Towards Securing Databases on Mobile Devices: A Log-based Approach

Subhasish Mazumdar, Anand Paturi
Computer Science and Engineering Department
New Mexico Institute of Mining and Technology
801 Leroy Place, Socorro, NM 87801, USA

Abstract

Mobile devices like smart phones empower their users to download, install, and execute a spectrum of attractive applications from both service providers and third parties. Consequently, users are increasingly relying on such devices not only for storing personal data but also for enabling purchases, managing finances, tracking diets, transacting business, and supporting social interactions. But the software systems on these devices are less robust than their laptop and desktop counterparts. These two factors can only entice attackers to manipulate the users' personal data via malicious code or malware that exploits the weaknesses of these mobile systems. Furthermore, mobile devices are often stolen or misplaced and the stored data, which users value more than the hardware, compromised. Since an application typically uses its own database on the mobile device to store pertinent user data, our aim is to enhance the security of those databases. We propose a consolidated logging scheme that reflects all transactions performed on all those databases. Only a part of this log needs to be stored on the mobile device, thus addressing the conflict between space limitation in these devices and the space requirement of logging. Our scheme is tamper-resistant; by modeling adversaries with various degrees of strength, we formally test our claim of tamper resistance against them. We explain how our scheme addresses the problems created by malware and theft and helps restore lost data.

1. Introduction

Modern smart phone operating systems like iOS, Blackberry OS and Android OS have introduced a new feature called the *App Store*, using which mobile device users can browse, download, and install third party applications. Most of these third-party applications make use of a lightweight database management system to create databases for storage, retrieval, and modification of application-specific information; for example, applications can provide an address book, manage a calendar, or the user's financial profile. Currently SQLite is being used as backend database management system for all mobile applications in the operating systems mentioned above. For some applications, such a database would be part of the mobile device and never require synchronization with the server; for others, such a database would frequently be uploaded on a stationary database server allowing the user to execute most operations even when disconnected from the server [2]. In this paper, we envision the SQLite database framework to be modified to incorporate a logging feature that can be used for forensic analysis.

Since every business is trying to attract its customers using the mobile environment and given the growing popularity and user-friendliness of devices like iPhone, Blackberry and Android-based phones, it is easy to predict that mobile applications will become the next hot target for attackers. Consider a mobile application that is used to download and store coupons or coupon codes or discount offers for shops in a locality [3]. Some of those coupons may even come from other users living in those areas. The application that organizes those coupons would use a mobile database. But, if a coupon contains malicious code, then the result could be unapproved accesses and alterations to the user's databases.

Primitive steps towards mobile device forensics have been taken in the form of tools for forensic analysis of PDAs (Personal Digital Assistants) [1]. But they do not apply to architectures of current mobile devices like iPhone, which runs a mini version of Mac OS X, or gphone, which has Linux-based Android. Moreover, a mobile database may be a stand-alone local database or a database embedded in a mobile application. This is a major difference: while in desktop database forensics, we examine a stand-alone database, in mobile database forensics, we need to look at a database that is part of

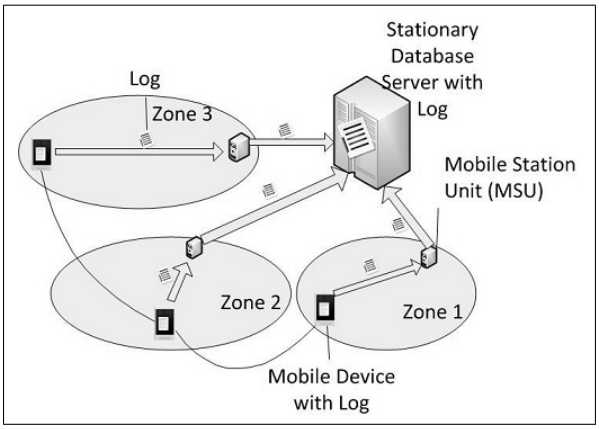

Figure 1. Log Transfer Scheme

an application, (one which may not need to contact a server at all).

The data in a mobile device is often more valuable to its owner than the hardware. If the mobile phone of a user that contains lots of mobile applications is lost and later found, the owner may be keen on learning which operations were performed by the thief using the owner's credentials. For example, if the device had a database of his clients (financial or health-related), the owner might want to know if anything was changed or deleted or just read.

Logging is a well-known technique for database recovery. But the problem is that as database management systems on mobile devices are not very robust, malware can also easily attack those logs. Also, logs require a great deal of storage. In this paper, we propose a tamper-resistant logging mechanism that is aware of these realities and can be used to detect tampering on such databases.

Our paper is structured as follows. In the next section, we outline our approach enumerating the modest enhancements we require and the structure of the log that is the basis of our tamper-resistant logging method; in the subsequent section, we focus on detection of tampering by formalizing our scheme, offering a theorem that expresses its tamper-detection properties, and showing that the entire log need not be stored at the device, thus expanding on our earlier work [4]. Next, we discuss some applications of our approach. After surveying related work, we offer concluding remarks.

2. Research Rationale

We envision mobile systems offering a lightweight database management system (DBMS) like sqlite shared by all the applications executing on the machine. A separate database D_j is created for each application A_j. We will refer to the tuple containing all these databases as D, *the database*.

We make two architectural requirements on that DBMS. First, it must support a *logwriter* process, which runs in the background. This process is responsible for recording in a log file all successful database operations along with precise timestamps. The major difference with conventional logging is that even read or select operations are logged. Second, we require that the DBMS will block all operations on all the databases in D if the *logwriter* process is either killed or stopped (owing to a software crash or malicious action); it will continue only after a restart.

Since a mobile device has space constraints, the log created by the *logwriter* is not stored in its entirety on such a device: its contents are periodically transferred to the stationary database server (SDS) via a mobile station unit (MSU) serving the cell or zone in which the mobile device is located (see Figure 1). Since the entire log is not stored on the mobile device, only a part, we call this *partial logging*. At any given time, the complete log is a combination of the log on the SDS and that on the mobile device.

We also assume system support for a unique secret value ω. It could perhaps be obtained by hashing the concatenation of the mobile IMEI (International Mobile Equipment Identity) number, certain geographical attributes during initial set up, and the owner's customer identification number. This secret value is read-only, i.e., no process can modify it once it has been created; and the only process that can access it is the *logwriter*. The secret value ω is stored in an encrypted form using a master key that could be generated in the following way. The user is required to enter a password during the initial set up of the device. This password is combined with his/her customer *id* to create a key for AES (Advanced Encryption Standard). This key is subsequently used for all log-related data encryption and decryption but need never be stored in non-volatile memory. (When and how often the system asks the user to reenter this password is an implementation detail we ignore here.) Calculating ω is almost impossible for an attacker because of several factors: the difficulty of guessing those exact geographical attributes; of accessing the stored value; and in the unlikely event of somehow obtaining that value, of decrypting it owing to ignorance of the user password, and consequently, the master key.

We compute dbh, a hash of all the databases along with ω. The hash dbh_i of database D_i is obtained by hashing each table in D_i using a secure hash algorithm and then hashing their concatenation; dbh is obtained

n	$\overline{t}$							h
Index	*Type &ID*	*App Name*	*Data Item*	*Oper-ation*	*DateTimestamp*	*Old Value*	*New Value*	*Hash*
1	c_0	h_0						dbh_0
2	T_1	jhk456g4	R_1	Create	12/18/2008,10:55:34:56	Null	Null	h_1
3	T_2	edf4545k	R_2	Insert	12/18/2008,12:34:56:32	Null	Fghghfghdfy 67456546546 5gbgf5676575	h_2
4	T_3	jhk456g4	R_4	Delete	12/18/2008,17:09:45:45	Fdf8d7fd8f hdkfdkjfd8 fdfjkdfdfkl	Null	h_3
5	c_1	h_3						dbh_1
6	T_4	56hght56	R_2, R_4	Update	12/18/2008,18:34:43:56	454jh4cx3f d3x4i4ou54 ok54b5c5	3FHGHgfgfb jdnbm,dsfh suiyjkkdj	h_4
7	T_5	5gsdgff6	R_2	Select	12/18/2008,18:34:47:06	Null	Null	h_5

Figure 2. Log before the first transfer to the server

by hashing the concatenation of all dbh_i and ω. Such a database hash aids in detecting surreptitious changes to the database without any corresponding record in the log.

We make use of checkpointing. Checkpoints are written to the log at periodic intervals, once N_{tr} transaction log entries have been recorded. A database hash is always calculated for committed data at that time. One possibility for optimization during checkpointing arises when most of the databases remain unchanged: only the changed databases need to be hashed. Checkpoints also help delineate a part of the log that gets transferred to the server. Periodically, a sequence of entries from the beginning up to the entry prior to the last checkpoint record is marked, transferred to the server, and deleted on the mobile device. In order to facilitate this process when the first record is the only checkpoint record in the log and neither checkpointing nor transfers have taken place in a while, dummy (no-change) transaction records may be added to the log so as to trigger immediate checkpointing.

2.1. Log Structure

There are two kinds of log entries: *checkpoint records* and *transaction records* arranged with one of the former followed by N_{tr} of the latter, e.g.,

$$< chk_0 >, < e_1 >, < e_2 >, \cdots, < e_{N_{tr}} >, < chk_1 >, < e_{N_{tr}+1} >, \cdots, < e_{2N_{tr}} >, < chk_2 >, \cdots$$

where $< chk_0 >, < chk_1 >, \cdots$ are checkpoint records while $< e_1 >, < e_2 >, \cdots$ are transaction records reflecting queries or updates of committed transactions. Figure 2 is an example of such a log with $N_{tr} = 3$. Each entry has a sequentially increasing unique integer index number $n \geq 1$ and a hash value h. We assume a function *last_index()* that returns the last used value of the index number, zero prior to the first use. The value would be read off a counter that is stored encrypted using the master key mentioned earlier. The *logwriter* process would typically make use of a procedure to increment that counter and assign the new value to the index field of a log entry atomically.

The m^{th} checkpoint record is of the form

$$< i, c_m, string, dbh_m >,$$

where i is the index number, *string* is obtained from the previous log entry by copying its hash value h_j, as explained below; if $m = 0$, then *string* is a random value h_0 shared between the server and the mobile device; dbh_m is a database hash computed at checkpoint time: it is a hash of all the databases along with the secret value ω.

The j^{th} transaction record has the form

$$< i, Type\&ID, AppName, DataItem, Operation, DateTimestamp, OldValue, NewValue, h_j >.$$

Type&ID is the identifier of the transaction; *AppName* is the name of the responsible mobile application; *DataItem* is the name of the accessed data item; *Operation* is the action applied on the data item; *DateTimestamp* is the date and time of the operation; *OldValue* and *NewValue* contain old and new values (if any, *Null* otherwise) resulting from the operation. We will refer to these components as a vector of transaction parameters $\overline{t_j}$; and thus denote the same record equivalently as

$$< i, \overline{t_j}, h_j >.$$

The hash component h_j is a hash value obtained by applying a secure hash function f_{sha} (e.g., a higher member of a SHA-n family) on the vector of transaction parameters, the secret value ω mentioned earlier, and the hash value h_{j-1} of the previous log entry. If the previous entry is a checkpoint record, h_{j-1} is substituted by the string component of that record. More precisely, if the log entry

- does not immediately succeed a checkpoint record, then

  $h_j = f_{sha}(\overline{t_j} + \omega + h_{j-1})$, where $+$ denotes concatenation (the vector components are concatenated).

- otherwise, i.e., if it immediately succeeds a checkpoint record $< i, c_k, str, dbh_k >$ containing string str, then

  $h_j = f_{sha}(\overline{t_j} + \omega + str)$.

Figure 2 is an example of such a log. In that example,
$h_1 = f_{sha}(T_1 + jhk456g4 + R_1 + Create + 12/18/2008, 10:55:34:56 + Null + Null + \omega + h_0)$;
$h_2 = f_{sha}(T_2 + edf4545k + R_2 + Insert + + 12/18/2008, 12:34:56:32 + Null + Fghghfghdfy674565465465gbgf5676575 + \omega + h_1)$.

2.2. Encryption of columns

We have omitted one detail for ease of exposition. The following components of the log entries appear in an encrypted form using the same master key mentioned earlier.

- Index values (n); attackers find it harder to fake the log or parts of it. (In order to enhance security by thwarting an examination of a sequence of numbers, we could increment by a constant $c \neq 1$, or use a more sophisticated number pattern. In this paper, we assume an increment-by-one policy.) In order to help explain the examples, we do not show the index entry encrypted in Figure 2.

- Application names; old and new values: hiding them enhances the privacy of user transactions.

2.3. Log Transfer

Upon initialization, through communication with the server, the $< chk_0 >$ checkpoint record is created and stored in the mobile device log. Its string value h_0 is a random value shared with the server. We refer to this as the *zero*th transfer of the log. Until the next transfer, the server will respond to a request for the index number and hash value of the last record transferred to it with zero and h_0 respectively.

Once the log on the mobile device reaches a certain size, at the next available opportunity to connect to the server, log transfer takes place, i.e., the sequence of entries in the log on the mobile device from the first entry (which must be a checkpoint) to the one just prior to the last checkpoint are marked, transferred to the server, and deleted from the mobile device. That last checkpoint entry now becomes the first entry in the new log on the device.

For example, suppose the log is in the state shown in Figure 2 before the very first transfer. Figures 3 and 4 show the resulting log on the device and the server respectively after the transfer. All entries up to the record just prior to checkpoint c_1, i.e., records with index num-

n	$\overline{t}$		h
5	c_1	h_3	dbh_1
6	$\overline{t_4}$		h_4
7	$\overline{t_5}$		h_5

Figure 3. Log on the mobile device after the first transfer

n	$\overline{t}$		h
1	c_0	h_0	dbh_0
2	$\overline{t_1}$		h_1
3	$\overline{t_2}$		h_2
4	$\overline{t_3}$		h_3

Figure 4. Log on the server after the first transfer

n	$\overline{t}$		h
121	c_{30}	h_{90}	dbh_{30}
122	$\overline{t_{91}}$		h_{91}
123	$\overline{t_{92}}$		h_{92}
124	$\overline{t_{93}}$		h_{93}
125	c_{31}	h_{93}	dbh_{31}
126	$\overline{t_{94}}$		h_{94}

Figure 5. A general snapshot of the log.

$$
\begin{aligned}
pairOK(1,\lambda) &= C(\lambda_1) \wedge (\lambda_1.s = LDBH()) \wedge (\lambda_1.n = 1 + LNDX()) \\
&\quad \wedge \ (undo\_hash(2, length(\lambda)) = \lambda_1.h) \\
pairOK(i,\lambda) &= [\lambda_i.n = 1 + \lambda_{i-1}.n] \wedge \\
&\quad [(C(\lambda_i) \wedge T(\lambda_{i-1}) \wedge (\lambda_i.s = \lambda_{i-1}.h) \wedge (undo\_hash(i+1,k) = \lambda_i.h)) \quad \vee \\
&\quad (T(\lambda_i) \wedge C(\lambda_{i-1}) \wedge (f_{sha}(\lambda_i.\overline{t}, \omega, \lambda_{i-1}.s) = \lambda_i.h)) \quad \vee \\
&\quad (T(\lambda_i) \wedge T(\lambda_{i-1}) \wedge (f_{sha}(\lambda_i.\overline{t}, \omega, \lambda_{i-1}.h) = \lambda_i.h))], \ \text{where} \ \ i > 1 \\
logOK(\lambda) &= (\lambda_k.n = last\_index()) \wedge \ \ (num\_checkpts(\lambda) = 1 + \lfloor \frac{k-1}{1+N_{tr}} \rfloor) \\
&\quad \wedge \ \ (num\_transrecs(\lambda) + num\_checkpts(\lambda) = last\_index() - LNDX() + 1) \\
&\quad \wedge \ \ (\bigwedge_{i=1}^{i=k} pairOK(i,\lambda)), \ \ \text{where} \ \ k = \text{length}(\lambda)
\end{aligned}
$$

Figure 6. Definition: predicates *pairOK* and *logOK*.

bers 1 through 4, are now at the server and all of them are deleted from the log on the device, leaving checkpoint c_1 as the first entry and its only checkpoint entry. Note that the string component of the c_1 checkpoint record is equal to the hash value of the previous log record and that record has been stored at the server. After several transfers, the log may look like Figure 5.

3. Detection of tampering

Let a log fragment λ of length k be a sequence of k records:

$$\lambda =< \lambda_1, \cdots, \lambda_k > \ \text{where} \ \textit{length}(\lambda) = k$$

Define predicates $C(\lambda_j)$ and $T(\lambda_j)$ recognizing λ_j as a checkpoint and transaction record respectively. Let functions *num_checkpoints*(λ) and *num_transrecs*(λ) return the number of checkpoint records and transaction records in the given log fragment. We use the dot notation to denote extraction of components from log entries, e.g., $\lambda_i.n$, $\lambda_i.h$, $\lambda_i.\overline{t}$, and $\lambda_i.s$, extract the index number, hash value, transaction parameters, and the string component respectively from a log entry λ_i. Nullary functions LDBH and LNDX represent requests to the server to return the hash and index value respectively from the last log entry transferred to it. The function *undo_hash*(p, q) computes a database hash after undoing the updates in the transaction records found in the log entries $\lambda_p, \lambda_{p+1}, \cdots, \lambda_q$ in reverse order. (Since the entire log may not be available within the mobile device, destructive data definition operators like `drop table` may have to be accompanied by the associated schema in order to create the table and perform the updates being undone.)

Now we can define two predicates *pairOK* and *logOK* whose second parameter λ is a fragment of the device log under examination. Let us walk through their definitions (given in Figure 6). The first equation defines *pairOK* when its first input parameter is equal to 1 corresponding to the first entry in the log fragment; the predicate checks that the first entry is a checkpoint record, that its string component and index number respectively equals and sequentially follows the hash value and the the index number of the last log entry recorded at the server, and that its hash component equals the database hash at that computational state. The second equation considers the same predicate when its first input parameter $i > 1$ corresponding to entries after the first one in the log fragment; it considers the i^{th} and the $(i-1)^{th}$ log entries and checks a pairwise integrity constraint imposed by the log structure. In the first line of this equation, it checks that the index numbers are in sequence; in the second line, that if the i^{th} is a checkpoint record preceded by a transaction record, then the i^{th} record's string component equals the hash value of the $(i-1)^{th}$ and that the database hash component of the i^{th} record equals the database hash at that computational state; in the third line, that if the i^{th} is a transaction record preceded by a checkpoint record, then the hash value of the i^{th} is correctly obtained from its own parameters and the string component of the $(i-1)^{th}$ (which should be equal to its preceding transaction record's hash value); in the fourth line, that if the i^{th} and the $(i-1)^{th}$ are both transaction records, then the hash value of the i^{th} is correctly obtained from its own parameters and the hash value of the $(i-1)^{th}$.

The second predicate, *logOK*, checks that the last

sequence number is correct, the number of checkpoints and transaction records are both correct, and that each pair of successive log entries satisfies *pairOK*.

Observation: In order to evaluate *logOK* on the mobile device, only two values are required from the server: the results of LNDX and LDBH.
Proof: By inspection of the predicate *logOK* and *pairOK*.
The data items required are D and λ, both available on the device. The component extraction functions indicated by the dot notation can be evaluated given a log record; the functions *num_checkpts* and *num_transrecs* can be evaluated given the log fragment; *last_index* can also be computed by looking up a counter and decrypting.

The function *undo_hash* checks that the hash of the database state in the last checkpoint c_m equals the hash computed from the current database by undoing the updates recorded in transaction records following c_m; if there is more than one checkpoint, their corresponding database hashes are similarly checked. Of course, the database is restored to its original state by *redo*ing the *undo*s.

The only two remaining functions LNDX and LDBH require values from the server.
End of proof.

Observation: If logging is error-free, then *logOK* evaluates to true.
Proof: This follows trivially from the close tie between the definition of *logOK* and the structure of the log.
End of proof.

Now we consider the more interesting question, can tampering be detected by *logOK* evaluating to false? First, we must be clear about the adversary $\mathcal{A}$, one who can change the databases, the log files, or both. Formally, we assume that $\mathcal{A}$ knows

1. the log structure and how to extract the components of a given log entry;
2. how to access the log file on the mobile device; and
3. how to access the database files;

but does not know the

1. master key used for encryption, since it is created using ingredients that are hard to guess plus an owner generated password we presume is protected by the user;
2. value of ω: it is a combination of multiple factors described above and unique for every mobile device making it hard to guess; accessible only to the *logwriter* process; and stored encrypted using the master key; and
3. value of *last_index()*, the last index number used, as it is stored encrypted. This is the result of the ignorance of the masterkey (item 1); we are making it explicit for clarity.

Second, we define *tampering* to be the result of an adversary bypassing the DBMS to perform any or all of (a) insertion / deletion / modification of one or more log records, or (b) modification of a database file D_j. (A sequence of deletes and inserts that leave the log or database files in the same state is not considered to be tampering.) Note that we implicitly assume that the adversary is neither able to alter the operating system nor the database management system.

Theorem: If *logOK* is true at each of the first m ($m \geq 0$) log transfers, then tampering by adversary $\mathcal{A}$ between the m^{th} and the $(m+1)^{th}$ log transfers will result in *logOK*=false with high probability (w.h.p.) at or before the $(m+1)^{th}$ log transfer.
Proof: We consider separate methods of tampering.

update of log record

- If $\mathcal{A}$ alters a transaction record in the log, owing to ignorance of ω, the hash value will not be correct.

  Note that the previous transaction record's hash value is used in the computation of the current record even if the previous record was a checkpoint (its string value equals the previous transaction record's hash value) or if it has already been transferred to the server (the function LDBH fetches it). This chaining thwarts replay attacks because $\mathcal{A}$ cannot copy an older transaction record corresponding to a known transaction update. Further, this incorrect hash value will make the hash values of all subsequent entries incorrect.

- When altering a checkpoint record, $\mathcal{A}$ can either alter the index number or the database hash. The former is easily detected because it is encrypted and part of a sequence. The latter will lead to a database hash value that will not match the hash computed from the database D; even if $\mathcal{A}$ bypasses the DBMS and alters D, ignorance of ω will leave $\mathcal{A}$ unable to construct the right database hash value.

insertion of log record

- If $\mathcal{A}$ inserts a non-terminal transaction record in the log, there will be two errors. First, the hash value will be incorrect owing to an ignorance of the ω value. Replay attacks are avoided since the previous entry's hash value is also used in the computation.

  Second, since $\mathcal{A}$ does not know how to decrypt and encrypt the correct index value, $\mathcal{A}$ will have to copy it off the next entry; this will result in two entries with the same index number.

- If $\mathcal{A}$ inserts a terminal transaction record, thus increasing the length of the log by one, the above two errors will result, except that the index number of the inserted record will be wrong (w.h.p.) since there is no next entry to copy from.

- If $\mathcal{A}$ inserts more than one transaction record, the same two errors will result.

- If $\mathcal{A}$ inserts a checkpoint record, the number of checkpoints as computed by the *num_checkpoints* function will be incorrect.

  Also, there will be an index number error just like the case of insertion of a transaction record.

  Further, the database hash will be incorrect (w.h.p.) owing to the ignorance of ω.

- If $\mathcal{A}$ inserts more than one checkpoint record, the number-of-checkpoints error will remain. While that error will be avoided if $\mathcal{A}$ inserts a checkpoint record in place of an existing checkpoint record, this would be equivalent to an update of a checkpoint record, a case we have covered already.

deletion of log record

- If $\mathcal{A}$ deletes a non-terminal transaction record from the log (perhaps to hide a transaction), the discrepancy in the sequence of index numbers will detect it.

  Further, the hash value of the next record will not be correct when chained with its previous record (the one before the deleted record) and hence *pairOK* will return false.

- If more than one non-terminal transaction records are deleted, the two above errors will result.

- If $\mathcal{A}$ deletes the last entry of the log, the comparison of the *last_index* function and the index of the last log record will detect it.

- If more than one transaction record including the terminal one is deleted, the above error will result.

- If $\mathcal{A}$ deletes a checkpoint record, then the number of checkpoints will not match the expected number. If it is not the terminal record, then there will also be a gap in the index numbers.

- The above holds if more than one checkpoint record is deleted.

insertion/deletion/update of multiple log records

- Clearly inserting, deleting, modifying multiple records is not beneficial owing to the fact that subsequent records will uncover the error either through index numbers or the hash value. So, the remaining strategy is to wipe out all subsequent records.

  Thus $\mathcal{A}$ can delete the entire log leaving only one checkpoint record (without that record, the *logwriter* will stop) containing $\mathcal{A}$'s own database hash. But, owing to ignorance of ω, that hash will not equal that expected during computation of *pairOK*. Adding more records after that checkpoint record will also be detected as shown above.

modification of the database

- If $\mathcal{A}$ alters the database file bypassing the log, then the database hash computed on the altered database will not equal those recorded with the checkpoint records (at least one checkpoint record and database hash is in the log at all times).

  After altering the database file(s), $\mathcal{A}$ could insert/replace a checkpoint entry containing the hash of the altered database. But, owing to ignorance of ω, $\mathcal{A}$ cannot compute the correct database hash.

End of proof.

3.1. Sensitivity to stronger adversaries

Now, we would like to explore the sensitivity of our proposed scheme to stronger adversaries, i.e., more knowledgeable ones than $\mathcal{A}$.

Adversary knows the user password.
It is not unusual to find users choosing passwords that are easy to guess or after choosing well, failing to guard

it adequately. Let us model an adversary $\mathcal{A}'$, who, with the knowledge of the user password, effectively knows the master key but still does not know the value of ω. To recap, ω is not available to processes other than *log-writer*. Formally, we assume that $\mathcal{A}'$ knows

1. the log structure and how to extract the components of a given log entry;
2. how to access the log file on the mobile device;
3. how to access the database files;
4. the master key used for encryption; and
5. the value of *last_index()*, the last index number used, through decryption using the master key;

but does not know the

1. value of ω, a value accessible only to the *logwriter* process.

Corollary: The theorem holds for tampering by adversary $\mathcal{A}'$.
Proof: Scanning the various update situations in the proof of the theorem, we find that without the knowledge of ω, the adversary fails in all update situations except in the one involving deletion of the last log entry.

But here too, while $\mathcal{A}'$ will be able to decrypt the index numbers, he will not be able to change the fact that the function *last_index* will return the index value of the deleted record and thus will not equal that of the current last record.

The only way that $\mathcal{A}'$ can compensate for this is by inserting a new last entry with that index value. But that entry can neither be a checkpoint record nor a transaction record as both will be detected owing to the ignorance of ω.
End of proof.

Adversary knows ω.
Now, consider the case of $\mathcal{A}''$ who knows the secret value ω but does not know the master key. Formally, we assume that $\mathcal{A}''$ knows

1. the log structure and how to extract the components of a given log entry;
2. how to access the log file on the mobile device;
3. how to access the database files; and
4. the value of ω;

but does not know the

1. master key used for encryption and
2. value of *last_index()*, the last index number used, as it is stored encrypted.

Corollary: The theorem does not hold for tampering by adversary $\mathcal{A}''$.
Proof: $\mathcal{A}''$ can delete the entire log, retaining the very first checkpoint record but altering its database hash to reflect the database modified by him or her. Owing to knowledge of ω, the database hash $\mathcal{A}''$ computes will look correct when *undo_hash* is used to compute *logOK*.
End of proof.

From the above, we learn that it is neither the user password nor the master key that provides the critical knowledge to the adversary: it is the knowledge of ω that will allow an adversary to tamper and yet remain undetected, i.e., leave *logOK* = true.

4. Applications of our Scheme

Malware access to mobile device. A mobile user's database can be attacked using malicious code or malware. With the user unaware, such malware can make use of the user's databases via the database management system. However, with our logging scheme, all database operations would get recorded in the log with and possibly stored at the server, thus enabling future analysis and restoration of the databases to a state close to that before the infection.

But what if the malware bypasses the log file or changes the database and deletes relevant entries in the log file? As we have shown, such attempts can be detected by computing the logOK predicate.

On the other hand, if the malware deletes the entire log file, the database itself stops functioning since the write process on the database gets blocked owing to the absence of the log.

Theft and subsequent retrieval of mobile device
Mobile devices are susceptible to theft or misplacement. But when that is followed by subsequent retrieval or recapture, the owner may be in a quandary about whether or not data on the device was accessed by the thieves or handlers during the interval of loss.

In our scheme, all the data created, stored, and read are logged, i.e., the create, update, and select statements are logged. Normally, logging reads are expensive in terms of performance. But, in our case, concurrency is not an issue; the only problem is a space overhead resulting from log entries; but almost the entire log gets transferred to the server where the problem of space overhead is

manageable and further, read-only transactions can be purged after a certain number of days.

A useful optimization here is to include the last access time of a tuple with the tuple itself. By analyzing the log, analysts can deduce if and when certain data in the database was accessed by the thief.

Finally, when and how often the system asks the user to re-enter the password used to create the master key is an implementation detail we have ignored here. It can be used to narrow the time interval the thief has from the time of theft to access the database without being asked for the password.

Restoration using the log The traditional use of a logging mechanism is to restore a database to the last committed transaction in the event of a crash.

Only a part of the log is stored in the device; hence, we need to combine the part stored at the server with the part remaining on the mobile device to obtain a complete valid log and restore the database. The part on the server is guaranteed to be valid (i.e., *logOK* = true). Hence, if the fragment on the mobile device is valid, then the entire log can be used to restore the database. Since the log can get very large, a well-known optimization is to create a dump of the database at a certain checkpoint and delete the log prior to that checkpoint. This dump can be stored encrypted at the server. However, if this raises privacy concerns, the dump can reside with the user in his/her own backup storage.

If the mobile device log is invalid (i.e., *logOK* = false), then we need to find the earliest entry at which tampering is detected. The entry just prior to that one is the last valid entry and hence we can restore the database state at that computation state. The restoration can be hastened by taking advantage of any existing dump.

One way of going beyond the earliest entry at which tampering was detected is to employ file-system level forensics to retrieve the deleted log file or deleted database files.

While our logging scheme cannot deliver the same guarantee as a traditional database management system, this is a known problem in mobile databases; it arises from the absence of continuous reliable connectivity with a server.

5. Related Work

We are motivated by the work of Snodgrass et al. [8] who pointed out the need for tamper-proof logging mechanisms. They proposed an audit log in the form of a database table with transaction time support; records would never be deleted, start and stop columns would specify the status of a record. They suggest the use of notarization: at database creation the schema is notarized; for each transaction, the data of the transaction is notarized.

Pavlou and Snodgrass [7] show how forensic analysis can be conducted after intrusion on an audit log has been detected; the aim is to find the exact time at which corruption of the log occurred and time at which corrupted data was originally stored. They introduce a *corruption diagram* that provide a graphical representation of corruption event (CE)s in terms of a temporal-spatial dimensions of the database.

Our proposal of encrypting log items to preserve privacy is motivated in part by the work of Miklau et al. [5] who argue for enhancements to database systems that allow users to securely manage history in order to balance the need for privacy and accountability. Crucial in this regard are questions like how and when data is retained by the system and who will be able to recover and analyze it.

We propose rolling back a database for analysis; this concept was suggested by Olivier [6] who showed similarities and differences between file system forensics and database forensics and then derived principles for the latter.

6. Conclusion

To summarize, we have proposed a tamper-resistant logging mechanism for mobile applications and mobile databases. We described how the log is implemented using a linked hashing technique and how it can be used to detect tampering of the log. The novelty of our approach is that only a part of the entire log is stored and examined at a time. Surprisingly, a secret value is the key to the strength of the system, not the user password. This approach enables a trustworthy logging system on a modern mobile device in spite of its limited storage while retaining the power of the log as a reliable source for recovery and forensic analysis. Of course, we have assumed that the DBMS itself is intact and so the logwriter process works. If an adversary can install his own DBMS, that could be detected at the server when no log fragments are transferred in a reasonable amount of time. We are working on an implementation of this mechanism on an existing mobile operating system.

7. References

[1] R. Ayers and J. Wayne. PDA forensic tools: An overview and analysis. http://csrc.nist.gov/publications/nistir/nistir-7100-PDAForensics.pdf (1 August 2004).

[2] E. Giguere. Why develop mobile database applications? http://ezinearticles.com/?Why-Develop-Mobile- Database- Applications?&id=1913994 (23 January 2009).

[3] Y. Luo, O. Wolfson, and B. Xu. The role of auto-id technologies in mobile e- commerce databases (vision paper). In *IEEE 24th International Conference on Data Engineering Workshop, 2008. ICDEW 2008.*, pages 108–109, 2008.

[4] S. Mazumdar and A. Paturi. Tamper-resistant database logging on mobile devices. In *Proceedings of the World Congress on Internet Security (WorldCIS-2011),*, 2011.

[5] G. Miklau, B. Levine, and P. Stahlberg. Securing history: Privacy and accountability in database systems. In *Conference on Innovative Data Systems Research (CIDR)*, 2007.

[6] M. S. Olivier. On metadata context in database forensics. *Digital Investigation*, 5:115–123, March 2009.

[7] K. Pavlou and R. Snodgrass. Forensic analysis of database tampering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 109–120, 2006.

[8] R. Snodgrass, S. Yao, and C. Collberg. Tamper detection in audit logs. In *Proceedings of the International Conference on Very Large Data Bases*, volume 30, pages 504–515, 2004.

