Computers & Security, 8 (1989) 229–245

Refereed Article

Prototyping to Explore MLS/DBMS Design

D. J. Thomsen,[1] W. T. Tsai[2] and M. B. Thuraisingham[3]

[1] *Honeywell, Secure Computing Technology Center, St. Anthony, MN, U.S.A.*
[2] *Department of Computer Science, University of Minnesota, Minneapolis, MN, U.S.A.*
[3] *Honeywell, Corporate Systems Development Division, Golden Valley, MN, U.S.A.*

This paper examines prototyping as a research tool for studying multilevel secure databases (MLS/DBMS). The paper proposes that an MLS/DBMS design can be quickly prototyped in Prolog. The prototype is then used as a research tool to experiment with the policies and models of the MLS/DBMS. To illustrate the principle, we built a Prolog prototype based on the Bell and LaPadula model. This prototype emphasizes a modular security policy to encourage reuse for other security mechanisms. The prototype also supports an inference control mechanism. It has proved to be a useful research tool for studying MLS/DBMS policies and models. Furthermore, since a prototype using Prolog can be built very quickly, we suggest that future MLS/DBMS models be prototyped and studied before costly mistakes are incurred in a full-scale implementation.

1. Introduction

Interest in database security is growing, and researchers have proposed many security policies, models, and designs. After a design is chosen, developing an MLS/DBMS is usually expensive. A sound design is critical in avoiding wasted effort. Various design issues should be explored at the design level, not at the implementation level. If the security policy is found to be incomplete or unenforceable during implementation costly changes must be made. This paper proposes that security policies and models be studied using a prototype before they are implemented. The prototype is an executable model that can be used as a research tool for developing and clarifying important issues. To illustrate the concept, we built an MLS/DBMS prototype based on the Bell and LaPadula model.

Even though, in general, Prolog is not a good implementation language, it is excellent for prototyping an MLS/DBMS. First, the language encompasses all the power of a relational database. This means many details of the MLS/DBMS are handled by Prolog. Secondly, since the prototype is only a model, Prolog's efficiency is not an important issue, but the performance is more than adequate for conducting experiments. Finally, researchers have proposed using logic to control inferences [7, 8], but a relational database and logic are often incompatible. In Prolog this problem does not occur, since it provides an environment where both can be used.

The paper is split into two main parts. The first discusses the design issues involved in building the Prolog prototype. The second discusses the features of the prototype as it stands today.

1. Design Issues

One of the major concerns in designing the prototype is to make it as generic as possible in the hope that parts can be used in other prototypes.

0167-4048/89/$3.50

The prototype stresses a modular view of the Bell and LaPadula model. Consider the mandatory and discretionary policies of Bell and LaPadula as fences surrounding the data. Both fences must be crossed before access is granted (see Fig. 1). In the Bell and LaPadula model the mandatory component requires the user to satisfy the simple security and *-properties. The discretionary component requires that the user's name be listed on an access control list (ACL). Enforcing the entire security policy then becomes a matter of checking to see if the user can cross all fences before granting access. Fulfilling the requirements of a security component then becomes a "key" for accessing data.

This idea of fences leads to a modular solution in Prolog, which further supports the choice of Prolog as the prototyping language. Another advantage of this modular view of the Bell and LaPadula model is that new fences can be added to create a more comprehensive security policy. For example, the current prototype has an additional fence for controlling inferences. If inferences have previously been detected, the prototype can control them by preventing access to data that allows inferences. It is important to note that this method can only be used for inferences that are known to exist.

The "key" required for the inference control fence is a list of conditions to be satisfied. If each condition can be satisfied, access is granted. The conditions check and update the audit information regarding the user's previous queries. Here is an example inference control component: "If a user has seen fact x he cannot see fact y, because this allows inference of sensitive data." If a user, John tries to retrieve fact y, the condition "not(seen(John, x))" must be satisfied.

Each fence is a modular component of the security policy. As a rule any component can be strengthened, weakened, replaced, or eliminated altogether without affecting the integrity of the other security components. The exception to this rule is the relationship between the mandatory and inference control components which will be discussed later in Section 3.1.5. Of course after a component is modified, the integrity and consistency of the com-

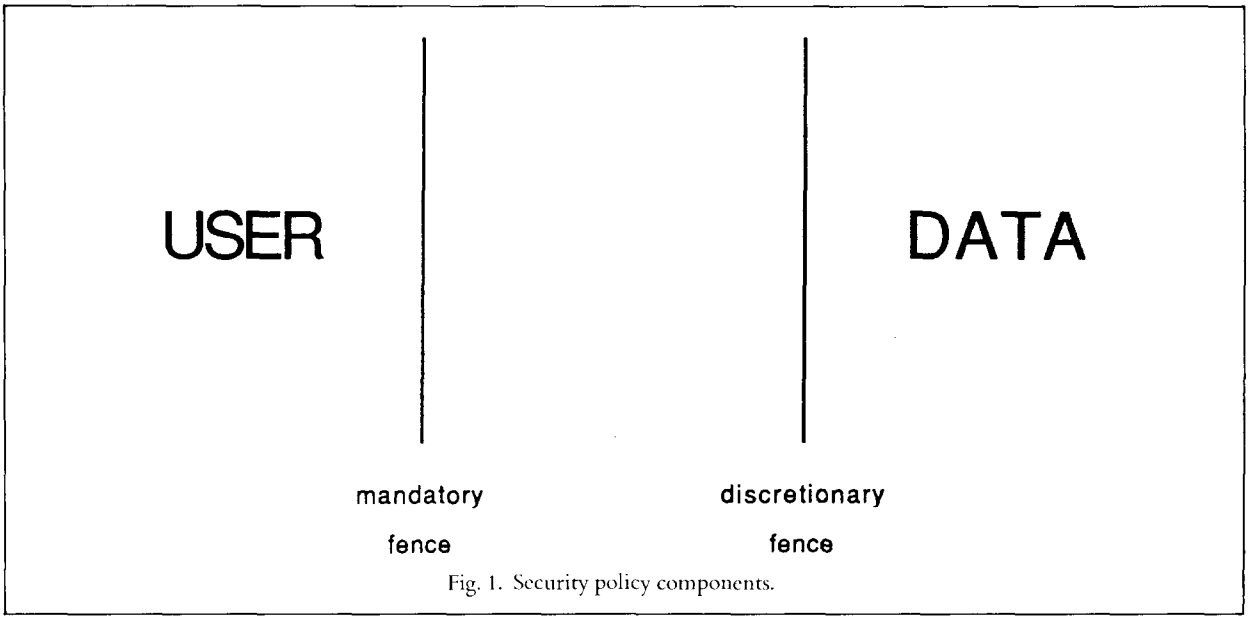

Fig. 1. Security policy components.

ponent must be checked to assure that the change has the desired effect on the overall security policy. As an example, if someone decides "Top secret information must be viewed during business hours," either a component can be modified or a new component created to enforce this statement. The component must be verified to insure it is doing its job, but the other components need not be verified again.

2.1 Commercial Design Issues

Commercial applications have been characterized as emphasizing the ability to modify data, while military applications emphasize the ability to read data [3]. Consider an accounting database; Which is more important, a user looking at the accounting records or a user modifying the accounting records to steal an untold amount of money? Certainly, the commercial world wants to control who can see their accounting records, but keeping the integrity of the information is more important. Data integrity is insured by two practices, well-formed transactions and separation of duties [3]. Well-formed transactions require that users cannot change the data arbitrarily, *i.e.* only program x can manipulate data y. Separation of duties requires that several people crosscheck the transaction. For example, a separation of duty policy might require that the person who creates the transaction cannot execute it, and the person who executes it cannot create the transaction. Thus two people are required to corrupt the data. Well-formed transactions and separation of duties form the basis of commercial security.

The interests of the military and the commercial world are not incompatible, because military applications are also concerned with data integrity. The goals of data integrity and data access can possibly be incorporated into a single secure database. Experiments with security policies using a Prolog prototype would be a good environment for experimenting with combining data integrity and data access.

3. The Current Prototype

This section discusses the current prototype in three parts. The first discusses the security policies used by the prototype. The second part discusses how data stored in the prototype are separated. The final part is a description of the prototype from an experimenter's viewpoint.

The prototype was written in a public domain version of Prolog called SB-Prolog. However, it can be easily ported to other varients as described in ref. [2], because it uses only basic Prolog constructs. The first version of the prototype was working in a few weeks. Many additional weeks were spent exploring different designs of the inference control component. Modular components made it possible to change the prototype quickly and explore ideas with immediate feedback.

Verification of Prolog and the Prolog prototype were not considered, because the goal is to test the soundness and functionality of the design. Only when the design has proven sound and useful should it be verified.

3.1 The Security Policy

The security policy of the prototype has three components: a mandatory component for access control across security levels, a discretionary component for controlling group accesses, and an inference component used to control inferences among security levels.

3.1.1 Mandatory Component

The mandatory access component is based on the simple security property and the *-property. Users can read data at or below their security level, and can write data at or above their security level. The prototype has four security levels: unclassified, confidential, secret, and top secret. The levels are ranked in the usual manner by the Prolog predicate security_level_at_or_above (L_1, L_2). The predicate succeeds if L_1 is at or above L_2. This predicate can be used for both the simple and *-properties by switching L_1 and L_2.

3.1.2 Discretionary Component

The discretionary component is implemented using ACLs. An ACL is associated with each fact in the database. If the user's name is on the ACL, he is allowed to read the data. Determining the ACL when adding data to the database poses a problem. The user is trusted to enter an ACL with the appropriate user names in it. If the user does not enter an ACL the prototype creates a default ACL consisting only of the user's name. This effectively fences the data into the user's private database.

3.1.3 Inference Component

The inference control component is a list of conditions to be satisfied before access is granted. These conditions must be specified when the database is created. As a result only inferences that are known to exist can be controlled.

The conditions are based on audit information of the user's past queries. If the user can infer sensitive data by combining facts released in the past with the fact being requested, the requested fact is not released. A list of conditions is associated with each fact in the database, and all conditions must be met before access is granted. These lists of conditions specify the inference component of the security policy. These conditions are also responsible for updating audit information when information with inference potential is released.

3.1.4 Polyinstantiation

It is quite likely that two users may create the same relation at different levels. This duplication of facts has been called polyinstantiation [4]. Here is a simple example of polyinstantiation: Matt, a secret user, adds the fact "meeting(10 am)" to the database, indicating that he has a meeting at ten o'clock in the morning. Then Joe, an unclassified user, also attempts to add the fact "meeting(10 am)." The fact already exists, and Joe does not have permission to write over Matt's fact. If Joe is denied the ability to add the fact he knows that a higher level user has a meeting at ten. The solution to this problem is polyinstantiation. Polyinstantiation is a necessary part of any multilevel security system [5].

Using the prototype we conducted brief experiments with polyinstantiation. One of the first results of the experiment was that the discretionary component can be used to distinguish between two polyinstantiated facts. If the relation's ACL contains only the author's name only the author can access it. To simplify the experiment the discretionary component was disabled to simulate the more general situation where the ACLs of the polyinstantiated facts overlap. When the discretionary component was disabled, the current prototype returned both polyinstantiated facts. It then became apparent that each fact must be labeled with its security level so that high level users can either distinguish between polyinstantiated facts themselves, or create queries to distinguish between the facts. This labeling allows the high level user to determine if the polyinstantiated facts are semantically different.

3.1.5 The Relationship Between the Mandatory and Inference Components

Controlling inferences has proven to be a difficult problem. Consider the following example: suppose facts x and y are unclassified, but a user seeing both x and y can infer confidential information. While unclassified users can see only x or y, users at the confidential or higher level should be able to see both facts. To prevent the inference component from denying access to legitimate users, the inference component must base its decisions on the security level of the user. Granting access based on security level is exactly what the mandatory component does, and therefore the two components must interact with each other.

The interdependence of the inference component and the mandatory component can be removed by duplicating the unclassified facts x and y at the confidential level without any inference conditions. Then users at the confidential or higher level can access the facts based only on the mandatory and discretionary components. The current prototype uses this method because it is an easy solution. The facts are simply duplicated in the database. It is important to note that this duplication of facts is

not polyinstantiation. In this case the database is duplicating semantically equivalent facts to enforce the security policy.

Duplicating facts to enforce the security policy leads to several problems. The first is that, if fact x is duplicated at the unclassified and confidential levels and the classified instance of fact x is modified, the facts are no longer identical. A trusted "downgrader" to update the unclassified version of x would be difficult to write, because of the potential downward information flow. Another problem is storage. In a large database, duplicating facts can require costly amounts of storage. Therefore duplicating facts to enforce the security policy is not acceptable.

The more acceptable solution then is to combine the mandatory and inference components by allowing the inference component to look at the security level of the user. There are two approaches to this solution. The first is to store the facts at the unclassified level. It would then be up to the inference component to protect the potential confidential information from unclassified users. The other approach is to have facts x and y stored at the confidential level. If an unclassified user tries to access only one fact, the inference component must downgrade the information. For practical purposes the two approaches are the same. In each approach, information is classified into two levels. Some of the information is marked as having inference potential, which means that under certain conditions the information must be treated as if it were on another level. Storing potentially classified material as unclassified may seem unacceptable, but since the inferences are known and the data were stored at the lower level it is assumed the lower level users needed to see it. On the other hand, if the facts are stored at the higher level, then the inference component must be trusted to downgrade sensitive data. Both approaches require that the inference component be a trusted part of the mandatory component. No matter what solution is chosen, the mandatory and the inference components must interact with each other, because both components rely on the security level of the data. Therefore the modularity between the inference control and mandatory components is lost. In other words, the consistency and integrity of one depends on the other.

3.1.6 Covert Channels Using the Inference Component

The prototype does not allow users to specify an inference control component when adding data to the database, because this opens a covert channel by use of audit information. The covert channel arises because the inference component has direct access to audit information. Thus, if users control the inference component, they can access audit information. Once a user has access to audit information a number of possible covert channels arise.

For example, suppose there is a high level Trojan horse (HLTH) that wants to communicate to a low level user (LLU). The HLTH creates a series of relations in the database, *to_LLU(1), to_LLU(2), ..., to_LLU(n).* Each relation corresponds to one bit in the n-bit binary message. The HLTH encodes a 1 by accessing the corresponding relation, or a 0 by not accessing it. To decode the string, a LLU adds a corresponding series of relations, *from_HLTH(1), from_HLTH(2), ..., from_HLTH(n).* Each *from_HLTH(x)* relation has an inference component that checks to see if the HLTH has seen the corresponding relation. If the HLTH has seen the relation, the inference component grants access to the relation. IF the HLTH has not seen the relation, it denies access to the *from_HLTH(x)* relation. Then the LLU tries to access the series of relations he just added. If the LLU can access a relation, he marks down 1; if he is denied access, he marks down 0. Thus the HLTH can send an arbitrary bit string to the LLU (see Fig. 2).

To prevent this covert channel, several things can be done. The biggest problem arises because the specification of the inference conditions is left up to the user. If the user was constrained to only writing general statements such as, facts x and y imply fact z, manipulation of audit data is more difficult. Some sort of trusted parser must be

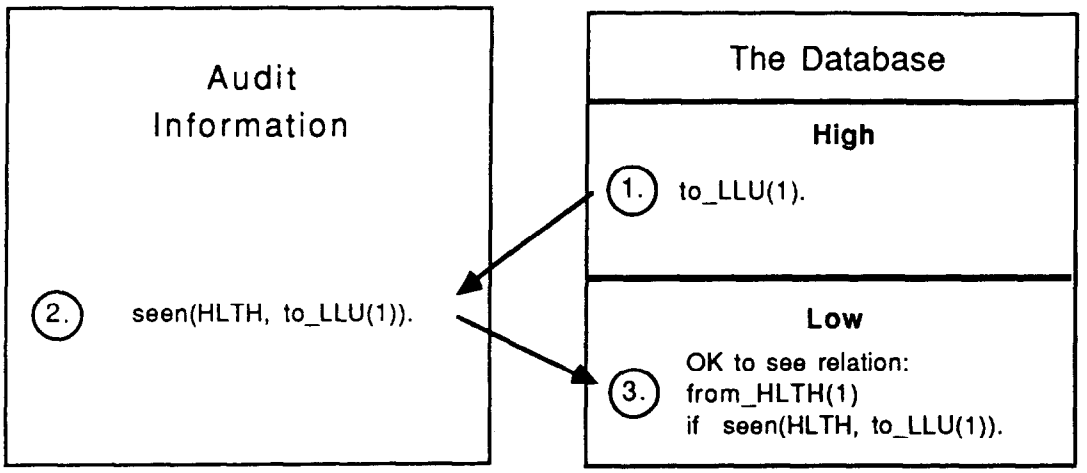

Fig. 2. A Trojan horse using audit information. 1. The high level Trojan horse adds the fact to the database. 2. The high level Trojan horse then accesses it, which creates an audit record. 3. A low level user then retrieves the information from the audit data.

written to turn these general statements into the proper inference conditions. It is important to note that these general rules must also be treated as multilevel data, because a covert channel arises if the LLU can see these general statements of a HLTH.

Another solution is to apply the same security policy to audit information that is applied to the rest of the database. Each piece of audit information would be tagged with a security level and an ACL. Queries about audit information are then forced to meet the mandatory and discretionary components of the security policy. Further study needs to be done to ensure that these restrictions provide the necessary security and do not limit the functionality of the inference component. If no suitable method for preventing this covert channel can be found, the conditions of the inference control component must continue to be trusted.

3.2 How Data are Separated

The prototype divides logical memory into four areas; program space, the user database, audit space, and query space (see Fig. 3). Each area is separate because it has special security requirements. An area is grouped together by Prolog predicates into different relations in the Prolog database.

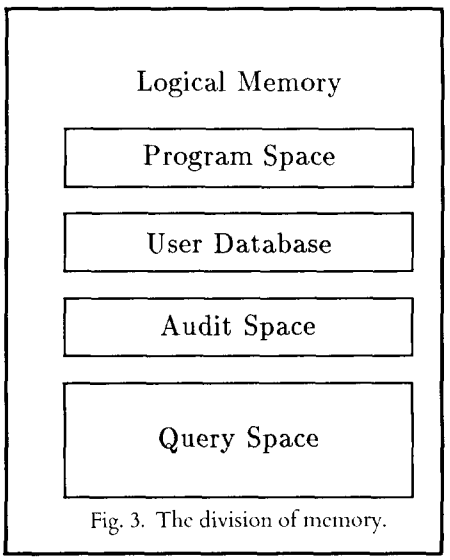

Fig. 3. The division of memory.

3.2.1 Query Space

Query space is where the user's data are stored, and is separated by the predicate "query." Facts stored

in query space are separated into the security levels and ACLs by fields in the query predicate. Each entry in the query predicate has four fields. The first field is the fact itself, such as "dog(snoopy)." The second field is the security level of the fact which is either unclassified, confidential, secret, or top_secret. The third field is the ACL used by the discretionary component. The fourth field is a list of inference control conditions to be executed by the inference control component (see the example in Fig. 4).

3.2.2 User Database

The user database is the multilevel equivalent of a password file. Entries in the user database contain the login name of the user, his password, and his security classification. The user database resides in its own area because users must access the database before their identity is known. Entries in the user database are sorted from other relations by the predicate "user_database." The "user_database" predicate has three fields; the user's name, the user's password, and a predicate called "key" containing information on the user's security level, group accesses etc. In the future, information to be used in the discretionary component regarding group membership can be added. Similarly, if the inference control component requires special background information on the user, it can be placed into the key as well.

3.2.3 Audit Space

Audit space contains information regarding a user's past queries. Information in audit space is separated by the "audit_space" predicate. Audit space is primarily used as a history mechanism for controlling inferences. The inference control component determines whether information in audit space needs updating. Currently, audit information is kept only on data with inference potential. Only the inference control component has access to audit space owing to the covert channel mentioned in Section 3.1.3. In a production MLS/DBMS, a great deal more audit information would be kept for general auditing.

```
query(dog(snoopy), unclassified, [dan, tom], [])
```

FACT	a dog named Snoopy,
LEVEL	unclassified,
ACL	users Dan and Tom,
Inference Component	none

Fig. 4. A database entry.

3.2.4 Program Space

Program space contains the executable code of the prototype. Since it is the part of the prototype that actually runs, it cannot be contained in a Prolog predicate. Therefore data belong to program space if they are not part of another area.

3.3 System Description

The speed of the prototype is acceptable for studying security policies on small databases. For the example database in Appendix B containing 60 facts, the only noticeable delay occurs when loading the database into memory. The delay is caused by the way Prolog hashes predicates for quick retrieval at run time. As a result the prototype is quite fast at searching the database.

The prototype contains all the necessary features for studying security policies. The most important feature is the prototype's simplicity. The entire prototype was done in a short period of time implying that even major alterations in the prototype can be done quickly. Because of the expressive power of Prolog, the prototype is only four pages long. However, because of the prototype's design, most experimental security policies require changes only in specific security components. See Appendix A for a sample run of the prototype, and Appendix B for the database that was used.

Several desirable features of a production DBMS were not implemented. Most of these features are not difficult to implement, but they are not essential for studying database security policies and models.

- Most notable is the output format of the queries. A query such as "dog(beagle, Who)" returns the

entire fact "dog(beagle, snoopy)" rather than just the name "snoopy."
- There is no mechanism for performing a query and storing the results into the database.
- Facts that users add to the database are not permanently stored in a file. In the future this can be added by dumping the contents of memory to a file when quitting.
- The user interface is difficult to work with because it requires a great deal of exact typing.
- The current prototype cannot do compound queries like "dog(beagle, Who)" and "famous(Who)." Compound queries only slightly complicate security since each element of the query can be requested separately and the results combined appropriately.
- There are no facilities for modifying the security attributes of data inside the prototype. All such modifications must be done outside the prototype.
- Users are forced to logout and login again to change security levels. This requirement avoids complications arising from saved internal states when the user changes levels.
- There are no groups of users for the discretionary component.
- The facts stored in the database can only have one security level. Data objects with components at different security levels can be simulated by multiple relations at different security levels.

4. Conclusion

While a Prolog prototype is by no means a production MLS/DBMS, it provides a basis for studying the security policies and models of MLS/DBMS. The prototype discussed in this paper contains a mandatory access component based on the simple security and the *-property from the Bell and LaPadula model. The discretionary component of the security policy consists of an ACL associated with each piece of data. More importantly, the prototype is being used as a tool for exploring various design issues for an inference control component of the security policy. Furthermore, each security component can be modified to study new security policies and models.

This prototype is one of the earliest prototype MLS/DBMS with a mandatory, discretionary, and inference component in its security policy. In our experience with prototyping, several issues arose that would not have surfaced until implementation. The most important issues were the necessity of polyinstantiation, the relationship between the mandatory and inference components, and the potential covert channel through audit space. These issues could only be found by careful analysis of the design, by implementation, or prototyping. Until a design is built or prototyped certain issues that were thought to be straightforward can be found to be more complex than anticipated. Building a prototype is the cheapest method to catch the largest number of issues.

Many design decisions can be explored by experimenting with a prototype. Prototypes that appear feasible can be used as models for building a production MLS/DBMS in a conventional language. Without using Prolog, building a prototype as a research tool with comparable utility may take years. A great deal of time would also be required to modify such a prototype for experiments. By using Prolog, a wide range of security policies for both military and commercial applications can be studied at low cost. Thus we recommend that any future MLS/DBMS model and policies be prototyped before full-scale implementation starts.

Acknowledgment

We wish to acknowledge Tom Keefe for his help in proofreading the paper.

References

[1] D. E. Bell and L. J. LaPadula, Secure computer systems: unified exposition and multics interpretations, *Technical Rep. MTR-2997,* March 1976 (Mitre Corporation).

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* Springer, New York, 2nd edn., 1984.

[3] D. D. Clark and D. R. Wilson, A comparison of commercial and military computer security policies, *Proc. 1987 Symp. on Security and Privacy, April 1987,* pp. 184–194.

[4] D. E. Denning, T. F. Lunt, R. R. Schell, M. Heckman and W. R. Shockley, A multilevel relational data model, *Proc. 1987 Symp. on Security and Privacy, April 1987,* pp. 220–234.
[5] D. E. Denning, T. F. Lunt, R. R. Schell, M. Heckman and W. R. Shockley, Secure distributed data views: the Sea View formal security policy model, *A003: Interim Rep.,* July 20, 1987 (SRI International).
[6] B. B. Dillaway and J. T. Haigh, A practical design for multilevel security in secure database management systems, *Aerospace Security Conf., December 1986,* pp. 44–57.
[7] M. Morgenstern, Security and inference in multilevel database and knowledge base systems, *ACM SIGMOD Conf. Proc., 1987,* pp. 357–373.
[8] M. B. Thuraisingham, Security checking in relational database management systems augmented with inference engines, *Comput. Secur.,* 6 (6) (1987) 470–492.

Daniel J. Thomsen is an associate research scientist at Honeywell's Secure Computing Technology Center. He received his undergraduate degree in computer science and mathematics from the University of Minnesota, Duluth, and is currently pursuing his Ph.D. degree at the University of Minnesota.

W. T. Tsai received his Ph.D. and M.S. degrees in computer science from the University of California, Berkeley, in 1982 and 1986 respectively, and an S.B. degree in computer science and engineering from MIT in 1979. He is currently an assistant professor in the Department of Computer Science at the University of Minnesota. His areas of interest are software engineering, artificial intelligence, computer security, computer systems, and neural networks. He is a member of IEEE.

Bhavani M. Thuraisingham is a lead staff member at The MITRE Corporation doing research in database security and integrity. Previously she was a principal research scientist at Honeywell Inc. researching in data/knowledge base systems and an adjunct professor and member of the graduate faculty in the Department of Computer Science at the University of Minnesota. Before that she was a senior analyst at Control Data Corporation developing network systems and was a faculty member in the Department of Mathematics, University of Minnesota and the Department of Computer Science, New Mexico Institute of Mining and Technology. She received the Ph.D. degree in recursive functions and computability theory from the University of Wales, Swansea, U.K., the M.Sc. degree in mathematical logic from the University of Bristol, U.K., the M.S. degree in computer science from the University of Minnesota and the B.Sc. degree in mathematics and physics from the University of Sri-Lanka. She has published over 30 refereed technical papers in the areas of database security, distributed processing, AI applications and computability theory. She is a member of IEEE Computer society, ACM and Sigma-Xi.

Appendix A: Sample Run

This appendix contains a sample run of the MLS/DBMS prototype. The database used for this run appears in Appendix B. The database is for a fictitious hospital. Appendix C contains the user database. Instead of supplying names for the users they have been called secretary1 or accountant2.

The first test exercises the mandatory security component by having a secret user try to read an unclassified, a secret, and a top secret fact. The same secret user then attempts to write an unclassified, a secret and a top secret fact. Notice that the user cannot read the top secret fact once they have written it.

```
|?-run.
login:accountant1.
password:foo.
Enter query > hospital_name(Name).
hospital_name(county_general)
```

```
Search for more (y./n.)?y.
no.
Enter query > budget(1988, Dollars).
budget (1988, 100000)
Search for more (y./n.)?y.
no.
Enter query > supporting_congressmen(List).
no.
Enter query > add_fact(budget(1989, 150000), unclassified).
Security Violation.
Fact not entered.
Enter query > add_fact(budget(1989, 150000), secret).
Fact asserted.
Enter query > add_fact(proposed_budget(1990, 200000), top_secret).
Fact asserted.
Enter query > proposed_budget(Date, Amount).
no.
```

The next section tests the discretionary security component. The same secret user tries to read a fact from an unclassified user, a secret user, and a top secret user. All of these attempts fail. The secret user then writes a new fact which only they can access, since the default ACL contains only their name.

```
Enter query > shopping_list(List).
no.
Enter query > career_goals(List).
no.
Enter query > appointments(List).
no.
Enter query > add_fact(appointments([8, 10]), secret).
Fact asserted.
Enter query > appointments(List).
appointments ([8, 10])
Search for more (y./n.)?y.
no.
Enter query > logout.
```

This section shows the inference control capabilities of the prototype. The first test is an example of facts x and y imply z where x and y are unclassified and z is confidential. Note that the prototype remembers that the secretary has seen fact x even after they log out. Also note that the top secret user can see both facts x and y.

```
login:secretary1.
password:foo.
Enter query > x.
x
Search for more (y./n.)?y.
no.
```

```
Enter query > y.
no.
Enter query > logout.
login:secretary2.
password:foo.
Enter query > y.
y
Search for more (y./n.)?y.
no.
Enter query > x.
no.
Enter query > logout.
login:administrator.
password:foo.
Enter query > x.
x
Search for more (y./n.)?y.
no.
Enter query > y.
y
Search for more (y./n.)?y.
no.
Enter query > logout.
login:secretary1.
password:foo.
Enter query > y.
no.
Enter query > x.
x
Search for more (y./n.)?y.
no.
```

In the following tests several general inference control policies are involved. Unclassified users cannot know both the surgeon's name and home address. Unclassified and confidential users cannot know both a surgeon's name and patient survival rate. So for example one secretary can see names and IDs for keeping various records while another secretary can see addresses, but not names for other records. In the first test the unclassified secretaries can only see either a surgeon's name and ID, or a surgeon ID and an address. Note that the schedulers can see both the surgeons' names and their addresses, but not the surgeons' patient survival rates.

```
Enter query > surgeon(surgeon1, ID).
surgeon(surgeon1, s1)
Search for more (y./n.)?y.
no.
Enter query > surgeon_address(s1, Address).
no.
```

```
Enter query > logout.
login:scheduler1.
password:foo.
Enter query > surgeon(surgeon1, ID).
surgeon(surgeon1, s1)
Search for more (y./n.)?y.
no.
Enter query > surgeon_address(s1, Address).
surgeon_address(s1, delmar_625)
Search for more (y./n.)?y.
no.
Enter query > survival_rate(s1, Deaths, Live).
no.
Enter query > logout.
login:scheduler2.
password:foo.
Enter query > survival_rate(s1, Deaths, Live).
survival_rate(s1, 0, 5)
Search for more (y./n.)?y.
no.
Enter query > surgeon(s1, ID).
no.
Enter query > logout.
```

In this test, once examiner2 sees any surgeon's name, examiner2 cannot see any surgeon's survival rate. Thus examiner2 cannot use the process of elimination to infer a surgeon's survival rate. In this test the top secret administrator has access to both names and survival rates.

```
login:examiner1.
password:foo.
Enter query > survival_rate(Surgeon, Deaths, Live).
survival_rate(s1, 0, 5)
Search for more (y./n.)?y.
survival_rate(s2, 4, 6)
Search for more (y./n.)?y.
survival_rate(s3, 5, 5)
Search for more (y./n.)?y.
no.
Enter query > surgeon(Surgeon, ID).
no.
Enter query > logout.
login:examiner2.
password:surgeon(Surgeon, ID).
login:examiner2.
password:foo.
Enter query > surgeon(Surgeon, ID).
```

```
surgeon(surgeon1, s1)
Search for more (y./n.)?y.
surgeon(surgeon2, s2)
Search for more (y./n.)?y.
surgeon(surgeon3, s3)
Search for more (y./n.)?n.
Enter query > survival_rate(s3, Deaths, Live).
no.
Enter query > logout.
login:adminstrator.
password:surgeon(Surgeon, ID).
login:administrator.
password:foo.
Enter query > surgeon(Surgeon, ID).
surgeon(surgeon1, s1)
Search for more (y./n.)?y.
surgeon(surgeon2, s2)
Search for more (y./n.)?y.
surgeon(surgeon3, s3)
Search for more (y./n.)?n.
Enter query > survival_rate(ID, Deaths, Lives).
survival_rate(s1, 0, 5)
Search for more (y./n.)?y.
survival_rate(s2, 4, 6)
Search for more (y./n.)?y.
survival_rate(s3, 5, 5)
Search for more (y./n.)?y.
no.
Enter query > surgeon(Surgeon, ID).
surgeon(surgeon1, s1)
Search for more (y./n.)?y.
surgeon(surgeon2, s2)
Search for more (y./n.)?y.
surgeon(surgeon3, s3)
Search for more (y./n.)?y.
Enter query > logout.
```

Appendix B: The Database

This is the sample database used in sample run in Appendix A. For people not familiar with Prolog the conditions in the inference control component have been paraphrased and the first few predicates have been paraphrased in English. For those familiar with Prolog the free variable "_" has been replaced with a predicate starting with "any" to make the database more readable.

```
/ ***************************************************************************************************************
*Database for A simple MLS/DBMS.
 ***************************************************************************************************************/

/*
 * query(Fact, Security_Level, Access_Control_List, Inference_Control_List).
 */
/* ------------------------------ unclassified ------------------------------*/
/*
 * hospital_name(Hospital_Name)
 */
query(hospital_name(county_general),
    unclassified,
    [examiner1, examiner2, scheduler1, scheduler2, secretary1, secretary2, surgeon1, surgeon2, surgeon3,
    accountant1, accountant2, administrator], [] ).
```

A hospital with the name county general stored at the unclassified level with an access control list consisting of examiner1, ... No inference component conditions are specified.

```
/*
 * surgeon(Surgeon_Name, ID)
 */
query(surgeon(surgeon1, s1),
    unclassified,
    [examiner1, examiner2, scheduler1, scheduler2, secretary1, secretary2, surgeon1, surgeon2, surgeon3,
      accountant1, accountant2, administrator],
    [not(audit_space( seen(User, any_survival_facts))),
      not(audit_space( seen(User, any_surgeon_address))),
      add_fact(audit_space( seen(User, surgeon(surgeon1, any_id))))] ).
```

A surgeon with name surgeon1 and ID s1 stored at the unclassified level with an access control list consisting of examiner1, ... The inference component conditions are that the user has not seen any survival facts, and the user has not seen any surgeons' addresses, and then add the fact that the user has seen surgeon1's name and id.

```
query(surgeon(surgeon2, s2),
    unclassified,
    [examiner1, examiner2, scheduler1, scheduler2, secretary1, secretary2, surgeon1, surgeon2, surgeon3,
      accountant1, accountant2, administrator],
    [not(audit_space( seen(User, any_survival_facts))),
      not(audit_space( seen(User, any_surgeon_address))),
      asserta(audit_space( seen(User, surgeon(surgeon2, any_id))))] ).
query(surgeon(surgeon3, s3),
    unclassified,
    [examiner1, examiner2, scheduler1, scheduler2, secretary1, secretary2, surgeon1, surgeon2, surgeon3,
      accountant1, accountant2, adminstrator],
```

```
    [not(audit_space( seen(User, any_survival_rate))),
     not(audit_space( seen(User, any_surgeon_address))),
     asserta(audit_space( seen(User, surgeon(surgeon3, any_id))))] ).
/*
 * surgeon_address (ID, Address)
 */
query(surgeon_address(s1, delmar_625),
    unclassified,
    [examiner1, examiner2, scheduler1, scheduler2, secretary1, secretary2, surgeon1, surgeon2, surgeon3,
     accountant1, accountant2, administrator],
    [not(audit_space( seen(User, any_surgeon))),
     asserta(audit_space( seen(User, surgeon_address(s1, any_address))))] ).
query(surgeon_address(s2, elm_1703),
    unclassified,
    [examiner1, examiner2, scheduler1, scheduler2, secretary1, secretary2, surgeon1, surgeon2, surgeon3,
     accountant1, accountant2, administrator],
    [not(audit_space( seen(User, any_surgeon))),
     asserta(audit_space( seen(User, surgeon_address(s2, any_address))))] ).
query(surgeon_address(s3, grant_101),
    unclassified,
    [examiner1, examiner2, scheduler1, scheduler2, secretary1, secretary2, surgeon1, surgeon2, surgeon3,
     accountant1, accountant2, administrator],
    [not(audit_space( seen(User, any_surgeon))),
     asserta(audit_space( seen(User, surgeon_address(s3, any_address))))] ).

/*----------confidential----------*/

/*
 * survival_rate(ID, Death, Live)
 */
query(survival_rate(s1, 0, 5),
    confidential,
    [examiner1, examiner2, scheduler1, scheduler2, surgeon1, surgeon2, surgeon3, administrator],
    [not(audit_space( seen(User, any_surgeon))),
     asserta(audit_space( seen(User, survival_rate(s1,
    any_number,
    any_number))))] ).
query(survival_rate(s2, 4, 6),
    confidential,
    [examiner1, examiner2, scheduler1, scheduler2, surgeon1, surgeon2, surgeon3, administrator],
    [not(audit_space( seen(User, any_surgeon))),
     asserta(audit_space( seen(User, survival_rate(s2,
    any_number,
    any_number))))] ).
query(survival_rate(s3, 5, 5),
```

```
    confidential,
    [examiner1, examiner2, scheduler1, scheduler2, surgeon1, surgeon2, surgeon3, administrator],
    [not(audit_space( seen(User, any_surgeon))),
     asserta(audit_space( seen(User, survival_rate(s3,
    any_number,
    any_number)))))] ).

/*-------------------------------secret-------------------------------*/
/*
 * budget(Year, Amount)
 */
query(budget(1988, 100000),
    secret,
    [accountant1, accountant2, administrator], [] ).
/*
 * career_goals([Goal_1, Goal_2, ...]),
 */
query(career_goals([prestige, early_retirement]),
    secret,
    [accountant2], [] ).

/*-----------------------------top secret-----------------------------*/
/*
 * supporting congressman([Name_1, Name_2, ...]),
 */
query(supporting_congressman([congressman1, congressman2]),
    top_secret, [accountant1, accountant2, administrator], [] ).
/*
 * appointments([Time_1, Time_2, ...]),
 */
query(appointments([2, 3, 5]),
    top_secret, [accountant1, accountant2, administrator], [] ).

/*------------------------ example inference control ------------------------*/
/* Here is an example of a rule that could be used to help solve the inference problem. Both facts x and y are
    unclassified, but together they are confidential. ie. x and y ⇒ confidential. */
query(x,
    unclassified, [secretary1, secretary2],
    [not(audit_space(key(_,User,_), seen(User, y))),
    asserta(audit_space(key(_,User,_), seen(User, x)))] ).

query(y,
    unclassified,
    [secretary1, secretary2],
    [not(audit_space(key(_,User,_), seen(User, x))),
    asserta(audit_space(key(_,User,_), seen(User,y)))] ).
```

```
query(x, confidential, [secretary1, secretary2, administrator], [] ).
query(y, confidential, [secretary1, secretary2, administrator], [] ).

/ ************************************************************************************************************
* User Database for A simple secure database.
 ************************************************************************************************************ /

/*
 * user_database(User_Name, Password, Key).
 */

/*----------------------------Unclassified----------------------------*/
user_database(secretary1, foo, key(unclassified, secretary1, [])).
user_database(secretary2, foo, key(unclassified, secreatry 2, [])).

/*----------------------------Classified----------------------------*/
user_database(examiner1, foo, key(classified, examiner1, [])).
user_database(examiner2, foo, key(classified, examiner2, [])).

user_database(scheduler1, foo, key(classified, scheduler1, [])).
user_database(scheduler2, foo, key(classified, scheduler2, [])).

/*----------------------------Secret----------------------------*/
user_database(accountant1, foo, key(secret, accountant1, [])).
user_database(accountant2, foo, key(secret, accountant2, [])).

/*----------------------------Top Secret----------------------------*/
user_database(administrator, foo, key(top_secret, administrator, [])).
```