for logic programs and default theories. *Journal of Automated Reasoning,* 8:345–366, 1992.

[2] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations and model. Report M74-244, Mitre Corp., Bedford, MA, 1975.

[3] S. Chen, D. Wijesekera, and S. Jajodia. Flexflow: A flexible flow control policy specification framework. Report ISE-TR-03-04, Center for Secure Information Systems, Fairfax, VA, 2003.

[4] D.E. Denning. A lattice model of secure information flow. *Communication of ACM,* pages 236–243, May 1976.

[5] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proceedings of the IEEE Symposium on Security and Privacy,* pages 130–140, Oakland, CA, May 1997. IEEE.

[6] S.N. Foley. A model for secure information flow. In *Proceedings of the IEEE symposium on Security and Privacy,* Oakland, CA, May 1989.

[7] A.V. Gelder. The alternating fixpoint of logic programs with negation. In *Proc. 8th ACM Symposium on Principles of Database Systems,* pages 1–10, 1989.

[8] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming,* pages 1070–1080,1988.

[9] S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems,* 26(4): 1–57, June 2001.

[10] C.J. McCollum, J.R. Messing, and L. Notargiacomo. Beyond the pale of mac and dac-defining new forms of access control. In *Proceedings of the IEEE symposium on Security and Privacy,* pages 190–200, Oakland, CA, May 1990.

[11] A.C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles,* pages 129–142, Saint-Malo, France, October 1997.

[12] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia. Information flow control in object-oriented systems. *IEEE Transactions on Knowledge and Data Engineering,* 9(4):524–538, July-Aug. 1997.