Architectures for Self-Healing Databases under Cyber Attacks

Peng Liu† and Jiwu Jing††

†College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802 USA
††State Key Lab of Information Security, Chinese Academy of Sciences, Beijing, China

Summary
In this paper, we propose five architectures for self-healing databases under malicious attacks. While traditional secure database systems rely on prevention controls, a self-healing database system can autonomically estimate, locate, isolate, contain, and repair damage caused by attacks in such a way that the database can "heal" itself on-the-fly and continue delivering essential services in the face of attacks. With a focus on attacks by malicious transactions, Architecture I can detect intrusions, and locate and repair the damage caused by the intrusions. Architecture II enhances Architecture I with the ability to isolate attacks so that the database can be immunized from the damage caused by a lot of attacks. Architecture III enhances Architecture I with the ability to dynamically contain the damage in such a way that no damage will leak out during the attack recovery process. Architecture IV enhances Architectures II and III with the ability to adapt the self-healing controls to the changing environment so that a stabilized level of healthiness can be maintained. Architecture V enhances Architecture IV with the ability to deliver differential, quantitative QoIA services to customers.

Key words:
Self-Healing Databases, Cyber Security, Architectures

1. Introduction

As society increasingly relies on database systems to store, manage, and access information digitally (e.g., database products are today a multi-billion dollar industry; database systems motivated 32% of the hardware server volume in 1995 [44], and 39% of the server volume in 2000), maintaining the integrity, availability, and confidentiality of databases is crucial. Many large-scale database systems critical to businesses are expected to be available continuously and can only be stopped for repair at great cost. However, fraudulent transactions can contaminate such databases and necessitate repairs, and traditional *prevention-centric* database security is very limited in tackling this problem. A *self-healing database* system would guarantee that under sustained malicious transaction attacks, the database is continuously accessible; the contamination or "wound" (on the data) is autonomically located, contained or isolated, and healed, without stopping the system; and the database's healthiness is carefully maintained in such a way that self-healing will not prevent (most) essential services from being provided correctly. The ITDB (Intrusion Tolerant Data Base) framework, which we will present shortly, combines a family of new database survivability or *intrusion tolerance* techniques to build self healing databases.

1.1 Technologies for Self-Healing Database Systems

A *database* is a set of *data objects*. The database *state* at time t is determined by the values of these data objects at that time. A data object x is *contaminated*, *damaged* or *corrupted* if its value is changed to a wrong value due to an attack (or a mistake). At this situation, the data *integrity* of x is jeopardized or degraded. A database is damaged if some data objects are damaged. A damaged data object x is *repaired* or *healed* if its value is restored to a *correct* value.

Self-healing does not always require on-the-fly repair. Self-healing systems can choose to do self healing either *offline* or *online*, based on the application's requirements and the overall cost-effectiveness. When the application does not have strict real-time constraints and is insensitive to short period of down time, a more cost-effective self-healing could be to autonomically select the best time to shut down the service, do the repair, then resume the service. Nevertheless, since many large-scale database systems critical to businesses are expected to be available continuously and can only be stopped for repair at great cost (i.e., they basically cannot tolerate any down time), in this paper we focus on online self-healing technologies.

Besides the ability to autonomically locate and repair the damage on-the-fly, online self-healing requires the system to be able to maintain its own *healthiness* or at least curable *fitness*, because if a self-healing system could not ensure that it is always *curable*, in some cases it cannot heal itself. Intuitively, a database is *healthy* when most of its data objects are not damaged. Similar to a human body, sometimes a system has too bad "health" to heal up. *Curability* means that every damaged data object can be "ultimately" repaired.

Existing database security mechanisms are very limited in maintaining healthiness. In particular, authentication and access control cannot prevent all attacks; integrity constraints are weak at prohibiting plausible but incorrect data; concurrency control and recovery mechanisms cannot distinguish legitimate transactions from malicious ones; and automatic replication facilities and active database triggers can even serve to spread the damage.

Besides curability, online self-healing also requires *availability*, that is, the database should be always *useful* and the system should always be able to deliver correct services. It should be noticed that being useful and being *accessible* are different. A totally contaminated database (i.e., every data object is damaged) can still be 100% accessible (e.g., it can still process transactions smoothly), but the database is no longer useful at all.

Moreover, note that curability and availability are not identical. Being useful implies making the database accessible as well as maintaining a certain level of data integrity of the current database state. In contrast, even if the current database state is seriously contaminated, the database may still be curable if the latest clean version of every damaged data object is kept in the log files (e.g., the *redo log*, the *checkpoint* files, etc.) intact and can be precisely located. Hence, availability has typically stronger data integrity requirements on current data versions than curability. On the other hand, note that when the database audits are seriously contaminated, even if the database is quite useful, it may not be curable.

Finally, since better healthiness in general not only implies easier and quicker self-healing with lower cost, but also implies better usability, maintaining good fitness of the system in the presence of attacks should be a crucial aspect of self healing database system development.

1.1.1 A Multi-Layer Approach to Self-Healing Databases

Building an attack resistant or self-healing database requires in general a *multi-layer* approach, since attacks could come from any of the following layers: hardware, OS, DBMS, and transactions (or applications). A multi-layer approach can be developed along two directions: (a) from scratch or (b) using "off-the-shelf" components.

Along the from-scratch direction, tamper-resistant processor environments, and trusted OS or trusted
DBMS loaders have been applied to close the door on hardware attacks and OS bugs; programming security technologies such as certified programs (e.g., [36]), bug-guarding compilers (e.g., [9]), and bug finding (e.g., [11]) can be applied to close the door on many DBMS bugs; and signed checksums (and a small amount of tamper-resistant storage to keep the signing key) have been used to detect OS-level data corruption [30]. Note that when the transaction logs are securely maintained, the corresponding OS-level repair can be efficiently done either online or offline.

Based on "off-the-shelf" components, OS-level attacks have been addressed by several efforts. In
[5], (signed) checksums are smartly used to detect data corruption. In [33], a technique is proposed to detect *storage jamming*, malicious modification of data, using a set of special *detect objects* which are indistinguishable to the jammer from normal objects. Modification of detect objects indicates a storage jamming attack.

Although the above techniques may effectively handle DBMS, OS, and hardware level intrusions, they cannot handle authorized but malicious transactions. For example, neither trusted OS nor signed checksums can detect or repair the data corruption caused by a malicious transaction issued by an attacker assuming the identity of an authorized user. The goal of this paper is to explore the self-healing database architectures that can handle fraudulent transactions.

1.1.2 Intrusion Detection Technologies

One critical step towards self-healing databases under attacks is intrusion detection, which has attracted many researchers. The existing methods of intrusion detection can be roughly classed as *signature-based detection* (e.g., [16]), *anomaly detection* based on *profiles* (e.g., [21]), or *specification-based detection* (e.g., [42]). Intrusion detection can supplement protection of network and information systems by rejecting the future access of detected attackers and by providing useful hints on how to strengthen the defense. However, intrusion detection has an inherent limitation in doing self-healing: Intrusion detection makes the system attack-aware but not attack-resistant, that is, intrusion detection itself cannot maintain integrity and availability of the database in the face of attacks. As a result, although intrusion detection and checkpoints can be used together to heal the database after an attack is detected, such self-healing requires the database to roll back its state to a clean checkpoint before the attack happens, is very difficult to be processed online, and will make all the good work done after the attack invalid.

1.1.3 Fault Tolerance Technologies

When the causes for self-healing are faults and failures, the corresponding self-healing technologies belong to the field of fault tolerance. However, when the causes for self-healing are malicious attacks, the corresponding self-healing technologies can be quite different from fault tolerance technologies, due to several fundamental differences between fault tolerance and intrusion tolerance. To illustrate, first, in fault tolerance, failures randomly happen; but in security, attacks are typically intentional and do not randomly happen. Moreover, attacks are more active

than failures, so more proactive techniques are needed for intrusion tolerance. Second, intrusion detection is typically more challenging and complicated than failure detection. Third, in traditional fault tolerance under fail stop fault model, being accessible means correct/quality services (i.e., being useful), but this is not true in online self-healing under attacks.

Accordingly, traditional database recovery mechanisms are fairly limited in healing the database under attacks. They do not address this problem, except for complete rollbacks, which undo the work of benign transactions as well as malicious ones, and compensating transactions, whose utility depends on application semantics. More important, they cannot maintain database healthiness under attacks in such a way that both the curability and usability requirements can be satisfied.

Nevertheless, fault tolerance technologies build a solid foundation for developing intrusion tolerant systems. Some specific fault tolerance technologies, such as Byzantine fault tolerance [7], have been found crucial in developing several types of intrusion tolerant systems.

1.1.4 Intrusion Tolerance Technologies

From the healthiness maintenance perspective, self-healing systems are indeed an intrusion tolerant system. We can classify existing intrusion tolerance technologies into two categories: *intrusion masking* and *defense-in-depth.*

Intrusion Masking The goal of intrusion masking is to creatively use enough *redundancy* (and maybe data fragmentation and distribution) to ensure that the system can function correctly even if part of it is hacked. In this sense, we say such systems can *mask* intrusions. Techniques in this category focus on how to enhance the inherent resilience of the system, and their effectiveness is typically much less sensitive to the agility and accuracy of intrusion detection than pragmatic intrusion response techniques. General principles in developing intrusion masking systems include but are not limited to (a) redundancy & replication; (b) diversity; (c) randomization; (d) fragmentation& threshold cryptography; and (e) increased layers of indirections. Techniques in this category include but are not limited to Byzantine intrusion masking techniques (e.g., [31]) and threshold-cryptography-based survivable systems (e.g., [50]).

Nevertheless, although Byzantine fault tolerance and threshold-cryptography-based survivability can be very effective when a group of replicated processing or storage servers are infected by outside attacks, they are very limited in surviving malicious user activities such as fraudulent transactions. In particular, these two technologies cannot exploit redundancy to distinguish a malicious transaction submitted by an attacker assuming the identity of a trusted user from a legitimate transaction submitted by the trusted user, and as a result, the system must treat them similarly, and this intrusion cannot be modeled as Byzantine faults.

Defense-in-depth The goal of defense-in-depth technologies is to arm the system with a set of intrusion response facilities which, with the help of intrusion detection, can respond to intrusions in such a way that the system can operate through attacks. Technologies in this category include (a) boundary controllers such as firewalls and access control; (b) intrusion detection; and (c) intrusion response. Boundary controllers cannot prevent every attack. Intrusion detection is already discussed. Intrusion response technologies can be classified into three categories:

- *Reactive response.* Facilities in this category are activated only when an intrusion is identified and their effectiveness is highly dependent on the accuracy and latency of intrusion detection.
- *Proactive response.* Facilities in this category are activated in a proactive manner based on suspicious activities (or signs) before an intrusion is confirmed. Although proactive response may consume more resources, it may immunize the system from the damage caused by many attacks.
- *Adaptive response.* Feedback based adaptation is a nice feature of many survivable systems, where the *defense posture* (i.e., security mechanism configurations) of the system is dynamically adjusted based on the changing environment.

Compared with intrusion masking technologies, where many attacks may be masked without causing any system security (e.g., integrity and availability) degradation, defense-in-depth technologies usually would introduce certain level of security degradation. On the other hand, the advantages of defense-in-depth technologies are that (a) they can be directly applied to legacy systems, (b) they may effectively handle malicious user activities, and (c) their overhead (or cost) can be much smaller than intrusion masking technologies. For example, defense-in-depth systems usually need much less redundancy. The key issues in defense-in-depth include but are not limited to: How to quickly contain/isolate the intrusions so that their infection will not be too serious to operate through? How to quickly distinguish the damaged part from the undamaged part of the system? How to quickly repair the contaminated part of the system without bringing it offline? How to handle the impact of false alarms, undetected intrusions, and detection latency? How to make the intrusion response facilities adaptive and proactive?

1.2 Overview of the ITDB Framework

The ITDB framework, which is composed of five architectures, as shown in Figures 1, 2, 3, 4 and 5, respectively, combines a family of novel defense-in-depth techniques to achieve database self-healing under attacks. In particular, Architecture I detects intrusions, and locates

and repairs the damage caused by intrusions. Architecture II enhances Architecture I with the ability to isolate attacks so that the database can be immunized from the damage caused by a lot of intrusions. Architecture III enhances Architecture I with the ability to dynamically contain the damage in such a way that no damage will leak out during the attack recovery process. Architecture IV enhances Architectures II and III with the ability to adapt the self healing controls to the changing environment so that a stabilized level of healthiness can be maintained. Architecture V enhances Architecture IV with the ability to deliver differential, quantitative QoIA services to customers.

The ITDB framework focuses on transaction-level intrusion tolerance, which, based on the fact that most attacks are from insiders [6], should be a major aspect of self-healing database systems. Although using "off-the-shelf" components, ITDB cannot (directly) defend against processor, OS, or DBMS-level attacks, when the lower-level attacks are not so serious and when most attacks are via malicious transactions, ITDB can still be very effective. Moreover, existing lower-level self-healing mechanisms, such as those proposed in [30, 5, 33], can be easily integrated into ITDB architectures to build a multi-layer, self-healing database system.

The remainder of the paper is organized as follows. Section 2 discusses some related work. In Sections 3, 4, 5, 6, and 7, we present five self-healing database system architectures. Section 8 concludes the paper.

2. Related Work

Database security concerns the confidentiality, integrity, and availability of data stored in a database. A broad span of research from authorization [15, 39, 18], to inference control [1], to multilevel secure databases [49, 41], and to multilevel secure transaction processing [4], addresses primarily how to protect the security of a database, especially its confidentiality. Intrusion tolerance, however, is seldom addressed.

One critical step towards intrusion-tolerant database systems is intrusion detection (ID), which has attracted many researchers [29, 35]. The existing methodology of ID can be roughly classed as *anomaly detection* [19, 40, 21, 43] or *misuse detection* [12, 17]. However, current ID research focuses on identifying attacks on OS and computer networks. Although there has been some work on database ID [8, 45], these methods are neither application aware nor at the transaction-level.

The need for intrusion tolerance has been recognized by many researchers in such contexts as *information warfare* [14]. Recently, extensive research has been done in general principles of survivability [20, 48, 13], survivability of networks [34], survivable storage systems [51], survivable application development via middleware [37], persistent objects [32], and survivable document editing systems [46].

Some research has also been done in database intrusion tolerance. In [3], a fault tolerant approach is taken to survive database attacks where (a) several useful survivability phases are suggested, though no concrete mechanisms are proposed for these phases; (b) a color scheme for marking damage (and repair) and a notion of integrity suitable for partially damaged databases are used to develop a mechanism by which databases under attack could still be safely used.

Some of the architectures presented in this paper are directly or indirectly proposed, investigated (using detailed system and algorithm designs), and evaluated (using prototypes) by our previous research. In particular, Architecture I is addressed in [2, 27]; Architecture II is addressed in [25, 23]; and Architecture III is proposed in [24, 26].

3. Scheme I

Since the property of database *atomicity* indicates that only committed transactions can really change the database, it is theoretically true that if we can detect every malicious transaction before it commits, then we can roll back the transaction before it causes any damage. However, this "perfect" solution is not practical for two reasons. First, transaction execution is, in general, much quicker than detection, and slowing down transaction execution can cause very serious denial-of-service. For example, the Microsoft SQL Server can execute over 1000 (TPC-C) transactions within one second (see *www.oracle.com*), while the average anomaly detection latency is typically in the scale of minutes or seconds. Detection is much slower since: (1) in many cases detection needs human intervention; (2) to reduce false alarms, in many cases a sequence of actions should be analyzed. For example, [21] shows that when using system call trails to identify *sendmail* attacks, synthesizing the anomaly scores of a sequence of system calls (longer than 5) can achieve much better accuracy than based on single system calls.

Second, some authorized but malicious transactions are very difficult to detect. They look and behave just like other legitimate transactions. Anomaly detection based on the semantics of transactions (and the application) may be the only effective way to identify such attacks, however, it is very difficult, if not impossible, for an anomaly detector to have a 100% detection rate with reasonable false alarm rate and detection latency.

Hence, a *practical* goal should be: "After the database is damaged, locate the damaged part and repair it as soon as possible, so that the database can continue being useful in the face of attacks." In other words, we want the database system to operate through attacks.

Architecture I, as shown in Figure 1, combines intrusion detection and attack recovery to achieve this goal. In particular, the *Intrusion Detector* monitors and analyzes the *trails* of database sessions and transactions in a real-time manner to identify malicious transactions as soon as possible. Alarms of malicious transactions, when raised, will be instantly sent to the *Repair Manager*, which will locate the damage caused by the attack and repair the damage. During the whole intrusion detection and attack recovery process, the database continues executing new transactions.

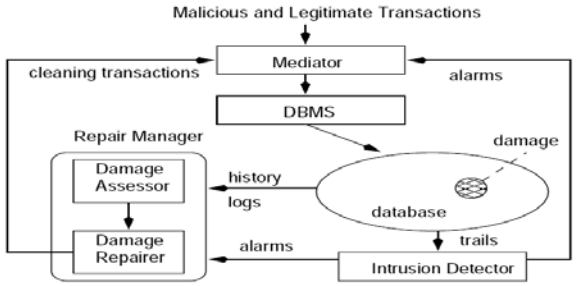


Fig. 1 Architecture I

Although there are lots of anomaly detection algorithms (for host or network based intrusion detection), they usually cannot be directly applied in malicious transaction detection, which faces the following unique challenges:

- Application semantics must be captured and used. For example, for a school salary management application, a $3000 raise is normal, but a $10000 raise is very *abnormal*. Application semantics based intrusion detection is *application aware*. Since different applications can have very different semantics, general application-aware database intrusion detection systems must support dynamic integration of application semantics. Since different anomaly detection algorithms may be good for different application semantics, a general application-aware database intrusion detection system must adapt itself to application semantics.
- Multi-layer intrusion detection is usually necessary for detection accuracy. First, proofs from application layer, session layer, transaction layer, process layer, and system call layer should be *synthesized* to do intrusion detection. Lower level proofs can help identify higher level anomalies. Second, OS-level and transaction-level intrusion detection should be coupled with each other.

We suggest a flexible *cartridge-like* detector to address these challenges. The detector is a cartridge which should be general enough to plug in a variety of (a) anomaly detection algorithms such as [8] and [45], (b) application semantics extraction algorithms, and (c) application semantics based adaptation policies. The user should be able to prepare some of these algorithms and policies. The detector should provide the interfaces for the user to pick existing and provide new *bullets*, and the detector should not be required to rebuild itself again and again to support each new bullet. (Here each bullet indicates an algorithm or a policy that the detector wants to plug in.) In this way, one detector can be used to meet the intrusion detection needs of multiple applications. Flexibility and expressiveness are the key challenges for developing such a detector.

Malicious transactions can seriously corrupt a database through a vulnerability denoted as *damage spreading*. In a database, the results of one transaction can affect the execution of other transactions. When a transaction T_i reads a data object x updated by another transaction T_j, T_i is directly *affected* by T_j. If a third transaction T_k is affected by T_i, but not directly affected by T_j, T_k is indirectly affected by T_j. It is easy to see that when a (relatively old) transaction B_i that updates x is identified as malicious, the damage to x can spread to every object updated by a *good* transaction that is affected by B_i, directly or indirectly. In a word, the read-from dependency among transactions forms the *traces* along which damage spreads.

The job of attack recovery is two-fold: damage assessment and repair. In particular, the job of the *Damage Assessor* is to locate each affected good transaction, i.e., the damage spreading traces; and the job of the *Damage Repairer* is to recover the database from the damage caused on the objects updated along the traces. In particular, when an affected transaction T is located, the Damage Repairer builds a specific *cleaning* transaction to *clean* each object updated by T (and not cleaned yet). Cleaning an object is simply done by restoring the value of the object to its latest undamaged version.

Temporarily stopping the database will certainly make the attack recovery job simpler since the damage will no longer spread and the repair can be done backwardly after the assessment is done, that is, we can repair the database by simply undoing the malicious as well as affected transactions in the reverse order of their commit order. An even simpler approach is to roll back the database (state) to a *check-point* taken before the attack [22], though all (legitimate) work done after the checkpointing time will be lost. However, since many critical database servers need to be 24*7 available and temporarily making the database shut down can be the real goal of the attacker, on-the-fly attack recovery which never stops the database is necessary in many cases.

On-the-fly attack recovery faces several unique challenges. First, we need to do repair forwardly since the assessment process may never stop. Second, cleaned data objects could be re-damaged during attack recovery. Finally, the attack recovery process may never *terminate*. Since as the damaged objects are identified and cleaned new transactions can spread damage if they read a damaged

but still unidentified object, so we face two critical questions. (a) Will the attack recovery process terminate? (b) If the attack recovery process terminates, can we detect the termination?

To tackle challenge 1, we must ensure that a later on cleaning transaction will not accidentally damage an object cleaned by a previous cleaning transaction. To tackle challenge 2, we must not mistake a cleaned object as damaged, and we must not mistake a re-damaged object as already cleaned. To tackle challenge 3, our study in [2] shows that when the damage spreading speed is quicker than the repair speed, the repair may never terminate. Otherwise, the repair process will terminate, and under the following three conditions we can ensure that the repair terminates: (1) every malicious transaction is cleaned; (2) every identified damaged object is cleaned; (3) further (assessment) scans will not identify any new damage (if no new attack comes).

From a state-transition angle, the job of attack recovery is to get a *state* of the database, which is determined by the values of the data objects, where (a) no effects of the malicious transactions are there and (b) the work of good transactions should be retained as much as possible. In particular, transactions transform the database from one state to another. Good transactions transform a good database state to another good state, but malicious transactions can transform a good state to a damaged one. Moreover, both malicious and affected (good) transactions can make an already damaged state even worse. We say a database state S_1 is *better* than another one S_2 if S_1 has fewer corrupted objects. The goal of on-the-fly attack recovery is to get the state better and better, although during the repair process new attacks and damage spreading could (temporarily) make the state even worse. (A state-oriented object-by-object attack recovery scheme is proposed in [38].)

Architecture I has the following properties: (1) It builds itself on top of an "off-the-shelf" DBMS. It does not require the DBMS kernel to be changed. It has almost no impact on the performance of the database server except that the *Mediator* can cause some service delay and the cleaning transactions can make the server busier. (2) The self-healing processes are all on-the-fly. (3) During attack recovery, the data integrity level can vary from time to time. When the attacks are intense, damage spreading can be very serious, and the integrity level can be dramatically lowered. In this situation, asking the *Mediator* to slow down the execution of new transactions can help *stabilize* the data integrity level, although this can cause some availability loss. This indicates that integrity and availability can be two conflicting goals in self-healing. (4) More availability loss can be caused when (a) the Intrusion Detector raises false alarms; or (b) a corrupted object is located (It will not be accessible until it is cleaned. Making damaged parts of the database available to new transactions can seriously spread the damage). (5) Inaccuracy of intrusion detection can cause some damage to not be located or repaired. (6) Architecture I is not designed to and cannot handle physical world attack recovery, which usually requires many additional activities. Logically repairing a database does not always indicate that the corresponding physical world damage can be recovered.

A major concern people may have is whether Architecture I can achieve better survivability when the Intrusion Detector is limited and whether the gained survivability, if any, is worth the corresponding performance degradation. To justify the cost-effectiveness of Architecture I, we have implemented a prototype of Architecture I on top of an Oracle database server. Our evaluation results suggest that when the performance of the Intrusion Detector is reasonable, Architecture I can effectively locate and repair damage on-the-fly with a reasonable amount of performance degradation (around 30%) [47].

4. Scheme II

One problem of Architecture I is that during the *detection latency* of a malicious transaction B, i.e., the duration from the time B commits to the time B is detected, damage can seriously spread. The reason is that during the detection latency many innocent transactions could be executed and affected. For example, if the detection latency is 2 seconds, then Microsoft SQL Server can execute over 2000 transactions during the latency on a single system, and they can access the objects damaged by B freely (since we do not know which objects are damaged by B during the latency).

Quicker intrusion detection can mitigate this problem, however, reducing detection latency without sacrificing the false alarm rate or the detection rate is very difficult, if not impossible. When the detection rate is decreased, more damage is left unrepaired. When the false alarm rate is increased, more denial-of-service will be caused. These two outcomes contradict the goal of Architecture I.

Architecture II, as shown in Figure 2, integrates a novel isolation technique to tackle this problem. In particular, first, the Intrusion Detector will raise two levels of alarms: when the (synthesized) anomaly of a transaction (or session) is above Level 1 anomaly threshold TH_m, the transaction is reported malicious; when the anomaly is above Level 2 anomaly threshold TH_s (but below TH_m), the transaction is reported *suspicious*. (The values of TH_m and TH_s are determined primarily based on the statistics about previous attacks). Suspicious transactions should have a significant probability that they are an attack. Second, when a malicious transaction is reported, the system works in the same way as Architecture I. When a suspicious transaction T_S is reported, the Mediator, with the help of the *Isolation*

Manager, will redirect T_s (and the following transactions submitted by the user that submits T_s) to a *virtually* separated database environment where the user will be isolated. Later on, if the user is proven malicious, the Isolation Manager will discard the effects of the user; if the user is shown innocent, the Isolation Manager will *merge* the effects of the user back into the main database. In this way, damage spreading can be dramatically reduced without sacrificing the detection rate or losing the availability to good transactions.

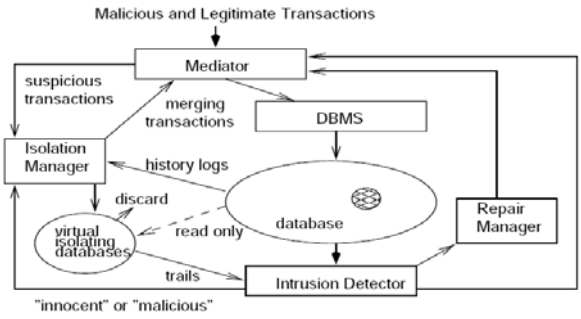


Fig. 2 Architecture II

We enforce isolation on an user-by-user basis because the transactions submitted by the same user (during the same session) should be able to see the effects of each other. And the framework should be able to isolate multiple users simultaneously. Isolating a group of users within the same virtual database can help tackle collusive attacks, however, a lot of availability may be lost when only some but not all members of the group are malicious. Using a completely replicated database to isolate a user has two drawbacks: (1) it is too expensive; (2) new updates of unisolated users are not visible to isolated users. In Architecture II, we use data *versions* to virtually build isolating databases. In particular, a data object x always has a unique trustworthy version, denoted $x[main]$. And only if x is updated by an isolated user can x have an extra suspicious version. In this way, the total number of suspicious versions will be much less than the number of main versions.

The isolation algorithm has two key parts: (1) how to perform the read and write operations of isolated users (Note that unisolated users can access only the main database); and (2) how to do merging after an isolated user is proven innocent. For part 1, we can enforce *one-way* isolation where isolated users can read main versions if they do not have the corresponding suspicious versions, and all writes of isolated users must be performed on suspicious versions. In this way, the data freshness to isolated users is maximized without harming the main database.

The key challenge in part 2 is the inconsistency between main versions and suspicious versions. If a trustworthy user and an isolated user update the same object x independently, $x[main]$ and the suspicious version will become inconsistent, and one update has to be backed out in order to do consistent merging. In addition, [25] shows that (1) even if they do not update the same object, inconsistency could still be caused; and (2) the merging of the effects of one isolated user could make another still being isolated history invalid. These inconsistencies must be resolved during a merging (e.g., [25] proposes a *precedence-graph* based approach that can identify and resolve all the inconsistencies).

Architecture II has the following set of properties. (1) Isolation is, to large extent, transparent to suspicious users. (2) The extra storage cost for isolation is extremely low. (3) The data consistency is kept before isolation and after merging. (4) During a merge, if there are some inconsistencies, some isolated or unisolated transactions have to be backed out to resolve these inconsistencies. This is the main cost of Architecture II. Fortunately, the simulation study done in [10] shows that the back-out cost is only about 5%. After the inconsistencies are resolved, the merging can be easily done by *forwarding* the left updates of the isolated user to the main database. (5) Architecture II has almost no impact on the performance of the database server except that during each merging process (a) the isolated user cannot execute new transactions; and (b) the main database tables involved in the update forwarding process will be temporarily locked.

We have been implementing an isolation subsystem prototype to further justify the cost-effectiveness of Architecture II. In order to transparently isolate a transaction on top of a commercial single-version DBMS such as Oracle, we need to (a) use extra tables to simulate multiple versions and (b) rewrite the SQL statements involved in this transaction in such a way that the one-way isolation policy can be achieved. Note that query rewriting could cause some service delay to isolated users but not to unisolated users.

5. Scheme III

Another problem of Architecture I is that its damage containment may not be effective. Architecture I *contains* the damage by disallowing transactions to read the set of data objects that are identified (by the Damage Assessor) as corrupted. This *one-phase* damage containment approach has a serious drawback, that is, it cannot prevent the damage caused on the objects that are corrupted but not yet located from spreading. Assessing the damage caused by a malicious transaction B can take a substantial amount of time, especially when there are a lot of transactions executed during the detection latency of B. During the *assessment latency*, the damage caused during the detection latency can spread too many other objects before being contained.

Architecture III, as shown in Figure 3, integrates a novel multi-phase damage containment technique to tackle

this problem. In particular, the damage containment process has one containing phase, which instantly contains the damage that *might* have been caused (or spread) by the intrusion as soon as the intrusion is detected and one or more later on uncontaining phases to uncontain the objects that are mistakenly contained during the containing phase, and the objects that are cleaned. In Architecture III, the *Damage Container* will enforce the containing phase (as soon as a malicious transaction is reported) by sending some containing instructions to the *Containment Executor*. The *Uncontainer*, with the help from the Damage Assessor, will enforce the uncontaining phases by sending some uncontaining instructions to the Containment Executor. The Containment Executor controls the access of the user transactions to the database according to these instructions.

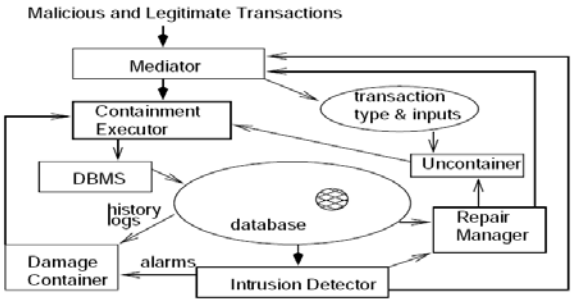


Fig. 3 Architecture III

When a malicious transaction B is detected, the containing phase must ensure that the damage caused directly or indirectly by B will be contained. In addition, the containing phase must be quick enough because otherwise either a lot of damage can leak out during the phase, or substantial availability can be lost. Time stamps can be exploited to achieve this goal. The containing phase can be done by just adding an access control rule to the Containment Executor, which denies access to the set of objects updated during the period of time from the time B commits to the time the containing phase starts. This period of time is called the *containing-time-window*. When the containing phase starts, every active transaction should be aborted because they could spread damage. New transactions can be executed only after the containing phase ends.

It is clear that the containing phase *overcontains* the damage in most cases. Many objects updated within the containing time window can be undamaged. And we must uncontain them as soon as possible to reduce the corresponding availability loss. Accurate uncontainment can be done based on the reports from the Damage Assessor, which could be too slow due to the assessment latency. [24] shows that transaction *types* can be exploited to do much *quicker* uncontainment. In particular, assuming that (a) each transaction T_i belongs to a transaction type $type(T_i)$ and (b) the *profile* for $type(T_i)$ is known, the *read set template* and *write set template* can be extracted from $type(T_i)$'s profile. The templates specify the kind of objects that transactions of $type(T_i)$ can read or write. As a result, the *approximate* read-from dependency among a history of transactions can be quickly captured by identifying the read-from dependency among the types of these transactions. Moreover, the type-based approach can be made more accurate by *materializing* the templates of transactions using their inputs before analyzing the read-from dependency among the types.

Architecture III has the following set of properties. (1) It can ensure that after the containing phase no damage (caused by the malicious transaction) leaks out. (2) As a result, the attack recovery process needs only to repair the damage caused by the transactions that commit during the containing time window, and the termination problem addressed in Architecture I does not exist any longer. (3) One phase containment and multi-phase containment are the two extremes of the spectrum of damage containment methods. In particular, one-phase containment has maximum damage leakage (so minimum integrity) but maximum availability, while multi-phase containment has zero damage leakage (so maximum integrity) but minimum availability. In the middle of the spectrum, there could be a variety of approximate damage containment methods that allow some damage leakage.

Architectures II and III share the same goal, that is, to reduce the extent of damage spreading, while they take two very different approaches. We are pleased to find that these two architectures are actually complementary to each other and can be easily integrated into one architecture, as illustrated in Figure 4.

Finally, in [3], a color scheme for marking damage and a notion of integrity suitable for partially damaged databases are used to develop a mechanism by which databases under attack could still be safely used. This mechanism can be viewed as a containment mechanism, however, it assumes that each object has an (accurate) initial damage mark, but Architecture III does not. In fact, Architecture III focuses on how to mark (and contain) the damage and how to deal with the impact of inaccurate damage marks.

6. Scheme IV

The self-healing components introduced in Architectures I, II, and III can behave in many different ways. At one point of time, the *resilience* or healthiness of a self-healing database system is primarily affected by four factors: (a) the current attacks; (b) the current workload; (c) the current system state; and (d) the current defense *behavior* of the system. It is clear that based on the same system state, attack pattern, and workload, two self-healing database systems (of the same Architecture) with different behaviors can yield very different levels of resilience. This suggests

that one defense behavior is only good for a limited set of *environments*, which are determined by factors (a), (b), and (c). To achieve the maximum amount of resilience, intrusion tolerant systems must *adapt* their behaviors to the environment.

Architecture IV, as shown in Figure 4, integrates a *reconfiguration* framework to handle this challenge. In particular, the *Adaptor* is deployed to *monitor* the environment changes and *adjust* the behaviors of the self-healing components in such a way that the adjusted system behavior is more (cost) effective than the old system behavior in the changed environment.

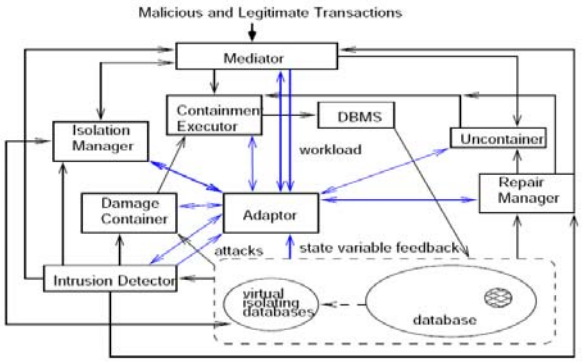


Fig. 4 Architecture IV

In Architectures I, II, and III, almost every self-healing component is reconfigurable and the *behavior* of each such component is *controlled* by a set of a *parameters*. For example, the major control parameters for the Intrusion Detector are TH_m and TH_s. The major control parameter for the *Damage Container* is the amount of allowed damage *leakage*, denoted DL. When $DL = 0$, multi-phase containment is enforced; when there is no restriction on DL, one-phase containment is enforced. The major control parameter for the Mediator is the transaction delay time, denoted DT. When $DT = 0$, transactions are executed in full speed; when DT is not zero, transaction executions are slowed down. At time t, we call the set of control parameters (and the associated values) for an intrusion tolerance component C_i, the *configuration* (vector) of C_i at time t, and the set of the configurations for all the self-healing components, the *configuration* of the self-healing system at time t. In Architecture IV, each reconfiguration is done by adjusting the system from one configuration to another configuration.

The goal of Architecture IV is to improve the healthiness or resilience of the system, which has three major aspects: (1) how well the level of data integrity is maintained in the face of attacks; (2) how well the level of data and system availability is maintained in the face of attacks; and (3) how well the level of cost effectiveness is maintained in the face of attacks.

To do optimal reconfiguration, we want to find the best configuration (vector) for each (new) environment. However, this is very difficult, if not impossible, since the *adaptation space* of Architecture IV systems contains an exponential number of configurations. To illustrate, the simplest configuration of an Architecture IV system could be $[TH_m, TH_s, DL, DT]$, then the size of the adaptation space is $domain(TH_m) \times domain(TH_s) \times domain(DL) \times domain(DT)$, which is actually huge. Moreover, we face conflicting reconfiguration criteria, that is, healthiness and cost conflict with each other, and integrity and availability conflict with each other. Therefore, we envision the problem of finding the best system configuration under multiple conflicting criteria a NP-hard problem.

Architecture IV focuses on near optimal heuristic adaptation algorithms which can have much less complexity. For example, a data integrity favored heuristic can work as follows: when the level of data integrity, i.e., LI, is below a specific warning threshold I_w, (a) switch the system to multi-phase containment, i.e., let $DL = 0$; (b) slow down the execution of new transactions by $DT = DT + \alpha(I_w - LI)$; and (c) lower the anomaly levels required for alarm raising, that is, $TH_m = TH_m - \beta(I_w - LI)$, and $TH_s = TH_s - \gamma(I_w - LI)$. In this way, we reject and isolate more transactions. Here the values of α, β, and γ are determined based on previous experiences. Note that it is very possible that different (value) combinations of (α, β, γ) are optimal for different environments. Hence it is worthy to have multiple such heuristics with different combinations of (α, β, γ).

It is clear that under different environments different heuristics are the most effective. For example, in some cases integrity favored heuristics can be better, but in some other cases availability favored heuristics can be better. Architecture IV systems should have a mechanism to guide the system to pick the right heuristic (for the current environment). For example, a rule-based mechanism such as [28] can be used for this purpose.

7. Scheme V

The resilience achieved by Architecture IV is *state-oriented* survivability, that is, the amount of resilience or healthiness achieved by Architecture IV is specified, measured, and delivered in terms of the database *state*. For example, at time t, an integrity level of 0.92 achieved by an self-healing database system that protects a database of 10,000 data objects can simply mean that 800 objects are corrupted, and an availability level of 0.98 can simply mean that only 200 objects are not accessible. Note than Architecture IV does not *differentiate* between data objects.

Unfortunately, state-oriented self-healing database systems have one serious drawback, that is, they are in

general not cost-effective in handling people's self-healing requirements in the real world. In the real world, different users usually have different healthiness or usability requirements on the shared database system. For example, in a bank, customer Alice could be able to tolerate much less fraud loss on her accounts than Bob on his. In other words, Alice has a much higher integrity level requirement than Bob. In this situation, to satisfy both Alice and Bob, Architecture IV has to achieve (and maintain) the integrity level required by Alice across the whole database, and as a result Architecture IV can waste substantial resources to protect Bob's accounts.

The drawback of state-oriented survivability motivates the idea of *service-oriented* survivability where users' intrusion-tolerant requirements are associated with each (transaction processing) *service*, and the database system's goal is to make sure that the amount of resilience requirement associated with a service is satisfied when the service is delivered. In particular, we call a service associated with a specific level of assurance a Quality of Information Assurance (QoIA) service. And from the viewpoint of users, the goal of a self-healing database system is enabling people to get the *QoIA services* that they have subscribed for even in the face of attacks. To illustrate, in the above example a QoIA balance inquiry service delivered to Alice could be associated with either one of the following two healthiness levels: (1) above 90% accounts involved in this service are not corrupted; (2) for each account involved in this inquiry, the balance reported is at least 90% of the correct balance.

It should be noticed that state-oriented survivability and service-oriented survivability are closely related to each other. Their relationship can be captured by the notions of *state trustworthiness* or healthiness, which is dependent on the extent to which the data objects can be corrupted or made unavailable and *service trustworthiness* or healthiness, which is dependent on the extent to which a service can be distorted by the attacker. If we assume that the DBMS and all transaction codes are trusted, then it is not difficult to see that the QoIA requirements associated with a service can be equivalently *mapped* to a set of state healthiness requirements since each service can be modeled as a function of the database state on which the service is executed.

Architecture V, as shown in Figure 5, extends state-oriented self-healing database systems to service oriented self-healing database systems. In particular, the *QoIA Reservation Console* enables users to subscribe for QoIA services. The *Observer* monitors (and measures) the trustworthiness or healthiness of the database state. The *Trustworthiness Assessor* uses the observed healthiness measurements to *infer* the "real" healthiness of the database state. The *QoIA Adaptor* enhances the Architecture IV Adaptor with the ability to map QoIA requirements associated with services to a set of state trustworthiness requirements and the ability to maintain *differential* state trustworthiness. The adaptation operations performed by the QoIA Adaptor are determined based on the difference between the inferred set of state trustworthiness measurements and the set of state trustworthiness requirements mapped from user QoIA requirements.

To develop an Architecture V system, we face several key challenges. First, although the QoIA requirements associated with a service can be straightforwardly specified based on the results and outputs of the service, delivering a set of QoIA services in a differential way is challenging. Our idea is to indirectly deliver QoIA services through differential state trustworthiness maintenance via the mapping from QoIA requirements to state trustworthiness requirements. Although it is not very difficult to map one service's QoIA requirements to a set of state trustworthiness requirements based on the "function" performed by the service, it could be difficult to resolve the inconsistencies among the set of different state trustworthiness requirements that the set of QoIA services have on a shared data object. Second, how can we maintain differential state trustworthiness? Our idea is to apply different self-healing controls on different parts of the database. To make this idea feasible, we need to make sure that one set of self-healing controls does not influence another set of self-healing controls. Third, how do we ensure that the (mapped) state trustworthiness requirements on a part of the database can be satisfied in the face of attacks? Our idea is through QoIA-aware adaptations where the set of self-healing controls enforced on a part of the database can adapt to the changing environment in such a way that the set of state trustworthiness requirements can be satisfied with minimum cost. To make this idea feasible, we need to be able to accurately *measure* state trustworthiness. However, this is not an easy job. The measurements observed by the Observer are usually incomplete and could even be misleading due to false negatives, false positives, and detection delays. New techniques are needed to infer the "real" trustworthiness of the database state based on the observed measurements. For example, a statistics based approach could work for this purpose.

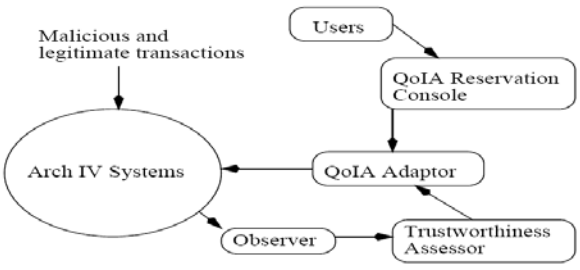


Fig. 5 Architecture V

8. Conclusion

In this paper, we have presented five self-healing database system architectures which can be built on top of COTS components. These architectures indicate that: (1) a multi-layer, defense-in-depth approach, as summarized in Figure 6, is usually more cost-effective than having the system's survivability depend on the effectiveness of one or two mechanisms such as intrusion detection; (2) adaptive intrusion-tolerant mechanisms are usually more cost-effective than pre-programmed intrusion tolerant mechanisms; (3) service-oriented, intrusion-tolerant database systems are usually more cost-effective than state-oriented, intrusion-tolerant database systems. Finally, we would like to restate that OS-level and transaction-level self-healing mechanisms should be seamlessly integrated to build multi-layer, self-healing database systems. This integration requires careful study of the relationships between these two layers of mechanisms. For example, although OS-level data corruptions cannot be detected using transaction-level approaches, transaction-level approaches can be very useful to recover from these corruptions.

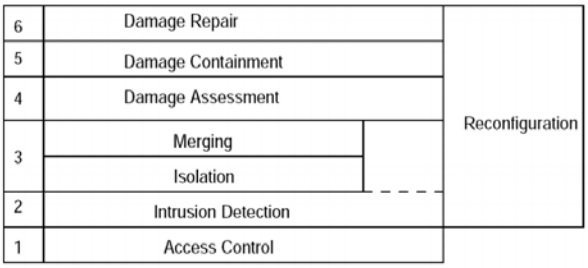


Fig. 6 Intrusion Tolerance in Depth

Finally, we would like to mention a couple of exciting future research directions that should be able to further improve the proposed architectures:

- Malicious transactions may be able to be *masked* by a set of partially replicated database servers where each server executes only a group of but not all transactions. The key challenge for such a masking framework should be the tradeoff between security and data consistency.
- It is in general true that the accuracy and latency of the Intrusion Detector can have a big impact on the overall cost-effectiveness of an intrusion-tolerant (database) system. Hence it is very desirable to know how "good" a detector needs to be (in terms of false positive rate, false negative rate, and detection latency) in order to make an intrusion tolerant database system (of Architectures I, II, III, IV, or V) that deploys the detector, cost-effective.
- OS-level and transaction-level intrusion-tolerance mechanisms should be seamlessly integrated to build multi-layer, intrusion-tolerant database systems. This integration requires careful study of the relationships between these two layers of mechanisms. For example, although OS-level data corruptions cannot be detected using transaction-level approaches, transaction-level approaches can be very useful to recover from these corruptions.

Acknowledgment

This work is supported by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-00-2-0575, by DARPA and AFRL, AFMC, USAF, under award number F20602-02-1-0216, by NSF CCR-TC-0233324, and by Department of Energy Early Career PI Award.

References

[1] M. R. Adam. Security-Control Methods for Statistical Database: A Comparative Study. *ACM Computing Surveys*, 21(4), 1989.

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

[3] P. Ammann, S. Jajodia, C.D. McCollum, and B.T. Blaustein. Surviving information warfare attacks on databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–174, Oakland, CA, May 1997.

[4] V. Atluri, S. Jajodia, and B. George. *Multilevel Secure Transaction Processing*. Kluwer Academic Publishers, 1999.

[5] D. Barbara, R. Goel, and S. Jajodia. Using checksums to detect data corruption. In *Proceedings of the 2000 International Conference on Extending Data Base Technology*, Mar 2000.

[6] Carter and Katz. Computer Crime: An Emerging Challenge for Law Enforcement. *FBI Law Enforcement Bulletin*, 1(8), December 1996.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proc. OSDI 99*, 1999.

[8] C. Y. Chung, M. Gertz, and K. Levitt. Demids: A misuse detection system for database systems. In *14th IFIP WG11.3 Working Conference on Database and Application Security*, 2000.

[9] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Symposium*, 1998.

[10] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–581, September 1984.

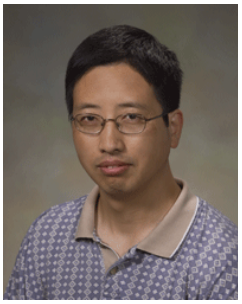
[11] D. Engler, Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. SOSP 01*, 2001.

[12] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, October 1991.

[13] K. Goseva-Popstojanova, F. Wang, R. Wang, G. Feng, K. Vaidyanathan, K. Trivedi, and B. Muthusamy.

Characterizing intrusion tolerant systems using a a state transition model. In *Proc. 2001 DARPA Information Survivability Conference (DISCEX)*, June 2001.
[14] R. Graubart, L. Schlipper, and C. McCollum. Defending database management systems against information warfare attacks. Technical report, The MITRE Corporation, 1996.
[15] P. P. Griffiths and B. W. Wade. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):242–255, September 1976.
[16] K. Ilgun. Ustat: A real-time intrusion detection system for unix. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1993.
[17] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.
[18] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 1997.
[19] H. S. Javitz and A. Valdes. The sri ides statistical anomaly detector. In *Proceedings IEEE Computer Society Symposium on Security and Privacy*, Oakland, CA, May 1991.
[20] J. Knight, K. Sullivan, M. Elder, and C. Wang. Survivability architectures: Issues and approaches. In *Proceedings of the 2000 DARPA Information Survivability Conference & Exposition*, pages 157–171, CA, June 2000.
[21] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.
[22] J. L. Lin and M. H. Dunham. A survey of distributed database checkpointing. *Distributed and Parallel Databases*, 5(3), 1997.
[23] P. Liu. Dais: A real-time data attack isolation system for commercial database applications. In *Proceedings of the 17th Annual Computer Security Applications Conference*, 2001.
[24] P. Liu and S. Jajodia. Multi-phase damage confinement in database systems for intrusion tolerance. In *Proc. 14th IEEE Computer Security Foundations Workshop*, Nova Scotia, Canada, June 2001.
[25] P. Liu, S. Jajodia, and C.D. McCollum. Intrusion confinement by isolation in information systems. *Journal of Computer Security*, 8(4):243–279, 2000.
[26] P. Liu and Y. Wang. The design and implementation of a multiphase database damage confinement system. In *Proceedings of the 2002 IFIP WG 11.3 Working Conference on Data and Application Security*, 2002.
[27] P. Luenam and P. Liu. Odar: An on-the-fly damage assessment and repair system for commercial database applications. In *Proceedings of the 2001 IFIP WG 11.3 Working Conference on Database and Application Security*, 2001.
[28] P. Luenam and P. Liu. The design of an adaptive intrusion tolerant database system. In *Proc. IEEE Workshop on Intrusion Tolerant Systems*, 2002.
[29] T.F. Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.
[30] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proceedings of 4th Symposium on Operating System Design and Implementation*, San Diego, CA, October 2000.
[31] D. Malkhi, M. Merritt, M. K. Reiter, and G. Taubenfeld. Objects shared by byzantine processes. *Distributed Computing*, 16(1), 2003.
[32] D. Malkhi, M. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the fleet system. In *Proc. 2001 DARPA Information Survivability Conference (DISCEX)*, June 2001.
[33] J. McDermott and D. Goldschlag. Towards a model of storage jamming. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 176–185, Kenmare, Ireland, June 1996.
[34] D. Medhi and D. Tipper. Multi-layered network survivability - models, analysis, architecture, framework and implementation: An overview. In *Proceedings of the 2000 DARPA Information Survivability Conference & Exposition*, pages 173–186, CA, June 2000.
[35] B. Mukherjee, L. T. Heberlein, and K.N. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, June 1994.
[36] G. C. Necula. Proof-carrying code. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, 1997.
[37] P. P. Pal, J. P. Loyall, R. E. Schantz, and J. A. Zinky. Open implementation toolkit for building survivable applications. In *Proc. 2000 DARPA Information Survivability Conference (DISCEX)*, June 2000.
[38] B. Panda and J. Giordano. Reconstructing the database after electronic attacks. In *Proceedings of the 12th IFIP 11.3 Working Conference on Database Security*, Greece, Italy, July 1998.
[39] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, 1994.
[40] D. Samfat and R. Molva. Idamn: An intrusion detection architecture for mibile networks. *IEEE Journal of Selected Areas in Communications*, 15(7):1373–1380, 1997.
[41] R. Sandhu and F. Chen. The multilevel relational (mlr) data model. *ACM Transactions on Information and Systems Security*, 1(1), 1998.
[42] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *Proc. 9th ACM Conference on Computer and Communications Security*, 2002.
[43] S. Sekar, M. Bendre, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proc. 2001 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.
[44] P. Stenstrom and et al. Trends in shared memory multiprocessing. *IEEE Computer*, (12):44–50, December 1997.
[45] S. Stolfo, D. Fan, and W. Lee. Credit card fraud detection using meta-learning: Issues and initial results. In *Proc. AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.
[46] M. Tallis and R. Balzer. Document integrity through mediated interfaces. In *Proc. 2001 DARPA Information Survivability Conference (DISCEX)*, June 2001.
[47] H. Wang, P. Liu, and L. Li. Evaluating the impact of intrusion detection deficiencies on the cost-effectiveness of attack recovery. In *Proc. 7th Information Security Conference*, 2004.

[48] F. Webber, P. P. Pal, R. E. Schantz, and J. P. Loyall. Defense-enabled applications. In *Proc. 2001 DARPA Information Survivability Conference (DISCEX)*, June 2001.
[49] M.Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACMTransactions on Database Systems*, 19(4):626–662, 1994.
[50] J.Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote, and P. Khosla. Survivable information storage systems. *IEEE Computer*, (8), 2000.
[51] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliccote, and P. K. Khosla. Survivable information storage systems. *IEEE Computer*, (8):61–68, August 2000.



Peng Liu is now an assistant professor of Information Sciences and Technology at Penn State University and the director of Cyber Security Lab. He received his BS and MS degree from the University of Science and Technology of China. He received his PhD degree from George Mason University in 1999. His research interests are in computer and network security. Dr. Liu has published a book and about 70 referred technical papers. Dr. Liu is the proceedings chair of the 2003 and 2004 ACM Conference on Computer and Communications Security. He is a program committee member of many conferences (e.g., 2004 International Conference on World Wide Web), and a referee for many journals (e.g., ACM Transactions on Information and Systems Security). Dr. Liu is a recipient of the United States DOE Early CAREER Award.

Jiwu Jing received his B.E. degree from Tsinghua University and his M.S. degree from the University of Science and Technology of China. He received his PhD degree from the Chinese Academy of Sciences. He is now the associate director of the Chinese State Key Lab of Information Security. His research interests are in computer and network security.