Tamper Detection in Audit Logs

Richard T. Snodgrass, Shilong Stanley Yao and Christian Collberg

University of Arizona
Department of Computer Science
Tucson, AZ 85721-0077
USA
{rts,yao,collberg}@cs.arizona.edu

Abstract

Audit logs are considered good practice for business systems, and are required by federal regulations for secure systems, drug approval data, medical information disclosure, financial records, and electronic voting. Given the central role of audit logs, it is critical that they are correct and inalterable. It is not sufficient to say, "our data is correct, because we store all interactions in a separate audit log." The integrity of the audit log itself must also be guaranteed. This paper proposes mechanisms within a database management system (DBMS), based on cryptographically strong one-way hash functions, that prevent an intruder, including an auditor or an employee or even an unknown bug within the DBMS itself, from silently corrupting the audit log. We propose that the DBMS store additional information in the database to enable a separate *audit log validator* to examine the database along with this extra information and state conclusively whether the audit log has been compromised. We show with an implementation on a high-performance storage engine that the overhead for auditing is low and that the validator can efficiently and correctly determine if the audit log has been compromised.

1 Introduction and Motivation

OLTP (on-line transaction processing) applications maintain *audit logs* so that the correctness of the transactions can be later checked. As a simple example, every deposit to and withdrawal from a bank account generates a separate audit record, so that the current balance in the account can be checked. As one test, the sum of all deposits minus the sum of all withdrawals should equal the change in total accounts over that period. If this auditing check fails the bank looks into the discrepancy further.

A variety of federal laws (e.g., Code of Federal Regulations for the Food and Drug Administration, Sarbanes-Oxley Act, Health Insurance Portability and Accountability Act, Canada's PIPEDA) and standards (e.g., Orange Book for security) mandate audit logs. The correctness of the auditing records themselves is critical. As Peha states, "An auditor should be able to retrieve a set of records associated with a given entity and determine that those records contain the truth, the whole truth, and nothing but the truth. There should be a reasonable probability that any attempt to record incorrect, incomplete, or extra information will be detected. Thus, even though many transactions will never be scrutinized, the falsification of records is deterred." [17]

The message is clear: given the central role of audit logs in performing auditing of interactions with the data (modification, exposure) as well as of the base data itself, it is critical that audit logs be correct and inalterable. It is not sufficient to say, "our data is correct, because we store all interactions in a separate audit log." The integrity of the audit log itself is still in question.

Recent experience has shown that sometimes the (human) auditors themselves cannot be trusted. (A recent security survey states that the source of cybersecurity "breaches appears fairly evenly split between those originating on the outside and those originating on the inside." [9, page 9].) In cases such as Enron and WorldCom (cases that inspired the Sarbanes-Oxley Act), supposedly independent auditing firms conspired with the company under audit to hide expenses and invent fictitious revenue streams. Just as the original databases can be manipulated to tell a story, so can

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

the audit logs be manipulated to be consistent with this new story.

This paper proposes mechanisms within a database management system (DBMS), based on cryptographically strong one-way hash functions, that prevent an intruder, including an auditor or an employee or even an unknown bug within the DBMS itself, from silently corrupting the audit log. We propose that the DBMS transparently store the audit log as a transaction-time database, so that it is available to the application if needed. The DBMS should also store a small amount of additional information in the database to enable a separate *audit log validator* to examine the database along with this extra information and state conclusively whether the audit log has been compromised. We propose that the DBMS periodically send a short document (a hash value) to an off-site digital notarization service, to bound when changes were made to the database.

In the following, we first review existing techniques for maintaining the integrity of the audit log. We then describe the context within which we have developed our approach. In Section 4, we summarize the threat model addressed by our approach. We then present a simplified approach that enables *validatable audit logs*. Section 6 discusses a range of refinements that provide increase performance. We then summarize an initial implementation of the general algorithm on Berkeley DB [22] and evaluate the performance of this prototype, showing that supporting validatable audit logs does not represent a significant performance hit in space or time. We end with a review of related work and discussion of outstanding problems.

Our contribution is to demonstrate that, with the system model, techniques and optimizations we introduce here, tamper detection in audit logs can indeed be realized within a high-performance DBMS.

2 Existing Audit Log Techniques

The traditional way [3] to protect logging data from tampering is to write it to an append-only device, such as a Write Once Read Multiple (WORM) optical drive or a continuous-feed printer. The security of such schemes assumes, however, that the computing site will not be compromised. If this is a possible attack scenario the logging data can be sent to a remote site over the network, so called *remote logging*. *Log replication* can be used to send the data to several hosts to require the attacker to physically compromise several sites.

Schneier and Kelsey [21] describe a secure audit log system. The idea is roughly as follows. An untrusted machine $\mathcal{U}$ (on which the log is kept) initially shares a secret authentication key A_0 with a trusted machine $\mathcal{T}$. To add the j:th log entry D_j, $\mathcal{U}$ computes $K = hash(A_j)$ (an encryption key), $C = E_K(D_j)$ (the encrypted log entry), $Y_j = hash(Y_{j-1}, C)$ (the j:th entry in a chain of hashes, where $Y_{-1} = 0$), and $Z_j = MAC_{A_j}(Y_j)$ (a keyed hash (*Message Authentication Code*) of Y_j). Then the j:th entry $\langle C, Y_j, Z_j \rangle$ is written to the log, a new authentication key $A_{j+1} = hash(A_j)$ is constructed, and A_j is destroyed. An attacker who compromises $\mathcal{U}$ at time t can delete (but not read nor modify) any of the first t log entries, since he will only have access to A_{t+1} but not to any of the previous $A_0 \ldots A_t$. While there is no way to prevent the attacker from deleting some or all of the log entries (or appending his own entries), any such attempted tampering will be detected by $\mathcal{T}$ on its next interaction with $\mathcal{U}$. Furthermore, since each log entry is encrypted with a key derived from A_0 (which is only stored permanently on $\mathcal{T}$) the attacker cannot read past log entries to find out if his attack was noticed or not.

As applications require access to the database (and often read access to the audit log), these existing techniques are not applicable to tamper detection of transactional database audit logs.

3 Context

We show how a database can be protected by having the DBMS maintain the audit log in the background and by using cryptographic techniques to ensure that any alteration of prior entries in this audit log can be detected. This approach thus applies to any application that uses a DBMS.

As an example, assume that we are a pharmacological research firm that develops new drugs, providing data to the FDA for approval of those drugs. As part of this effort we have a relational table, `Administer`, that records what drugs were administered to which patients during a drug trial. 62 FR 13430 requires a computer-generated, time-stamped audit trail. We define the table as follows, in the MySQL DBMS [7].

```
CREATE TABLE Administer (...)
AS TRANSACTIONTIME
TYPE = BDB
AUDITABLE = 1
```

The first line—which also specifies the columns and primary and foreign key constraints for the table—is supported by the conventional MySQL release, as is the third line, which specifies that the Berkeley DB storage system be used. The second line specifies that this `Administer` table includes transaction-time support (this is an open-source extension that we have implemented). A transaction-time database records the history of its content [11]. All past states are retained and can be reconstituted from the information in the database. This is ensured through the *append-only* property of a transaction-time database: modifications only add information; no information is ever deleted. It is this basic property that we exploit to validate the audit log (in fact, the table *is* the audit log).

The last line specifies that this transaction-time table be *auditable*, that is, that the system take additional steps so that an audit log in maintained and so that later a separate *audit log validator* can examine the database and state conclusively whether the audit log has been compromised. These additional steps are the primary topic of this paper. We have implemented support for auditable tables and independent validation in MySQL and Berkeley DB.

It is important to emphasize that the applications that update and query this table need not be aware of the last three lines of the CREATE TABLE statement. Behind the scenes, transparently to the application, the DBMS creates an audit log and ensures auditability. This is because our approach ensures *temporal upward compatibility* [2].

A transaction-time table includes all the columns declared in the CREATE TABLE statement, along with two additional columns, which are not normally directly visible to the application: the Start and Stop columns. These latter columns are maintained by the DBMS. Specifically, the value of the Start column is the time (to a granularity of, say, a millisecond) at which that row was inserted into the table. The Stop time of a newly inserted row will be the special value "until changed," or *UC*. When a row is deleted, the deletion time is recorded in the Stop column; the row itself is retained. A modification of a row is treated as a deletion of the old value of the row and an insertion of the new value. A special form of SELECT is available for the application to see these columns and past versions of the table.

The rows that are currently relevant (that is, those that have yet to be modified or deleted) all have a Stop time of UC. These rows, along with the rest, which have an explicit Stop time, form an audit log of the table. A SELECT statement evaluated by an application will, because of temporal upward compatibility, see only current rows. A modification or deletion will only affect current rows. Again, the Start and Stop columns and the audit information are maintained behind the scenes by the DBMS.

4 Threat Model

In this work, we assume a Trusted Computing Base (TCB) consisting of correctly booted and functioning hardware and a correctly installed operating system and DBMS. More precisely, we assume that the TCB is correctly functioning until such a time t when a penetration occurs. Similarly, until time t the DBMS is created, maintained, and operated in a secure manner, and all network communication is performed through secure channels (such as SSL), ensuring the correctness of the internal state of the DBMS. Since the DBMS is a transaction-time database, all previous states can be reconstructed.

A penetration by an adversary ("Bob") can take many forms. An intruder (or an insider) who gains physical access to the DBMS server will have full freedom to corrupt any database file, including data, timestamps, and audit logs stored in tuples.

Of course, physical access is not necessary to compromise a system. Malware (such as viruses, worms, and trojans) can penetrate a machine by exploiting bugs (such as buffer overflows) in trusted software. The infection can occur over the network or through DBMS extensions (Oracle cartridge, Sybase plugin, DB2 extender). Regardless, the result is typically to allow Bob full root access to the machine and the DBMS server software. Once such access has been established, the integrity of any data or software on the machine is in question: the DBMS source code (including any audit log algorithms), the data in the database, and the audit log trails themselves could all have been compromised. (The information stored in the off-site digital notarization service is assumed to remain secure and unaltered.)

Our scheme assumes the existence of a trusted notarization service which, given a digital document, will return a unique identifier. We also assume a trusted and independent audit log validation service which, given access to (a copy of) the database, will verify the validity of the audit log. The integrity of these services is assumed to remain intact even in the event of a full DBMS compromise.

5 Basic Idea

We first outline our approach, an adaptation of Peha's "verifiable audit trails" to databases, making many assumptions and simplifications. We then remove some of these assumptions and simplifications to achieve a more practical and robust system.

Figure 1(a) illustrates the normal operation of our approach. The user application performs transactions on the database, which insert, delete, and update the rows of current state. Behind the scenes, the DBMS retains for each tuple hidden Start and Stop times, recording when each change occurred. The DBMS ensures that only the current state of the table is accessible to the application, with the rest of the table serving as the audit log. Alternatively, the table itself could be viewed by the application as the audit log. In that case, the application only makes insertions to the audited table; these insertions are associated with a monotonically increasing Start time. Our approach and implementation support both usages.

The basic idea is to store a "check field" in each tuple. This check field cannot be computed directly from the data (and timestamps) of the tuple, because then Bob could simply recompute the check field after he has altered the tuple. Indeed, if needed he could replay all of the transactions, making whatever changes he wanted to the data or the timestamps.

We use a *digital notarization service* that, when pro-

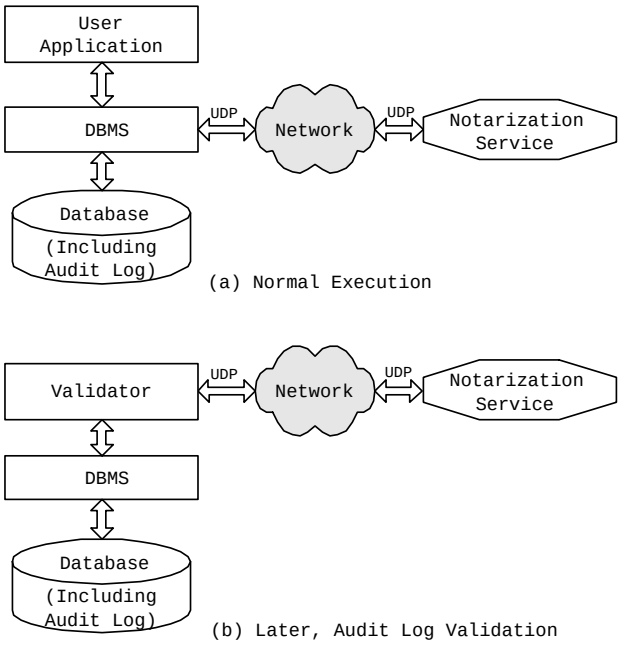

Figure 1: Normal Operation and Audit Log Validation

vided with a digital document, provides a *notary ID*[1]. Later, during audit log validation, the notarization service can ascertain, when presented with supposedly unaltered document and the notary ID, whether that document was notarized, and if so, when.

On each modification of a tuple, the DBMS obtains a timestamp, computes a *cryptographically strong one-way hash function* of the (new) data in the tuple and the timestamp, and sends that hash value, as a digital document, to the notarization service, obtaining a notary ID. The DBMS stores that ID in the tuple.

Later, Bob gets access to the database. If Bob changes the data or a timestamp, the ID will now be inconsistent with the rest of the tuple. Bob cannot manipulate the data or timestamp so that the ID remains valid, because the hash function is one-way. Note that this hold even when Bob has access to the hash function itself. Bob can instead compute a new hash value for the altered tuple, but that hash value won't match the one that was notarized.

If an independent audit log validation service was provided with the database (as illustrated in Figure 1b), that service could, for each tuple, hash the data and the timestamp, provide it with the ID to the notarization service, which will check the notarization time with the stored timestamp. (There is a timing issue in that the database obtains a timestamp first and later notarizes the tuple. The notarization time will be slightly later than the timestamp. Also, there will be many tuples with identical timestamps, as they were all modified within a single transaction. These details will be discussed in later sections.) The validation service would then report whether the database and the audit log are consistent. If not, either or both had been compromised.

Few assumptions are made on the threat model. The system is secure until Bob gets access, at which point he has access to everything: the DBMS, the operating system, the hardware, and the data in the database. We still assume that the notarization and validation services remain in the trusted computing base. This can be done by making them geographically and perhaps organizationally separate from the DBMS and the database.

The basic mechanism just described provides correct tamper detection. If Bob modifies even a single byte of the data or its timestamp, the independent validator will detect a mismatch with the notarized document, thereby detecting the tampering. Bob could simply re-execute the transactions, making whatever changes he wanted, and then replace the original database with his altered one. However, the notarized documents would not match in time. Avoiding tamper detection comes down to inverting the cryptographically-strong one-way hash function.

However, the basic approach exhibits unacceptable performance. Interactions with the notarization service should be infrequent, because such interactions are slow, requiring non-local network transmissions, and expensive, in that each notarized document results in a charge. Additionally, the space overhead is somewhat excessive, in that a notary ID must be stored in each tuple. Such IDs, on the order of 256 bits (32 bytes), are onerous for very small tuples.

In subsequent sections, we refine this approach to address these performance limitations. As we will see, it is challenging to achieve adequate performance while retaining the desirable properties of the basic approach. Our goal is to realize tamper detection in the context of a high-throughput transaction processing system. Existing approaches are simply inadequate in this context.

6 Minimizing Notarization Service Interactions

The basic approach just described interacts with the notarization service for each tuple that is modified. This interaction requires that a packet containing the hashed value of the tuple's data and timestamp (32 bytes) be sent to the service, which responds with another 32-byte notary ID, which is stored in the tuple. A transaction could easily modify thousands or even

[1]Surety (`www.surety.com`) provides online digital notary service. Secure one-way hashing is used to generate notary IDs locking the contents and time of the notarized documents [10]. Telia's Digital Notarization Service (`www.trust.telia.com`) uses VeriSign's Digital Receipt and Digital Timestamping technology to create a tamper-proof digitally signed timestamp of a document, and store the records of that transaction.

millions of tuples, rendering this approach impractical.

Throughout, we attempt to minimally impact the actual transaction processing and also attempt to minimize the validation time, at a cost of more work should tampering be detected. Our expectation is that an involved forensic analysis would be necessary upon detecting actual evidence of tampering. Rather, we attempt to minimize the effort to certify that the audit log has not been altered.

6.1 Opportunistic Hashing

The first step is to reduce interactions with the notarization service to one per transaction, rather than one per tuple. In this section, we first give an overview of the design and then explain the two key techniques we utilized.

As a refinement of the basic approach, we hash *all* the tuples modified by a transaction to compute a 20-byte hash value that is sent to the notarization service when the transaction commits. The notary ID returned from the notarization service is written to a separate *Notarization History Table*, which contains one tuple for each transaction. (This potential hot spot will be addressed in the next section.) Validation must also be on a per-transaction basis. The audit log validator (referred to from now on as simply the validator) scans all of the pages of the audited tables, maintaining a running hash value for each transaction, with each transaction identified by a transaction start or stop time within a tuple. After the database has been scanned, the validator has the hash value for every transaction. It can then check the notarization history table to see which transaction(s) were corrupted. This requires storing in main memory the hash value for each transaction. For high-performance systems completing hundreds or thousands of transactions a second, there may not be a sufficient amount of memory to store all of these values, and multiple passes over the database may be required. Note that if Bob changes a transaction time within a tuple, say to the transaction time of a different tuple, neither of these two transactions will match their notarized documents.

This seemingly innocuous change has wide-ranging implications. The data for the transaction may comprise many pages, which are generally not simultaneously in main memory. (Pages are written to disk when they are replaced by the buffer manager, either before or after the transaction commits [19].) One could during commit processing read in all of the pages that had been modified by the transaction and hash those tuples written by the transaction, to compute a single hash value for the entire transaction. (One could also stamp those pages, replacing the tuple ID with the commit time.) However, this imposes additional I/O on the commit, potentially doubling the I/O (if all the pages had been written out by the time the transaction committed).

Our solution is to *opportunistically hash* the transaction's data. Each tuple that is modified by a transaction is individually hashed at the time of the modification, when that tuple is present in main memory.

There are two key techniques to support the opportunistic hashing. The first technique, *incremental hashing*, is used to address the apparent contradiction of tuple-based hashing and transaction-based hash values. As an example, transaction T touches three tuples t_1, t_2 and t_3. A single hash value $H(T)$ needs to be computed from the three tuples as a whole. However, opportunistic hashing requires that each tuple t_i $(1 \leq i \leq 3)$ of T is hashed independently as soon as it is processed by the auditing module. This implies that the hashing needs to be done incrementally. Whenever an unhashed tuple is encountered, it is incrementally hashed so that the hashing of that transaction progresses one more step. When the last tuple of a transaction is incrementally hashed, the hashing of that transaction is then fully accomplished and the final hash value is produced.

We utilize the well-accepted cryptographically strong hash function SHA-1 [25]. One of the advantages of SHA-1 is that its API explicitly supports incremental hashing. In SHA-1, the document to be hashed is divided into 512-bit blocks. Each block is processed in sequential order. For the first block, a 160-bit constant is used as the initial hash value. For each block, starting with the intermediate hash value from the previous block and based on the data in the current block, a new intermediate hash value is computed. If there are no subsequent blocks, the intermediate hash value of the current block is the final hash value. To leverage the SHA-1 hash function to build our incremental hash algorithm, we maintain the intermediate hash state of the SHA-1 function for each transaction. The intermediate hash state includes the intermediate hash value and the left-over data (a block). Along with a few other bookkeeping variables in the intermediate hash state (e.g., the offset of the left-over data), the total space required by a transaction to store the intermediate hash state is less than 100 bytes. This is far less than the space overhead of keeping all the tuples until the transaction commits.

The space overhead of storing the intermediate hash state can be further optimized down to 20 bytes. If we pad each tuple to the block boundary, there will never be left-over data. Thus we can eliminate the 64 bytes overhead of storing the left-over data and some bookkeeping variables.

Although the incremental hashing enables opportunistic hashing, it also introduces a new problem. In incremental hashing, the order in which tuples of a transaction are hashed is critical. Different hashing orders will result in different hash values. It has been proven that all known associative (order-independent)

hash functions are cryptographically weak [18], so we have to ensure the same hashing order for the application and the validator. To enable the validator to recompute the same hash value as the one computed when the application was running, the hashing order during validation should be the same as when the application was running.

The second key technique to support opportunistic hashing is motivated by the hashing ordering problem just mentioned. In order to maintain the order consistency, a *tuple sequence number* is used to indicate the hashing order.

When the transaction is running, whenever a tuple is hashed, a sequence number, unique within the transaction and monotonically increasing, is assigned to the tuple and is stored in the tuple header. This sequence number indicates the order in which tuples were hashed during the transaction was running. We can further optimize the space allocation in the tuple header for storing sequence numbers. We can have two bits in the existing flag in the tuple header to indicate how many bytes are needed to store the sequence numbers of this tuple. For most small transactions with less than 256 tuples, only one extra byte is needed to store the sequence number.

When the validator is running, to hash the tuples of a transaction in exactly the same order as they were hashed when the transaction was running, tuples of a transaction are sorted in ascending sequence number order and are incrementally hashed in that order. Because tuples do not necessarily arrive in sequence number order while scanning the database, there will be holes (noncontinuous sequence numbers) of unseen tuples; these holes are gradually filled as the scan approaches the end of the database. To reduce the space overhead caused by storing the tuples, whenever a tuple is inserted into the sorted list we check the list to see if we can hash a subset of the consecutive tuples. Whenever there are no holes at the beginning of the queue, we can hash the leading tuples until the first hole of an unseen tuple and discard the hashed tuples so as to reclaim the memory. This sorted tuple list can be efficiently implemented with a priority queue.

At this point, only one special case is left to be explained about the opportunistic hashing. A transaction inserts a tuple. The page containing the newly inserted tuple is "stolen" by the DBMS (evicted before the transaction commits), which causes the tuple to be hashed towards the transaction hash value. Then the transaction deletes the same tuple, whose default behavior in DBMS is that the tuple is physically deleted. At the end the transaction commits. In this case, during the validation the expected transaction hash value can never be obtained, because the deleted tuple does not physically exist in the database and thus can not be hashed towards the transaction. This problem is solved by carefully handling deletions. When a tuple is deleted it is not physically deleted. Instead its start and stop time are both filled with the transaction's ID, which later is stamped with the transaction time. The tuples with identical start and stop times are regarded as transient inside the transaction. They never appear in the query results, but they participate in the hashing of the transaction twice, one is for insertion and the other for deletion.

Opportunistic hashing reduces the space overhead tremendously, as there is little per-tuple space required. (Specifically, two bits are required per tuple to indicate whether the Start and Stop times, respectively, have been hashed, two bits are needed for the sequence number size, and one to four bytes are needed for the sequence number itself.) The interaction with the notarization service is reduced to one per transaction. While the data itself remains secure, the notarization service knows for each period of time how many transactions were executed. A downside is that validator no longer can ascertain the immutability of each individual tuple. Rather, it can only check the consistency of the tuples in that transaction (identified by the transaction time stored in those tuples) with the associated hash value stored in the notarization history table, either of which may be corrupted by the attacker.

An alternative to the approach of hashing the tuples of a transaction was also considered. This alternative hashes the tuples on a page instead. Unfortunately, tuples can migrate between pages (say, because of a B-tree node split); maintaining sufficient information for the validator to determine which page a particular tuple was on when a particular hash value was computed turned out to be doable, but quite difficult. Utilizing a sequence number nicely avoids this problem.

6.2 Linked Hashing

Interacting with the notary service on each transaction is still quite expensive, especially in modern high-performance systems, which can complete thousands of transactions a second.

An attractive solution is to use *partial result authentication codes* [3] to link transactions. At database creation we get the timestamp, hash the schema and the timestamp, notarize this value, and store it in the notarization history table. Then, for each transaction, we hash the data of the transaction as before, using opportunistic hashing. At commit, we rehash a document containing this value and the previous hashed value, to obtain a new hashed value. Periodically, say at midnight, we notarize the hashed value of the most recent transaction, resulting in a notary ID. We hash the most recent hashed value with its notary ID, to compute a new value used to link subsequent transactions.

To check the validity, we repeat the hashing, in transaction-time sequence, checking the values we ob-

tain for the transaction at midnight with the notarization service. Doing so requires a single linear scan of the database, followed by the linked hashing.

The benefit is greatly reduced interaction with the notarization service, from one per transaction to, say, one interaction per day, with a concomitant reduction of notarization history table to one tuple per day. The downsides are reduced information on attacks (we only know that the information stored on a particular day was corrupted) and an inability to do checking after an attack.

6.3 Interacting with Timestamping

The time assigned by the notarization service will be (slightly) after that assigned by the database stamp module. To address this problem we introduce the *Transaction Ordering List* (TOL). Before a transaction is committed, the user data (that is, the explicit columns, as we do not yet know the timestamp of the transaction) of the tuples modified by the transaction is hashed. When the transaction commits, the transaction time (which has just been determined) and the hash value are hashed together to get the final hash value of the transaction. Thus we do not have to delay the tuple hashing until the transaction is committed.

7 Implementation

In our implementation, the audit facility is built on top of the TUC (Temporal Upward Compatibility), CLK (Clock), and STP (Stamper) modules we previously added to the underlying Berkeley DB so that it can support transaction time. We also added several more transaction-time related commands (e.g., AS TRANSACTIONTIME) and flags in MySQL so that MySQL can support transaction time. We then added an audit (AUD) module. A notarization service requester (NSR) utility and a database validator (DBV) utility were implemented as separately running programs.

While Berkeley DB is reading or writing a disk page, or while a tuple that can be stamped is accessed, STP will scan the page to do the stamping (replacing the transaction IDs with their transaction time) and some bookkeeping. This is the perfect opportunity to piggyback the hashing of the tuples onto this STP page scan. With further analysis of the relationship of the STP and AUD modules, we found we can completely put AUD behind STP to form a clean, modularized architecture.

The auditable DBMS architecture and the AUD module internal structure are shown in Figure 2. In this figure, each process is represented by a dotted line rectangle, each piece of software is represented by a solid line rectangle and each internal module is represented by a rounded solid line rectangle. Berkeley DB serves as the storage manager for MySQL and is

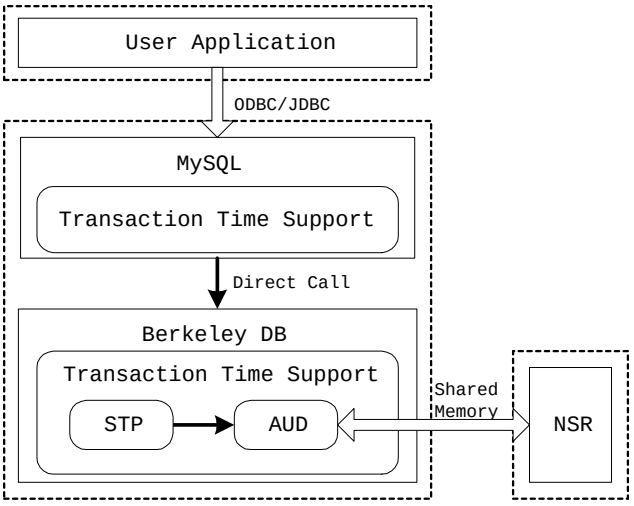

Figure 2: Auditable DBMS via MySQL+Berkeley DB

linked into the MySQL as a library. The user application running in another process interacts with MySQL through various interfaces, such as ODBC and JDBC. The NSR runs in a separate process but shares its memory region with Berkeley DB, where the database environment information is stored and shared.

AUD hashes each tuple into the transaction hash value and maintains the hash value chain across transactions. We call the last transaction on the hash chain the *tail transaction*, its transaction time the *tail time*, and its hash value the *tail hash value*. NSR is triggered every day. When triggered, it obtains the tail hash value and tail time from AUD, hashes them and sends the hash value as a document to the notarization service to be notarized. The notarization service will reply with a notary ID (NID) consisting of the notarization time and a secure timestamp computed from the document received and the time when the notarization occurred (the notarization service has its own trusted time source). The NSR hashes the NID along with the tail hash value to compute a new value to serve as the beginning of the next transaction hash chain. It also stores the (*tailtime*, *NID*) tuple in a notarization history table.

DBV (the audit validator) is run when a database audit is needed. It reads the entire audit log, recomputing the transaction hash values. According to the tail time information stored in the notarization history table, all the transaction hash values are grouped into hash chains, so that for each hash chain we have a corresponding tuple in the notarization history table. For each hash chain, transaction hash values are linked and the tail time and tail hash value are recomputed and sent to the notarization service. Encountering a validation that is refused by the notarization service implies something is corrupted after the beginning of the

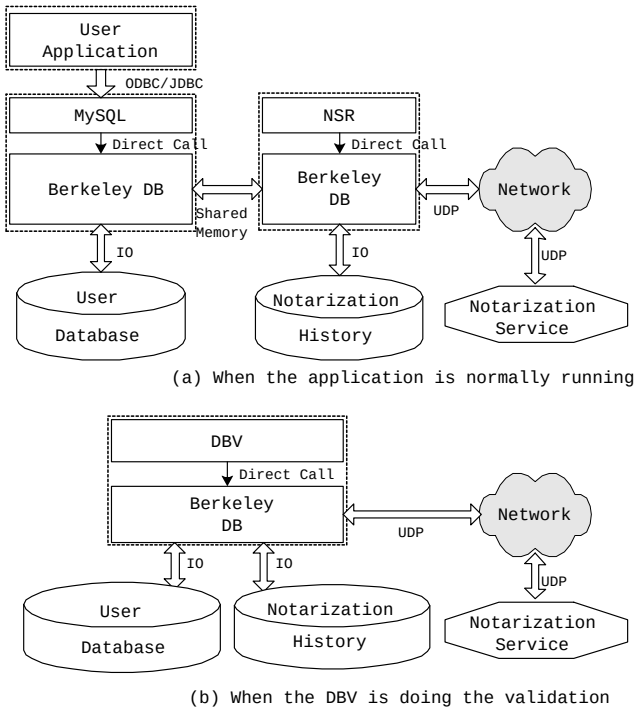

Figure 3: AUD Module Architecture

transaction chain. DBV then displays the time when the invalid transaction is discovered in the database and a list of pages that are potentially involved in the unmatched hash value.

Figure 3(a) illustrates the detailed architecture of the auditing module during a normal run of the application. Figure 3(b) shows the validator validating the audit log. In the figure, each process is again represented by a dotted line rectangle. Different pieces of software are represented by solid line rectangles.

8 Performance

We studied the performance of the auditing system and the various database parameters' impact on the auditing system performance.

8.1 Opportunistic Hashing

We first compare the time and space complexity of the opportunistic hashing approach and the naive approach, during transaction processing and during validation. In the naive approach, for both phases, all tuples are kept and sorted in main memory until being hashed at transaction commit.

As far as main memory space is concerned, when the transaction is running, the naive approach has $O(n)$ (where n is the number of tuples per transaction) space overhead to store the tuples of a transaction, while the optimized approach only has constant space overhead to store the intermediate hashing state. When the validator is running, the naive approach has $O(n)$ space overhead to sort and hash the tuples, which is the same as that of the optimized approach. However the actual space overhead of the optimized approach is lower, because tuples are discarded as the validator scans the database.

As far as CPU time is concerned, when the transaction is running, the optimized approach does not sort tuples at all, while the naive approach has $O(n \lg n)$ cost to sort the tuples in a priority queue. When the validator is running, the time complexity of the sorting is the same for both naive and optimized approaches.

Since the performance improvement due to the opportunistic hashing is obvious from this analysis, we did not run experiments on it.

8.2 Notarization Service Interaction

A trivial analysis can illustrate the dramatic performance improvement due to the minimization of the notarization service interaction. If there are t transactions per day, the basic approach requires t notarization service interactions per day. However, with the opportunistic hashing and linked hashing, only one such interaction is needed. This is a dramatic reduce in the network load. If the notarization service cost is calculated based on the number of interactions, the total service cost is reduced by a factor of t. The same analysis applies to the validator. This analysis suggests that the optimization to minimize the notarization service interaction is critical.

Concerning asynchronous notarization, suppose the round trip time (from the time a request is send to the time a response is received) of a notarization is r seconds and the system throughput is t transactions per second. For each notarization, the audit system with asynchronous notarization can finish $r \cdot t$ more transactions per notarization than one without it.

8.3 Experiment Design

We now turn to a more detailed, empirical analysis of our implementation, which includes all of the optimizations just discussed. We simulated a bank account balance scenario. The database was populated with tuples inserted in random order. Each tuple represented a bank account and the key represented the unique account number. The data represents the account balance.

For all of our experiments the disk page was 8 KB. The tuple size was 250 bytes, implying 32 tuples per page. The experiments were all run on a 3.0GHz Pentium IV running Fedora 1 Linux, with the default buffer pool size of 264KB, accessing data from a local EIDE disk.

The notarization was done every five seconds. The notarization service response time was two seconds (from the DBMS sending the notarization request until

receiving the notary ID back from the notarization service, a quite conservative estimate). This reflects the worst case, because in the real world application, the notarization will not be done that frequently. It would usually be around one notarization per day, which imposes much less notarization overhead than what we did here. We chose five seconds here just to accommodate the experiments, which run for a relatively short amount of time (much less than a day!). We implemented our own notarization service, rather than using one of the commercial services.

8.4 Validation Overhead

As discussed in Section 6.1, validation involves a linear scan of the audit log, which costs $O(n)$ time according to the size of the audit log. (Since an intruder could have changed any byte of the audit log, necessarily the entire audit log must be read by the validator.) As an initial experiment, we ran small transactions (inserting four tuples each), and then validated the audit log. We varied the number of transactions that were run before validation and observed the total running time, which consists of CPU time, sleep time and I/O time. Validation time was under 1% of the time required to create the audit log.

8.5 Impact of the Number of Transactions per Application

We then studied the scalability of the auditing system as the number of transactions grows. The database was initially populated with 4M tuples to simulate that many different bank accounts. implying a starting database of approximately 1GB. Then according to different experiments, insertion and/or deletion operations wrapped in transactions were applied on randomly selected tuples. In order to simulate the data access hot spots (i.e., some of the accounts are very active and represent a greater percentage of the total accesses), the access to the tuples follows a normal distribution with an average equal to half of the largest key and standard deviation equal to $\frac{1}{8}$ of the largest key.

We ran an application that performed updates on this database changing the balance of accounts to a new value (with the audit log retaining the old value in an archival tuple). Each transaction updated four tuples. We ran the application with different number of transactions, and observed the total running time and the number of I/Os with and without the auditing system.

We can see from Figure 4 that the auditing running time overhead increases proportionally to the number of transactions, at about 9% over the non-auditing database (for all the experiments in this paper, the auditing overhead was between 9% and 16%).

The auditing system introduces essentially no I/O overhead (either no or one I/O, for a slightly larger log). For example, at 12,000 transactions, the NSR made only 44 I/O requests.

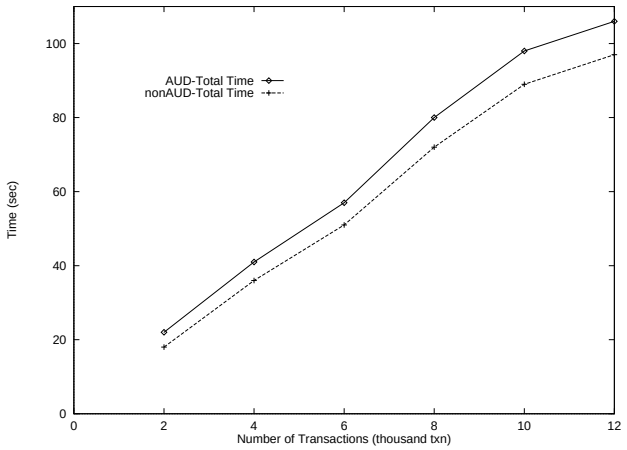

Figure 4: Performance: Changing the Number of Transactions per Application

8.6 Impact of Transaction Size

We then studied the impact of transaction size (number of tuples modified per transaction) on the auditing system performance. The database configuration was identical to the above. We ran the application with 10,000 transactions, but varied the number of tuples updated per transaction, from 2 to 64. Then we observed the total running time and the number of I/Os with and without auditing system. From Figure 5, we see that the overhead of auditing is again about 11%, independent of transaction size.

8.7 Impact of Tuple Size

We then turned to the impact of the tuple size on the auditing system performance. The database configuration was identical to that above. Here each transaction modified four tuples. We fixed the data bandwidth (total bytes of data manipulated) and varied the number of transactions in inverse proportion to the tuple size, namely (10 bytes/tuple, 10,000 transactions) to (1000 bytes/tuple, 100 transactions). Note that the x-axis of Figure 6 is logarithmic.

Here the time overhead for hashing ranged from 16% in the worst case (for very small tuples) to an insignificant overhead for large tuples. From this experiment and the fact that hashing operations count for most of the auditing system overhead, we can infer that the number of hashing operations instead of the total bytes of data hashed dictates to first order the auditing system overhead.

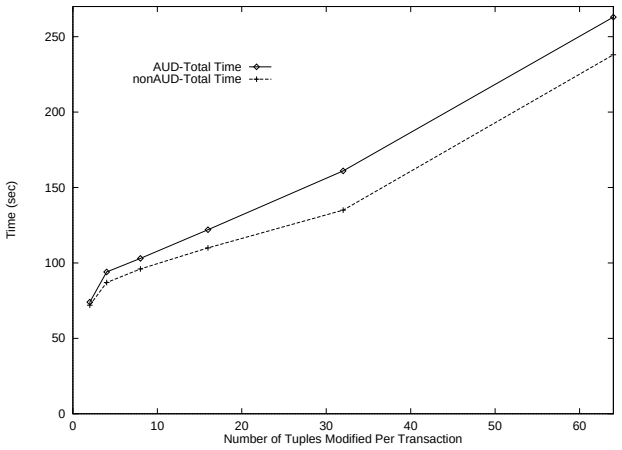

Figure 5: Performance: Changing the Transaction Size

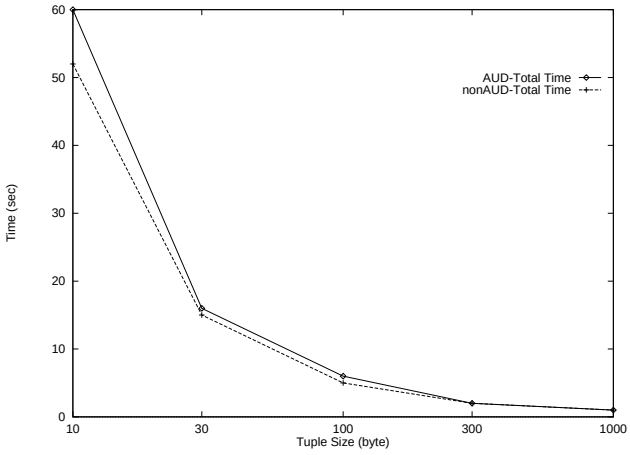

Figure 6: Performance: Changing the Tuple Size While Fixing the Data Bandwidth

8.8 Impact of the Notarization Service Response Time

In this experiment, we studied the impact of the notarization service response time on the auditing system. The "response time" is defined as the time interval from sending the notarization service request until the DBMS received the notary ID back from the notarization service. There were 10,000 transactions in this application and each updated four tuples. We ran the application with various notarization response time, namely 1, 2, 3, and 4 seconds (the system was executing about 100 transactions per second, so there are many hundreds of transactions backed up when the response time is four seconds) and observed the CPU time, total running time and number of I/Os with the auditing system.

The notarization service response time does not affect the auditing system performance. The total time was around 98 seconds, independent of the response time. When the notarization takes longer time to return the notary ID, the currently running transactions will accumulate in the TOL in the AUD module, which causes an undetectable CPU overhead to maintain the TOL data structure management.

9 Related Work

There has been related work in several fields: security, operating systems, and databases. We address each in turn.

Mercuri raises the need to audit the audit log [14]. Peha [17] uses, as we do, one-way hash functions and a "trusted" notary to hash and store every transaction. Our approach differs in that we make no assumptions about the DBMS, or even the hardware it executes on, remaining in the trusted computing base following an intrusion; Peha on the other hand advocates a "notary on a chip". Unlike Peha, we integrate hashing with stamping of tuples in the table, and we consider system issues such as the need to hash tuples and using partial result authentication codes to link transactions. Peha simply batches transactions together by hashing all the data in all the transactions, which will undoubtedly result in very poor performance, as we discussed in detail in Section 6.1. Peha goes into more detail on how customers, notarizers, validators, and auditors can use public key encryption to coordinate. Note that since we send the notarization service only hash values, no private data that is revealed to that external service. It may still be useful to encrypt the tuples that flow from the database to the validator, if that process communicates with the DBMS over non-secure channels.

As mentioned in Section 2, Schneier and Kelsey address audit logs that are used for later forensic investigations into detected intrusions [21]. Their requirements differ considerably from ours. In particular, they render the log entries impossible for the attacker to read. They use a hash linking in a similar way to our algorithm. They do not consider efficiency issues, which are critical in our situation where an online transactional database is being logged.

Merkle proposed a digital signature system based on a secure conventional encryption function over a tree of document fragments [15]. This work could be utilized within an notarization service, but is not directly applicable to our problem of hashing the data of individual transactions.

Devanbu et al. applied the Merkle Tree authentication mechanism to both relational [6] and XML [5] data. Here the model is different: queries over static data which has been previous digested are evaluated by an insecure server. The query results are sent to clients, which can independently verify, using the digest, that the result contains all the requested records and no superfluous records. While our approach also

uses a hashing approach similar to Merkle and this work (though not tree-based), we focus on *modifications* made by the perhaps compromised server.

The file system equivalent of a transaction-time database is a *read-only file system*, in which new files can only be added; existing files cannot be modified [8], or a read-write system in which the files are logged, such as in the Ivy system [16]. These systems sign the data, so that programs that read a file can be assured that it has not been corrupted. They share with the present paper the need for high performance.

POSTGRES [24] and Dali [4] were two different approaches of protecting critical DBMS data structures against software errors. The former used hardware protection, while the latter used a wordcode software mechanism and also proposed a recovery algorithm for the corrupted transactions. They mainly dealt with abnormal corruptions caused by software errors instead of human hackers. And the protection was limited to the in-memory data structures instead of the whole database on disk. If Bob directly changed the data in the disk file, this intrusion would not be detected by these mechanisms.

Liu, Ammann and Jajodia propose an algorithm that rewrites the execution history backing out malicious transactions while preserving the good transactions [1, 13]. This assumes that the malicious transactions had been identified according to the application semantics; they did not deal with intrusion detection. Perhaps our detection approach could be coupled with their repair technology.

10 Summary and Future Work

Motivated by audit log requirements, we have presented a new approach to providing audit logs for transaction processing systems that can effectively and efficiently detect tampering. We based our approach on existing cryptographic techniques such as strong cryptographic hashing, partial result authentication codes, and off-site digital notarization services. Our contributions are as follows.

- We showed how a transaction-time database can be used as the basis for tamper-detecting audit logs, transparent to the application.
- To reduce the expense of interacting with the notarization service on a per-tuple basis, an opportunistic hashing algorithm was proposed to compute the data hash value of transactions, so as to enable per-transaction notarization. Incremental hashing and tuple sequence numbers are proposed to support opportunistic hashing.
- To further minimize the expense of interacting with notarization services, linked hashing of transactions was introduced, by means of partial result authentication codes.
- We used an in-memory data structure, the transaction ordering list, to handle non-monotonic assignment of transaction times that result from sophisticated timestamping algorithms as well as handling delays from the off-site notarization service.
- We designed our validator to make a single scan of the audited tables, interacting with the notarization service infrequently.
- We developed an implementation within the high-performance Berkeley DB data storage manager, and showed through experiments that the overhead never exceeds 16%, that to a first approximation the tuple size dictates the auditing system overhead, rather than the total number of bytes hashed, and that the notarization service response time minimally impacts system performance.

There is more work that can be done. We've focused in this paper on certifying in a definitive fashion that an audit log is pristine. If an audit log has been tampered with, a detailed forensic analysis would be required and would involve going back to other records, perhaps even to physical records, to document the fraud. In such circumstances, it would be helpful for the validator to provide information enabling the analysis to determine fairly precisely *who* performed the attack, *when* the attach occurred, and *which* data was compromised. To provide this information about the intrusion, various hints, such as page-level hash values, and the algorithms of inferring detailed intrusion information from these hints need to be developed. It is also possible for the DBMS and applications to log activities through additional data stored in the database. That additional data would be helpful in analyzing intrusions. Of course, such data should be stored in audited tables.

The validator should be enhanced to differentiate corruption of the hints from corruption of the data, and should be able to contend with multiple intrusions. The appropriate notarization granularity (an hour, a day?) should be investigated. And we want to look at partitioning to eliminate the need to scan all pages of audited tables.

Finally, we want to produce and evaluate mechanisms that leverage tamper detection of audit logs to produce *tamper-resistant audit logs*, which cannot be corrupted, yet are still accessible to the application, and accord with our very general threat model.

References

[1] P. Ammann, S. Jajodia, and P. Liu, "Recovery from Malicious Transactions," IEEE *Transactions on Knowledge and Data Engineering* 15(5):1167–1185, September 2002.

[2] J. Bair, M. Böhlen, C. S. Jensen, and R. T. Snodgrass, "Notions of Upward Compatibility of Temporal Query Languages," *Business Informatics (Wirtschafts Informatik)*, Vol. 39, No. 1, February, 1997, pp. 25–34.

[3] M. Bellare and B. Yee, "Forward Integrity for Secure Audit Logs," Technical Report, Computer Science and Engineering Department, University of California at San Diego, November 1997.

[4] P. Bohannon, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan, "Using Codewords to Protect Database Data from a Class of Software Errors," in *Proceedings of the IEEE International Conference on Data Engineering*, 1999, pp. 276–285.

[5] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. G. Stubblebine, "Flexible Authentication of XML documents," in *Proceedings of the ACM Conference on Computer and Communications Security*, Philadelphia, PA, November, 2001, pp. 136–145.

[6] P. Devanbu, M. Gertz, C. Martel and S. G. Stubblebine, "Authentic data publication over the Internet," *Journal of Computer Security* 11(3):291–314, 2003.

[7] P. DuBois, **MySQL**, Second Edition, SAMS, 2003.

[8] K. Fu, M. F. Kaashoek and D. Mazieres, "Fast and secure distributed read-only file system," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 181-196, October 2000.

[9] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson, **2004 CSI/FBI Computer Crime and Security Survey**, Computer Security Institute, 2004.

[10] S. Haber and W. S. Stornetta, "How To Time-Stamp a Digital Document," *Journal of Cryptology* 3, pp. 99–111, 1999.

[11] C. S. Jensen and C. E. Dyreson (eds), A Consensus Glossary of Temporal Database Concepts—February 1998 Version," in **Temporal Databases: Research and Practice**, O. Etzion, S. Jajodia, and S. Sripada (eds.), Springer-Verlag, pp. 367–405, 1998.

[12] C. S. Jensen and D. B. Lomet, "Transaction Timestamping in (Temporal) Databases," in *Proceedings of the International Conference on Very Large Databases*, Roma, Italy, pp. 441–450, 2001.

[13] P. Liu, P. Ammann and S. Jajodia, "Rewriting Histories: Recovering from Malicious Transactions," *Distributed and Parallel Databases Journal* (8):1:7–40, January 2000.

[14] R. T. Mercuri, "On Auditing Audit Trails," *CACM* 46(1):17–20, January 2003.

[15] R. C. Merkle, "A Certified Digital Signature," *Advances in Cryptography—Annual International Cryptography Conference*, Vol. 435, pp. 218–238, 1989.

[16] A. Muthitacharoen, R. Morris, T. M. Gil and B. Chen, "Ivy: A Read/Write Peer-to-Peer File System," in *Proceedings of USENIX Operating Systems Design and Implementation*, 2002.

[17] J.M. Peha, "Electronic commerce with verifiable audit trails," in *Proceedings of ISOC*, 1999. `www.isoc.org/isoc/conferences/inet/99/-proceedings/1h/1h_1.htm`, viewed on March 26, 2003.

[18] M. Rabi and A. T. Sherman, "An observation on associative one-way functions in complexity theory," in *Information Processing Letters*, Vol 64, Issue 5, pages 239-244, 1997.

[19] R. Ramakrishnan and J. Gehrke, **Database Management Systems**, Third Edition, 2003.

[20] B. Salzberg, "Timestamping After Commit," in *Proceedings of PDIS Conference*, pp. 160–167, 1994.

[21] B. Schneier and J. Kelsey, "Secure Audit Logs to Support Computer Forensics," ACM *Transactions on Information and System Security* 2(2):159–196, May 1999.

[22] Sleepycat Software Inc., **Berkeley DB**, 2001.

[23] M. Stonebraker, "The Deisgn of the POSTGRESS Storage System," in *Proceedings of the International Conference on Very Large Databases*, pp. 289–300, 1987.

[24] M. Sullivan and M. Stonebraker, "Using Write Protected Data Structures to Improve Software Fault Tolerance in Highly Available Database Management Systems," in *Proceedings of the International Conference on Very Large Dsta Bases*, pp. 171–180, 1991.

[25] National Institute of Standards and Technology, "Secure Hash Signature Standard," August 2002.