Proc. Second European Symposium on Research in Computer Security, Toulouse, France, November 1992, pages 307-328.

Polyinstantiation for Cover Stories

Ravi S. Sandhu and Sushil Jajodia[1]

Center for Secure Information Systems &
Department of Information and Software Systems Engineering
George Mason University
Fairfax, VA 22030, USA
email: {sandhu, jajodia}@sitevax.gmu.edu

Abstract. In this paper we study the use of polyinstantiation, for the purpose of implementing cover stories in multilevel secure relational database systems. We define a particular semantics for polyinstantiation called PCS (i.e., polyinstantiation for cover stories). PCS allows two alternatives for each attribute (or attribute group) of a multilevel entity: (i) no polyinstantiation, or (ii) polyinstantiation at the explicit request of a user to whom the polyinstantiation is visible. PCS strictly limits the extent of polyinstantiation by requiring that each entity in a multilevel relation has at most one tuple per security class. We demonstrate that PCS provides a natural, intuitive and useful technique for implementing cover stories. A particularly attractive feature of PCS is its run-time flexibility regarding the use of cover stories. A particular attribute may have cover stories for some entities and not for others. Even for the same entity, a particular attribute may be polyinstantiated at some time and not at other times.

1 INTRODUCTION

Polyinstantiation has generated a great deal of controversy lately. Some have argued that polyinstantiation and integrity are fundamentally incompatible, and have proposed alternatives to polyinstantiation. Others have argued about the correct definition of polyinstantiation and its operational semantics. Much has been written about this topic, as can be seen from the bibliography of this paper.

There are two extreme positions that can be identified with respect to polyinstantiation.

[1] The work of both authors was partially supported by the U.S. Air Force, Rome Air Development Center through contract #F-30602-92-C-002. We are indebted to Joe Giardono for making this work possible.

– Polyinstantiation and integrity are fundamentally incompatible, and steps must be taken to avoid polyinstantiation in multilevel relations regardless of the cost.
– Polyinstantiation is an intrinsic phenomenon, inevitable in the multilevel world. Therefore, multilevel relations must be polyinstantiated whenever necessary.

Extreme proponents of the former view are apparently willing to tolerate information leakage and/or severe denial-of-service in order to totally banish polyinstantiation. Extreme proponents of the latter view appear similarly willing to generate large numbers of spurious tuples and data associations, whenever the opportunity is presented.

As is often the case in such situations, the truth lies somewhere in between. To reconcile these extreme views, it is useful to draw an analogy with the debate in the early 1970's regarding **goto** statements in programming languages. Today it is well understood that indiscriminate use of **goto**'s is harmful, but also that the complete eliminations of **goto**'s creates more problems than it solves. Polyinstantiation should similarly be viewed as a technique which can be used for better or for worse.

It is important to understand that there is nothing fundamental about the occurrence of polyinstantiation. Jajodia and Sandhu [16, 23] have shown how it is possible to prohibit polyinstantiation securely (i.e., without leakage of secret information or denial-of-service). In other words, if you don't like it you can get rid of it completely and securely.

At the same time, it is equally important to understand that there is no fundamental incompatibility between polyinstantiation and integrity. A properly designed database management system (DBMS) can limit the occurrence of polyinstantiation to precisely those instances where it is explicitly requested by a user[2] to whom the polyinstantiation is visible. The early work on polyinstantiation allowed an unclassified user to insert information which propagated into several polyinstantiated tuples at the secret and higher levels. The resulting "spaghetti relations" do remind one of the all too familiar spaghetti code riddled with **goto**'s. But, much as the elimination of **goto**'s is not fundamental to structured programming, the elimination of polyinstantiation is not fundamental to database integrity.

The principal objective of this paper is to demonstrate that careful use of polyinstantiation is a natural, intuitive and disciplined method for implementing cover stories in multilevel secure relational databases. Polyinstantiation should, of course, be used only where it is appropriate. Therefore polyinstantiation must be prevented in the many situations where there is no need for cover stories. In other words, even within the same database or relation we should be able to

[2] Strictly speaking we should be saying subject rather than user. For the most part we will loosely use these terms interchangeably. Where the distinction is important we will be appropriately precise.

allow or disallow polyinstantiation selectively. We also reiterate the importance of limiting the occurrence of polyinstantiation to precisely those instances where it is explicitly requested by the user to whom it will be visible.

This paper defines a particular semantics[3] for polyinstantiation called PCS (i.e., polyinstantiation for cover stories). In developing PCS we have refined many of our previously published ideas, included some new ones; as well as borrowed and adapted concepts from other researchers who have published on this topic. Our principal contribution is in the total package we have produced, by combining and refining various ideas into a consistent, intuitive, and flexible aggregate.

PCS allows two alternatives for each attribute (or attribute group) of a multilevel entity: (i) no polyinstantiation, or (ii) polyinstantiation by explicit request. PCS offers run-time flexibility of when to use cover stories, and uniformity of the query interface. These are not available in other proposals for implementing cover stories, such as having a separate attribute for the true facts and the cover story. A particularly attractive feature of PCS is that the same attribute may be polyinstantiated or not for different entities in the same relation. For example, the Destination of the Starship Enterprise can be polyinstantiated for a cover story, while polyinstantiation for the Destination of the Voyager is forbidden. Furthermore, PCS can readily accommodate the situation where on different occasions the same entity does or does not have a cover story for a particular attribute, as the need changes. For example, the Destination of the Starship Enterprise can be polyinstantiated for a cover story today, but tomorrow its polyinstantiation can be forbidden.

The remainder of this paper is organized as follows. Section 2 reviews the concept of polyinstantiation emphasizing those aspects which are important to our objective in this paper. Section 3 discusses how polyinstantiation can be eliminated in a secure manner, i.e., without introducing signaling channels[4] for leakage of secret information or incurring serious denial-of-service costs. Section 4 introduces and motivates the concepts of PCS. (A formal model for PCS, including its entity integrity and referential integrity properties, is given in the appendix.) Section 5 gives our conclusions.

[3] We do not claim that PCS is the only useful semantics for polyinstantiation.

[4] A signaling channel is distinct from a covert channel. A signaling channel is a means of information flow which is inherent in the data model, and will occur in *every* implementation of the model. A covert channel, on the other hand, is a property of a specific implementation; not a property of the data model. In other words, even if the data model is free of downward signaling channels, a specific implementation may well contain covert channels due to implementation quirks. It is therefore most important for the data model to be free of downward signaling channels. Otherwise there is *no* implementation of the model, however idealized, which can be free of information leakage.

2 POLYINSTANTIATION

In this section we discuss some basic concepts of polyinstantiation by means of examples. We assume that the readers are familiar with the basic concepts of the standard (single-level) as well as multilevel relations. We refer the readers to [14] or [15] for a detailed exposition.

A multilevel relation is said to be polyinstantiated when it contains two or more tuples with the same "apparent" primary key values. The concept of *apparent primary key* was introduced by Denning et al. in [3]. While the notion of a primary key is simple and well understood for classical (single-level) relations, it does not have a straightforward extension to multilevel relations. The apparent primary key of a multilevel relation are those attributes which are asserted by the user as being the primary key. The *real primary key* (i.e., the minimal set of attributes which is unique in each tuple) of the multilevel relation is obtained by adding one or more classification attributes to the apparent primary key. The exact manner in which this is done is closely related to the precise polyinstantiation behavior of the relation (see [2] for a detailed discussion).

In multilevel relations, a major issue is how access classes are assigned to data stored in relations. One can assign access classes to relations, to individual tuples in a relation, to individual attributes (i.e., "columns") of a relation, or to the individual data elements of a relation. Polyinstantiation does not arise explicitly when access classes are assigned to relations or individual attributes of a relation. For generality, we consider the case where access classes are attached to the individual data elements themselves. Systems which attach access classes to the tuples in a relation have limited expressive power and will not be discussed in this paper.

There are two different types of polyinstantiation in multilevel relations with element level labeling [19], as follows:

- entity polyinstantiation, and
- element polyinstantiation.

Our proposal in PCS is to disallow entity polyinstantiation[5], and allow element polyinstantiation in a carefully controlled manner, as explicitly requested by users.

2.1 Entity Polyinstantiation

Entity polyinstantiation occurs when a relation contains multiple tuples with the same apparent primary key values, but having different access class values

[5] Entity polyinstantiation can actually be allowed without significantly impacting PCS. There may be situations in which entity polyinstantiation is desirable. However, it should be understood that entity polyinstantiation is particularly detrimental to referential integrity as noted in [7].

for the apparent primary key. As an example, consider the relation SOD given below:

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Enterprise	S	Spying	S	Rigel	S	S

Here, as in all our examples, each attribute in a tuple not only has a value but also a classification. In addition there is a tuple-class or TC attribute. This attribute is computed to be the least upper bound of the classifications of the individual data elements in the tuple. We assume that the attribute Starship is the apparent primary key of SOD.

The name "entity polyinstantiation" arises from the interpretation that these two tuples refer to two distinct entities in the external world. That is, there are two distinct Starships with the same name Enterprise. We will discuss how to prevent entity polyinstantiation in section 3.

2.2 Element Polyinstantiation

The following relation illustrates element polyinstantiation:

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Enterprise	U	Spying	S	Talos	U	S

With element polyinstantiation, a relation contains two or more tuples with identical apparent primary keys and the associated access class values, but having different values for one or more remaining attributes. As shown in the above example, the objective of the starship Enterprise is different for U- and S-users.

What are we to make of this last relation given above? There are at least two reasonable interpretations that have been proposed in the literature.

- The objective of Exploration is a *cover story* (at the U-level) for the real objective of Spying (at the S-level).
- We have an inconsistency in the database which needs to be resolved.

We will show in section 3 how to securely prevent element polyinstantiation from arising due to inconsistencies. As a result the only occurrence of polyinstantiation will be when it is deliberately requested for the purpose of implementing cover stories.

To appreciate the intuitive notion of a cover story consider the eight instances of SOD shown below [9].

No.	Starship	Objective	Destination	TC
1	Enterprise U	Exploration U	Talos U	U
2	Enterprise U	Exploration U	Talos U	U
	Enterprise U	Spying S	Talos U	S
3	Enterprise U	Exploration U	Talos U	U
	Enterprise U	Exploration U	Rigel S	S
4	Enterprise U	Exploration U	Talos U	U
	Enterprise U	Spying S	Rigel S	S
5	Enterprise U	Exploration U	Talos U	U
	Enterprise U	Exploration U	Rigel S	S
	Enterprise U	Spying S	Rigel S	S
6	Enterprise U	Exploration U	Talos U	U
	Enterprise U	Spying S	Talos U	S
	Enterprise U	Spying S	Rigel S	S
7	Enterprise U	Exploration U	Talos U	U
	Enterprise U	Spying S	Talos U	S
	Enterprise U	Exploration U	Rigel S	S
8	Enterprise U	Exploration U	Talos U	U
	Enterprise U	Spying S	Talos U	S
	Enterprise U	Exploration U	Rigel S	S
	Enterprise U	Spying S	Rigel S	S

These instances can be partitioned into three classes as follows.

- Instance 1 has no polyinstantiation and is therefore straightforward.
- Instances 2, 3, and 4 are also relatively straightforward. In each case there is a single U-tuple and a single S-tuple for the Enterprise. The U-tuple can therefore be reasonably interpreted as being a cover story for the S-tuple. Instances 2, 3, and 4 differ in the extent to which the U cover story is actually true or false at the S level. Instance 2 has a cover story for the objective, but the U destination is correct. Instance 3 conversely has a cover story for the destination, but the U objective is correct. Instance 4 has a cover story for both the objective and destination.
- Instances 5, 6, 7, and 8 are, however, confusing to interpret from a cover story perspective. Each of these cases has more than one S-tuple for the Enterprise, but only one U-tuple. It is possible to give a meaningful and consistent interpretation and update semantics for such relations [9, 12]. However, these interpretations loose the basic intuitive simplicity of the relational model.

The intuitive appeal of instances 2, 3 and 4 is that they have *one tuple per tuple class*. We will adhere to this requirement in the rest of this paper.

It should be noted that certain problems with the concept of one-tuple-per-tuple-class in context of a partially ordered lattice were identified in [21]. These problems arise because [21] takes the following view: those attributes in a tuple that are classified below the tuple class are *automatically* derived from lower-level polyinstantiated tuples. PCS, however, takes the view that such attributes are *explicitly* derived by the user when constructing the higher-level tuple. PCS therefore does not suffer from the problems identified in [21].

2.3 Update Propagation

One of the subtleties involved in maintaining plausible cover stories is consistency across different levels. To illustrate this issue consider the following relation instances:

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
Enterprise U	Exploration U	Rigel S	S

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
Enterprise U	Exploration S	Rigel S	S

We will treat these relations as being different, even though the values of the individual data elements are the same in both cases. In other words, there is a difference between the objective being <Exploration,U> versus <Exploration,S>. To understand this difference, consider what happens when a U-user updates the objective of the Enterprise to be Mining. These two relations will respectively be updated as follows:

Starship	Objective	Destination	TC
Enterprise U	Mining U	Talos U	U
Enterprise U	Mining U	Rigel S	S

Starship	Objective	Destination	TC
Enterprise U	Mining U	Talos U	U
Enterprise U	Exploration S	Rigel S	S

3 ELIMINATING POLYINSTANTIATION

In this section we show how polyinstantiation can be completely prevented. We discuss the prevention of entity and element polyinstantiation separately below.

3.1 Source of Entity Polyinstantiation

Entity polyinstantiation can occur in basically two different ways, which we respectively call *polyhigh* and *polylow* for ease of reference [23].

1. Polyhigh: A high user inserts a tuple with a primary key that already exists at the low level.
2. Polylow: A low user inserts a tuple with a primary key that already exists at the high level.

Polyhigh is easily prevented without disclosing secret information. The DBMS simply rejects the attempted insertion. The real challenge is in preventing polylow.

To be concrete, let us illustrate polyhigh by considering the following instance of SOD.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U

Now suppose a S-user attempts to insert the following tuple in this relation: (Enterprise, Spying, Rigel). A polyinstantiating DBMS will allow this insertion giving us the following result.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Enterprise	S	Spying	S	Rigel	S	S

There is, however, no fundamental need to polyinstantiate in this situation. The DBMS can simply reject this insertion by the S-user. The key conflict is visible to the S-user without any secrecy violation. Since the name Enterprise is already in use, it is only proper to ask the S-user to choose another name for the new ship, say, Enterprise′. In other words, there is no serious denial-of-service to the S-user; so long as the user can rename the new Starship to be Enterprise′ and enter the following tuple: (Enterprise′, Spying, Rigel) to obtain

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Talos	U	U
Enterprise′	S	Spying	S	Rigel	S	S

without polyinstantiation.

Similarly, let us illustrate polylow by considering the following relation instance.

Starship		Objective		Destination		TC
Enterprise	S	Spying	S	Rigel	S	S

Note that due to simple-security this tuple is not visible to U-users, who therefore see an empty relation. Now suppose a U-user attempts to insert the following tuple in this relation: (Enterprise, Exploration, Talos). This insertion cannot be rejected without some security compromise. Once we allow the database to come to this point, we can get out of the situation only by compromising some aspect of security. Various solutions have been proposed but none are really palatable. We can identify the following alternatives.

1. *Tolerate Loss of Secrecy.* Proponents of this approach consider it better to disallow the insertion and leak information, by inference, that the Enterprise is being used as a key at some level above U. Unfortunately the signaling channels opened up by this tolerance preclude such systems from attaining a high rating (i.e., B2 or above [6]) for multilevel security.
2. *Tolerate Loss of Integrity.* This is the entity polyinstantiation route and would give us the following result.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
Enterprise S	Spying S	Rigel S	S

   It is possible to maintain an appearance of integrity in this case by deleting the existing S-tuple for the Enterprise and inserting the new U-tuple to obtain

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U

   For obvious reasons, no one has proposed this "solution" seriously.
3. *Tolerate Denial of Service.* The SWORD project [25] has proposed that in such situations we forbid all further insertions for all time! For instance, a U-user is prevented from even inserting a tuple such as (Voyager, Mining, Mars) which does not cause any key conflict. Thus, the moment a S-key has been inserted no more Starships can be created by *any* user in this relation. Moreover, there is no way of recovering from this state. This is clearly serious denial-of-service.

The main point to note, for our purpose, is that is *too late to securely prevent this insertion at the point where the insertion is about to take place.* The insertion can be securely prevented only by taking proactive steps in advance of its imminent occurrence.

3.2 Prevention of Entity Polyinstantiation

There are three basic techniques for eliminating entity polyinstantiation.

1. *Make all the keys visible.* In this method the apparent primary key is required to be labeled at the lowest level at which a relation is visible. For example, we can require that all keys be unclassified. Consequently, the following relation

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
Enterprise S	Spying S	Rigel S	S

   would be forbidden. Note that we can represent the same information in two different relations called USOD and SSOD as follows

UStarship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U

SStarship	Objective	Destination	TC
Enterprise S	Spying S	Rigel S	S

In other words we horizontally partition the original SOD relation, putting all the U-Starships in USOD and all the S-Starships in SSOD.

2. *Partition the domain of the primary key.* Another way to eliminate entity polyinstantiation is to partition the domain of the primary key among the various access classes possible for the primary key. For our example, we can say require that starships whose names begin with A-E are unclassified, starships whose names begin with F-T are secret, and so on. Whenever a new tuple is inserted, we enforce this requirement as an integrity constraint. In this case we would need to rename the secret Enterprise, perhaps as follows.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
FEnterprise S	Spying S	Rigel S	S

The DBMS can now reject any attempt by a U-user to insert a Starship whose name begins with F-Z, without causing any information leakage or integrity violation.

3. *Limit insertions to be done by trusted subjects.* A third way to eliminate entity polyinstantiation is to require that all insertions are done by a system-high user, with a write-down occurring as part of the insert operation. (Strictly speaking, we only need a relation-high user, i.e., a user to whom all tuples are visible.) In context of our example this means that a U-user who wishes to insert the tuple: (Enterprise, Exploration, Talos), must request a S-user to do the insertion. The S-user does so by invoking a trusted subject which can check for key conflict, and if there is none insert a U-tuple by writing down. If there is a conflict the S-user informs the U-user about it, so the U-user can, say, change the name of the Starship to Enterprise$'$.

The first approach is available in any DBMS which allows a range of access classes for individual attributes (or attribute groups), by simply limiting the classification range of the apparent key to be a singleton set. The second approach is available to any DBMS that can enforce domain constraints with adequate generality. The third approach is always available but requires the use of trusted code, and tolerates some leakage of information (although with a human in the loop). The best approach will depend upon the characteristics of the DBMS and the application, particularly concerning the frequency and source of insertions.

3.3 Source of Element Polyinstantiation

Element polyinstantiation can occur by polyhigh or polylow, similar to the occurrence of entity polyinstantiation. Let us again consider concrete examples to make these notions clearer.

Polyhigh occurs when an S-user attempts to update the destination of the Enterprise in the following relation to be Rigel.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U

The existing destination of Talos cannot be overwritten without violating the $\star$-property. Therefore, either the update must be rejected or the destination must be polyinstantiated to get the following result.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
Enterprise U	Exploration U	Rigel S	S

In either case U-users see no change in the relation and there is no information leakage.

Polylow arises in the opposite situation, where the Enterprise previously already has an S-destination and a U-destination is entered later. Specifically, consider the following relation.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Rigel S	S

U-users see this relation with the secret data filtered out as follows.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	null U	U

Now suppose a U-user attempts to update the destination of the Enterprise to be Talos. This update cannot be rejected on the grounds that a S-destination for the Enterprise already exists, because that amounts to establishing a downward signaling channel. It is possible to overwrite the secret destination, so that both U- and S-users see the following relation after the update takes place.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U

This option has major problems for the integrity of secret data, and has never been seriously considered. The remaining option is to polyinstantiate the destination attribute for the Enterprise, so that S-users see the following relation; whereas, U-users see the relation above.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
Enterprise U	Exploration U	Rigel S	S

To summarize, we can deal with polylow using the same three alternatives identified for polylow in entity polyinstantiation.

1. *Tolerate Loss of Secrecy.* This precludes a high degree of assurance (i.e., B2 or above [6]) for multilevel secure DBMS's.
2. *Tolerate Loss of Integrity.* This is the polyinstantiation route. (Or, the clearly unacceptable route of overwriting secret data by unclassified data.)
3. *Tolerate Denial of Service.* Once any secret data has been entered in a relation, we can prohibit further entry of *any* unclassified data.

It is again important to understand that it is *too late to securely prevent the polylow update at the point where the update is about to take place.* The update can be securely prevented only by taking proactive steps in advance of its imminent occurrence.

3.4 Prevention of Element Polyinstantiation

In this section we show how to prevent element polyinstantiation without compromising on confidentiality, integrity or denial-of-service requirements. The basic idea is to introduce a special symbol denoted by "restricted" as the possible value of a data element [23]. The value "restricted" is distinct from any other value for that element and is also different from "null." In other words the domain of a data element is its natural domain extended with "restricted" and "null." We define the semantics of "restricted" in such a way that we are able to eliminate both polyhigh and polylow.

Consider again the polyhigh scenario of section 3.3. We have the following relation to begin with.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U

Next a S-user attempts to modify the destination of the Enterprise to be Rigel. As we have argued we can reject this update securely. But what if the true destination has changed to <Rigel,S>? Surely there must be some way to enter this information. We require the S-user to first login as a U-subject[6] and mark the destination of the Enterprise as restricted giving us the following relation.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	restricted U	U

[6] Alternately the S-user logs in at the U-level and requests some properly authorized U-user to carry out this step. Communication of this request from the S-user to the U-user may also occur outside of the computer system, by say direct personal communication or a secure telephone call.

The meaning of <restricted,U> is that this field can no longer be updated by an ordinary U-user.[7] U-users can therefore infer that the true value of Enterprise's destination is classified at some level not dominated by U. The S-user then logs in as a S-subject and enters the destination of the Enterprise as Rigel giving us the following relations at the U and S levels respectively.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	restricted U	U

Starship	Objective	Destination	TC
Enterprise U	Exploration U	restricted U	U
Enterprise U	Exploration U	Rigel S	S

Note that this protocol does not introduce a signaling channel from a S-subject to an U-subject. There is information flow, but from a S-user (logged in as an U-subject) to an U-subject. This is an extremely important distinction. The paramount threat in computer security (at least, in terms of the Orange Book [6]) is from Trojan Horse infected subjects. Information leakage due to the activities of users in carrying out their jobs is of concern to overall system security. However, computerization cannot eliminate leakage that is intrinsically part of the application domain (such as setting some data element to be restricted). The point is that an information flow channel with a trusted S-user in the loop can be exercised only by Trojan Horses that are capable of manipulating the real world! This entails the manipulation of real trusted people making real decisions and not merely the manipulation of bits in a database. Finally, it is important to understand that information flow which includes humans in the loop always exists. For example, the above scenario can be played out replacing "restricted" with "null" and the same information flow occurs.

Next consider how the polylow scenario of section 3.3 plays out with the restricted requirement. In this case the Enterprise can have a secret destination only if the destination has been marked as being restricted at the unclassified level. Thus either the S- and U-users respectively see the following instances of SOD,

Starship	Objective	Destination	TC
Enterprise U	Exploration U	restricted U	U
Enterprise U	Exploration U	Rigel S	S

Starship	Objective	Destination	TC
Enterprise U	Exploration U	restricted U	U

or both S- and U-users see the following instance

[7] As discussed in section 4.6, only those U-users with the unrestrict privilege for this field can update it.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	null U	U

In the former event an attempt by a U-user to update the destination of the Enterprise to Talos will be rejected, whereas in the latter event the update will be allowed (without causing polyinstantiation).

The concept of restricted is straightforward, so long as we have a totally ordered lattice. In the general case of a partially ordered lattice some subtleties arise. How to completely eliminate polyinstantiation using restricted is discussed at length in [23]. In general, updating the value of a data element to restricted is a safe operation from a polyinstantiation viewpoint; that is, it cannot cause polyinstantiation. On the other hand, updating the value of a data element to a data value, say, at the c-level can be the cause of polyinstantiation. If polyinstantiation is to be completely prohibited, this update must require that the data element is restricted at all levels which do not dominate c. The fact that the data element is restricted at all levels below c can be verified by the usual integrity checking mechanisms in a DBMS [22]. However, to guarantee this at levels incomparable with c is more tricky. In preparing to enter a data value at the c level, we would need to start a system-low (really data element low) process which can then write-up. A protocol for this purpose is described in [23].[8]

4 SEMANTICS OF PCS

We now describe and motivate the intuitive semantics underlying our concept of PCS (i.e., polyinstantiation for cover stories). A formal model is given in the appendix. In a nutshell, PCS combines the "one tuple per tuple class" concept discussed in section 2 with the "restricted" concept of section 3. The basic motivation for PCS can be appreciated by considering the following instance of SOD.

Starship	Objective	Destination	TC
Enterprise U	restricted U	Talos U	U
Enterprise U	Spying S	Rigel S	S

In this case the Destination attribute of the Enterprise is polyinstantiated, so that <Talos,U> is a cover story for the real S destination of Rigel. The Objective is not polyinstantiated.

In the rest of this section we will discuss various aspects of PCS in turn, leading up to a summary at the end of this section which reiterates the main points.

[8] It should be noted that this protocol works for any arbitrary lattice, and does not require any trusted subjects. The use of trusted subjects will allow simpler protocols for this purpose.

4.1 Polylow Revisited

Let us reconsider the occurrence of polyinstantiation due to polylow, as discussed by example in section 3.3. This example begins with S- and U-users respectively having the following views of SOD.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Rigel S	S

Starship	Objective	Destination	TC
Enterprise U	Exploration U	null U	U

So far there is no polyinstantiation. Polyinstantiation occurs in the example when a U-user updates the destination of the Enterprise to be Talos.

In developing PCS we will take a slightly different perspective on this example. The shift in viewpoint, although very small, is extremely significant for the semantics of polyinstantiation. In our opinion it is a mistake to say that polyinstantiation does not exist in the S-instance of SOD given above. Indeed this instance should be correctly shown as follows.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	null U	U
Enterprise U	Exploration U	Rigel S	S

But then polyinstantiation already exists prior to the U-user updating the destination of the Enterprise to be Talos! This update merely modifies an already polyinstantiated relation instance to the one given below.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
Enterprise U	Exploration U	Rigel S	S

With this perspective, *element polyinstantiation can occur only due to polyhigh.* Polylow simply cannot be the cause of element polyinstantiation. Consequently, polyinstantiation will occur only by the deliberate action of a user to whom the polyinstantiation is immediately available. In other words, polyinstantiation does not occur as a surprise.

4.2 The Semantics of Null

The issue here is a subtle one, but one that is very important to resolve properly; so as to get a good semantics for PCS. Our proposal is remarkably simple: a "null" value should be treated just like any data value (except in the apparent key fields where "null" should not occur). Previous work on the semantics of null in polyinstantiated databases has taken the view that null's are subsumed by non-null values independent of the access class [9, 21]. In this case the first tuple in the following relation

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	null	U	U
Enterprise	U	Exploration	U	Rigel	S	S

is subsumed by the second tuple, resulting in the following relation used in our polylow example of section 3.3.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	Rigel	S	S

In PCS the former relation is quite acceptable. The latter can be acceptable, but only if the lower limit on the classification of the destination attribute is S.

To further illustrate the semantics of null in PCS, consider the following relation.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	null	U	U
Enterprise	U	Exploration	U	null	S	S

PCS will consider this to be a polyinstantiated relation. The fact that we have null's rather than data values in the polyinstantiated field has no bearing on this issue. We note that the semantics of null in [9, 21] require all null values to be classified at the level of the apparent key (U in this case), thereby deeming the second tuple as illegal.

4.3 The Semantics of Update

Our interpretation of the semantics of an SQL UPDATE command is identical to the one in the standard relational model: An update command is used to change values in tuples that are already present in a relation. In short, UPDATE does not cause polyinstantiation. UPDATE is a set level operator; i.e., all tuples in the relation which satisfy the predicate in the update statement are to be updated (provided the resulting relation satisfies integrity constraints).

The UPDATE statement executed by a c-user (i.e., a user with clearance c) has the following general form.

UPDATE R
SET $\quad A_i = s_i[, A_j = s_j] \ldots$
[WHERE p]

Here, s_k is a scaler expression, and p is a predicate expression which identifies those tuples in R_c that are to be modified. The predicate p may include conditions involving the classification attributes, in addition to the usual case of data attributes. The assignments in the SET clause, however, can only involve the data attributes. The corresponding classification attributes are implicitly determined to be c. In PCS this statement is interpreted, from the c-users perspective, to apply only to tuples with TC = c as follows.

```
UPDATE $R$
SET    $A_i = s_i\,[, A_j = s_j] \ldots$
WHERE  $[p \wedge ]$ TC $= c$
```

To be specific consider the following relation instance

Starship	Objective	Destination	TC
Enterprise U	Exploration U	null U	U

to which a U-user applies the following UPDATE command

```
UPDATE SOD
SET    Destination = “Talos”
WHERE  Starship = “Enterprise”
```

This statement is interpreted in PCS as follows

```
UPDATE SOD
SET    Destination = “Talos”
WHERE  Starship = “Enterprise” ∧ TC = U
```

giving us the following result

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U

Next, suppose a S-user executes the following UPDATE statement

```
UPDATE SOD
SET    Destination = “Rigel”
WHERE  Starship = “Enterprise”
```

PCS interprets this as follows

```
UPDATE SOD
SET    Destination = “Rigel”
WHERE  Starship = “Enterprise” ∧ TC = S
```

Since there is no secret tuple for the Enterprise, this UPDATE has no effect.

4.4 Propagation of Updates

As discussed in section 2.3 the update of an attribute must propagate into polyinstantiated tuples. For example, consider the above UPDATE by a U-user in context of the following relation.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	null U	U
Enterprise U	Spying S	null U	S

The S-tuple is invisible to the U-user who therefore sees exactly the scenario described above, i.e., the destination of the Enterprise is set to <Talos,U>. The point of update propagation is that this change must also be reflected in the S-tuple. That is, S-users should now see the following relation.

Starship	Objective	Destination	TC
Enterprise U	Exploration U	Talos U	U
Enterprise U	Spying S	Talos U	S

In this relation the Destination attribute of the U-tuple has been explicitly updated by a U-user. The Destination attribute of the S-tuple is implicitly updated unknown to the U-user. The fact that the Destination attribute of the S-tuple is classified U indicates that this implicit update is desired.

Suppose further, that a U-user executes the following UPDATE statement.

```
UPDATE SOD
SET    Objective = "Mining"
WHERE  Starship = "Enterprise"
```

After this update S-users will see the following relation

Starship	Objective	Destination	TC
Enterprise U	Mining U	Talos U	U
Enterprise U	Spying S	Talos U	S

This behavior can be implemented in a kernelized architecture using decompositions similar to [14]. Detailed discussion of this is beyond the scope of this paper.

4.5 Polyinstantiating Updates

In addition to the usual UPDATE statement, whose interpretation is given above, we propose in PCS to introduce a polyinstantiating update statement to allow users to explicitly request polyinstantiation. This statement is called a PUPDATE (i.e., polyinstantiating UPDATE) statement and has the same general format as an UPDATE, as shown below.

PUPDATE R
SET $A_i = s_i[, A_j = s_j] \ldots$
[WHERE p]

The interpretation of this statement in PCS is intuitively speaking, to polyinstantiate whenever a write-down is imminent. To be concrete consider the following relation instance

Starship	Objective	Destination	TC
Enterprise U	Exploration U	null U	U

to which a S-user applies the following UPDATE command

```
PUPDATE SOD
SET        Destination = "Rigel"
WHERE      Starship = "Enterprise"
```

Since overwriting the null destination in place would result in a write-down, PCS will interpret this statement as a request to polyinstantiate a secret destination. It will therefore insert a new tuple into the relation, identical to the one above, but with a secret destination of Rigel. This will give us the following relation.

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	null	U	U
Enterprise	U	Exploration	U	Rigel	S	S

Subsequent PUPDATEs by S-users to the Enterprise are treated just like UPDATES. For example, the following PUPDATE statement

```
PUPDATE SOD
SET        Destination = "Sirius"
WHERE      Starship = "Enterprise"
```

by an S-user will be interpreted in PCS as

```
UPDATE SOD
SET      Destination = "Sirius"
WHERE    Starship = "Enterprise" ∧ TC = S
```

giving us the following result

Starship		Objective		Destination		TC
Enterprise	U	Exploration	U	null	U	U
Enterprise	U	Exploration	U	Sirius	S	S

In other words a PUPDATE requests polyinstantiation if necessary to prevent a write-down, but otherwise is identical to an UPDATE.

4.6 The Semantics of Restricted

Our proposal in PCS is to treat "restricted" for the most part as just another data value. The main difference comes about when restricted is changed to unrestricted (i.e., some value other than restricted), and vice versa. The usual write privilege for the data item in question should not authorize these special updates which change restricted to unrestricted, and vice versa. Otherwise restricted provides no additional protection. Our proposal in PCS is to provide two additional access privileges as follows.

1. The *restrict* privilege on a data item authorizes a write operation which changes unrestricted to restricted.
2. The *unrestrict* privilege on a data item authorizes a write operation which changes restricted to unrestricted.

The possession of restrict and unrestrict privileges must be carefully controlled by non-discretionary means to make this effective. One possibility is to tie the use of these privileges to some kind of a mandatory integrity label on the subject. Another possibility is to control the propagation of these privileges, by non-discretionary means such as described in [20, 24], so it can be determined who can possess them (i.e., with efficient safety analysis).

The meaning of <restricted,c> in a data element is that ordinary c-users cannot modify this field. Only a c-user with the unrestrict privilege for that field is allowed to write into it. Similarly, ordinary c-users cannot write the restricted value into a data field in the first place.

4.7 Summary

In summary we can describe the salient features of PCS as follows.

1. No entity polyinstantiation (which greatly facilitates referential integrity).
2. Element polyinstantiation only by explicit polyhigh PUPDATE requests.
3. One tuple per tuple class for a given apparent primary key.
4. UPDATEs apply only to tuples at the user's access class, and propagate to higher level tuples.
5. Nulls are treated like any other data value.
6. Polyinstantiation is further controlled by restricted.
7. Changing restricted to unrestricted and vice versa requires special privileges. The former is specially dangerous in terms of possible polyinstantiation and should be executed only with proper protocols.

Finally, we note that PCS can be implemented in a kernelized architecture using decompositions similar to [14]. Detailed discussion of this will require another paper.

5 CONCLUSION

In this paper we have brought together a number of our previously published ideas, along with some new ones. We have also incorporated some concepts proposed by other researchers in the polyinstantiation arena. We have adapted and refined these ideas, while combining them into a consistent, intuitive and flexible package called PCS (i.e., polyinstantiation for cover stories).

PCS has several advantages over other proposals for incorporating cover stories in a multilevel relational DBMS. Most noteworthy are its uniform query interface and its flexibility. One can ask the same query and expect to be shown cover stories only when they exist, rather than having to explicitly ask for them. The cover stories are created upon need and may disappear and reappear from time to time for the same entity. Using explicit attributes to accommodate such situations makes for a rigid structure in which the DBA (Database Administrator) is the ultimate authority regarding creation of cover stories. PCS puts this power in the users' hands, where it properly belongs.

There is little, if any, experience regarding the use of multilevel DBMSs. Until recently, there have been no systems to use. Systems are now emerging but with many ad hoc features built into their data models. We must give users a flexible vehicle to experiment with. We believe PCS provides a useful data model for this purpose.

References

1. Rae K. Burns, "Referential Secrecy." *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, May 1990, pages 133-142.
2. F. Cuppens and K. Yazdanian, "A "natural" decomposition of multi-level relations," *Proc. IEEE Symposium on Security and Privacy*, May 1992, pages 273-284.
3. Dorothy E. Denning, Teresa F. Lunt, Roger R. Schell, Mark Heckman, and William R. Shockley, "A multilevel relational data model." *Proc. IEEE Symposium on Security and Privacy*, April 1987, pages 220-234.
4. Dorothy E. Denning, Teresa F. Lunt, Roger R. Schell, William R. Shockley, and Mark Heckman, "The SeaView security model." *Proc. IEEE Symposium on Security and Privacy*, April 1988, pages 218-233.
5. Dorothy E. Denning, "Lessons Learned from Modeling a Secure Multilevel Relational Database System." In *Database Security: Status and Prospects,* (C. E. Landwehr, editor), North-Holland, 1988, pages 35-43.
6. Department of Defense National Computer Security Center. *Department of Defense Trusted Computer Systems Evaluation Criteria.* DoD 5200.28-STD (1985).
7. Gajnak, G.E. "Some Results from the Entity-Relationship Multilevel Secure DBMS Project." *Aerospace Computer Security Applications Conference*, pages 66-71 (1988).
8. J. Thomas Haigh, Richard C. O'Brien, and Daniel J. Thomsen, "The LDV Secure Relational DBMS Model." *Database Security IV: Status and Prospects*, S. Jajodia and C. E. Landwehr (editors), North-Holland, 1991, pages 265-279.
9. Sushil Jajodia and Ravi S. Sandhu, "Polyinstantiation integrity in multilevel relations." *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, May 1990, pages 104-115.
10. Sushil Jajodia and Ravi S. Sandhu, "A formal framework for single level decomposition of multilevel relations." *Proc. IEEE Workshop on Computer Security Foundations*, Franconia, New Hampshire, June 1990, pages 152-158.
11. Sushil Jajodia and Ravi S. Sandhu, "Polyinstantiation integrity in multilevel relations revisited." *Database Security IV: Status and Prospects*, S. Jajodia and C. E. Landwehr (editors), North-Holland, 1991, pages 297-307.
12. Sushil Jajodia, Ravi S. Sandhu, and Edgar Sibley, "Update semantics of multilevel relations." *Proc. 6th Annual Computer Security Applications Conf.*, December 1990, pages 103-112.

13. Sushil Jajodia and Ravi S. Sandhu, "Database security: Current status and key issues," *ACM SIGMOD Record,* Vol. 19, No. 4, December 1990, pages 123-126.
14. Sushil Jajodia and Ravi S. Sandhu, "A novel decomposition of multilevel relations into single-level relations." *Proc. IEEE Symposium on Security and Privacy,* Oakland, California, May 1991, pages 300-313.
15. Sushil Jajodia and Ravi S. Sandhu, "Toward a multilevel secure relational data model," *Proc. ACM SIGMOD Int'l. Conf. on Management of Data,* Denver, Colorado, May 29-31, 1991, pages 50-59.
16. Sushil Jajodia and Ravi S. Sandhu, "Enforcing Primary Key Requirements in Multilevel Relations," *Proc. 4th RADC Workshop on Multilevel Database Security,* Rhode Island, April 1991.
17. Teresa F. Lunt, Dorothy E. Denning, Roger R. Schell, Mark Heckman, and William R. Shockley, "The SeaView security model." *IEEE Transactions on Software Engineering,* Vol. 16, No. 6, June 1990, pages 593-607.
18. Teresa F. Lunt and Donovan Hsieh, "Update semantics for a multilevel relational database." *Database Security IV: Status and Prospects,* S. Jajodia and C. E. Landwehr, (editors), North-Holland, 1991, pages 281-296.
19. Teresa F. Lunt, "Polyinstantiation: an inevitable part of a multilevel world." *Proc. IEEE Workshop on Computer Security Foundations,* Franconia, New Hampshire, June 1991, pages 236-238.
20. Ravi S. Sandhu, "The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes." *Journal of ACM* 35(2):404-432 (1988).
21. Ravi S. Sandhu, Sushil Jajodia, and Teresa F. Lunt, "A new polyinstantiation integrity constraint for multilevel relations." *Proc. IEEE Workshop on Computer Security Foundations,* Franconia, New Hampshire, June 1990, pages 159-165.
22. Ravi S. Sandhu and Sushil Jajodia, "Integrity Mechanisms in Database Management Systems." *Proc. 13th NIST-NCSC National Computer Security Conference,* Washington, D.C., October 1990, pages 526-540.
23. Ravi S. Sandhu and Sushil Jajodia, "Honest Databases That Can Keep Secrets." *Proc. 14th NIST-NCSC National Computer Security Conference,* Washington, D.C., October 1991, pages 267-282.
24. Ravi S. Sandhu, "The Typed Access Matrix Model." *Proc. IEEE Symposium on Research in Security and Privacy,* Oakland, California, May 1992, pages 122-136.
25. Simon R. Wiseman, "On the Problem of Security in Data Bases." In *Database Security III: Status and Prospects,* (Spooner, D.L. and Landwehr, C.E., editors), North-Holland, 1990, pages 143-150.

APPENDIX: FORMAL MODEL OF PCS

A *multilevel relation* consists of the following two parts.[9]

Definition 1. [RELATION SCHEME] A state-invariant multilevel relation scheme

$$R(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

[9] For simplicity the formal model is stated in terms of individual attributes. It can be generalized by replacing each A_i by an attribute group in a straightforward manner.

where each A_i is a *data attribute* over domain D_i, each C_i is a *classification attribute* for A_i and TC is the *tuple-class* attribute. The domain of C_i is specified by a range $[L_i, H_i]$ which defines a sub-lattice of access classes ranging from L_i up to H_i. The domain of TC is $[\text{lub}\{L_i : i = 1 \ldots n\}, \text{lub}\{H_i : i = 1 \ldots n\}]$ (where lub denotes the least upper bound). Let $A_1 = AK$ be the apparent primary key.

Definition 2. [RELATION INSTANCES] A collection of state-dependent *relation instances*

$$R_c(A_1, C_1, A_2, C_2, \ldots, A_n, C_n, TC)$$

one for each access class c in the given lattice. Each instance is a set of distinct tuples of the form $(a_1, c_1, a_2, c_2, \ldots, a_n, c_n, tc)$ where each $a_i \in D_i$ or $a_i = \text{null}$, $c \geq c_i$ and $tc = \text{lub}\{c_i : i = 1 \ldots n\}$. Moreover, if a_i is not null then $c_i \in [L_i, H_i]$. We require that c_i be defined even if a_i is null, i.e., a classification attribute cannot be null.

Property 3. [Entity Integrity] Let AK be the apparent key of R. A multilevel relation R satisfies entity integrity if and only if for all instance R_c of R and $t \in R_c$

1. $A_i \in AK \Rightarrow t[A_i] \neq \text{null}$,
2. $A_i, A_j \in AK \Rightarrow t[C_i] = t[C_j]$, i.e., AK is uniformly classified, and
3. $A_i \notin AK \Rightarrow t[C_i] \geq t[C_{AK}]$ (where C_{AK} is defined to be the classification of the apparent key).

Property 4. [No Entity Polyinstantiation] A multilevel relation R has no entity polyinstantiation if and only if $AK \rightarrow C_{AK}$.[10]

Property 5. [Tuple Integrity] A multilevel relation R has tuple integrity if and only if $AK, C_{AK}, TC \rightarrow A_i$. (In context with Property 4, it suffices to require $AK, TC \rightarrow A_i$.)

Property 6. [Entity Element Integrity] A multilevel relation R has entity element integrity if and only if $AK, C_{AK}, C_i \rightarrow A_i$. (In context with Property 4, it suffices to require $AK, C_i \rightarrow A_i$.)

Property 7. [Inter-Instance Integrity] R satisfies inter-instance integrity if and only if for all $c' \leq c$ we have $R_{c'} = \{t | t \in R_c \wedge t[TC] \leq c'\}$.

Property 8. [Referential Integrity] Let A_j be a foreign key of $R(A_1, C_1, \ldots, A_n, C_n)$ referencing an entity in $Q(B_1, C_1, \ldots, B_m, C_n)$. R and Q satisfy referential integrity if and only if for all c, if $t \in R_c$ with $t[A_j] \neq$ null or restricted then there exists $q \in Q_c$ such that $t[A_j] = q[B_1]$.

[10] The notation $X_1, \ldots, X_n \rightarrow Y$ signifies that Y is *functionally dependent* on $X_1, \ldots, X_n$, that is, it is not possible to have two tuples with the same values for $X_1, \ldots, X_n$ but different values for Y.

Formalization of UPDATE and PUPDATE semantics is omitted due to lack of space. The formalization is very similar in outline to the minimal propagation semantics of [12].