A Portable Implementation Framework for Intrusion-Resilient Database Management Systems

Alexey Smirnov
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
alexey@cs.sunysb.edu

Tzi-cker Chiueh
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
chiueh@cs.sunysb.edu

Abstract

An intrusion-resilient database management system is the one that is capable of restoring its consistency after being compromised by a malicious attack or a human error. More specifically, an intrusion-resilient mechanism helps to quickly repair a database by nullifying the damage caused by malicious or erroneous transactions, while preserving the effects of unaffected legitimate transactions that take place between intrusions/errors and their detection. The goal of this project is to develop a portable implementation framework that can augment a commercial database management system with intrusion resilience without requiring any modifications to its internals. The intrusion resilience mechanism described in this paper significantly improves the availability of modern DBMSs by facilitating and sometimes even automating the post-intrusion damage repair process. In addition, it can be embodied in a reusable implementation framework, whose portability is demonstrated by its successful application to three different DBMSs: PostgreSQL, Oracle, and Sybase. Performance measurements on the fully operational prototypes under the TPC-C benchmark show that the run-time overhead of the intrusion-resilience mechanism is between 6% and 13%.

1. Introduction

Consistency of an information system can be compromised due to a hardware failure, a malicious attack, or a human mistake. Standard recovery mechanisms in modern database management systems are designed to recover from hardware failures, which can be detected as soon as they occur. For malicious attacks and human mistakes, where there is typically a time gap between occurrence and detection, these recovery mechanisms are inadequate because they can neither roll back committed transactions nor keep track of inter-transaction dependencies. As a result, to clean up a compromised database using existing tools takes time-consuming human efforts and typically results in a long mean time to repair (MTTR) and thus database down time. We call an information system intrusion-resilient if it can quickly repair the damage caused by a malicious attack or human error and maximize the overall system availability.

A wide variety of information systems can be made intrusion-resilient. For instance, Zhu and Chiueh [2] described a general intrusion-resilience implementation framework for network file servers. The system, called RFS (Repairable File Service), aims at facilitating the post-intrusion repair process for network file servers. RFS is not a file server on its own. Instead, it is a special file server that acts as a proxy between a protected network file server and its clients and logs all the file updates. The resulting log is used at recovery time to determine the extent of the damage and to undo any detrimental side effects.

Because there is a time gap between when an attack/error occurs and when it is detected, legitimate transactions that are not related to the attack/error could be committed to the compromised database during this period. The key problem that an intrusion-resilient DBMS needs to address is how to completely undo the damage caused by an attack or an error while preserving the effects of these good transactions as much as possible. More specifically, an intrusion-resilient DBMS should be able to:

- Determine the exact extent of database damage from an initial set of attack/error transactions identified by the database administrator, including transactions that are benign in nature but are polluted by attack/error transactions.
- Perform a selective rollback of those transactions that are considered corruptive to undo the database damage caused by the attack or the error.

Because standard recovery mechanisms in modern DBMSs performs neither of the above functions, today's database administrators (DBA) have to perform these two tasks manually to repair a compromised DBMS. A typical post-

intrusion repair procedure involves restoring the compromised database to a state before the attack/error, analyzing the transaction log in detail to identify the corrupting transactions, and redoing only those transactions that are legitimate and unaffected by the attack/error. In most cases, this is a time-consuming, error-prone and labor-intensive process.

In this paper, we develop a fast database damage repair mechanism that can quickly repair a database compromised by an intrusion or an error and thus greatly improve the database availability. This mechanism keeps track of inter-transaction dependencies at run time in order to determine the exact extent of the database damage at repair time, and performs a selective rollback of those and only those corrupting transactions. Moreover, this fast database damage repair mechanism does not require any modification to the DBMS internals, and thus could be embodied in a reusable implementation framework that can be easily ported to different DBMSs with no or minor customization.

2. Related Work

The previous research on survivable and intrusion-resilient systems has evolved in both hardware and software fields and has addressed such areas as file systems, storage systems, and database systems.

Wylie et al. [3, 4] describes a survivable storage system S4, which is a network-attached object store with an access interface based on storage of objects. The Repairable File System (RFS) project [2] aims at improving the speed and precision of post-intrusion damage repair for NFS servers. Traditionally, file system recovery uses signatures generated by systems such as Tripwire [5] to determine the corrupted system files or complete point in time restoration from backups. Instead, RFS maintains file system operation logs and carries out dependency analysis to provide fast and accurate repair of damage caused by NFS operations issued by attackers.

Traditional database recovery methods have been discussed in many database textbooks [9, 10]. Combined with data replication, WAL presents an efficient way for a database to recover after media failures.

The problem of database post-intrusion recovery has been addressed from both theoretical [6, 7] and practical [1, 8] points of view. Liu [7], develops a family of architectures for intrusion-tolerant database systems. Subsequent architectures enhance the first basic architecture by addressing various problems such as attack isolation, damage confinement, and quality of information assurance provision.

Ammann et al. [6] proposes various algorithms for recovery from malicious transactions. The authors address two problems: the problem of inter-transaction dependency tracking and the problem of database repair. Inter-transaction dependency tracking requires knowing the data read and written by each transaction. The latter problem is relatively simple since this information is logged by modern DBMSs. However, the former problem is much more difficult. Two solutions are proposed in this paper — comprehensive logging of transaction reads and extracting the read information from transaction profiles. The authors admit that modern database systems do not support read logging and, therefore, the first approach requires changing the source code of existing DBMS. The second solution has also some limitations, because it is possible to come up with a transaction whose read set profile will not provide a complete information on the data read by this transaction. However, the authors claim that this solution will work in many cases by providing the read set templates for TPC-C transactions. Two versions of repair algorithms are provided for each of the tracking approaches — static and dynamic. The static algorithm brings the whole database offline during the repair time, whereas the dynamic algorithms performs on-the-fly repair. There is a trade-off between the two algorithms: the dynamic algorithm provides better service availability, but it can initially mark some benign transactions as malicious (to prevent the damage from being spread over the database) thus preventing the user from accessing the data modified by these transactions. Based on this work, an intrusion tolerant database system [8] was implemented as an enhancement to Oracle database server.

Pilania et al. [1] describes an intrusion resilience mechanism for PostgreSQL. The problem of transaction dependency tracking is solved by modifying the internals of PostgreSQL to allow the transaction read information to be captured. Although the system has a relatively small overhead, its main drawback is that the technique used in it cannot be directly applied to other DBMSs.

3. Intrusion-Resilient DBMS System Architecture

3.1. Inter-Transaction Dependency

Let us call the set of rows that an SQL statement retrieves from the database for further processing the *read set* of the statement. The read set of a `SELECT`, `UPDATE` or `DELETE` statement is the set of rows satisfying its `WHERE` clause. An `INSERT` statement has an empty read set. An SQL statement S_2 depends on another SQL statement S_1 if the read set of S_2 was modified by S_1. Transaction T_1 depends on transaction T_2 if there exists one statement S_1 in T_1 and another statement S_2 in T_2 such that S_1depends on S_2.

This definition of transaction dependency could lead to both false positives and false negatives. For example, a false positive occurs if two transactions, T_1 and T_2, update different attributes of a row. Even though they do not share any data, one is still considered dependent on the other. This problem can be solved by tracking inter-transaction depen-

dencies on a column by column basis, which incurs a much higher overhead. A false negative occurs if a transaction T_1 updated the balance of an account from \$50 to \$500, and later T_2 charged a service fee from all accounts whose balance is less than \$100. Each account is represented by a table row. In this case, T_2 does not depend on T_1, because the read set of T_2 does not include the row that T_1 updates. However, were T_1 not to update the row, the read set of T_2 would have included that row. Therefore, if T_1 is a malicious transaction, the right repair operation is to roll back both T_1 and T_2, even though dependency analysis suggests only T_1 needs to be undone.

There are also scenarios in which implicit inter-transaction dependencies will not be caught, for instance, dependencies that arise as a result of internal application logic or inter-application interactions. In general, transaction dependencies cannot be tracked only by just analyzing SQL statements issued to a DBMS. Because of all these issues, it is still advisable for the DBA to be intimately involved in the determination of the final set of corrupting transactions, using the transaction set derived from the dependency analysis as the starting point.

3.2. Transaction Dependency Tracking

An intrusion-resilient DBMS needs to keep track of inter-transaction dependencies constantly so that it can use this information to determine the damage perimeter at repair time. To record inter-transaction dependencies in a way independent of the underlying DBMS, we propose a transaction dependency tracking mechanism that is based on intercepting and rewriting SQL statements sent from a database client to its database server. One way to transparently intercept SQL statements from DBMS client to DBMS server is to put a proxy program between them. Another alternative to transaction dependency tracking is to use database triggers, but this approach is not feasible because modern DBMSs do not support read triggers (therefore, it is not possible to intercept `SELECT` statements).

If a DBMS client uses an open database connectivity protocol such as JDBC to connect to the DBMS server, a proxy JDBC driver sitting on the client side can perform query interception and rewriting, as shown in Figure 1. Putting the intercepting proxy on the server side is infeasible because data transmitted over the network is in a DBMS-specific and typically proprietary format. Putting the proxy on the client side makes the database vulnerable to an attack in which an attacker uses a standard JDBC driver bypassing the proxy. In this case, the malicious transactions executed by the attacker will not be tracked, and, therefore, it will not be possible to identify them and roll them back. This problem can be solved by using two proxies as shown in Figure 2. One of these proxies resides on the client side, the other resides on the server side. The goal of the client-side proxy is to transmit the data to the server-side proxy in some format known to both proxies. The transaction dependency tracking is performed by the server-side proxy. The server-side proxy establishes a local connection to the database through a standard JDBC driver.

Transaction dependencies are stored as regular database tables and are committed to the database together with the transactions that these dependencies relate. The following changes are made to a database when it is created:

- The table `trans_dep(tr_id INTEGER, dep_tr_ids VARCHAR)` is added to the database. For each transaction ID, it stores the set of IDs of the transactions it depends on as a string with IDs separated by spaces.
- The table `annot(tr_id INTEGER, descr VARCHAR)` is added to the database. It contains a symbolic name for each transaction, which is used in the visualization of the inter-transaction dependency graph, as shown in Figure 3.
- A new field `trid:INTEGER` is added to each database table transparently. The field of each row stores the ID of the last transaction that modified the row.

By rewriting incoming SQL statements in a transaction in the way shown in Table 1, the intercepting proxy can track and record inter-transaction dependencies. For a `SELECT` statement, the proxy additionally retrieves the `trid` attribute from each table involved in the statement. These attributes contain the IDs of the transactions that update the rows being read most recently. When the DBMS server returns a set of rows, the proxy reads these rows' `trid` field and store them in a local array. For an `UPDATE` statement, the intercepting proxy updates the `trid` attribute of all the rows the statement modifies with the current transaction ID. Because an `UPDATE` statement implicitly involves a `SELECT` operation, the transaction containing the `UPDATE` statement thus depends on the transactions whose IDs are stored in the `trid` attribute of the rows being updated. In theory, these transaction IDs can be retrieved by executing a `SELECT` statement before the `UPDATE` statement. However, we decide to skip this step to reduce the run-time performance overhead. This does not affect correctness as DBMS logs `UPDATE` operations and these dependencies can be reconstructed at repair time from the transaction log. For the same reason the proxy does not record the associated inter-transaction dependencies associated with a `DELETE` statement.

For a `COMMIT` statement, the proxy issues an `INSERT` operation that records the current transaction ID and the IDs of all transactions it depends on in the `trans_dep` table. Having done so, the proxy commits the current transaction by sending the `COMMIT` operation to the DBMS server.

Because inter-transaction dependency tracking involves only SQL query rewriting, it is highly portable across different DBMSs as long as they support standard SQL interface. In addition, most DBMSs also support open database

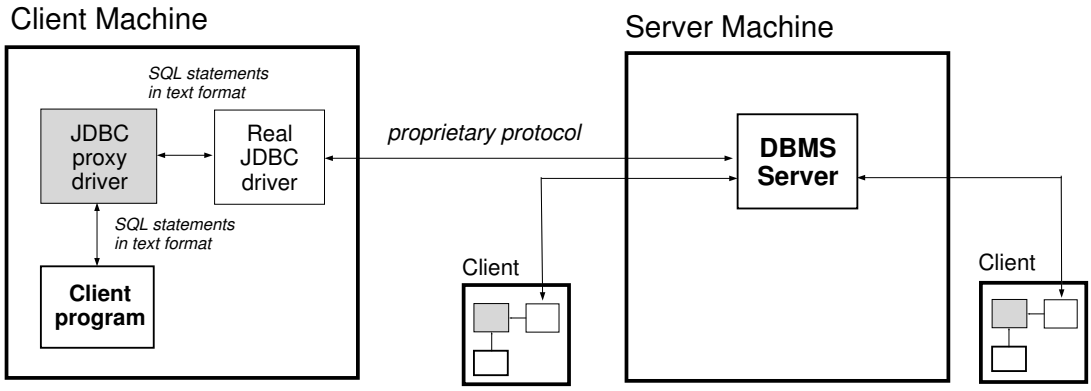

Figure 1. ***Client-side single-proxy architecture for inter-transaction dependency tracking, where shaded boxes represent new elements to be added to a standard client-server DBMS system.***

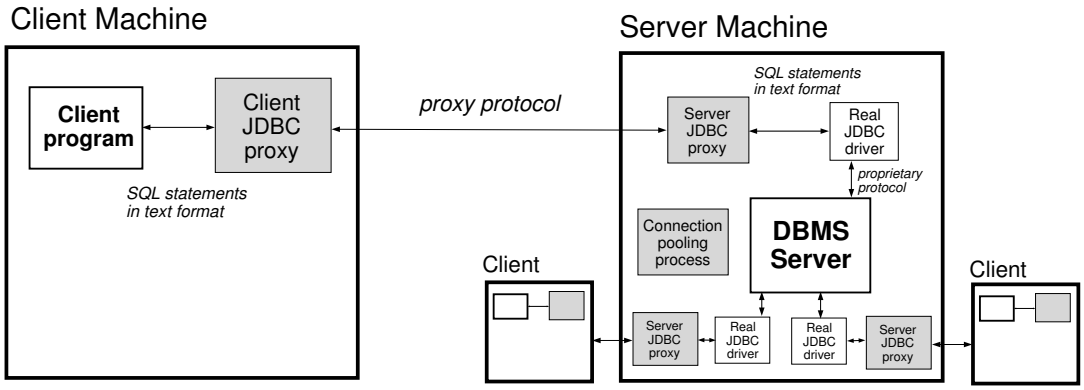

Figure 2. ***Dual-proxy architecture for inter-transaction dependency tracking, where shaded boxes represent new elements to be added to a standard client-server DBMS system.***

connectivity protocol such as JDBC. Therefore, our inter-transaction dependency tracking module is fully reusable across Oracle, Sybase, and PostgreSQL.

3.3. Selective Undo of Committed Transactions

Modern DBMSs perform transaction logging on a per-row basis, and create a separate log entry for each row being modified. As a result, multiple log entries could be created from a single SQL statement that affects multiple rows. Each log entry contains the operation type, e.g., `INSERT`, `DELETE` and `UPDATE`, internal transaction ID, the ID of the table that the row belongs to, and the database data affected by this operation. For an `INSERT` and `DELETE` operation, the entire row is saved into the log. The amount of data saved into the log in the case of an `UPDATE` operation varies from DBMS to DBMS. It can be either complete pre-update and post-update image of an updated row or only those attributes that were actually modified. Each log record also contains a reference to the position in the database to which the change it describes is applied. For the three DBMSs that we have studied, this position is a physical location within the disk file containing the database, and is described by a logical page number and an offset within this page. This addressing format is convenient for recovering a database after a crash or media failure when the transaction log needs to be rolled forward.

The effect of a transaction can be nullified with a compensating transaction. To do so, we consider each row affected by an original transaction, create a compensating

Original statement	Modified statement(s)
SELECT $t_1.a_1, ..., t_1.a_{n_1}, ..., t_k.a_{n_k}$ FROM $t_1, ..., t_k$ WHERE c	SELECT $t_1.a_1, ..., t_1.a_{n_1}, ..., t_k.a_{n_k}, t_1.trid, ..., t_k.trid$ FROM $t_1, ..., t_k$ WHERE c
SELECT SUM($t.a$) FROM t WHERE c GROUP BY $t.b$	SELECT $t.trid$ FROM t WHERE c SELECT SUM($t.a$) FROM t WHERE c GROUP BY $t.b$
UPDATE t SET $a_1 = v_1, ..., a_n = v_n$ WHERE c	UPDATE t SET $a_1 = v_1, ..., a_n = v_n$, $trid = curTrID$ WHERE c
INSERT INTO $t(a_1, ..., a_n)$ VALUES $(v_1, ..., v_n)$	INSERT INTO $t(a_1, ..., a_n, trid)$ VALUES $(v_1, ..., v_n, curTrID)$
COMMIT	INSERT INTO trans_dep($curTrID, ...$) COMMIT

Table 1. ***Modifications to SQL statements that the intercepting proxy makes to track and record inter-transaction dependency information.***

statement for each affected row, and form a compensating transaction from these per-row compensating statements. For instance, if a particular row was deleted, its compensating statement is an INSERT that puts this row back into the database. Similarly, if a row was inserted, the compensating action is a DELETE statement. Finally, if a row was updated, a compensating action is another UPDATE statement restoring the pre-update image of the row. Although the transaction log can be used to generate compensating transactions, it does not contain sufficient information to address any given row precisely so that each compensating statement is applied to that row only. Fortunately, most DBMSs support a read-only row ID attribute in each table. We can use this attribute in WHERE clause of UPDATE and DELETE compensating statements to ensure that the change is applied to a particular row only.

The intercepting proxy generates its own transaction IDs at run time because it is not always possible to access the internal transaction ID of the underlying DBMS, if it exists at all. To correlate a transaction's internal ID with its proxy-generated ID, one searches for the last log entry right before the transaction's commit operation, which should be an insert operation into the trans_dep table. The proxy-generated ID contained in this inserted row and the internal transaction ID recorded in this log entry establish the desired correspondence. Once the correspondence between two types of transaction ID is established, transaction dependencies due to UPDATE and DELETE statements are generated. For each entry in the transaction log that is due to a DELETE and UPDATE statement, one builds up a dependency between the transaction to which the log entry belongs and the transaction whose ID is stored in the pre-update row image associated with the log entry.

After all transaction dependencies are identified, the complete transaction dependency graph is visually presented to the DBA. An example transaction dependency graph display is shown in Figure 3. The current prototype uses GraphViz [13], an open source graph drawing software from AT&T, for graph display. The DBA determines the final undo set of transactions by analyzing the transaction dependency graph using the application-specific domain knowledge. Ideally, this transaction dependency graph rendering software should be part of an interactive database damage repair tool, which allows the DBA to include certain inter-transaction dependencies into dependency analysis and ignore others, thus avoiding both false positives and negatives. For instance, if the database contains a temporary table that does not have any semantic significance, the DBA may choose to ignore all the dependencies among transactions due to this table. As another example, one transaction may depend on another due to false sharing, i.e., touching different attributes of the same row. In this case, the DBA may choose to ignore this type of dependencies in the repair process.

After the undo transaction set is determined, each entry in the transaction log is checked from the end to the beginning. If the proxy transaction ID of a log entry belongs to the undo set its corresponding compensating statement is executed immediately. Special care is required for rows inserted to the database during the repair process. For example, when a DELETE log entry is to be undone, a new row is inserted into the database, and the DBMS assigns it a unique row ID, which may be different from the row ID that was used to refer to this row in the transaction log previously. As a result, the old row ID needs to be mapped to the new row ID when processing all subsequent log entries associated with this row. When an INSERT log entry associated with this row is encountered, the mapping has to be discarded. Each table has its own row ID mapping. Consequently, the same row ID can be mapped to different row IDs in different tables.

Unlike inter-transaction dependency tracking, the repair-time logic of an intrusion-resilient DBMS is very database-specific, because many of the following data structures are proprietary: the transaction log format, transaction ID and row ID encoding, pre-update row image representation, etc.

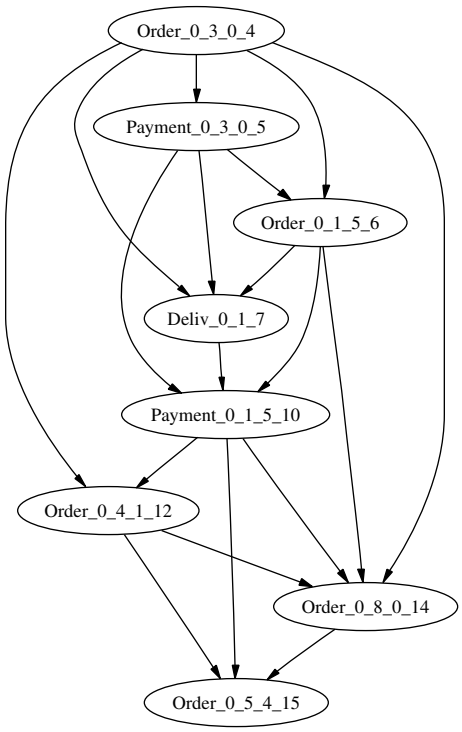

Figure 3. ***Visualization of a sample inter-transaction dependency graph. Nodes correspond to transactions and edges to inter-transaction dependencies. Each node has a label describing the transaction type, e.g., `Order` for an order placement transaction, `Payment` for an order payment transaction, `Deliv` for an order delivery transaction. Numbers that are part of each label are the warehouse ID, the district ID, the client ID, and the transaction ID.***

4. Implementation Issues

The most challenging part of our prototype implementation efforts is transaction log parsing, analysis, and reconstruction. In this section, we discuss in greater detail how reconstructing transaction log entries in Oracle, Sybase, and PostgreSQL is done.

4.1. Transaction Log Processing in Oracle

Oracle provides a set of PL/SQL procedures called LogMiner [14] which is designed to convert a binary transaction log into a database table called `v$logmnr_contents`, which is accessible via SQL inside the database. This database table contains one row per transaction log entry. Each row has attributes such as operation type, user ID, transaction ID, as well as a corresponding redo and undo SQL statement. In order to roll back a particular transaction, one needs to execute all undo SQL statements available in the `v$logmnr_contents` for this transaction.

4.2. Transaction Log Processing in PostgreSQL

PostgreSQL [15] does not have any tools for accessing its transaction log. However, it is possible to reverse-engineer its log format since PostgreSQL is an open source DBMS. It turns out that for each row operation (`UPDATE`, `DELETE`, `INSERT`), PostgreSQL stores complete contents of the before and after images (if required) for that row. We have implemented a plugin for PostgreSQL that provides a Logminer-kind functionality.

4.3. Transaction Log Processing in Sybase

The major implementation issue in Sybase [16] is the fact that Sybase does not have a row ID attribute in its tables. One has to add an attribute of type `numeric(n) identity` to provide a row ID for each row. The dependency tracking proxy for Sybase intercepts all `CREATE TABLE` statements and adds such a column to each new table.

Sybase provides the `dbcc log` command to read the contents of the transaction log. For each row operation this command outputs the contents of the row being modified in the binary format without dividing it into attributes. If the row operation is `INSERT` or `DELETE`, Sybase stores complete row contents in the log. However, for an `UPDATE` operation (called `MODIFY` in Sybase), only those attributes that were modified are stored in the log. Therefore, the row ID attribute we introduced to identify a row is never saved in the transaction log. For the repair purpose, the entire content of each row appearing in the log needs to be fully restored to obtain the row ID attribute.

Sybase stores the page number and the offset within the page for each row operation. Given these two values, one can read the contents of the page by using `dbcc page` command, retrieve complete row content and access the row ID attribute of the required row. However, a row can migrate within a page when some other rows are deleted. Therefore, a row's current location at the moment when the transaction log is read might be different from the row's current location when it is stored in the transaction log. The rule of row migration is as follows: when a row is deleted from the middle of a page, all rows located closer to the end of the page are moved closer to the beginning of the page, overwriting the row being deleted so that no gaps ever exist in the page. Rows cannot migrate from one page to another. These observations allow one to develop an algorithm to keep track of the location of each row in a page and thus to identify the row ID attribute of every row associated with log en-

tries of type MODIFY. In this algorithm, $rec.len$ denotes the length of data (in bytes) of a log record rec, and $rec.off$ denotes the offset of data (in bytes) of a log record rec within a data page.

1. Read the transaction log using dbcc log command. For each log record that modifies a table that needs to be repaired, store its record type and data offset in the repair tool's memory.
2. Go through the list of all log records again and do the following: for each record rm of type MODIFY go through all log records of type DELETE located after rm. For each DELETE record rd, decrement $rm.off$ by $rd.len$ if $rd.off + rd.len \leq rm.off$. If $rd.off \leq rm.off < rd.off + rd.len$, then the current DELETE operation deletes the row being modified, and since the log entry associated with a DELETE operation keeps a complete image of the deleted row, this image could be used as the before image for the MODIFY statement as well in this case.
3. Go through the list of all records and for each record rm of type MODIFY, issue a command dbcc page with an appropriately adjusted $rm.off$ to read the full row content.

Having restored the complete row data for each operation, it is now relatively straightforward to generate all compensating statements. We have implemented the algorithm presented above and were able to generate compensating statements correctly.

4.4. Limitations of Current Prototype

Our current prototype has several limitations. First, there is no support for stored procedures. However, the code stored on the server's side can be modified in advance to support transaction dependency tracking. Second, almost all DBMS vendors provide custom extensions to standard SQL. Our current prototype supports only a subset of it, thus making it impossible to use the tracking proxy in an arbitrary database. However, the tracking proxy can be customized for each DBMS vendor. The current prototype has also several limitations due to its inter-transaction dependency tracking mechanism. Essentially, the tracking is row-based rather than field-based. This can result in false inter-transaction dependencies. Another similar problem is the dependencies that are the result of application logic (such as inter-process communication). Such dependencies cannot be tracked by our current tracking proxy.

5. Performance Evaluation

5.1. Experimental Setup

We used the TPC-C benchmark [12] to evaluate the run-time performance overhead of the intrusion resilience mechanism added to Oracle, Sybase, and PostgreSQL. This benchmark simulates activities of a wholesale supplier. The supplier operates a number (W) of warehouses each of which has its own stock. A warehouse is comprised of a number of districts, each of which in turn has a number of clients. The TPC-C benchmark also describes the set of transactions that are issued during benchmarking runs: order placement, payment, order delivery, order status inquiry, stock level inquiry. Orders can only be made by clients already in the database. We created a TPC-C database and populated it with random data. The parameters of the database are shown in Table 2. For the DBMSs that support disk space pre-allocation, we have pre-allocated a sufficiently large data file of size 4.5 GB to avoid dynamic allocations at run time. All databases were configured with a block size of 8 KB.

Number of warehouses	10
Districts per warehouse	30
Clients per district	5000
Items per warehouse	100000
Orders per district	5000

Table 2. *Test database parameters and their values.*

The following machines are used in the experiments. A Pentium-4M 1.7GHz laptop with 512MB of RAM and a 30GB hard drive spinning at 4200 RPM is used as a client issuing transactions requests. A Pentium-4 1.5GHz desktop with 384MB of RAM and a 120GB hard drive spinning at 7200 RPM is used as a server running the DBMS under test. The two machines were placed in the same 100Mbps local network. Both machines used Mandrake Linux 9.1. We have measured the following DBMS servers: Oracle 9.2.0, Sybase ASE 12.5, PostgreSQL 7.2.2. In all our experiments, we used a single-proxy dependency tracking architecture, as shown in Figure 1.

In this performance study, we are mainly interested in answering the following two questions. First, what is the run-time performance cost of transparently augmenting an existing DBMS with the proposed intrusion resilience mechanism? Second, how much value does such an intrusion resilience mechanism help in the post-intrusion or post-error repair process, in terms of the percentage of transactions whose effects can be preserved in the repair process because they are legitimate and unaffected by the intrusion or error?

5.2. Dependency Tracking Overhead

The run-time overhead of inter-transaction dependency tracking includes query interception and modification overhead as well as additional SQL query processing because of additional fields and tables introduced for dependency

tracking. We vary the following workload parameters when measuring the dependency tracking overhead for different DBMSs:

- *Transaction Mix*: We used a read-intensive workload (consisting of 100 read intensive `Stock Level` transactions) and a read/write intensive workload (consisting of 200 `New Order`, 200 `Payment` and 100 `Delivery` transactions).
- *DBMS Client-Server Connection*: In *local* configuration both the DBMS client and server were placed on the server machine. In the *networked* configuration the DBMS client ran on the client machine and the DBMS server ran on the server machine.
- *Total Footprint Size*: We varied the total footprint size of the input workload so that in one case, a small amount of data is accessed repeatedly and the data accessed is mostly in the database cache once it is loaded, and in the other case, a large amount of data is accessed randomly and mostly once.

Figure 4 presents the relative throughput penalty of transaction dependency tracking with respect to the original DBMS without any intrusion resilience mechanism, where the overall throughput is the ratio of the number of transactions completed within a period of time over the time period. In a typical on-line transaction processing environment, which the TPC-C benchmark attempts to emulate, the DBMS server and client are connected through a network, the transaction mix is read-intensive, and the total footprint size is large so that most accesses require disk I/O access. The results in the upper left corner of Figure 4 correspond to this scenario, and show that the transaction dependency tracking overhead in this case is between 6% to 13% for all three DBMSs.

There is no clear trend as to whether the throughput penalty of transaction dependency tracking is higher or lower when comparing the networked configuration with the local configuration. There are two factors at work here. On the one hand, running the DBMS server and client on the same machine, i.e., local configuration, means that the DBMS server has access to less CPU resource and thus lower base-case performance. As a result, the percentage overhead should be lower in the networked configuration than in the local configuration. On the other hand, running the DBMS server and client on separate machines means that the average transaction latency is increased due to network transfer delay. This increase in latency does not in itself decreases the throughput as long as the DBMS client can always keep sufficient transactions outstanding in the pipeline. When this is not the case, the base-case throughput suffers and the throughput penalty of transaction dependency tracking could be increased.

Decreasing the total footprint size and thus increasing the database cache hit ratio significantly increases the throughput penalty of transaction dependency tracking for the read/write intensive load, but matters very little for the read intensive load. The reason is that when the footprint is small, the only disk I/O required is writes to the transaction log, and each transaction log write becomes more expensive when the transaction dependency tracking mechanism is turned on.

5.3. Accuracy of Database Damage Repair

One way to minimize the number of legitimate transactions that are incorrectly flagged as corruptive is to allow the DBA to specify transaction dependencies that should be ignored in the determination of the final undo set. Because fewer dependencies are considered, fewer transactions are judged corruptive and put into the undo set. We call dependencies that can be safely ignored a *false dependency*.

One example of a false dependency is when a dependency is based on an attribute of a table that can be computed from other data in the database. For instance, the `warehouse` table in the TPC-C benchmark's test database contains a `w_ytd` attribute, which is the total sum of money spent by all clients on a warehouse. This value could have been computed by using information from the `orders` table by summing up all orders that aim at a particular warehouse.

Let us use T_{detect} to refer to the interval between when an intrusion/error takes place and when it is detected. Figure 5 shows how the number of corruptive (those that need to be undone) transactions and percentage of saved transactions (those that survive repair) correlate with T_{detect}, under different warehouse factor (W) values, where T_{detect} is expressed in terms of the number of transactions committed since the intrusion/error. As expected, the number of transactions that are affected by the initial attack/error transaction increases with T_{detect}, but the percentage of saved transactions remain flat except when T_{detect} is small. Moreover, ignoring false dependency can significantly increase the number of benign transactions that can be saved from a repair process. The difference in the number of transactions that need to be undone or rolled back can be more than a factor of 5, and the improvement in the percentage of saved transactions ranges from 20% to 30%. The saved transaction percentage improvement decreases with W because larger W tends to have less false sharing and thus reduce the benefit of eliminating false dependency. This result suggests that it is crucial for an intrusion resilience engine to incorporate site-specific domain knowledge from the DBA and improve its repair accuracy by minimizing false positives and negatives.

6. Conclusion and Future Work

The most important contribution of this work is the development of a reusable implementation framework that adds intrusion resilience to existing DBMSs without re-

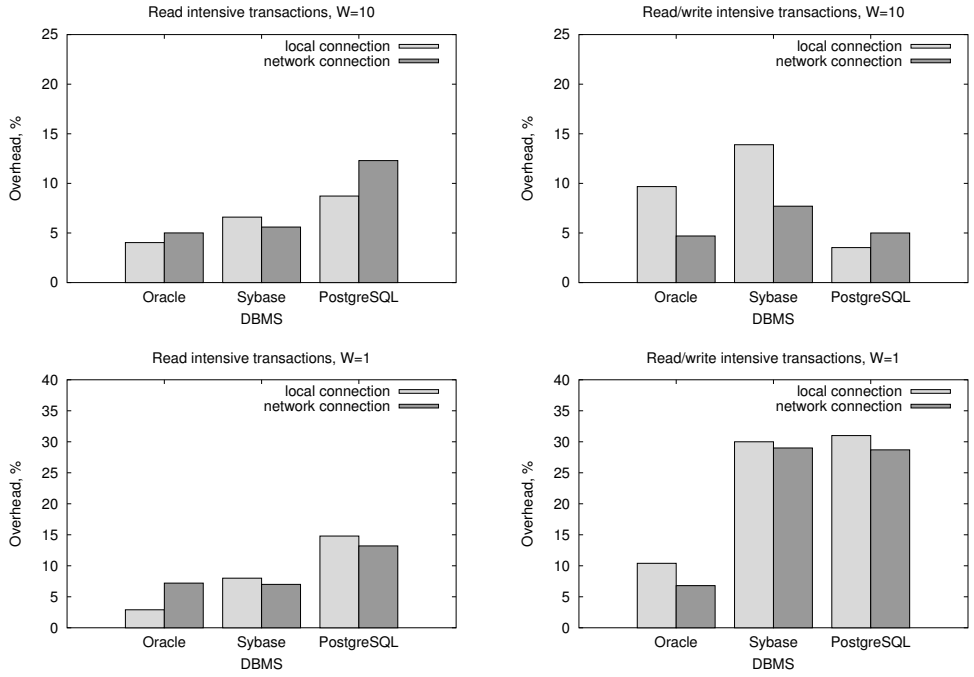

Figure 4. ***The inter-transaction dependency tracking overhead differs depending on whether the workload is read-intensive (left column) or read/write intensive (right column). It is also affected by the total footprint size, which is determined by the warehouse factor*** W***. The upper row corresponds to the large footprint case (*** $W = 10$ ***and low database cache hit ratio) and the lower row corresponds to the small footprint case (*** $W = 1$ ***and high database cache hit ratio).***

quiring any modifications to their internals, and a demonstration of this framework's portability to three different DBMSs, Oracle, Sybase, and PostgreSQL. Moreover, we also show the performance overhead of this portable approach to intrusion-resilient DBMS is quite reasonable, between 6% to 13% for a typical on-line transaction processing environment. We believe this framework is one of the first, if not the first portable intrusion resilience implementation framework that is also efficient and fully operational.

There are several directions we are pursuing currently. First, we plan to build a full-scale interactive database damage repair tool that allows a DBA to interact with the transaction dependency graph through a GUI, and explore the damage perimeter by conducting "what if" analysis. This tool will make the database damage repair process more flexible, accurate and convenient to use. Second, we are planning to build a transaction dependency tracking appliance that can be put in front of a DBMS server, and performs SQL query interception and re-writing without any additional configuration. Such an appliance minimizes the disruption to existing IT infrastructure, and thus offers a smoother migration path. Third, the current query re-writing algorithms can be further optimized to reduce the tracking overhead. For example, a single `trans_dep` table may become a bottleneck when the DBMS server runs on a multiprocessor machine; the `tr_id` attribute probably should be put in the middle of each row, rather than at the left or right end, to minimize the additional logging penalty; keeping a `tr_id` attribute per attribute rather than per row is required to minimize false sharing and to support suppression of false dependency, and how to implement it efficiently deserves more investigation. Finally, the current prototype does not support intrusion detection. We plan to develop a DBMS-specific intrusion detection tool and integrate it with the proposed intrusion resilience mechanism to form an end-to-end database security solution.

Acknowledgement

This research is supported by NSF awards ACI-0083497, CCF-0342556, ACI-0234281, SCI-0401777, USENIX student research grants, as well as fundings from Reuters Information Technology Inc., Computer Associates Inc., National Institute of Standards and Technologies, Siemens, and Rether Networks Inc.

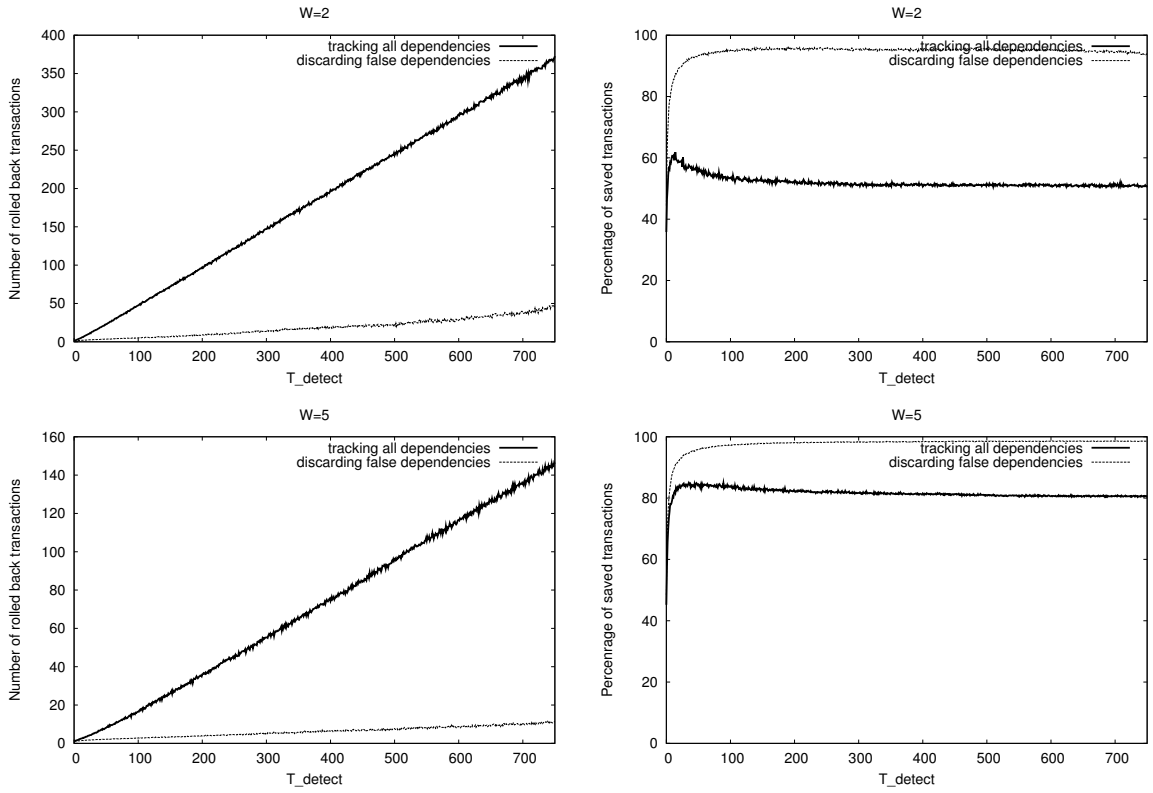

Figure 5. ***Number of transactions that need to be rolled back during the repair process (left column) and percentage of saved benign transactions (right column) versus*** T_{detect} ***under different*** W ***values.***

References

[1] Dhruv Pilania, Tzi-cker Chiueh, "Design, Implementation, and Evaluation of an Intrusion-Resilient Database System", *Technical Report TR-123*, ECSL, Department of Computer Science, SUNY at Stony Brook.

[2] Ningning Zhu, Tzi-cker Chiueh, "Design, Implementation, and Evaluation of Repairable File Service"*,* in *Proc. of DSN 2003*, June 2003.

[3] Jay Wylie, Michael Bigrigg, John Strunk, Gregory Ganger, Han Kiliccote, Pradeep Khosla, "Survivable Information Storage Systems", *IEEE Computer, 2(1)*, August 2000, pp. 61–68.

[4] J. Strunk, G. Goodson, M. Scheinholtz, Craig Soules, Gregory Ganger, "Self-Securing Storage: Protecting Data in Compromised Systems"*,* in *Proc. of OSDI 2000*, October 2000.

[5] Tripwire Open Source Project*, http://www.tripwire.org.*

[6] P. Ammann, S. Jajodia, P. Liu, "Recovery from Malicious Transactions"*, IEEE Transactions on Knowledge and Data Engineering,* Vol. 15, No. 5, 2002, pp. 1167–1185.

[7] P. Liu, "Architectures for Intrusion Tolerant Database Systems"*,* in *Proc. of 18th Annual Computer Security Applications Conference,* December 2002, pp. 311-320.

[8] P. Liu*,* "ITDB: An Attack Self-Healing Database System Prototype"*,* in *Proc. of DISCEX 2003*, Volume 2, April 2003, pp. 131–133.

[9] M. Kifer, P. Lewis, A. Bernstein*, Database and Transaction Processing: An Application Oriented Approach,* Addison-Wesley, 2002.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, 1987.

[11] Tzi-cker Chiueh, Ningning Zhu, and Dhruv Pilania, "How to Repair a Compromised Information System Quickly", *Technical Report TR-135*, ECSL, Department of Computer Science, SUNY at Stony Brook, January 2003.

[12] TPC-C benchmark, *http://www.tpc.org/tpcc/default.asp.*

[13] GraphViz — an open source graph drawing software, *http://www.research.att.com/sw/tools/graphviz/.*

[14] Oracle9i LogMiner *http://otn.oracle.com/products/oracle9i/daily/oct25.html.*

[15] PostgreSQL *http://www.postgresql.org.*

[16] Sybase Adaptive Server Enterprise *http://www.sybase.com/products/databaseservers/ase.*