CRYPTOGRAPHIC CHECKSUMS FOR MULTILEVEL DATABASE SECURITY

Dorothy E. Denning

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025.

The 1982 Air Force Summer Study on Multilevel Data Management Security recommended several approaches to designing a multilevel secure database system. One of the approaches uses an untrusted database system to manage the data, and an isolated trusted filter to enforce security. The filter attaches a security classification label to each data record, computes an unforgeable cryptographic checksum over the record (including the label), and stores the checksum in the database. The checksum protects against modification to the data and its classification label. This paper discusses the implementation, security, and limitations of the approach.

Introduction

In July 1982, the Air Force sponsored a summer study on Multilevel Data Management Security at Woods Hole, Massachusetts[1]. The objective of the study was to make recommendations leading to the development of secure database systems that handle classified data at different levels, where the users of the system are also cleared to different levels.

A study group headed by Clark Weissman examined and proposed several near-term approaches to this problem (see Chapter 1 of the study report[1]). One of their approaches is designed to use a commercially available untrusted database system whose security is not guaranteed.

Because the database system may have security holes or Trojan Horses, all accesses to it are constrained to go through a trusted (verified), and isolated, filter (or guard), which is a security kernel responsible for enforcing the requirements for multilevel security. This includes authenticating users and ensuring that all information flow in and out of the database is authorized. The filter thus implements the reference monitor concept, mediating all access to the database. Being a security kernel, it is nonbypassable, tamperproof, and verifiable.

To achieve multilevel security, the filter stores a label in each record (or field within a record) giving the classification level (TOP SECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED) of the data in the record (or field). The classification labels are then used by the filter to determine what data a given user is allowed to access, based on the user's clearance. If a query for data includes data that the user is not cleared to see, then the filter removes that data from the view returned to the user.

Of course, simply storing classification labels in the records is not enough to prevent top secret data, for example, from reaching an unclassified user: the database system could simply change the labels, or move top secret data into a record marked unclassified. To protect against changes to the data or classification labels, the filter computes a *cryptographic checksum* (also called *message authentication code* or MAC) for each record, which is stored in the record or a separate file. The checksums are validated on retrieval to detect tampering. The cryptographic checksums are computed and validated in the filter using an encryption chip and a secret key K known only to the filter. Note that the data stored in the database is in the clear; this is so the database system can perform operations such as select and join on the database.

Figure 1 illustrates the structure of the system. The figure shows two classes of users, "Lo" and "Hi", each connected to a separate interface. In practice, there is a separate user interface for each distinct class of users. The interfaces are untrusted, and perform operations such as query parsing and output formatting. These operations are separated from the filter in order to simplify the design of the filter. An important objective of the approach is that the filter be kept simple to minimize the verification effort. If the filter becomes as complex as the database system itself, the attraction of the approach is lost.

To be presented at the 1984 IEEE Symposium on Security and Privacy, Oakland, Calif., April 30-May 2, 1984.

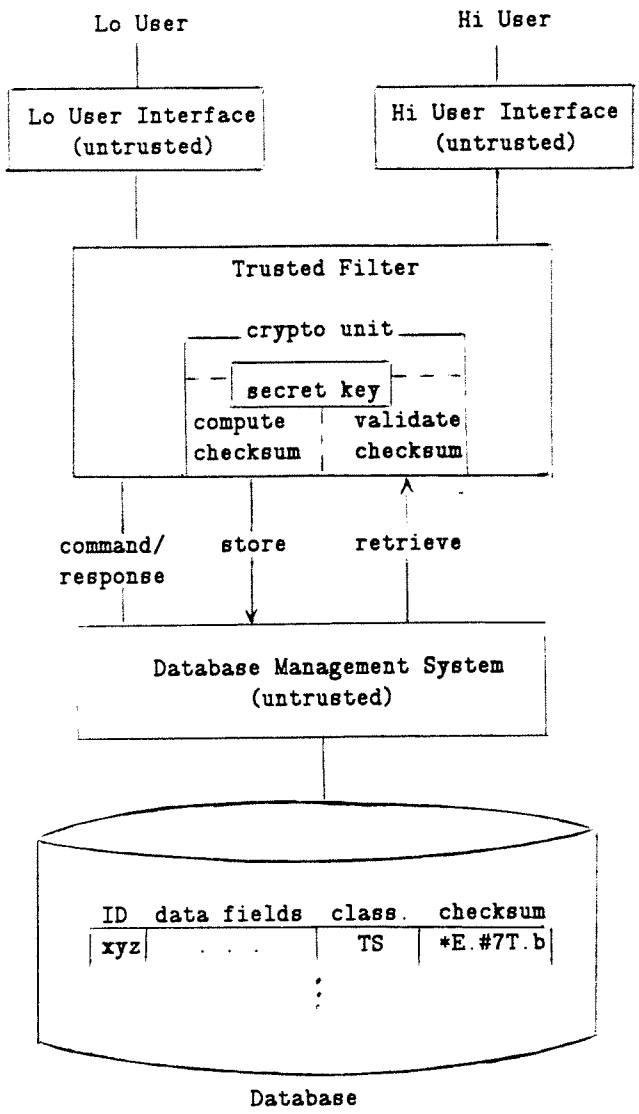

Figure 1. System Structure.

The subsystems shown in Figure 1 must be physically or logically isolated so that the users and user interfaces cannot bypass the filter and obtain direct access to the database system or the database, and so that the untrusted database system cannot interfere with the operation of the filter. We shall discuss this further later.

The idea of protecting a database with a minimal amount of trusted code in the overall system was studied in the mid-late 1970s by Downs and Popek[2]. They proposed a system structure considerably different from the one proposed here, using two trusted modules (security kernels): a base kernel, which manages the physical database and serves as a filter, and a kernel input controller, which performs some interface functions and manages the protection data. Although they did not propose storing cryptographic checksums of the data, they proposed storing tags in the database, which are encoded forms of logical entity names. The tags ensure the integrity of the mapping from the physical identifiers, managed by the base kernel, to the logical identifiers, managed by the kernel input controller.

Roger Schell introduced the concept of using cryptographic checksums in a guard environment in the early 1980s, while at the Naval Postgraduate School working on the application of security kernels to microprocessors. Schell was doing a study for the Korean Air Intelligence System (KAIS), and observed that cryptographic checksums (using DES) could be used to reliably label records stored in the untrusted database with their security classifications. Jim Anderson noted that a security kernel provided the means of meeting the verifiability requirements of the guard.

Anderson[3] suggested applying the guard approach to the RECON bibliographic system. The RECON guard is similar in concept to the filter described here, but uses a different key for each classification level. The channels connecting the guard to the users are assigned keys in accordance with their classification levels, and a record is not transmitted over a channel unless the checksum computed with the channel key matches that stored with the record (a channel can have multiple keys).

Both the KAIS and RECON guards are physically isolated from the the untrusted system managing the database. This is to ensure the integrity of the cryptographic process.

The KAIS and RECON systems both have limited database management capabilities. They are essentially record storage and retrieval systems. The question addressed in this paper is the extent to which the guard approach carries over into general purpose database systems.

Schell helped bring the guard approach to the attention of the Air Force summer study. The study group liked the idea, and nicknamed it "spray paint" because the checksums indelibly "color" the records with their classifications. They also referred to the approach as "integrity locking", since the records are effectively write-locked in that any modifications are detectable.

The concept of using cryptographic checksums to authenticate cleartext data was independently proposed by Gligor and Lindsay[4] in the late 1970s as a way of protecting the objects belonging to type managers in an operating system. Specifically, they proposed that type managers, whose security is under kernel control, "cryptographically seal" their objects so that the

representation and state of a migrated object, which is no longer under kernel control, can be authenticated and reinstated. Cryptographic seals (checksums) provide one way of implementing the "seal" operation proposed earlier by Morris[5] for protecting a type manager's objects (see also Denning[6]).

The remainder of this paper discusses different implementation strategies in terms of cost and security, with the aim of evaluating the applicability of the approach to general database systems. Some of our conclusions have been independently reached by Graubart [7]. We will assume in our discussion that the database system is a relational system, though the ideas are easily applied to any type of database system.

Implementing Classifications and Checksums

We shall first discuss implementation strategies for the classification labels, and then discuss strategies for the cryptographic checksums.

Classification Labels

Classification labels can be applied at four levels: 1) the relation level, 2) the attribute (column, field) level, 3) the record (row, tuple) level, or 4) the data element level. The storage and processing requirements for the labels are least at the relation level and greatest at the data element level. For a given application, the appropriate level is determined by the requirements of the application. In this note, we will assume that the labels are at the attribute, record, or element level.

There are three approaches to storing the labels: 1) directly in the database, 2) in the database, but encrypted, and 3) in a separate database managed by the filter. Approach (1) allows the database system to retrieve data with a particular classification. For example, the filter can request the database system to return only SECRET or lower data satisfying some logical formula. This can greatly reduce the volume of data transferred from the database system to the filter, which would otherwise be required to discard all the higher level data. (Of course, the filter must still check the classifications before it transmits the data to a user.) This is the approach taken in the RECON guard[3].

The study group observed, however, that the approach was vulnerable to Trojan Horses in the untrusted database system[1]. Since the classification labels are exposed, the Trojan Horse can, for example, identify all TOP SECRET data in the database. It could then leak the TOP SECRET data in its responses to queries for SECRET records. For example, the value for an attribute A in a TOP SECRET record might be leaked by returning a SECRET record with an identical value for A, or, more subtly, by encoding the TOP SECRET value in a sequence of SECRET records (e.g., using a binary field). For the RECON guard, Anderson[3] observed that such leaks could occur through error messages returned to the user from the database system at the rate of .085 bits/second (assuming 100-byte error messages), or 7.8 hours to transmit a 300-character record.

Approaches (2) and (3) are designed to protect against such Trojan Horse leaks by concealing the classification labels from the database system. How well this objective is achieved is discussed later.

Approach (3), which conceals the labels by storing them in a separate database managed by the trusted filter, was suggested by the summer study group[1]. This approach has two disadvantages. First, it can degrade performance of the overall system because the labels must be retrieved separately from the data. Second, the approach is somewhat counter to the overall objective of keeping the filter simple. The filter would require a small database capability, complete with indexes, so that the label for a record could be efficiently retrieved using the record's identifier (Id).* This will complicate the verification effort for the filter. Note that we cannot avoid verifying the classification database manager. An untrusted classification manager could fail to update classification labels, and, simultaneously, the untrusted database manager could fail to update checksums, thereby allowing data to be released at a lower classification (see later discussion). The security properties that must be verified for the classification database manager, however, are much simpler than for the remaining part of the filter.

Approach (2), storing encrypted labels in the database, has generally been thought to be insecure. If the same encryption key is used to encrypt all the labels, the data can be vulnerable to ciphertext searching, i.e., scanning for identical ciphertext labels (which implies identical cleartext labels). Classification labels applied at the record or data element level are particularly vulnerable to ciphertext searching, since most labels are likely to repeat many times. Moreover, if the labels on some of the data are known, or if the distribution of labels is known (e.g., that 90% of the data is SECRET and 10% TOP SECRET), then it would be easy to deduce the exact classifications. A Trojan Horse in the database system could make such deductions, and then leak classified data to unauthorized users.

---

* In database terminology, this is called a key; we will use Id instead to avoid confusion with cryptographic keys.

The apparent weakness with encrypted labels, however, can be resolved with our field based encryption scheme[8]. The scheme can be used to encrypt any (or all) of the individual fields within a record. The basic idea is to use a different encryption key to encrypt each element, thereby hiding identical plaintext values. The encryption key is itself a cryptographic function of the record Id, the field Id (attribute name), and the secret key K stored in the filter. Encryption is done using a block encryption algorithm such as the DES[9]; thus, each ciphertext block is 8 bytes. If the plaintext labels are short (e.g., 4 bytes or less), the field can be replicated before encrypting; this will produce plenty of redundancy so that the ciphertext can provide authenticity as well as secrecy.

Our encryption scheme is easily applied to the problem of concealing classification labels. If the labels are applied at the record level, then each record has an encrypted field containing the record's ciphertext classification label; if they are applied at the data element level, then each record has an encrypted label field associated with each data field. Finally, if the labels are applied at the attribute level, then the encrypted labels are stored with the meta data in the database system (in a relational database system, this is usually a column of the ATTRIBUTE relation). The only disadvantage of the approach is some data expansion, since the ciphertext labels will be 8 bytes, even if the cleartext labels are less.

Cryptographic Checksums

We now turn to the problem of computing cryptographic checksums that bind data to its classification label. Letting X denote a datum and its label, the checksum for X is given by $S = \text{last}(E_k(X))$, where $E_k(X)$ denotes the encryption of X with key k using, say, the DES in cipher block chaining mode (e.g., see Denning[6]), and "last" returns the last block.

Checksums can be applied at the same four levels that apply to classification labels: (1) entire relations, (2) attributes, (3) records, or (4) individual data elements. Moreover, the checksums need not be applied at the same level that the classification labels are applied; this is an independent parameter of the database. For example, we can label the individual elements of a record with distinct classifications, but compute a checksum across the entire record. Alternatively, we can label a complete record, but compute checksums for the individual fields by concatenating the record label with the data in the field when the checksum is computed. Similarly, we can label an attribute, and then concatenate the attribute's label with the corresponding value in some record to compute a checksum for the individual data element.

Thus, we can independently consider the level at which checksums should be applied, and here the decision will be determined by the expected access patterns to the database. Approach (1), which computes a checksum over an entire relation, will be very costly unless most accesses are for complete relations. Similarly, approach (2), which computes a checksum over an entire attribute, will be costly unless most accesses are for complete columns, and classification is at the attribute level.

Approach (3), which computes a checksum for each record, was proposed by the study group[1]. Assuming that the classification labels are also applied at the record level, then the label is concatenated to the other data in the record when the checksum is computed. If the label is stored in the filter rather than the record, then, of course, it must be separately fetched. Applying the checksums at the record level has the disadvantage of requiring the database system to always return complete records; that is, the database system cannot project out any fields. If queries are frequently for only a subset of the fields stored in a record, this could impact the performance of the system.

Approach (4), which computes a checksum for each data element, would allow the database system to return only the data requested in a query. This approach, however, has usually been rejected for reasons similar to those for rejecting encrypting individual data elements: If the same key is used to compute all checksums, then the approach is vulnerable to substitution of the value and checksum in one record with that in another.

Checksums can be securely implemented at the field level[8]. The solution is similar to that for field encryption; the checksum for each element is computed using a distinct key, which is a cryptographic function of the record Id, the attribute Id, and the secret key K in the filter. Since it is important to bind each element with its security classification, we concatenate the label to the data before computing the checksum, even if the labels are stored at the record or attribute level.

Figure 2 illustrates how classified data in record i is protected when classification is at the record level, and field M gives the classification label. The stored record shows the ciphertext classification $C_{iM} = E_{K_{iM}}(X_{iM})$ and the checksum $S_{ij} = \text{last}(E_{K_{ij}}(X_{ij} \| X_{iM}))$, where $K_{ij} = g(R_i, F_j, K)$, g is a key generator, and "||" denotes concatenation.

Because the fields are individually authenticated, the database system can perform projections to suppress unwanted fields. The unique identifier and classification label fields, however, are never projected out.

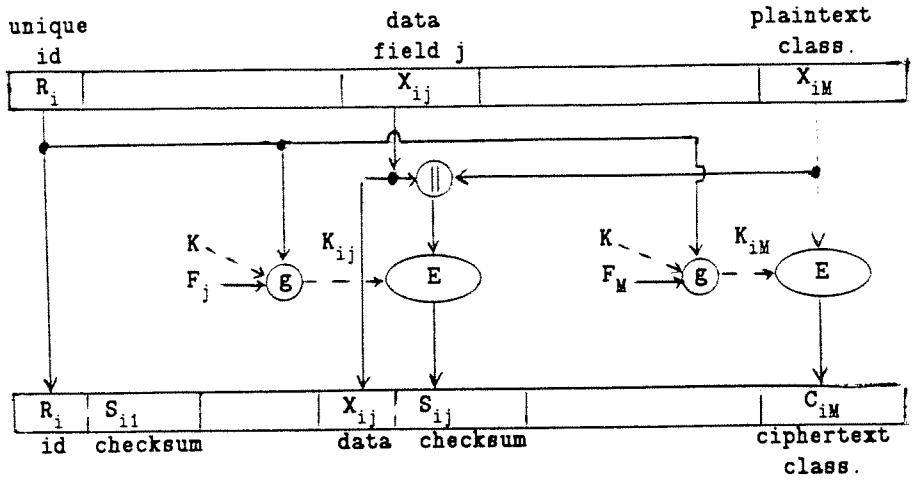

Figure 2. Checksums Applied at the Data Element Level.

When the data is returned to the filter, only the returned fields are checked for changes. Changes to other fields are not detected until those fields are retrieved. Individual fields can be updated by the filter without the need to obtain the entire record.

An attractive property of field checksums is that it is possible to tell which field (or fields) have been changed when validation fails. With record checksums, it is not possible to pinpoint changes to specific fields.

One disadvantage of field checksums over record checksums is that they require more storage (say, 32-64 bits per element, rather than 32-64 bits per record). If there are many short fields, checksums could be computed over groups of fields rather than individual fields.

Trojan Horses

We now discuss the extent to which a Trojan Horse in the database system could leak data or otherwise cause damage.

Deduction of Classifications

Recall that the reason for concealing classification labels was to thwart a Trojan Horse in the untrusted database system from leaking classified data through lower-classified responses. Unfortunately, encrypting the labels or storing them in the filter does not rule out a Trojan Horse deducing them by other means.

One way might be through dependency constraints known to exist in the database. For example, an attribute CODE might determine the classification of a record, with CODE = ULTRA or GEMINI implying TOP SECRET, CODE = PURPLE, RED, or BLUE implying SECRET, and so forth. Because the data attributes are not encrypted, a Trojan Horse could then easily deduce each record's classification.

Another way might be through access patterns to the data. Suppose most accesses are for data with the same classification; and that 90% of all accesses are for TOP SECRET data, with 10% for SECRET data. By observing access patterns, a Trojan Horse can deduce the classifications.

User Assistance

Even if the classification labels are concealed and a Trojan Horse is unable to deduce the classifications on its own, this does not rule out a user assisting the Trojan Horse in the attack. For example, in the following scenario, a SECRET user helps the Trojan Horse determine what records are TOP SECRET, so that the Trojan Horse can then leak the TOP SECRET data.

1. The user asks for a set of records satisfying a formula F.

2. The database system locates the set of records satisfying F. Call it X. The database system records the Ids of X, and sends the set X back to the filter. We assume that some subset Y of these records is SECRET, and the remaining subset Z = X - Y is TOP SECRET. However, at this point, the Trojan Horse cannot tell which records are in which set, or even if there are any TOP SECRET records.

3. The filter discards the TOP SECRET records in Z and sends the SECRET subset Y back to the user.

4. The user next asks for the set Y of records, which he has just received, by supplying their record Ids.

5. The Trojan Horse, remembering the Ids for X, computes Z, and leaks their contents back to the user.

Failure to Update

A more serious Trojan Horse threat arises when we consider updates to the database. A Trojan Horse could simply fail to update the database!

Consider first the case where data changes, but its classification remains the same; and suppose that the Trojan Horse fails to update the data and its checksum. If the checksums are stored in the database, the filter cannot detect this regardless of whether the classifications are stored in the filter or stored encrypted in the records. This threat could be eliminated by storing the checksums in the filter, since the old data will no longer pass the validation test. Indeed, if we are concerned enough about Trojan Horses to store the classification labels in the filter, then we should store the checksums there as well. Note, however, that keeping classification labels and checksums in the filter adds to the complexity of the filter and the response time for a query. Figure 3 illustrates.

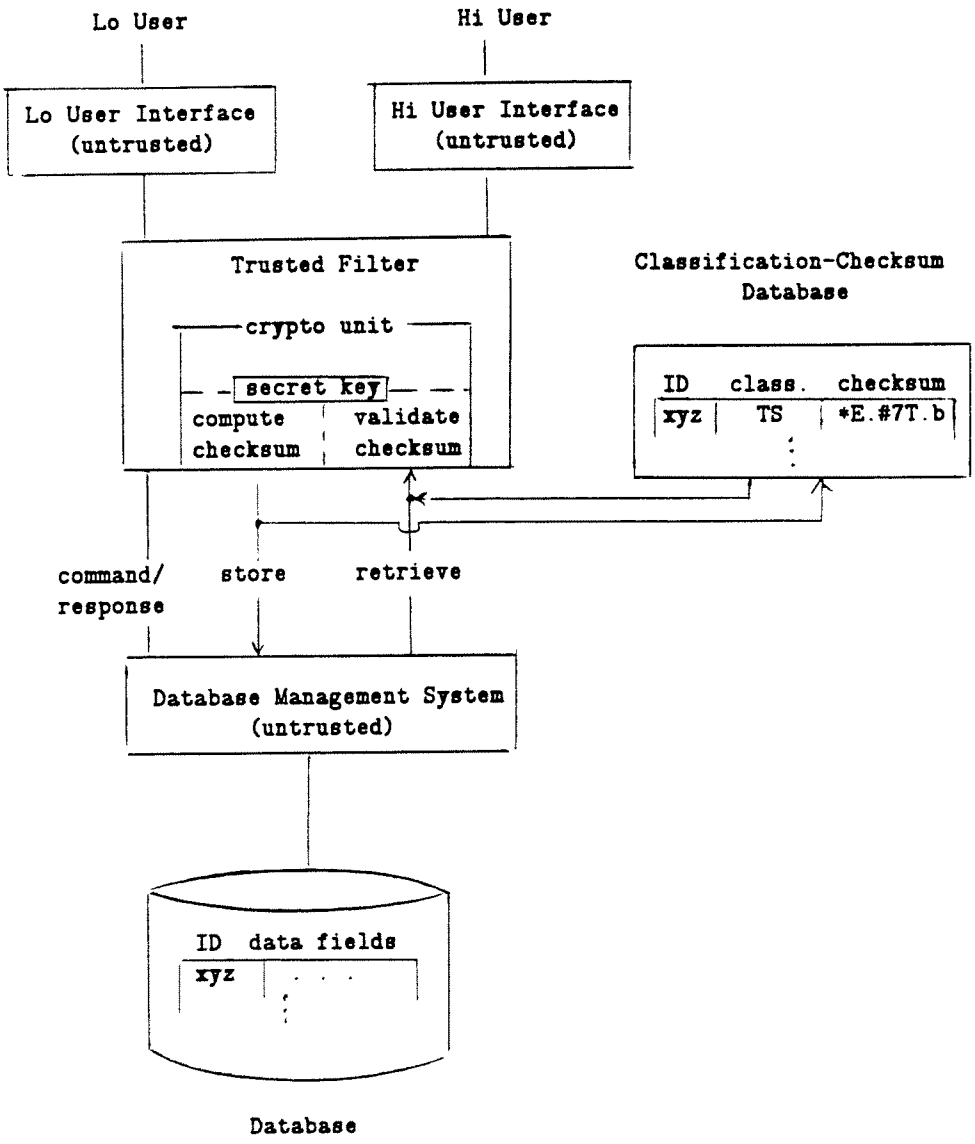

Figure 3. Classification Labels and Checksums Managed by Filter.

Consider next the case where the classification of the data changes (say increases), but the data itself may or may not change. If checksums and cleartext or encrypted classifications are stored in the records, then by failing to perform the update, the Trojan Horse can cause the data to be released at its lower classification. (Note that with encrypted classifications, the Trojan Horse can detect changes of classification even though it cannot decrypt them because the classifications and checksums will change while the remaining data remains fixed.) If the checksums or classifications are stored in the filter, the Trojan Horse cannot get away with this, because the old checksums would not be valid with the new classifications.

It is worth noting that we could protect against the update threat, without the need for a filter managed classification or checksum database, by computing a checksum over the entire database, and storing this checksum in the filter. The problem with this approach is that it would seem to require that the database system always return the complete database to the filter, an approach that clearly would be unacceptable. We are investigating whether there might be practical techniques that achieve the same objective.

Erroneous Results

As noted by the summer study[1], a Trojan Horse in the database system can potentially cause serious damage simply by returning erroneous results, but of the proper classification. This type of attack can be detected, however, if the database system is required to return complete records, so that the user can then check that the records returned satisfy the query asked.

Summary

Figure 4 summarizes the leakage and update threats for the various implementation strategies. Although the greatest security is provided by storing checksums and classifications in the filter, this also requires the greatest verification effort and causes the greatest performance penalty. For most applications, the security provided by storing the labels and checksums in the database is probably acceptable. Moreover, there does not seem to be any way of preventing all leakages, short of verifying enough of the database system to show that such leakages are not possible, and then maintaining tight controls over updates to the code. Verifying a commercial database system that has not been designed to be verified, however, may be nearly impossible, so we may have to settle for trusting the vendor supplied code.

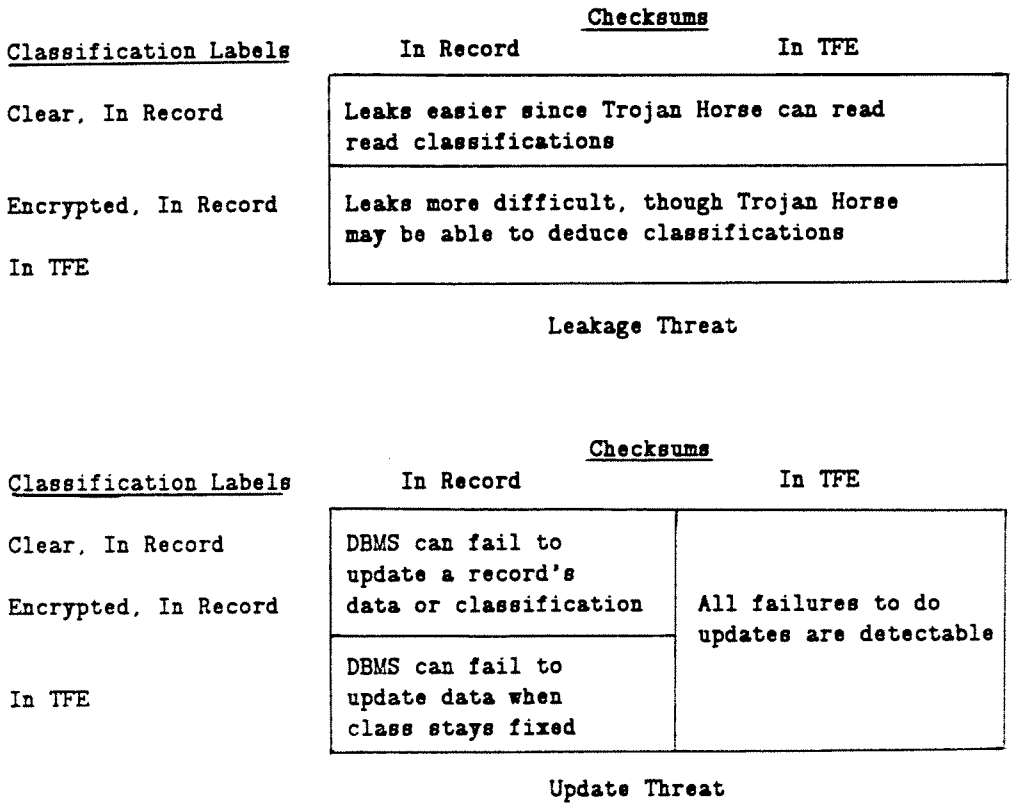

Figure 4. Summary of Threats for Different Implementation Options.

Other Issues

We now consider other aspects of the approach.

Functions and Subqueries

In the introduction, we noted that the user interface should be responsible for query parsing and output formatting, in order to minimize the code in the filter. We will now consider the placement of other functions normally provided by a database system.

Suppose the user asks for an average, total, or some other function of the data. We first observe that the untrusted database system cannot perform these functions and return the result to the filter, since the filter would have no way of verifying the security classification of the result. Moreover, because the database system does not have access to the classifications of the data, it cannot know what values to include and exclude in the computation. This suggests that the modules that perform statistical and nonstatistical functions (e.g., for sorting) should be placed in the user interface, since there is no reason why they must be trusted.

Many database systems have a capability for forming subqueries. For example, the following query, expressed in SQL (e.g., see[10]), uses a subquery to retrieve the names of all persons who have gone to Rhodesia.

```
SELECT NAME
FROM EMPLOYEE
WHERE EMPID IN
   SELECT EMPID
   FROM TRIPS
   WHERE DEST="RHODESIA"
```

Graubart[7] observes that such nested queries can disclose classified information if the database system processes the nested query and returns only the final result. For example, if every record in the TRIPS relation is TOP SECRET when the destination is Rhodesia, but all records in EMPLOYEE are SECRET, then the answer to the above query will be a list of SECRET records naming all personal going to Rhodesia, even though the information contained in the response is TOP SECRET.

To prevent such disclosures, the database functions that support subqueries should also be placed in the user interface. All subqueries will then pass through the filter, and the filter will only return intermediate results that are authorized to the user. For example, the subquery above, which returns the Ids of all employees going to Rhodesia, would not be returned to a user interface operating at the SECRET level.

Similar reasoning shows that query processing functions that perform selects and projects must be placed in the user interface. For example, the preceding query can be rewritten as a single-level query as follows:

```
SELECT NAME
FROM EMPLOYEE, TRIPS
WHERE EMPLOYEE.EMPID=TRIPS.EMPID
   AND DEST="RHODESIA"
```

This query is a standard "select-project-join", which would normally be performed in one step by a database system.

We conclude that the filter must see all data used to form the response to a query. Selections must be performed in the user interface. If the database system performs a join, then both of the joined attributes must be returned to the filter, since they could have different classifications. This conclusion lessens the attraction of applying classifications and checksums to individual elements.

Isolation

The untrusted user interfaces must not be allowed to bypass the filter and access the database system (or database) directly. Moreover, they must be isolated from each other, lest the interface connected to high level users leak data to a lower level interface.

Isolation can be achieved either physically or logically. Physical isolation between the user interfaces and the database system could be obtained by dedicating a separate processor to the filter, and forcing all interface connections to go through the filter (i.e., users would not be allowed terminal access to the processor containing the database system). Physical isolation among the interfaces could be obtained by dedicating a different processor to each user class. These processors could be general purpose systems operating in dedicated mode, which are connected to the filter over a shared network that uses encryption to isolate different logical channels. The RECON guard uses physical isolation, with a separate microprocessor implementing the guard.

Logical isolation would allow the modules to share a single processor, but would be feasible only on a machine that is is certified to be multilevel secure; i.e., on a trusted computing base.

Sanitization

Some applications require more complex policies for classifying information than can be handled with labels attached to records or even fields within records. As an example, suppose a field A is classified SECRET, but we would like to be able to release averages computed over A

at the CONFIDENTIAL level. To do this securely, some mechanism is needed for ensuring that individual values cannot be inferred from released averages (e.g., see Denning[6] and Denning and Schlorer[11]). In general, if sanitized views can be constructed from classified data, a mechanism is needed for classifying such views and verifying that they do not disclose data at higher levels. Cryptographic checksums do not, by itself, provide such a mechanism.

Conclusions

Classification labels and cryptographic checksums can be applied to complete relations, records, attributes, or individual data elements within a record. Applying checksums at the data element level allows the database system to return only that data needed to respond to a query. The cryptographic weaknesses normally present with element based checksums can be overcome by using a different key with each checksum.

There are, however, security reasons for requiring the database system to always return complete records. If data used by the database system during record selection has a higher level than that returned in the response, inference problems can arise. Returning complete records also helps protect against the threat of the system returning erroneous results.

But returning complete records, and every record accessed by the database system in response to a query, effectively reduces the functional support provided by the database system to a storage and retrieval system. Many complex database functions must be delegated to the user interface systems, including subquery handling, query optimization, and statistical computations.

It is not practically possible to eliminate all leaks. Concealing the classification labels helps, but only at a cost. Placing the labels in the filter complicates the filter, and is not recommended. Although the labels can be encrypted efficiently, concealing them has the disadvantage of not allowing the database system to retrieve records by their classification.

Verification and isolation of the filter is essential. This is easiest to achieve if the filter is implemented as a security kernel on a dedicated processor.

The conclusion to be drawn from the above is that the application of cryptographic checksums in a guard environment is ideally suited to applications, such as RECON, that are multilevel record based storage and retrieval system. Since the RECON guard has been successfully implemented, little additional research is needed to implement the approach in other environments. The effort should be largely developmental.

The problem of designing a multilevel secure general purpose database system that provides the capabilities found in modern relational database systems remains to be solved. Moreover, if the system is to address inference, aggregation, and sanitization problems, another approach is needed. A group at the summer study headed by this author and Peter Neumann (see Chapter 3 of the summer study report) proposed a database model for addressing these issues based on protected views. Where this approach will lead is not yet known.

Acknowledgments

The ideas in this paper have been assembled from many sources. I would especially like to acknowledge Jim Anderson and Roger Schell, who provided historical background and many helpful suggestions on an earlier version of this paper. I would also like to acknowledge Steve Kent, Bob Morris, Clark Weissman, and the other participants at the Air Force Summer Study; and the inquisitive audience at my Berkeley talk. Finally, I thank the NSF for their generous support through NSF Grant MCS-8313650.

References

1. Committee on Multilevel Data Management Security, "Multilevel Data Managment Security," Tech. report, Air Force Studies Board, National Research Council, 1982.

2. Downs, D. and Popek, G. J., "A Kernel Design for a Secure Data Base Management System," *Proc. 3rd Conf. Very Large Data Bases*, IEEE and ACM, New York, 1977, pp. 507-514.

3. Anderson, J. P., "On the Feasibility of Connecting RECON to an External Network," Tech. report, James P. Anderson Co., March 1981.

4. Gligor, V. D. and Lindsay, B. G., "Object Migration and Authentication," *IEEE Trans. on Soft. Eng.*, Vol. SE-5, No. 6, Nov. 1979, pp. 607-611.

5. Morris, J. H., "Protection in Programming Languages," *Comm. ACM*, Vol. 16, No. 1, Jan. 1973, pp. 15-21.

6. Denning, D. E., *Cryptography and Data Security*, Addison-Wesley, Reading, Mass., 1982.

7. Graubart, R. D., "The Integrity-Lock Approach to Secure Database Management," *Proc. 1984 Symp. on Security and Privacy*, IEEE, 1984.

8. Denning, D. E., "Field Encryption and Authentication," *Proc. of CRYPTO 83*, Plenum Press, Aug. 1983.

9. National Bureau of Standards, "Data Encryption Standard," Tech. report FIPS PUB 46, Washington, D.C., Jan. 1977.

10. Date, C. J., *An Introduction to Database Systems*, Addison-Wesley, 1981.

11. Denning, D. E. and Schlorer, J., "Inference Controls for Statistical Database Security," *IEEE Computer*, Vol. 16, No. 7, July 1983, pp. 69-82.