Static Detection of Security Flaws in Object-Oriented Databases

Keishi Tajima
Research Institute for Mathematical Sciences*
Kyoto University, JAPAN
tajima@kurims.kyoto-u.ac.jp

Abstract

Access control in function granularity is one of the features of many object-oriented databases. In those systems, the users are granted rights to invoke composed functions instead of rights to invoke primitive operations. Although primitive operations are invoked inside composed functions, the users can invoke them only through the granted functions. This achieves access control in abstract operation level. Access control utilizing encapsulated functions, however, easily causes many "security flaws" through which malicious users can bypass the encapsulation and can abuse the primitive operations inside the functions. In this paper, we develop a technique to statically detect such security flaws. First, we design a framework to describe security requirements that should be satisfied. Then, we develop an algorithm that syntactically analyzes program code of the functions and determines whether given security requirements are satisfied or not. This algorithm is sound, that is, whenever there is a security flaw, it detects it.

1 Introduction

User access to a database is either an action to get some information from the database, or an action to give some information to the database in order to make it reflected by the database state. Access control is to impose restrictions on those actions in order to meet the requirements concerned with security. In many theoretical researches on security analysis, those two types of access are represented by read and write operations for simplicity. The basis of this simplification is the fact that any information flow between the users and the database originates in those operations. In practice, however, it is often the case that security requirements cannot be expressed in terms of such simple operations. For example, in some cases a user should be allowed to get just partial information on some data but should not know the exact value of it. In other cases, a user should be allowed to update some data in some specific procedure, but should not be able to write any value he wants. We can impose these kinds of restrictions by defining appropriate functions and by authorizing users to invoke those functions instead of authorizing them to directly execute read or write operations. Those functions read the data but return a processed data through some computation, or they write the data following the required procedure. Although primitive read or write operations are invoked inside those functions, the users can invoke them only indirectly. In other words, the functions encapsulate those primitives into some procedures. The access control in the abstract operation level by using encapsulated functions (or "methods" in the object-oriented terminology) is one of important features of many object-oriented database systems [ADG92, Ber92, GGF93].

We use functions in order to encapsulate some sensitive primitive operations inside them. Therefore, it is essential that those functions certainly hide those primitives. If one can infer the result of a read operation encapsulated in some function, or can control (i.e. can change to any value he wants) the argument of a write operation inside some function, those functions are not actually hiding those operations: the user effectively obtains the same capability that he would have when he were allowed to invoke those primitives directly. Such flaws easily occur especially when the user exercises multiple capabilities together. For example, suppose a stock company has a database about all stockbrokers of the company. In this company, each stockbroker is given a budget for his stock dealing and there is a regulation that the budget of each broker should not be higher than ten times his salary. One clerk is assigned a job to periodically examine whether the budget of each broker is illegally high against this regulation, but he should not be able to know the exact amount of the salary of each broker. Then, the database administrator defines a function that reads out the salary and the budget of a broker, compares them, and returns true or false. The clerk is authorized to invoke this function but is not authorized to directly read the salary data. In this situation, however, if that clerk can know the amount of the budget of some broker, he can know a little about the salary of that broker: "his salary is at least higher (lower) than this". Or in a worse case, if that user can change the amount of the budget

*The author is currently at Kobe University. E-mail address above is still available.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

to any value he wants, he can infer the exact amount of the salary by repeatedly changing the budget to several values and invoking the testing function. Those are security flaws. Another example is concerned with write access. Suppose the salary of each broker is updated once a week to a new value calculated from the budget given to him last week and the profit he made last week. Then, the database administrator defines a function that reads the budget and the profit of each broker, calculates a new salary value, and writes it in. An clerk is authorized to invoke this function. In this situation, if the employee is also able to change the budget of each broker to any value he wants, and as a consequence of it, can change the new salary value to any value he wants, then he can write any value he wants as the new salary. When we use encapsulation of functions for access control, many of these types of security flaws may occur.

In this paper, we develop a technique to statically detect those flaws. We authorize users to invoke functions, and we also describe security requirements as negative authorizations like "one should not be able to infer the result of this read operation", or "one should not be able to control the argument of this write operation". We call the former capability *inferability* and call the latter *controllability*. These two capabilities effectively correspond to the abilities to directly invoke read or write operation. Because read and write operations can be considered as special cases of function invocations, we can naturally generalize the notion of inferability onto returned values of any functions and the notion of controllability onto arguments of any functions. In fact, we sometimes want to encapsulate a composed function into another function and to describe requirements in terms of the encapsulated composed function, such as "one should not be able to infer the result of this function invocation". Then, security requirements are described in the following forms: (1) the user u should not have inferability on the returned value of the function f, or (2) the user u should not have controllability on the argument a of the function f. Inferability on returned values of functions and controllability on arguments of functions precisely represent two kinds of user abilities in database access: (1) the ability to get data from the database and (2) the ability to give data to the database.

Based on these two notions, first, we establish a framework for description of security requirements. To give a formal definition of inferability, we design an inference system that models the inference ability of users. Next, we develop an algorithm that statically detects security flaws violating the given requirements. The main part of the algorithm is another inference system that simulates the inference system above by syntactical analysis of the program code of the functions.

As mentioned in [ADG92, Ber92, GGF93], when we control access in function granularity, there are two ways to impose restrictions: verifying only direct function invocations or verifying indirect invocations as well. The advantage of the former is that we can grant users encapsulated abstract operations, and the advantage of the latter is that security flaws are less likely to occur because every access is verified. We achieve both advantages by choosing the former approach and by providing a static security flaw detection mechanism.

1.1 Related Work

There are many researches on statically determining whether a user can infer sensitive information in the database especially in the context of relational database systems [Bur90, MJ88, LDS+90, JS91, SO91, HD92, QSK+93, Qia94]. Those researches focus on whether users can know the existence of some entities or can make sensitive associations between entities or values in a database, while we focus on different aspect, i.e. whether a user can compute sensitive values from supplied values. [Mor87, Row89, MSS88] propose frameworks to detect the possibility of user inference on sensitive values through the knowledge on semantic dependency or on the integrity constraints defined in the database. On the other hand, our mechanism deals with dependency between arguments or returned values of functions, and data in the database which are referred to in those functions. Although all researches deal with dependency, functions can represent wider range of dependency than semantic dependency or integrity constraints. In fact, our mechanism can include integrity constraints by describing each integrity constraint in the form of a function with no argument and returning a boolean value. On the contrary, a function cannot always be described in the form of a constraint because functions may have arguments, to which users can assign several values, and returned values can be any type. Arguments can be simulated by database values that the users can freely update. In [Mor87, Row89], however, they do not consider the effect of update by the user. [MSS88] takes into consideration only limited kind of situations: situations where users can infer some data by monotonously updating some data. But their model does not analyze whether the user can actually realize needed update, and does not include other kind of cases where update affects on inferability. In our model, the notion of controllability, which is introduced to analyze the user's write capability, is also used to examine more elaborately how the user can update the data, and to investigate how it interacts with inferability.

There are many researches on access control using "views" in object-oriented databases [TYI88, HZ90, AB91, Run92, OT94]. Although the idea of access control using views defined by functions is essentially the same concept with access control using functions, those researches do not discuss security issues. Our technique can be applied to the verification of view definitions in those systems as well. In [DAH+87], they provide a mechanism to automatically compute security levels of computed attributes in views in the context of relational databases. Their method is, however, simply to compute the least upper bound of security levels of all data used in the computation. Therefore, their method

cannot be used for our purpose: to give users not total but partial information on some data through some computations.

[Den76, DD77] proposed techniques to analyze program code in order to detect all flow of information. Their methods is also to compute the least upper bound of the security levels of source data, and therefore, cannot be applied for our purpose.

The rest of this paper is organized as follows. Section 2 briefly explains the basic data model used as the base of the development. Section 3 discusses and formally defines the concept of inferability and controllability. Section 4 describes the algorithm that analyzes program code and detects security flaws. Finally, Section 5 mentions some further issues and concludes this paper.

2 The Basic Model

In this section, we explain the basic model of database on which we develop our framework for access control. Although in this paper we assume a simple data model with mutable objects and classes, the only essential point of our development is that users access data objects by invoking functions. We believe that our mechanism can be translated onto several other data models, such as the relational data model with abstract data types.

The data model is defined as follows:

$$\begin{aligned} scm &= \langle\{c\_name : [att : t, \ldots, att : t]\}, \\ &\quad \{\langle f\_name(arg : t, \ldots, arg : t) : t,\ body\rangle\}\rangle \\ t &= b \mid c\_name \mid \{t\} \\ db &= \langle\{\langle c\_name, \{obj\}\rangle\}, \\ &\quad \{\langle u\_name,\ \{f\_name\}\rangle\}\rangle \end{aligned}$$

A schema scm is a pair of a set of class definitions and a set of function definitions. Each class definition has the form $c\_name : [att : t, \ldots, att : t]$, which declares that instances of the class c_name, i.e. objects belonging to c_name, have attributes att of type t. Objects are mutable entities, and we can read out current values of their attributes, update their attributes, pass them to functions as arguments, and store them in attributes of other objects. Each function definition is a pair of a signature of the form $f\_name(arg : t, \ldots, arg : t) : t$ and the definition of its body. The users access the database by invoking those functions. We call them *access functions*. We can interpret an access function also as a "method" by regarding the first argument as the receiver. Some additional consideration that would be needed if we introduced subtyping and overloading will be explained later. t stands for types in this model. t is either a basic type b such as integer, a class name c_name that is interpreted as the type of its instances, or a set type of some type. A database db is a pair of (1) a set of pairs of a class name and its extension, i.e. a set of all its instances, and (2) a set of pairs of a user name and his capability list. A capability list is a set of all access function names (or names of special functions explained below) that the user is allowed to invoke in the query. (The query language is defined later in this section.)

Bodies of access functions are described using the *function definition language* defined by the following syntax:

$$\begin{aligned} e \ ::= \ & c \mid a \mid f_b(e,\ldots,e) \mid f_a(e,\ldots,e) \mid \\ & \mathsf{r}\_att(e) \mid \mathsf{w}\_att(e, e) \end{aligned}$$

c stands for constants, a stands for the arguments of the access function. $f_b(e,\ldots,e)$ is an invocation of a basic function f_b with arguments $e,\ldots,e$. Basic functions are primitive operations on basic types, such as addition on integers. $f_a(e,\ldots,e)$ is an invocation of another already defined access function f_a. We do not consider recursive functions. r_*att* and w_*att* are *special functions* that read or write attributes of objects. For example, r_salary(x) returns the current value of the attribute salary of the object x, and w_salary(x, 100) writes 100 into the attribute salary of the object x and returns a special value null.

Other than those constructs, the complete version of our development includes persistent global variables, to which we can store any object, a special function to create new objects, let construct to define a local variable, and two special functions to get inputs from the console and to output values to the console. In this paper, however, we omit them for the brevity.

The users issue query using the following SQL-like query language:

select $item, \ldots, item$ from $A_1 \in C_1, \ldots, A_n \in C_n$ where $condition$

In this syntax, $C_1, \ldots, C_n$ is a class name. $A_1, \ldots, A_n$ are called from-*clause variables* and are bound to each combination of instances of $C_1, \ldots, C_n$. $item$ is $f(v,\ldots,v)$ where f is an access function or a special function (r_*att* or w_*att*) and where v is either a constant of some basic type or one of $A_1, \ldots, A_n$. If object identifiers have some printable form, such as ⟨id:730710⟩, we can also use them in place of v. In this development, however, we assume object identifiers do not have any printable form. We explain the reason of this choice later in Section 3. Items in a select clause are evaluated in order from left to right. $condition$ consists of boolean terms connected by and and or where each boolean term has the form either of "$f(v,\ldots,v)$ op v" or "$f(v,\ldots,v)$ op $f(v,\ldots,v)$". op is a binary predicate for basic types, such as $>$ for integers. For example, if the class Person:[name:*string*, age:*int*, ...] and the access function profile(x:*Person*):*string* are defined, and a user has r_name, profile, and r_age in his capability list, he can issue a query;

select r_name(p), profile(p) from p ∈ Person where r_age(p) > 20

which returns a set of pairs of a name and a profile for all Person instances whose age are greater then 20. select ...

construct can be nested, and set valued functions (or read operations of set valued attributes) can be used in place of class names in from clause. For example, suppose $Person$ has an attribute child:$\{Person\}$. Then, the query below returns a set of names of children of a person named 'John':

```
select (select r_name(q) from q ∈ child(p))
from p ∈ Person where r_name(p) = 'John'
```

3 Specification of Security Requirements

In this section, we explain how we describe security requirements using the notion of inferability and controllability. We discuss the properties of those two kinds of capability and give the formal semantics of them.

3.1 Basic Concepts

In Section 1, we introduced the notion of inferability on returned values and controllability on arguments as ability effectively equivalent to ability to invoke functions directly. Controllability on an argument intuitively means that one can use any value he wants as that argument. Having controllability on an argument, therefore, implies being able to change the value of that argument to any value. We call this ability *alterability*. Alterability only, however, is not enough to imply controllability. If one can change a value of an argument indirectly, but cannot know which value it takes now, it is similar to walking in the dark. He can move, but he cannot know where he is now. For example, suppose there is an access function f(x, o) defined as g(+(x, r_age(o)), and one user is allowed to invoke f directly in the query. Then, he can change the value of the argument of g, i.e. the value of the expression +(x, r_age(o)), to any values by changing the value of x. However, if he cannot know the value of r_age(o), he cannot know the current value of the argument of g. We consider, therefore, having controllability equals to having both inferability and alterability. From now on, we decompose controllability into these two capabilities.

We further classify inferability into the ability to infer the exact value, *total inferability*, and the ability to infer a set of values to which the value must belong, *partial inferability*. Partial inferability is, in other words, the ability to infer at least one value that an expression can NOT be. Similarly, we classify alterability into the ability to change the value to any value of its type, *total alterability*, and the ability to change the value only within some limited subset, *partial alterability*. Total inferability implies partial inferability, and total alterability do partial alterability.

Using these notions, we describe security requirements in the following syntax:

$$
\begin{array}{rcl}
req & ::= & (u, f(x_1 : clist, \ldots, x_n : clist) : clist) \\
clist & ::= & cap : cap : \ldots : cap \\
cap & ::= & \textbf{ti} \mid \textbf{pi} \mid \textbf{ta} \mid \textbf{pa}
\end{array}
$$

A requirement $(u, f(x_1 : c_1^1 \cdots : c_1^{m_1}, \ldots, x_n : c_n^1 \cdots : c_n^{m_n}) : c_0^1 \cdots : c_0^{m_0}$ means that the user u should not be able to invoke the function f in a context where he can simultaneously achieve all specified capabilities c_i^j on each argument and on the returned value. f can be any of basic functions, access functions, or special functions explained in Section 2. Capabilities are total inferability **ti**, partial inferability **pi**, total alterability **ta**, or partial alterability **pa**. We give the formal semantics of the description of security requirements later in this section.

We show examples of security requirements using the examples explained in Section 1. Suppose the class Broker:[name:$string$, salary:int, budget:int, profit:int] representing stockbrokers is defined. The administrator defines the access function checkBudget as below:

```
checkBudget(broker:Broker):bool
    >=(r_budget(broker), *(10, r_salary(broker)))
```

The first line is the signature of this function, and the following line is the body. This function takes an argument broker, reads out budget and salary of it, compares the budget with ten times the salary, and returns true or false. A user u is authorized to invoke this function, but should not be able to know the exact amount of each broker's salary. This requirement is described as (u, r_salary(x):**ti**). As we explained before, a flaw occurs if u can control (i.e. can infer and can alter) the value of r_budget(broker). For example, if he also has w_budget in his capability list, he can infer the salary of each broker by issuing the query below:

```
select w_budget(b, 1), checkBudget(b),
       w_budget(b, 2), checkBudget(b), ...
from b ∈ Broker where r_name(b) = 'John'
```

which will yield a set {(null, false, ..., null, false, null, true, ...)}. (Here, we assume there is only one broker named John.) We detect this security flaw by examining whether the requirement above is satisfied or not. Similarly, the administrator defines the access function updateSalary as below:

```
updateSalary(broker:Broker):null
    w_salary(broker, calcSalary(r_budget(broker),
                                r_profit(broker)))
```

This function takes a broker object, read budget and profit of it, calculates his new salary by invoking another access function calcSalary, and writes the result to salary. A user u is authorized to invoke this access function but should not be able to change the new value of salary. In this case, however, if the user can alter the value of the expression r_budget(broker), and as a consequence of it, can alter the returned value of calcSalary, he can eventually alter the value of salary, which means there is a security flaw. This flaw can be detected by describing the security requirement (u, w_salary(a, v):**pa**), and by examining whether this requirement is satisfied or not.

If we introduced subtyping and overloading, a single access function name would correspond to multiple implemen-

tations defined on different subclasses. Even in such a situation, it is natural to interpret those security requirements as referring not to specific implementations but to conceptual functionalities. We, therefore, interpret each security requirement as requiring to be satisfied by every implementations of the function name. Under this interpretation, we determine whether the requirement is satisfied or not by examining all implementations of all possible subclasses. That process cannot be infinite because our language is recursion-free. If we also introduced recursion, the analysis would be quite hard, but that difficulty is common to various static analysis of languages with overloading and recursion.

Note that we assume the users can read the program code of the access functions. Even if we prohibit it, the users should know the semantics of the access functions in order to use them, and therefore, can infer the contents of the access functions. To take that knowledge into consideration, we assume that the users can read the program code.

3.2 Causality between capabilities

As we can see in the examples above, in order to analyze inferability and alterability, we must consider inferability and alterability on every subexpression in the program code, and must trace how they are propagated to other subexpressions. Therefore, we generalize the notions of inferability and controllability to any expressions of the function definition language. Before giving formal semantics of them, we give some intuitive discussion about causality of those two kinds of capability.

There are two kinds of expressions, those of basic types and those of object types. First, we consider capability on basic type expressions. The base cases of inferability on basic type expressions are expressions whose values are directly shown to the user, i.e. constants in the program code, or the arguments and the returned values of the functions directly invoked in the query. Inferability on basic type expressions is propagated through the "dependency" between arguments and returned values of basic functions. If the relation between the values of two expressions is one-to-one correspondence, total inferability on either expression causes total inferability on the other. If it is one-to-many correspondence, total inferability on the side of "many" causes total inferability on the other side, and total inferability on the "one" side causes partial inferability on the other side. If it is many-to-many, total inferability on either side causes partial inferability on the other. Another property that transmits inferability is equality. If the user knows that the values of two expressions must be the same, then inferability on either one causes inferability on the other. The user can know that the values of two expressions are equal if (1) two expressions denote the same variable, or (2) they are the value of the same attribute of the same object and no update occurs between those two expressions.

Similarly, alterability on basic type expressions is propagated through dependency between returned values and arguments of basic functions. Suppose there is a expression $f(e_1, \ldots, e_n)$. If the returned value of f changes to all values of its type while the value of e_1 is changing to all values of its type, then total alterability on e_1 causes total alterability on $f(e_1, \ldots, e_n)$. If the returned value of f changes to only limited values, only partial alterability on $f(e_1, \ldots, e_n)$ is caused. Alterability is also propagated through persistent data. If one has alterability on the argument of some write operation, he has alterability on the returned value of the following read operations that will read out the value written by that write operation. The user can alter the result of read operations also by changing the objects to be accessed, i.e. by utilizing alterability on the argument of the read operations. The base cases of alterability are expressions whose values are directly specified by the user, that is, the arguments of the functions directly invoked in the query.

Alterability can be analyzed independent of inferability because alterability is only caused by another alterability. Inferability never causes alterability. On the other hand, alterability plays a role to cause inferability. Suppose inferability on $e_1, \ldots, e_n$ causes partial inferability on e, that is, one can infer some set of values about e. If he can change $e_1, \ldots, e_n$ to several values, and can infer different subsets from different values of $e_1, \ldots, e_n$, then he can infer that the value of e must be in the intersection of those sets. If the intersection is a singleton set, he can infer the exact value of e. For example, suppose there is an expression >=(x, y). Total inferability on the whole expression and on x cause partial inferability on y. If one also has alterability on x, then he can infer the exact value of y by repeatedly changing x to several values and check the result of >=(x, y). Other examples are the division and the reminder operator on integer.

Next, we consider capability on expressions of object types. The actual value of an object type expression is some form of object identifier. We can consider two kinds of situations. One is where object identifiers have some printable form, such as ⟨id:730710⟩, and another is where they do not have any printable form. In the former case, the user can directly specify an object in a query, and when the result of a query is an object, it can be output to output devices. In such a situation, capability on object type expressions can be treated in the same way as that on basic type expressions. Because the development for this situation is rather simple, we assume the latter case in this development. In the latter case, when the result of a query was an object, it may be shown in some form like ⟨a Person object⟩, and the only way to pass an object to function in a query is to give it through from-clause variables. For example, in the example of query in Section 2, the argument for r_name is given through the from-clause variable p. In such a situation, inferability on object identifiers does not make sense. Even in this situation, however, we can recognize equality between two objects. For example, in the query in Section 2, we can recognize that p in r_name(p) and p in profile(p) is identical within each evaluation of the select clause for one broker object. Similarly, in the access function checkBudget

above, we can recognize that broker in r_budget(broker) and broker in r_salary(broker) are identical. As for alterability, we consider a user has total alterability on arguments of access functions directly invoked in a query even when the arguments are of object types. For example, in the query in Section 2, the user can alter an object passed to profile to any Person instance by changing the predicate in where clause (or rather he can invoke profile for all Person objects simply by using true in where clause). Therefore, he has alterability on the argument of profile.

3.3 Formalization

Now we give the formal semantics of the capability and the security requirement descriptions. We define the semantics of inferability by defining an inference system that formulates the inference ability of the users.

First, we clarify what a user can do in queries by select ... from ... where construct. In select clause, a user can invoke the access functions or special functions in his capability list. Although he can also get some information or can update some data within where clause as well, what he can do in the where clause is a subset of what he can do in select clause. Therefore, what a user can do in a query is essentially to invoke a sequence of functions in his capability list. In those invocations, as explained before, a user can pass arbitrary basic values and arbitrary objects as the arguments, and he can make two different arguments be the same object by using the same from-clause variable for those arguments.

To describe the semantics of capability, we introduce some notions. *A function sequence available to the user* u is a sequence of some functions in the capability list of u. These sequences correspond to what u can do in queries. $\langle f_1, \ldots, f_n \rangle$ denotes a function sequence consisting of $f_1, \ldots, f_n$. It may include the same function more than twice. $\mathcal{L}(u)$ denotes the set of all function sequences available to the user u. Given a sequence, we unfold each function in the sequence with replacing it to its body. We also recursively unfold all access function invocations inside it by replacing an access function invocation $f(e_1, \ldots, e_n)$ with an expression let(f) $x_1 = e_1$, $x_n = e_n$ in e end, where e is a body of f and $x_1, \ldots, x_n$ are argument variables of f. We use let(f) to record that this expression was an invocation of f. Then, we number all subexpressions like ${}^k e$ where k is a serial number corresponding to the order of the evaluation in the actual execution of the sequence. For example, if L =⟨f,f⟩, and if f and g are defined as f(x) = +(g(x),1) and g(y) = r_age(y), the unfolded and numbered form of L is;

⟨^{6}+(^{4}let(g) y = ^{1}x in ^{3}r_age(^{2}y) end, ^{5}1),
^{12}+(^{10}let(g) y = ^{7}x in ^{9}r_age(^{8}y) end, ^{11}1)⟩

We number all subexpressions in order to distinguish different occurrences of the syntactically same subexpressions. Next, $\mathcal{S}(L)$ denotes the set of all the numbered subexpressions in the sequence L. For example, $\mathcal{S}(L)$ for L above is

{^{1}x, ^{2}y, ^{3}r_age(^{2}y), ^{4}let(g) y=^{1}x in ^{3}r_age(^{2}y) end,
^{5}1, ^{6}+(^{4}let(g) y=^{1}x in ^{3}r_age(^{2}y) end, ^{5}1),
^{7}x, ^{8}y, ^{9}r_age(^{8}y), ^{10}let(g) y=^{7}x in ^{9}r_age(^{8}y) end,
^{11}1, ^{12}+(^{10}let(g) y=^{7}x in ^{9}r_age(^{8}y) end, ^{11}1) }

Because the signature of every access function is defined in the schema, the type of any subexpression can be statically determined. $Dom({}^k e)$ denotes the set of all values of the type of ${}^k e$.

Let $L = \langle f_1, \ldots, f_n \rangle$. Then *an execution instance of* L is a tuple of the form $\langle D,\ f_1(v_1^1, \ldots, v_1^{m_1}) = r_1,\ \ldots,\ f_n(v_n^1, \ldots, v_n^{m_n}) = r_n \rangle$, which represents an execution of L with D as the initial database state, with $v_1^1, \ldots, v_1^{m_1}, \ldots, v_n^1, \ldots, v_n^{m_n}$ as the arguments, and resulting to the returned values $r_1, \ldots, r_n$. If every r_i equals to the result of the actual execution, that execution instance is said to be *valid*. $\mathcal{E}(D, L)$ denotes the set of all valid execution instances of L with D as the initial database state. $[\![{}^k e]\!]_E$ denotes the value to which ${}^k e$ would be evaluated in the actual execution of the execution instance E. For example, let f(x) be defined as +(x,1), let L be ⟨f⟩ and unfolded into ⟨^{3}+(^{1}x,^{2}1)⟩, and let E be ⟨D, f(3)= 4⟩, then E is a valid execution instance of L, and $[\![{}^1\mathsf{x}]\!]_E = 3$.

Using these notations, we define the semantics of the security requirement descriptions as follows:

Definition 1 *A requirement* $(u, f(x_1 : c_1^1 \cdots : c_1^{m_1}, \cdots, x_n : c_n^1 \cdots : c_n^{m_n}) : c_0^1 \cdots : c_0^{m_0})$ *is not satisfied iff the following holds:*

$\exists D.\ \exists L \in \mathcal{L}(u).$
$\exists\ {}^{k_0}$let(f) $x_1 = {}^{k_1}e_1, \ldots, x_n = {}^{k_n}e_n$ in ... end $\in \mathcal{S}(L).$
$\{\forall i (1 \le i \le n). \forall j (1 \le j \le m_i).\ Can(D, L, c_i^j, {}^{k_i}e_i)\} \wedge$
$\{\forall j (1 \le j \le m_0).\ Can(D,\ L,\ c_0^j,\ {}^{k_0}$let($f$) ... end$)\}$ □

Here, $Can(D,\ L,\ c,\ {}^k e)$, which is formally defined below, intuitively means that the capability c on ${}^k e$ can be achieved in L when the initial database state is D. The definition above says if there exists a database state, if there exists a function sequence available to u which includes the subexpression ^{k_0}let(f) $x_1 = {}^{k_1}e_1, \ldots, x_n = {}^{k_n}e_n$ in ... end, and if all the specified capabilities are achieved on that subexpression, then the requirement is not satisfied. Another considerable way of the definitions is to use $\forall D$ instead of $\exists D$. Although we do not deal with those definitions in this paper, we can develop a similar (but a little more complicated) flaw detection algorithm based on those definitions as well. If f is not an access function but a special function, then ^{k_0}let(f) ... in ... end in the definition above is replaced with ${}^{k_0} f({}^{k_1}e_1, \ldots, {}^{k_n}e_n)$.

$Can(D,\ L,\ c,\ {}^k e)$ is defined for each capabilities as follows:

Definition 2 $Can(D,\ L,$ **ta**, ${}^k e)$ *holds iff:*
$\forall v \in Dom({}^k e).\ \exists E \in \mathcal{E}(D, L).\ [\![{}^k e]\!]_E = v$ □

Definition 3 $Can(D,\ L,$ **pa**, ${}^k e)$ *holds iff:*
$\exists v_1, v_2 \in Dom({}^k e).\ (v_1 \neq v_2)\ \forall v \in \{v_1, v_2\}.$
$\exists E \in \mathcal{E}(D, L).\ [\![{}^k e]\!]_E = v$ □

1, axioms for $\in$

$\rightarrow [\langle {}^k c\rangle \in \{c\}]$

$\rightarrow [\langle {}^k x\rangle \in \{v_i^j\}]$ (if x is the jth argument of f_i and has a basic type)

$\rightarrow [\langle {}^k e\rangle \in \{r_i\}]$ (if ${}^k e$ is the entire body of f_i and has a basic type)

$\rightarrow [\langle {}^k e\rangle \in Dom({}^k e)]$ (if ${}^k e$ has a basic type)

$\rightarrow [\langle {}^{k_1} e_1, \ldots, {}^{k_n} e_n, {}^{k_0} f_b({}^{k_1} e_1, \ldots, {}^{k_n} e_n)\rangle \in \{\langle v_1, \ldots, v_n, r\rangle \mid f_b(v_1, \ldots, v_n) = r\}]$

2, axioms for $=$

$\rightarrow [{}^{k_1} x = {}^{k_2} x]$ (if they are different occurrences of the same argument variable)

$\rightarrow [{}^{k_1} x_1 = {}^{k_2} x_2]$ (if they are passed values through the same from-clause variable)

$\rightarrow [{}^{k_1} x = {}^{k_2} e]$ (if let(f) $\ldots, x = {}^{k_2} e, \ldots$ in $\ldots$ end$\in \mathcal{S}(E)$ and ${}^{k_1} x$ is in the scope of that x)

$\rightarrow [{}^{k_1} e = {}^{k_2}$let($F$) $\ldots$ in ${}^{k_1} e$ end]

3, rules for $\in$: join and projection

$[\langle {}^{k_\alpha} e_\alpha, {}^{k_\beta} e_\beta\rangle \in S_1], [\langle {}^{k_\beta} e_\beta, {}^{k_\gamma} e_\gamma\rangle \in S_2] \rightarrow [\langle {}^{k_1} e_1, \ldots, {}^{k_n} e_n\rangle \in \{\langle v_1, \ldots, v_n\rangle \mid \langle v_\alpha, v_\beta\rangle \in S_1, \langle v_\beta, v_\gamma\rangle \in S_2\}]$

$[\langle {}^{k_1} e_1, \ldots, {}^{k_n} e_n\rangle \in S] \rightarrow [\langle {}^{k_\alpha} e_\alpha\rangle \in \{\langle v_\alpha\rangle \mid \exists \langle v_\beta\rangle .\, \langle v_1, \ldots, v_n\rangle \in S\}]$

4, rules for $=$

$[{}^{k_1} e_1 = {}^{k_2} e_2] \rightarrow [{}^{k_3} e_3 = {}^{k_4}r_att({}^{k_2} e_2)]$ (if ^{k_5}w_att$({}^{k_1} e_1, {}^{k_3} e_3) \in \mathcal{S}(E)$, $k_5 < k_4$)

$[{}^{k_1} e_1 = {}^{k_2} e_2] \rightarrow [{}^{k_3}r_att({}^{k_1} e_1) = {}^{k_4}r_att({}^{k_2} e_2)]$

$[{}^{k_1} e_1 = {}^{k_2} e_2] \rightarrow [{}^{k_2} e_2 = {}^{k_1} e_1]$

$[{}^{k_1} e_1 = {}^{k_2} e_2], [{}^{k_2} e_2 = {}^{k_3} e_3] \rightarrow [{}^{k_1} e_1 = {}^{k_3} e_3]$

5, rule for $\in$ based on $=$

$[{}^{k_1} e_1 = {}^{k_2} e_2] \rightarrow [\langle {}^{k_1} e_1, {}^{k_2} e_2\rangle \in \{\langle v, v\rangle | v \in Dom({}^k_1 e_1)\}]$

Table 1: The axioms and the inference rules of $\mathcal{I}(E)$

Definition 4 *$Can(D, L,$ **ti**, ${}^k e)$ holds iff:*
$\exists v \in Dom({}^k e).\ \exists E \in \mathcal{E}(D, L)$.
the inference system $\mathcal{I}(E)$ defined below deduce $[{}^k e \in \{v\}]$ □

Definition 5 *$Can(D, L,$ **pi**, ${}^k e)$ holds iff:*
$\exists S \subset Dom({}^k e).\ \exists E \in \mathcal{E}(D, L)$.
the inference system $\mathcal{I}(E)$ defined below deduce $[{}^k e \in S]$ □

Note that $\subset$ in the definitions above means non-equal subset. The definition of total alterability says "for any value v in $Dom({}^k e)$, one can make ${}^k e$ be evaluated to v by executing L with some arguments". The definition of partial alterability says "there exist at least two values v_1 and v_2 in $Dom({}^k e)$, and one can make ${}^k e$ be evaluated to either of them by executing L with some arguments". Inferabilities are defined by using $\mathcal{I}(E)$. $\mathcal{I}(E)$ is an inference system that formulates the inference that the users do from the observation of the execution of E. It performs inference on the terms defined by the syntax below:

$$term \ ::= \ [\langle {}^{k_1} e_1, \ldots, {}^{k_n} e_n\rangle \in S] \mid [{}^{k_1} e_1 = {}^{k_2} e_2]$$

$\langle {}^{k_1} e_1, \ldots, {}^{k_n} e_n\rangle$ denotes tuples of any number of expressions in $\mathcal{S}(E)$, and $[\langle {}^{k_1} e_1, \ldots, {}^{k_n} e_n\rangle \in S]$ is a term saying that users can infer that $\langle [\![{}^{k_1} e_1]\!]_E, \ldots, [\![{}^{k_1} e_1]\!]_E\rangle \in S$, for some S which is a subset of $Dom({}^{k_1} e_1) \times \cdots \times Dom({}^{k_n} e_n)$. $[{}^{k_1} e_1 = {}^{k_2} e_2]$ is a term saying that users can infer that $[\![{}^{k_1} e_1]\!]_E = [\![{}^{k_2} e_2]\!]_E$.

Let $E = \langle D, f_1(v_1^1, \ldots, v_1^{m_1}) = r_1, \ldots, f_n(v_n^1, \ldots, v_n^{m_n}) = r_n\rangle$. Then, the axioms and the inference rules of $\mathcal{I}(E)$ are as listed in Table 1. In that description, we use some macro expressions. When we use α and β (or α, β, and γ), it means that the inference rule holds for any A, B (or A, B, C) such that $A \cap B = \emptyset$ $(= B \cap C = C \cap A)$ and $A \cup B$ $(\cup C) = \{1, \ldots, n\}$. e_α means the row of e_α for all α in A. For example, if $A = \{1, 2\}$, then e_α means "e_1, e_2".

Base cases of $\in$-terms are the constants in the program code c, the arguments v_i^j or the returned values r_i of the functions directly invoked by the user, and the constraints on the arguments and the returned values of basic functions f_b. Starting those terms, the users proceed inference using the rules in 3. Those two rules in 3 intuitively correspond to join and projection for $\in$-terms. The users also use the knowledge on equality of two expressions as shown in the rule 5. The axioms 2 say that two expressions are equal (1) if they are the same argument variable, (2) if they are different argument variables given values through the same from-clause variable, (3) if they are an argument variable of indirectly invoked functions and an expression bound to that variable, or (4) if they are a body of an indirectly invoked function and its returning value. The rules in 4 also say that two expressions are equal if they are values of the same attribute of the same object. If update has occurred, two values of the same attribute of the same object are not necessarily same. In a model that allows multiple attributes

to share an object, the users are not always able to determine whether an attribute has been updated or not, but sometimes they are. To include cases where they can determine it, we assume that the values of the same attribute of the same object are always same.

4 An Algorithm for Flaw Detection

Although we gave formal semantics of security requirements in the last section, we cannot determine whether given requirements are satisfied or not by directly following that definition because that definition includes the notion of infinite sequences of functions. Examining all possible initial database state is also impractical. In this section, we develop an algorithm that detects security flaws in a reasonable amount of computation. This algorithm syntactically analyze program code using an inference system that simulates $\mathcal{I}(E)$ by taking some pessimistic assumptions. This algorithm is sound, that is, this algorithm always judges that the requirement is not satisfied when really it is not.

4.1 The Algorithm

Instead of a sequence of functions, which can be infinite, our algorithm considers a set of all the functions in the user's capability list, which must be finite. Given a set of function, our algorithm determines whether there exist a function sequence and its execution instance that consists of only those functions in the set and where the user can achieve each capabilities on each expressions. We pessimistically assume that when the user can achieve some capabilities separately, there always exist one initial database state, one function sequence, and one execution instance of it, with which he can achieve all those capabilities simultaneously. (This, in fact, does not always hold. For example, if a user can achieve one capability only when the initial database state is D_1 and can achieve another capability only when the initial database state is $D_2(=\neq D_1)$, he cannot achieve those two capability simultaneously.) Based on this assumption, our algorithm separately determines whether users can achieve each capability.

As a first step, in the same way we did for function sequences, we recursively unfold all access function invocations in the given set of functions, and number all subexpressions like ${}_l e$. $\mathcal{S}'(F)$ denotes a set of all the numbered subexpressions in the set of function F. For example, let F be {f(x), r_name(person)} and f(x) be +(r_age(x), 1). Then unfolded and numbered form of F is:

$$\{{}_4+({}_2\text{r\_age}({}_1\text{x}), {}_3 1), {}_6\text{r\_name}({}_5\text{person})\}$$

and $\mathcal{S}'(F)$ is

$$\{{}_1\text{x}, {}_2\text{r\_age}({}_1\text{x}), {}_3 1, {}_4+({}_2\text{r\_age}({}_1\text{x}), {}_3 1), {}_5\text{person}, {}_6\text{r\_name}({}_5\text{person})\}$$

Note that ${}_l e$ denotes each occurrence in some set F, and is different from ${}^k e$ denoting each occurrence in some sequence L. Each ${}^k e$ corresponds to one ${}_l e$ in $\mathcal{S}'(F)$ while each ${}_l e$ may correspond to multiple ${}^k e$ in one L. We call those ${}^k e$ *correspondents of* ${}_l e$.

Then we develop an inference system $\mathcal{J}(F)$ that syntactically analyzes program code of functions in F, and pessimistically determine whether the user can achieve each capability on each ${}_l e$. $\mathcal{J}(F)$ performs inference on terms defined by the following syntax:

$$\begin{aligned} term &::= \mathbf{ta}[{}_l e] \mid \mathbf{pa}[{}_l e] \mid \\ &\quad \mathbf{ti}[{}_l e, num, dir] \mid \mathbf{pi}[{}_l e, num, dir] \mid \\ &\quad \mathbf{pi}*[\langle {}_{l_1} e_1, \ldots, {}_{l_n} e_n\rangle, num, dir] \mid \\ &\quad =[{}_{l_1} e_1,\ {}_{l_2} e_2] \\ dir &::= + \mid - \end{aligned}$$

ta, **pa**, **ti**, and **pi** are terms saying that there may exist a function sequence L where he can achieve each capability on some ${}^k e$ which is correspondent of ${}_l e$. **pi**$*$ is a term saying that $\mathcal{I}(E)$ may deduce $[\langle {}^{k_1} e_1, \ldots, {}^{k_n} e_n\rangle \in S]$ with some S such that $S \subset Dom({}^{k_1} e_1) \times \cdots \times Dom({}^{k_n} e_n)$ but $\{v_i \mid \exists \langle \ldots, v_i, \ldots\rangle \in S\} = Dom({}^{k_i} e_i)$ for any i. In other words, this term says that the user may not have partial inferability on any ${}^k e_i$ but can infer some subset for their combination $\langle {}^{k_1} e_1, \ldots, {}^{k_1} e_n\rangle$. **ti**, **pi**, and **pi**$*$ have extra fields num and dir, which are used to record how that inferability has been achieved. Inferability originates in (1) inference on the returned value of some ${}_l f_b$ from the knowledge on its arguments, or (2) inference on the arguments e of ${}_l f_b$ using the knowledge on its other arguments and its returned value. When the inferability is achieved through the former type of inference, we record it using l and $+$, and when it is achieved through the latter type of inference, we record it using l and $-$. The reason why we need to record them is explained later. $=[{}_{l_1} e_1,\ {}_{l_2} e_2]$ is a term saying that there exists a function sequence where the users can deduce $[{}^{k_1} e_1 = {}^{k_2} e_2]$ where ${}^{k_1} e_1$ and ${}^{k_2} e_2$ are correspondent of ${}_{l_1} e_1$ and ${}_{l_2} e_2$ respectively.

When defining the axioms and the inference rules of $\mathcal{J}(F)$, we take the following pessimistic assumptions:

- $\forall L.\ \forall {}^k e \in \mathcal{S}(L). \forall v \in Dom({}^k e). \exists D. \exists E \in \mathcal{E}(D, L).\ [\![{}^k e]\!]_E = v$
- if $\mathcal{I}(E)$ can deduce $[{}^k e \in S]$ and $[{}^k e \in S']$ through two different ways, $S \cap S'$ may be a singleton set. In other words, if a user can achieve partial inferability through two different ways, he may have total inferability.
- when there are two read operations (or one write operation and one read operation) on the same attributes, update obstructing the inference does not occur between those two operations.

The first assumption means we assume each expression may have any values in its domain.

We need to record how partial inferability is caused for two reasons. The first reason is that we assumed that if the user can deduce $[{}^k e \in S]$ through two "different" ways, the

1, axioms and rules for alterability

$\rightarrow$ **ta**$[_l x]$ (x is an argument variable of an outer-most function)

pa$[_l e_1] \rightarrow$ **ta**[r_$att(_l e_1)$]

pa$[_l e_1] \rightarrow$ **ta**[r_$att(e_2)$] (if w_$att(_l e_1, e_3) \in \mathcal{S}'(F)$)

ta$[_l e_3] \rightarrow$ **ta**[r_$att(e_2)$] (if w_$att(e_1, _l e_3) \in \mathcal{S}'(F)$)

pa$[_l e_3] \rightarrow$ **pa**[r_$att(e_2)$] (if w_$att(e_1, _l e_3) \in \mathcal{S}'(F)$)

ta$[_{l_1} e] \rightarrow$ **ta**$[_{l_2} x]$ (if let $\ldots, x = {}_{l_1}e, \ldots$ in $\ldots$ end $\in \mathcal{S}'(F)$ and $_{l_2}x$ in in the scope of that x)

pa$[_{l_1} e] \rightarrow$ **pa**$[_{l_2} x]$ (if let $\ldots, x = {}_{l_1}e, \ldots$ in $\ldots$ end $\in \mathcal{S}'(F)$ and $_{l_2}x$ in in the scope of that x)

ta$[_l e] \rightarrow$ **ta**[let $\ldots$ in $_l e$ end]

pa$[_l e] \rightarrow$ **pa**[let $\ldots$ in $_l e$ end]

2, axioms and rules for inferability

$\rightarrow$ **ti**$[_l c, l, +]$

$\rightarrow$ **ti**$[_l x, l, +]$ (if x is an argument variable of an outer-most function)

$\rightarrow$ **ti**$[_l e, 0, -]$ (if $_l e$ is the entire body of an outer-most function)

$=[e_1, e_2] \rightarrow$ **pi***$[\langle e_1, e_2\rangle, 0, +]$

$=[e_1, e_2]$, **ti**$[e_1, n, d] \rightarrow$ **ti**$[e_2, n, d]$

$=[e_1, e_2]$, **pi**$[e_1, n, d] \rightarrow$ **pi**$[e_2, n, d]$

$=[e_1, e_2]$, **pi***$[\langle \ldots, e_1, \ldots\rangle, n, d] \rightarrow$ **pi***$[\langle \ldots, e_2, \ldots\rangle, n, d]$

pi*$[\langle e_\alpha, e_\beta\rangle, n_1, d_1]$, **pi***$[\langle e_\beta, e_\gamma\rangle, n_2, d_2] \rightarrow$ **pi***$[\langle e_\alpha, e_\gamma\rangle, n_1, d_1]$

pi$[_l e, n_1, d_1]$, **pi**$[_l e, n_2, d_2] \rightarrow$ **ti**$[_l e, n_1, d_1]$ ($\langle n_1, d_1\rangle \neq \langle n_2, d_2\rangle$)

3, axioms and rules for $=$

$\rightarrow =[_{l_1} x, {}_{l_2} x]$ (if they are different occurrences of the same argument variable)

$\rightarrow =[_{l_1} x_1, {}_{l_2} x_2]$ (if they are argument variables of outer-most functions of the same type)

$\rightarrow =[_{l_1} x, {}_{l_2} e]$ (if let $\ldots, x = {}_{l_2}e, \ldots$ in $\ldots$ end $\in \mathcal{S}'(F)$ and $_{l_1}x$ in in the scope of that x)

$\rightarrow =[_l e,$ let $\ldots$ in $_l e$ end]

$=[e_1, e_2] \rightarrow =[e_3,$ r_$att(e_2)]$ (if w_$att(e_1, e_3) \in \mathcal{S}'(F)$)

$=[_{l_1} e_1, {}_{l_2} e_2] \rightarrow =[_{l_3}$r_$att(_{l_1} e_1), {}_{l_4}$r_$att(_{l_2} e_2)]$

$=[e_1, e_2] \rightarrow =[e_2, e_1]$

$=[e_1, e_2], =[e_2, e_3] \rightarrow =[e_1, e_3]$

4, rules for implications between capabilities

ta$[_l e] \rightarrow$ **pa**$[_l e]$

ti$[_l e, n, d] \rightarrow$ **pi**$[_l e, n, d]$

Table 2: The axioms and the inference rules of $\mathcal{J}(F)$

intersection of those two S can be a singleton set. Therefore, we need to record how each partial inferability is caused. The second reason is that we must not feed back inferability to its cause. For example, suppose there is an expression $+(e, 1)$, and the user has partial inferability on e. Then, the user can have partial inferability on $+(e, 1)$. Then, it seems that the user can achieve partial inferability on e again via inferability on $+(e, 1)$. However, this does not mean the user can achieve partial inferability on e in two "different" ways. What was wrong is to feed back the inferability caused by e to itself again.

The axioms and the inference rules of $\mathcal{J}(F)$ are listed in Table 2. In this list, numbers of subexpressions are omitted when they are not important. Most rules for equality and inferability directly corresponds to rules of $\mathcal{I}$. The rule **pi**$[e, n_1, d_1]$, **pi**$[e, n_2, d_2] \rightarrow$**ti**$[e, n_1, d_1]$ corresponds to the special case of the join of $[\langle e\rangle \in S]$, and represents that we take a pessimistic assumption that the intersection of two different subsets can be always a singleton set.

In addition to those rules, we must also specify the rules on basic functions. Because those rules depend on the semantics of each basic function, we must define those rules by hand. We can, however, provide "metarules" of the form "if the semantics of the basic function f_b satisfies this condition, then this rule must be added". Only if rules on basic functions are correctly given following those metarules, our algorithm can analyze any composed functions. Here, we show only a couple of examples of metarules for the limitation of the space:

if $\exists v_2. \forall r \in Dom(f_b). \exists v_1. f_b(v_1, v_2) = r$
ta$[e_1] \rightarrow$ **ta**$[f_b(e_1, e_2)]$

if $\exists r. \exists v_1. \forall v_2 \in Dom(e_2). f_b(v_1, v_2) = r$
ti$[e_1, n, d] \rightarrow$ **ti**$[_l f_b(e_1, e_2), l, +]$ $\langle n, d\rangle \neq \langle l, -\rangle$

if $\exists v_1 . \bigcap_{v_2 \in Dom(e_2)} \{w_1 \in Dom(e_1) | \exists w_2 . f_b(w_1, w_2) = f_b(v_1, v_2)\} = \{v_1\}$
$\mathbf{ti}[_l f_b(e_1, e_2), n, d]$, $\mathbf{ta}[e_2] \rightarrow \mathbf{ti}[e_1, l, -]$ $\langle n, d \rangle \neq \langle l, + \rangle$ $\langle n, d \rangle \neq \langle l, - \rangle$ and $\langle n, d \rangle \neq \langle l, + \rangle$ are restrictions to avoid feedback. All examples above are metarules for basic functions with two arguments. Metarules for functions with more arguments can be defined in a similar way. The metarules for inferability are examining whether the set S in the term $[e \in S]$ deduced by $\mathcal{I}(E)$ can be a singleton set or not. "$\bigcap\{\ldots\}$" corresponds to the repetition of inference on the same expression with changing some alterable arguments. We explain the effect of such repetition in Section 3.

As an example of rules based on basic functions, we show the rules on the basic function $>=(e_1, e_2)$:

$$\begin{aligned}
\mathbf{pa}[e_1] &\rightarrow \mathbf{ta}[>=(e_1, e_2)] \\
\mathbf{pi}[e_1], \mathbf{pi}[e_2] &\rightarrow \mathbf{ti}[>=(e_1, e_2)] \\
\mathbf{pi}*[\langle e_1, e_2 \rangle] &\rightarrow \mathbf{ti}[>=(e_1, e_2)] \\
\mathbf{pi}[e_1] &\rightarrow \mathbf{pi}*[\langle e_2, >=(e_1, e_2) \rangle] \\
\mathbf{ti}[e_1], \mathbf{pa}[e_1], \mathbf{ti}[>=(e_1, e_2)] &\rightarrow \mathbf{ti}[e_2] \\
\mathbf{pi}[e_1], \mathbf{ti}[_l >=(e_1, e_2)] &\rightarrow \mathbf{pi}[e_2] \\
\mathbf{ti}[>=(e_1, e_2)] &\rightarrow \mathbf{pi}*[\langle e_1, e_2 \rangle]
\end{aligned}$$

Here, we omit *num*, *dir*, and conditions on them. We can get more rules, but we omit redundant ones, such as "$\mathbf{pa}[e_1]$, $\mathbf{pa}[e_2] \rightarrow \mathbf{ta}[>=(e_1, e_2)]$", and omit those that can be got by replacing e_1 and e_2. In the same way, the rules for $*(e_1, e_2)$, multiplication on integers, are as follows:

$$\begin{aligned}
\mathbf{ta}[e_1] &\rightarrow \mathbf{ta}[*(e_1, e_2)] \\
\mathbf{pa}[e_1] &\rightarrow \mathbf{pa}[*(e_1, e_2)] \\
\mathbf{ti}[e_1] &\rightarrow \mathbf{ti}[*(e_1, e_2)] \\
\mathbf{pi}[e_1] &\rightarrow \mathbf{pi}[*(e_1, e_2)] \\
\mathbf{pi}[e_1] &\rightarrow \mathbf{pi}*[\langle e_2, *(e_1, e_2) \rangle] \\
\mathbf{pi}[e_1], \mathbf{pi}[*(e_1, e_2)] &\rightarrow \mathbf{ti}[e_2] \\
\mathbf{pa}[e_1], \mathbf{pi}[*(e_1, e_2)] &\rightarrow \mathbf{ti}[e_2] \\
\mathbf{pi}[*(e_1, e_2)] &\rightarrow \mathbf{pi}[e_2] \\
\mathbf{pi}*[\langle e_1, *(e_1, e_2) \rangle] &\rightarrow \mathbf{ti}[e_2] \\
\mathbf{pi}[*(e_1, e_2)] &\rightarrow \mathbf{pi}*[\langle e_1, e_2 \rangle]
\end{aligned}$$

The first rule holds because $*(e_1, e_2) = e_1$ when $e_2 = 1$. The third rule holds because if one can infer $e_1 = 0$, then he can infer $*(e_1, e_2) = 0$. The sixth rule holds because if he can infer $e_1 \in \{2, 3\}$ and $*(e_1, e_2) \in \{4, 5\}$, then he can infer that $e_2 = 2$.

Using $\mathcal{J}(F)$, we define the algorithm $\mathcal{A}(R)$ that determines whether the requirement R is satisfied or not:

Definition 6 $\mathcal{A}(R)$
Given $R = (u, f(x_1 : c_1^1 \ldots : c_1^{m_1}, \ldots, x_n : c_n^1 \ldots : c_n^{m_n}) : c_0^1 \ldots : c_0^{m_0})$, $\mathcal{A}(R)$ *calculates the closure set of all inferable terms of* $\mathcal{J}(F)$ *where* F *is a set of all functions in the capability list of* u. *Then, if there exists some* $_l$let$(f)x_1 =_{l_1} e_1, \ldots, x_n =_{l_n} e_n$ in $\ldots$end $\in \mathcal{S}'(F)$ *for which all terms corresponding to capabilities specified in* R *are included in the closure set,* $\mathcal{A}(R)$ *determines that* R *is not satisfied. Otherwise* $\mathcal{A}(R)$ *determines that* R *is satisfied.* □

Calculating a closure corresponds to calculating all capabilities the user may achieve. $\mathcal{A}(R)$ can be proved to be sound:

Theorem 1 *Soundness of* $\mathcal{A}(R)$
If R *is not satisfied,* $\mathcal{A}(R)$ *always determines that* R *is not satisfied.*

(Proof outline) As for alterability, it is proved by induction on the structure of the subexpression. It is rather simple. As for inferability, it is proved by induction on the length of inference sequence of $\mathcal{I}(E)$ deducing $[e \in S]$. The soundness of the algorithm is essentially comes from the fact that "the rules for basic functions are defined so that the algorithm be sound". We briefly describe how $\mathcal{J}(F)$ simulates $\mathcal{I}(E)$. In $\mathcal{I}$, terms of the form $[\langle \ldots \rangle \in S]$ are deduced by (1) axioms on constants, or arguments and returned values of directly invoked functions, (2) the rule deducing $[\langle e_1, e_2 \rangle \in \ldots]$ from equality, and (3) the rule corresponding to dependency among arguments and results of basic functions. Results got by (1) are simulated by axioms in $\mathcal{J}$. Results got by (2) are simulated by the rule $=[e_1, e_2] \rightarrow \mathbf{pi}*[\langle e_1, e_2 \rangle]$. Results got by joining (1) and (2) are simulated by the rule $=[e_1, e_2], \mathbf{ti}[e_1] \rightarrow \mathbf{ti}[e_2]$. Results got by (3) or by joining (3) and others are simulated by the last four rules in 2 in Table 2 and the rules on basic functions. ▮

4.2 An Example

We briefly explain one example of the analysis. Suppose the user u can directly invoke checkBudget(broker) and w_budget(o, v), and requirement (u, r_salary(broker):**ti**) is specified. In this situation, as explained in Section 3, the user u can infer the exact value of the salary of each broker. Therefore, there is a security flaw. It can be detected in the following way. First, we unfold and number the code of checkBudget and w_budget(o, v) as below:

checkBudget(broker)
$_7$ >=($_2$r_budget($_1$broker), $_6$*($_3$10, $_5$r_salary($_4$broker)))

$_{10}$w_budget($_8$o, $_9$v)

Then, $\mathcal{J}(\{$checkBudget(broker), w_budget(o, v)$\})$ deduce $\mathbf{ti}[^5$r_salary(^{4}broker)] as shown in Figure 1. (In this figure, we omit *num* and *dir* because they are not important in this example.) Therefore, $\mathcal{A}(R)$ determines that (u,r_salary(broker):**ti**) is not satisfied. This means the security flaw is successfully detected.

5 Conclusion

We defined a framework for access control in the abstract operation granularity, and developed an mechanism that

	$\rightarrow$	$=[_8o, {}_1\text{broker}]$	(axiom for $=$)
$=[_8o, {}_1\text{broker}]$	$\rightarrow$	$=[_9v, {}_2\text{r\_budget}({}_1\text{broker})]$	(rule for $=$)
	$\rightarrow$	$\mathbf{ti}[_9v]$	(axiom)
$=[_9v, {}_2\text{r\_budget}({}_1\text{broker})], \mathbf{ti}[_9v]$	$\rightarrow$	$\mathbf{ti}[_2\text{r\_budget}({}_1\text{broker})]$	(inferability based on $=$)
	$\rightarrow$	$\mathbf{pa}[_9v]$	(axiom)
$=[_8o, {}_1\text{broker}], \mathbf{pa}[_9v]$	$\rightarrow$	$\mathbf{pa}[_2\text{r\_budget}({}_1\text{broker})]$	(alterability based on $=$)
	$\rightarrow$	$\mathbf{ti}[_7 >=(\ldots)]$	(axiom)
$\mathbf{ti}[_2\text{r\_budget}({}_1\text{broker})], \mathbf{pa}[_2\text{r\_budget}({}_1\text{broker})], \mathbf{ti}[_7 >=(\ldots)]$			
	$\rightarrow$	$\mathbf{ti}[_6*({}_3 10, {}_5\text{r\_salary}({}_4\text{broker}))]$	(basic function)
	$\rightarrow$	$\mathbf{ti}[_3 10]$	(axiom)
$\mathbf{ti}[_3 10], \mathbf{ti}[_6*({}_3 10, {}_5\text{r\_salary}({}_4\text{broker}))]$	$\rightarrow$	$\mathbf{ti}[_5\text{r\_salary}({}_4\text{broker})]$	(basic function)

Figure 1: An example of analysis by $\mathcal{J}$

detects security flaws caused by functions not hiding primitive operations inside them. The most important contribution of this research is that we introduced the notions of inferability on returned values and controllability on arguments, demonstrated that they properly model the problem of security flaws, investigated their properties, and gave the formal semantics of them. We think that these notions are proper generalization of traditional read/write capability and can work as a basis for various researches on access control in the function granularity.

Although we also showed a static analysis algorithm which is sound and sufficiently practical, it is not necessarily the only way to avoid security flaws. In fact, the algorithm shown in this paper is quite pessimistic. More accurate analysis with more complex computation could be developed using existing techniques for program analysis. Another alternative is to develop a mechanism to dynamically detect security flaws during execution of queries. Those are future issues.

Our model achieves name-dependent access control, and a kind of context-dependent control [FSW81]. Content-dependent control, which is control depending on the contents of the actual data, could be also integrated by introducing some existing techniques, such as classes with predicates [TYI88, AB91, SLT91, Run92, OT94].

The function definition language we defined in this research is quite simple language. Including more language features into it, such as conditional branch, recursion, and polymorphism, is also an important issue for future researches. We also assume a rather simple data model in this development. In [RBKW91, Spo89, GGF93], how various data modeling concepts, such as versions or inheritance, affect the authorization mechanisms is discussed. The integration of the techniques we show in this paper and the mechanisms proposed in those researches is also an interesting issue.

In this paper, we do not discuss properties of aggregate functions on sets. Interesting studies on that topic has been shown in the context of statistical databases [KU77, Chi78, DDS79, DJL79, Bec80, CO82]. The result of these researches say, in short, that aggregate functions on a set of data almost always reveal the information on the individual elements of the set.

Acknowledgments

I would like to thank Atsushi Ohori for his valuable suggestions and discussions through the research. This work was done while the author was working under him at Kyoto University. The author is also indebted to Takashi Masuda for his continuous support and encouragement. I also would like to thank Masakazu Soshi for his help in surveying this research area. The author is also supported by International Information Science Foundation, Japan (Grant No. 96.1.2.587).

References

[AB91] Serge Abiteboul and Anthony Bonner. Objects and views. In *Proc. of ACM SIGMOD*, pages 238–247, Jun. 1991.

[ADG92] Rafiul Ahad, James Davis, and Stefan Gower. Supporting access control in an object-oriented database language. In *Proc. of EDBT*, volume 580 of *LNCS*, pages 184–200. Springer-Verlag, Mar. 1992.

[Bec80] Leland L. Beck. A security mechanism for statistical databases. *ACM TODS*, 5(3):316–338, Sep. 1980.

[Ber92] Elisa Bertino. Data hiding and security in object-oriented databases. In *Proc. of IEEE ICDE*, pages 338–347, Feb. 1992.

[Bur90] Rae K. Burns. Referential secrecy. In *Proc. of IEEE Symp. on Research in Security and Privacy*, pages 133–142, 1990.

[Chi78] Francis Y. Chin. Security in statistical databases for queries with small counts. *ACM TODS*, 3(1):92–104, Mar. 1978.

[CO82] Francis Y. Chin and Gultekin Ozsoyoglu. Auditing and inference control in statistical database. *IEEE Trans. on Soft. Eng.*, 8(6):574–582, Nov. 1982.

[DAH+87] Dorothy E. Denning, Selim G. Akl, Mark Heckman, Teresa F. Lunt, Matthew Morgenstern, Peter G. Neumann, and Roger R. Schell. Views for multilevel database security. *IEEE Trans. on Soft. Eng.*, 13(2):129–140, Feb. 1987.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *CACM*, 20(7):504–512, Jul. 1977.

[DDS79] Dorothy E. Denning, Peter J. Denning, and Mayer D. Schwartz. The tracker: A threat to statistical database security. *ACM TODS*, 4(1):76–96, Mar. 1979.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *CACM*, 19(5):236–243, May 1976.

[DJL79] David Dobkin, Anita K. Jones, and Richard J. Lipton. Secure databases: Protection against user influence. *ACM TODS*, 4(1):97–106, Mar. 1979.

[FSW81] Eduardo B. Fernandez, Rita C. Summers, and Christopher Wood. *Database Security and Integrity*, chapter 5, pages 55–64. The Systems Programming Series. Addison-Wesley, 1981.

[GGF93] Nurith Gal-Oz, Ehud Gudes, and Eudardo B. Fernandez. A model of methods authorization in object-oriented databases. In *Proc. of VLDB*, pages 52–61, Aug. 1993.

[HD92] Thomas Hinke and Harry S. Delugach. Aerie: An inference modeling and detection approach for databases. In *Database Security VI: Status and Prospects*, pages 179–194. IFIP WG 11.3, Aug. 1992.

[HZ90] Sandra Heiler and Stanley B. Zdonik. Object views: Extending the vision. In *Proc. of IEEE ICDE*, pages 86–93, Feb. 1990.

[JS91] Sushil Jajodia and Ravi Sandhu. Toward a multilevel secure relational data model. In *Proc. of ACM SIGMOD*, pages 50–59, May 1991.

[KU77] John B. Kam and Jeffrey D. Ullman. A model of statistiacl database and their security. *ACM TODS*, 2(1):1–10, Mar. 1977.

[LDS+90] Teresa F. Lunt, Dorothy E. Denning, Roger R. Schell, Mark Heckman, and William R. Shockley. The SeaView security model. *IEEE Trans. on Soft. Eng.*, 16(6):593–607, Jun. 1990.

[MJ88] Catherine Meadows and Sushil Jajodia. Ingetrity versus security in multi-level secure databases. In *Database Security II: Status and Prospects*, pages 89–101. IFIP WG 11.3, North-Holland, Oct. 1988.

[Mor87] Matthew Morgenstern. Security and inference in multilevel database and knowlege-base systems. In *Proc. of ACM SIGMOD*, pages 357–371, Dec. 1987.

[MSS88] Subhasish Mazumdar, David Stemple, and Tim Sheard. Resolving the tension between integrity and security using a theorem prover. In *Proc. of ACM SIGMOD*, pages 233–242, Sep. 1988.

[OT94] Atsushi Ohori and Keishi Tajima. A polymorphic calculus for views and object sharing. In *Proc. of ACM PODS*, pages 255–266, May 1994.

[Qia94] Xiaolei Qian. Inference channel-free integrity constraints in multilevel relational databases. In *Proc. of IEEE Symp. on Research in Security and Privacy*, pages 158–167, 1994.

[QSK+93] Xaiolei Qian, Mark E. Stickel, Peter D. Karp, Teresa F. Lunt, and Thomas D. Garvey. Detection and elimination of inference channels in multilevel relational database systems. In *Proc. of IEEE Symp. on Research in Security and Privacy*, pages 196–205, 1993.

[RBKW91] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrell Woelk. A model of authorization for next-generation database systems. *ACM TODS*, 16(1):88–131, Mar. 1991.

[Row89] Neil C. Rowe. Inference-security analysis using resolution theorem-proving. In *Proc. of IEEE ICDE*, pages 410–416, Feb. 1989.

[Run92] Elke A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proc. of VLDB*, pages 187–198, Aug. 1992.

[SLT91] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updatable views in object-oriented databases. In *Proc. of Int. Conf. on Deductive and Object-Oriented Database Systems*, volume 566 of *LNCS*, pages 189–207. Springer-Verlag, Dec. 1991.

[SO91] Tzong-An Su and Gultekin Ozsoyoglu. Controlling FD and MVD inferences in multilevel relational database systems. *IEEE Trans. on Know. and Data.*, 3(4):474–485, Dec. 1991.

[Spo89] David L. Spooner. The impact of inheritance on security in object-oriented database systems. In *Database Security II: Status and Prospectus*, pages 141–150. Elsevier Science Publ., 1989.

[TYI88] Katsumi Tanaka, Masatoshi Yoshikawa, and Kozo Ishihara. Schema virtualiztion in object-oriented databases. In *Proc. of IEEE ICDE*, pages 23–30, Feb. 1988.