# Intrusion Confinement by Isolation in Information Systems

Peng Liu[1] Sushil Jajodia[2,3] Catherine D. McCollum[2]

[1]Department of Information Systems
University of Maryland, Baltimore County
Baltimore, MD 21250
pliu@umbc.edu

[2]The MITRE Corporation
McLean, VA 22102-3481
{jajodia, mccollum}@mitre.org

[3]Center for Secure Information Systems
and
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030-4444
jajodia@isse.gmu.edu

**Abstract**

System protection mechanisms such as access controls can be fooled by authorized but malicious users, masqueraders, and misfeasors. Intrusion detection techniques are therefore used to supplement them. However, damage could have occurred before an intrusion is detected. In many computing systems the requirement for a high degree of soundness of intrusion reporting can yield poor performance in detecting intrusions and cause long detection latency. As a result, serious damage can be caused either because many intrusions are never detected or the average detection latency is too long. The process of bounding the damage caused by intrusions during intrusion detection is referred to as *intrusion confinement*. We justify the necessity for intrusion confinement during detection by using a probabilistic analysis model, and propose a general solution to achieve intrusion confinement. The key idea of the solution is to isolate likely suspicious actions before a definite determination of intrusion is reported. We also present two concrete isolation protocols in the database and file system contexts, respectively, to evaluate the feasibility of the general solution, which can be applied to many types of information systems.

**Key Words:** Intrusion Confinement, Isolation, Intrusion Detection.

# 1 Introduction

Recently, increasing emphasis has been placed on supplementing protection of networks and information systems with intrusion detection [Lun93, MHL94, LM98], and numerous intrusion detection products have emerged commercially. Recognizing that access controls, filtering, and other protection mechanisms can be defeated or bypassed by would-be attackers who take advantage of remaining vulnerabilities, intrusion detection systems monitor system or network activity to discover attempts to disrupt or gain illicit access to systems. Intrusion detection must be concerned both with attempts by external penetrators to enter or interfere with the system and by authorized users to exceed their legitimate access or abuse the system in some way. The latter case also includes *seemingly* authorized users, such as masqueraders operating under another user's identification (id) and password, or outside attackers who successfully gained system access but eluded detection of the method of entry. The methodology of intrusion detection can be roughly classed as being either based on *statistical profiles* or on known patterns of attacks, called *signatures*.

Statistical profile-based systems compare relevant data by statistical or other methods to representative profiles of normal, expected activity on the system or network. Deviations indicate suspicious behavior. In these systems, stringent requirements apply not only to reporting an intrusion accurately (this is necessary because abnormal behavior is not always an intrusion) but also to detecting as many intrusions as possible (usually, not all intrusions can be detected). However, these two requirements can often result in conflicting design goals.

Based on the assumption that the more significant the deviation, the larger the possibility that the behavior of a user is an intrusion, in order to ensure a high degree of soundness of intrusion reporting, a significant anomaly is required to raise a warning. However, this requirement usually decreases the number of intrusions that can be detected because intrusions characterized by a gradual anomaly can be overlooked by the detector (a formal analysis is presented in Section 2), in which case many intruders may stay at large and cause substantial damage. Moreover, when the anomaly of an intrusion is accumulated slowly, detecting it can still cause a long latency even if it is characterized by a significant anomaly. As a result, substantial damage can be caused by an intruder within the latency. Relaxing the significancy requirement on deviations can mitigate these problems; however, the soundness of intrusion reporting can be dramatically degraded. In other words, innocent users may be mistaken for malicious ones and legal service requests may be denied in many situations. This situation occurs because trustworthy users may gradually change their behavior in non-significant ways.

With the requirement of a given degree of soundness, the process of bounding the damage caused by undetected intrusions and/or detected intrusions during their latencies is referred to as *intrusion confinement*. Intrusion confinement also encompasses corresponding strategies and mechanisms taken to enable the process.

Signature-based detection examines sniffer logs, audit data, or other data sources for evidence of operations, sequences, or techniques known to be used in particular

types of attacks. Signature-based detection techniques cannot be used to detect new, unanticipated patterns that could be detected by statistical profile-based detection techniques. However, they are used to detect known attacks. Although a behavior that partially matches a signature will not cause damage (see that the behavior is not an intrusion because otherwise the signature can be shorter), serious damage may have already been caused by the time a signature is matched. The process of bounding this type of damage is also referred to as *intrusion confinement*.

This paper makes two contributions: (1) It gives a simple probabilistic model to measure the effectiveness of an intrusion detection system and to justify the necessity of intrusion confinement. Based on the degree of effectiveness, the system security officer (SSO) can decide whether to enforce intrusion confinement and, if so, what mechanisms and strategies to apply for intrusion confinement. (2) It gives a general solution of intrusion confinement that is closely coupled with intrusion detection mechanisms and can be applied to many types of information systems. The basic idea is to isolate suspicious actions before an intrusion is reported. The solution is illustrated in the contexts of a database system and a file system, respectively.

When a suspicious behavior is discovered, the intrusion detector or the SSO must decide how to react and whether to allow continued access by the associated subject, say $B$, such as a process, a user, or a host. The system can let $B$ continue to access the system just as systems do today, risking further damage, or take action to deny $B$ continued access to the system, which may also be undesirable, for a couple of reasons. Further investigation may show that the suspicious behavior was actually unusual but trustworthy activity. If this is the case, denying $B$ access could result in unwarranted denial of service. It is even possible that an attacker is intentionally spoofing $B$ to provoke a denial-of-service response against $B$. On the other hand, if $B$ proves guilty, immediately denying access to the system may mean losing the opportunity to gather more information that would help identify the attacker and the objectives of the attack. Experience with system and network penetrations has shown the usefulness of another alternative of intrusion confinement: *isolation*.

Isolating $B$ transparently into a separate environment that still appears to $B$ to be the actual system allows $B$'s activities to be kept under surveillance without risking further harm to the system. An isolation strategy that has been used in such instances is known as *fishbowling*. Fishbowling involves setting up a separate lookalike host or file system and transparently redirecting the suspicious entity's requests to it. This approach allows the incident to be further studied to determine the real source, nature, and goal of the activity, but it has some limitations, particularly when considered at the application level. First, the substitute host or file system is essentially sacrificed during the suspected attack to monitor $B$, consuming significant resources that may be scarce. Second, since $B$ is cut off from the real system, if $B$ proves innocent, denial of service could still be a problem. While some types of service $B$ receives from the substitute, fishbowl system may be adequate, in other cases the lack of interaction with the real system's resources may prevent $B$ from continuing to produce valid results. On the other hand, if the semantics of the application are such that $B$ can continue producing valid work, this work will be lost when the incident concludes even if $B$

is deemed innocent and reconnected to the real system. The fishbowling mechanism makes no provision for re-merging updates from the substitute, fishbowl system back into the real system.

Within the general solution, we offer an isolation strategy that, while in some sense a counterpart to fishbowling, takes advantage of action semantics to avoid some of these limitations. In this isolation approach, as in the case of fishbowling, when $B$ comes under suspicion, we let $B$ continue working while we attempt to determine whether there is anything to worry about. At the same time, we isolate the system from any further damage $B$ might have in mind. However, we provide this isolation without consuming duplicate resources to construct an entirely separate environment, we allow options for partial interaction across the boundary, and we provide algorithms for smoothly merging $B$'s work back into the real system should $B$ prove innocent.

This work has relevance to information warfare (IW) defense [AJMB97, GSM96, MG96a, MG96b, PG99]. As pointed out by Ammann, et al. [AJMB97], IW defense does everything possible to prevent attacks from succeeding, but it also recognizes that attempting to prevent information attack is insufficient; attacks that are successful to some degree must be recognized as unavoidable, and comprehensive support for identifying and responding to attacks is required.

The rest of the paper is organized as follows. In Sections 2 and 3, we use a probabilistic model to justify the necessity of intrusion confinement and to indicate when intrusion confinement should be enforced. Section 4 presents a general solution of intrusion confinement. In Section 5 and 6, we evaluate the feasibility of our solution by presenting two concrete isolation protocols that can be applied to a database system and a file system, respectively. Section 7 discusses related work. In Section 8, we conclude the paper.

## 2 Why Is Intrusion Confinement Necessary?

Informally, suspicious behavior is the behavior that may have already caused some damage, or may cause some damage later on, but was not reported as an intrusion when it happened. In our model, suspicious behavior emerges in four situations:

1. In statistical profile-based detection, as discussed above, to achieve a high degree of soundness of intrusion reporting, some intrusions characterized by gradual deviations may stay undetected. The corresponding behaviors can be reported as suspicious.

2. In statistical profile-based detection, for a detection with a long latency, the corresponding behavior can be reported as suspicious in the middle of the latency.

3. In statistical profile-based detection, trustworthy behavior can be reported as suspicious if it is sufficiently unlike the corresponding profile.

4. In signature-based detection, partial matching of a signature can trigger a report of suspicious behavior.

In the remainder of this section, we present a probabilistic model for justifying the necessity of intrusion confinement in a statistical profile-based detection system (see Section 2.1) and a signature-based detection system (see Section 2.2), respectively. Note that all of the statistics used in this section are computed based on the audit trail and/or transaction log, where the entire intrusion history is assumed to be recorded.

## 2.1 Intrusion Confinement for Statistical Profile-Based Detection

Consider a statistical profile-based detection system where a user $U_i$ accesses the system through *sessions*. A session of $U_i$ begins when $U_i$ logs in and ends when $U_i$ logs out. A *behavior* of $U_i$ is a sequence of *actions* that can last across the boundaries of sessions. An action is the basic unit of the audit trail (or the transaction log). A short-term behavior of $U_i$ is a behavior that is composed of a sequence of $U_i$'s most recent actions. The length of the sequence, which is usually small, is determined by the SSO. In contrast, a long-term behavior of $U_i$ is also a sequence of $U_i$'s most recent actions but it is usually much longer than a short-term behavior. We assume that the *profile* of $U_i$ at time $t$ is determined by the long-term behavior of $U_i$ at time $t$. We further assume that the intrusion detector is triggered in every $m$ actions (or $m$ audit/log records), that is, after $m$ new actions are executed, both the current short-term behavior and long-term behavior of $U_i$ will be upgraded and the deviation of the new short-term behavior from the new long-term behavior, namely, the profile, will be computed to see if the current behavior of $U_i$ is abnormal (or suspicious). When a short-term behavior is upgraded, its oldest $m$ actions will be discarded and the newest $m$ actions will be appended.

We assume that statistical methods, such as the method proposed in NIDES [JV94], are used in the system. We assume that both the short-term and long-term behaviors of $U_i$ at time $t$ are described by a vector of $n$ intrusion detection measures (or variables). We denote them as $\vec{v_s} = (v_{s1}, ..., v_{sn})$ and $\vec{v_l} = (v_{l1}, ..., v_{ln})$, respectively. The deviation of $\vec{v_s}$ from $\vec{v_l}$ is specified by the distance, denoted $d(\vec{v_s}, \vec{v_l})$, from the point defined by $\vec{v_s}$ to the point defined by $\vec{v_l}$ in the $n$-dimension space*. We further assume that the longer the $d(\vec{v_s}, \vec{v_l})$, the bigger the probability that $\vec{v_s}$ is an intrusion. In the system, a warning is raised if $d(\vec{v_s}, \vec{v_l})$ is sufficiently long. In particular, we assume the system will report $\vec{v_s}$ as an intrusion if $d(\vec{v_s}, \vec{v_l}) \geq D_i$, and the system will reject all the following accesses of $U_i$ after this report. Here $D_i$ is a specific threshold determined by the SSO. Interested readers can refer to [JV94] for such details as to which measures can be used, how these measures can be quantified, and how $d(\vec{v_s}, \vec{v_l})$ can be computed.

Now we present a simple probabilistic model to evaluate the effectiveness of such a statistical profile-based detection system at an arbitrary point of time $t_i$. Although the effectiveness of the detection system may vary from time to time depending on how the

*In addition to Euclidian distances, other types of distances can also be used here.

system has been accessed and attacked, the effectiveness of the detection system at $t_i$ can represent the performance of the system over a period of time pretty well in many cases, especially when the system is accessed and attacked in a consistent way. For brevity, in the following presentation we use the term *behavior* to denote a short-term behavior.

In the model, users access the system in a unit of $m$ actions, called a *m-unit*, and the intrusion detector is thus triggered after a m-unit is executed. We assume the actions of one user are independent of those of others. Therefore, concurrent m-units can be equivalently viewed as being executed in a serial order. For simplicity, we assume the system executes one m-unit at a time. We further assume there is a *m-unit space* that consists of the set of all possible m-units of interest. Note that each m-unit in the space *belongs* to a specific user. We assume that based on the state of the system at time $t_i$ we exactly know whether the execution of a m-unit will generate an intrusion or not; and we exactly know the value of $d(\vec{v_s}, \vec{v_l})$ of the behavior resulted from the execution [†]. We finally assume that there is a specific probability for each m-unit in the space to be chosen for execution.

We use the following two measures to do the evaluation:

- The *rate of detection*, denoted $R_d$, is the conditional probability that when the execution of a m-unit generates an intrusion, the corresponding behavior is reported as an intrusion. $R_d$ indicates the general ability of a detector in detecting intrusions.
- The *rate of errors*, denoted $R_e$, is the conditional probability that a reported intrusion is actually not an intrusion. $R_e$ indicates the soundness of a detector. The smaller the $R_e$, the higher degree of soundness can be achieved.

$R_d$ and $R_e$ can be computed as follows based on the parameters listed in Table 1, which can be easily computed based on the probability for each m-unit to be chosen for execution. For example, $P_i$ equals the summary of the probabilities for one set of m-units to be chosen for execution divided by the summary of the probabilities for another set of m-units to be chosen for execution such that (1) the behavior resulted from the execution of each m-unit in the first set is an intrusion; (2) the value of $d(\vec{v_s}, \vec{v_l})$ of the behavior resulted from the execution of each m-unit in the first set is greater than or equal to $D_i$; and (3) the value of $d(\vec{v_s}, \vec{v_l})$ of the behavior resulted from the execution of each m-unit in the second set is greater than or equal to $D_i$. Here, we assume the values of $D_i$ and $D_s$ are the same for every user. The model can be easily extended to support different values of $D_i$ and $D_s$. We introduce $D_s$, $P_s$, and $A_s$ to model intrusion confinement. A formal definition of *suspicious* behaviors is given in Definition 1. Based on these parameters, from the viewpoint of the intrusion detector at time $t_i$, user behaviors, according to their values of $d(\vec{v_s}, \vec{v_l})$, can be classified into three categories which are shown in Figure 1, although as implied by $P_i$ and $P_g$, a

[†]For simplicity, we assume the behavior resulted from the execution of a m-unit $mu_i$ is independent of the behaviors resulted from the execution of the m-units preceding $mu_i$.

trustworthy behavior can be mistakenly viewed by the intrusion detector as *malicious* and a malicious behavior can be mistakenly viewed as *trustworthy*.

$R_d = \frac{A_i P_i}{A_i P_i + A_s P_s + (1 - A_i - As) P_g}$
$R_e = 1 - P_i$

**Example 1** *Assume the SSO decides that $P_i$ should be at least 0.90 to ensure a high degree of soundness. As a result, the corresponding $D_i$ can be determined to ensure such a $P_i$. However, ensuring such a high $P_i$ does not mean we can also detect most of the intrusions. If we consider the situation where $A_i = 0.03$, $A_s = 0.20$, $P_s = 0.50$, and $P_g = 0.01$, then the rate of detection is $R_d = 0.20$, which is very low. On the other hand, if we decrease the value of $P_i$ to detect more intrusions, for example, we set the value of $D_i$ to that of $D_s$, then the rate of error is $R_e = 1 - P_s = 0.50$, which is too high.*

The above example shows that in many situations if we want to achieve a low rate of errors, then we cannot achieve a high rate of detection. Therefore, the two requirements can often result in conflicting design goals. Since the rate of errors cannot be very high because otherwise substantial trustworthy service requests will be denied, a high rate of detection cannot be achieved in many cases. Thus, many intrusions can stay undetected (in Example 1, the undetected intrusion rate is 0.80), and the latency of an intrusion can be very long. As a result, serious damage can be caused.

Intrusion confinement can bound this damage. Based on the parameters listed in Table 1, the set of actions that should be isolated can be specified as follows. Note that since $D_s$ is determined by the SSO based on the value of $P_s$ he or she prefers, intrusion confinement systems can be flexibly configured.

**Definition 1** A short-term behavior described by $\vec{v_s}$ is *suspicious* if $D_s \leq d(\vec{v_s}, \vec{v_l}) < D_i$. Here, $D_s$ is determined by the value of $P_s$, which is chosen by the SSO.

By isolating suspicious behaviors, we can often protect the system from the damage caused by most of the intrusions. The effectiveness of isolation can be roughly measured by the *rate of isolation*, denoted $R_i$, which is the probability that an intrusion is either detected or isolated (note that an isolated intrusion could be detected later on). $R_i$ can be formalized as follows. It is clear that $R_i$ is never less than $R_d$. However, it should be noticed that it is not guaranteed that an isolated intrusion will always be detected later on.

$R_i = \frac{A_i P_i + A_s P_s}{A_i P_i + A_s P_s + (1 - A_i - As) P_g}$

In Example 1, if we keep $P_i$ at 0.90, then $R_i$ is 0.94.

### 2.2 Intrusion Confinement for Signature-Based Detection

We define a *signature* as a sequence of events leading from an initial limited access state to a final compromised state [PK92, Ilg93, IKP95, SG91, SG97, LWJ98]. Each event causes a transition from one state to another. We identify a signature with length $n$, denoted $Sig(n)$, as $Sig(n) = s_0 E_1 s_1 ... E_n s_n$, where $E_i$ is an event and $s_i$ is a state, and $E_i$ causes the state transition from $s_{i-1}$ to $s_i$. For simplicity, intra-event conditions are not explicitly shown in $Sig(n)$, although they are usually part of a signature.

A *partial matching* of a signature $Sig(n)$ is a sequence of events that matches a prefix of $Sig(n)$. According to previous discussion, a partial matching is usually not an intrusion. However, it can predict that an intrusion specified by $Sig(n)$ may occur. The accuracy of the prediction of a partial matching, denoted $s_0 E_1 s_1 ... E_m s_m$, can be measured by the following parameter:

$P_m$**:** the probability that the partial matching can lead to an intrusion later. Assume the number of the behaviors that match the prefix is $N_p$ and the number of the intrusions that match the prefix is $N_i$, then $P_m \approx N_i / N_p$.

Intrusion confinement in signature-based detection is necessary because by the time an intrusion is reported, serious damage may have already been caused by the intrusion. In signature-based detection, the set of actions that should be isolated is defined as follows. Isolating suspicious behaviors can surely confine damage in signature-based detection because the behavior that is actually an intrusion will, with a high probability, be prevented from doing harm to the system.

**Definition 2** In signature-based detection, a behavior is *suspicious* if it matches the prefix of a signature but not the whole signature, and $P_m$ of the prefix is greater than or equal to a threshold that is determined by the SSO.

## 3 When Should Intrusion Confinement Be Enforced?

The decision of whether to enforce intrusion confinement depends on the amount of damage that can be caused. The more serious the damage, the more efforts the SSO would like to take. In signature-based detection, the decision of whether to enforce intrusion confinement on a known attack that is specified by a signature is dependent on the seriousness of the attack and the value of $P_m$ for each prefix of the signature. For example, if the damage is strongly undesirable, and there exists a prefix whose $P_m$ is sufficiently large, then intrusion confinement can be enforced by isolating the behavior that matches the prefix.

In statistical profile-based detection, however, making this decision can be difficult. As shown in Section 2, since degrading the requirement on $R_e$ (the rate of errors) usually can improve $R_d$ (the rate of detection), the SSO may want to find a trade-off between $R_e$ and $R_d$; thus, the cost of isolation would be avoided. However, a

satisfactory trade-off may not be achievable in some systems since the relationship between these two effectiveness measures can dramatically differ from one system to another.

Consider two systems with the same set of parameters and associated values as in Example 1. When $P_i$ is degraded from 0.90 to 0.85 in both of the systems (so the value of $D_i$ must be decreased), the other parameters may take the values listed in Table 2. Here we assume the value of $D_s$ is unchanged. It is easy to get the following result: In system 1, $R_d = 0.315$ and $R_e = 0.15$; In system 2, $R_d = 0.63$ and $R_e = 0.15$. It is clear that when $P_i$ is degraded to 0.85, system 2 performs much better than system 1. At a result, the SSO may need only to enforce intrusion confinement in system 1.

It should be noted that the relationship between $R_e$ and $R_d$ can be influenced by many factors, such as the distribution of the number of intrusions on the distance $d(\vec{v_s}, \vec{v_l})$ and the distribution of the number of behaviors on $d(\vec{v_s}, \vec{v_l})$. Three typical types of relationships between $R_e$ and $R_d$ are specified as follows (here, $a$ and $b$ are real numbers):

- $R_e = 1 - ae^{(b-R_d)}$. Intrusion confinement is very necessary since a small improvement in $R_d$ can cause a significant degradation of $R_e$.
- $R_e = 1 - (a - bR_d)$. If $b$ is large, then intrusion confinement is necessary; if $b$ is small, then a satisfactory trade-off between $R_e$ and $R_d$ can be achieved.
- $R_e = 1 - a\log(b - R_d)$. A satisfactory trade-off is usually achievable since a small degradation of $R_e$ can cause a significant improvement of $R_d$.

Of course, there are many systems in which the relationship between $R_e$ and $R_d$ is none of these. However, the SSO can make a sound decision by a similar analysis.

# 4 How Can Intrusion Confinement Be Enforced?

In this section, we present a general solution for enforcing intrusion confinement in information systems. In particular, an intrusion confinement system architecture is proposed and its characteristics are investigated. It should be noticed that the architecture is not restricted to any specific type of information system.

## 4.1 Architecture Support

The architecture of an intrusion confinement system from the perspective of information warfare [AJMB97] is shown in Figure 2.

The *Policy Enforcement Manager* enforces the access controls in accordance with the system security policy on every access request. We assume no data access can bypass these access controls. We further assume that users' accesses (including updates)

will be audited in the *audit trail*. For database systems and transactional file systems, users' accesses on data *objects* such as tables and files are usually recorded in specific transaction logs maintained for recovery purposes, in addition to the audit trail. For simplicity, these logs are not explicitly shown in Figure 2. We assume that they are associated with each trustworthy or suspicious data version store.

The *Intrusion Detection and Confinement Manager* applies either statistical profile-based or signature-based detection techniques, or both to identify suspicious behavior as well as intrusions. The detection is typically processed based on the information provided by the audit trail and the logs.

Initially, when there are no suspicious or malicious transactions identified, each data object has only one *data version* (*version* for short), called the *main* data version; and all user accesses are processed on the *Main Data Version Store* where the main versions of all data objects are stored. The main version of each object is believed to be undamaged.

When a suspicious behavior is identified, the corresponding user is marked *suspicious*. At this point, first we need to deal with the effects that the behavior has already made on the Main Data Version Store because these effects may have already caused some damage. In signature-based detection systems, we can accept these effects because a partial matching is not an intrusion. In statistical profile-based detection systems, if the SSO does not think the effects can cause any serious damage, we can accept these effects. If the SSO thinks these effects can cause intolerable damage, we can isolate and move these effects from the Main Data Version Store to a separate *Suspicious Data Version Store*[‡], which is created to isolate the user. The process of isolation may need to roll back some trustworthy actions that are dependent on the actions contained in the behavior. At this point, we can apply another strategy that moves the effects of the behavior as well as the affected trustworthy actions to the Suspicious Data Version Store.

Second, the Intrusion Detection and Confinement Manager notifies the Policy Enforcement Manager to direct the subsequent suspicious actions of the user to the separate Suspicious Data Version Store. Since we focus on the isolation itself, we can simply assume that when a suspicious behavior starts to be isolated, no damage has been caused by the behavior. Since there can be several different suspicious users, e.g., $S_1$, ..., $S_n$, being isolated at the same time, multiple Suspicious Data Version Stores can exist at the same time. However, since Suspicious Data Version Stores contain only versions of the objects that are suspected to be damaged, they are usually much smaller than the Main Data Version Store.

When a suspicious user turns out to be *malicious*, that is, his/her behavior has led to an intrusion reporting, the corresponding Suspicious Data Version Store can be discarded to protect the Main Data Version Store from harm. On the other hand, when the user turns out to be innocent, the corresponding Suspicious Data Version Store is merged into the Main Data Version Store. A suspicious behavior can be malicious

[‡]A Suspicious Data Version Store is a collection of data versions which are believed to be suspicious.

in several ways: (1) In signature-based detection, a complete matching can change a behavior from suspicious to malicious; (2) Some statistics of gradual anomaly, such as frequency and total number, can make the SSO believe that a suspicious behavior is malicious; (3) The SSO can find that a suspicious behavior is malicious based on some non-technical evidence.

A suspicious behavior can be innocent in several ways: (1) In signature-based detection, when no signatures can be matched, the behavior proves innocent; (2) The SSO can prove the behavior to be innocent by some non-technical evidence. For example, the SSO can investigate the user directly; (3) Some statistics of gradual anomaly can also make the SSO believe that a behavior is innocent.

Since intrusion confinement cannot isolate every intrusion in most cases (see that the rate of isolation, $R_i$, is usually less than 1.0), intrusions do happen on the Main Data Version Store. When such an intrusion is detected (this type of intrusion is usually detected by some other approaches beyond the standard mechanisms), the corresponding user is marked as *malicious*. The Intrusion Detection and Confinement Manager then notifies the *Damage Confinement and Assessment Manager* to confine and assess the damage caused by the intrusion. The confinement can be done by notifying the Policy Enforcement Manager to reject the subsequent access of the user and to restrict the access of other users to the damaged data. For example, damaged data may be forbidden to be read for a specific period of time. However, concrete damage confinement mechanisms are beyond the scope of the paper.

After the damage is assessed, the *Reconfiguration Manager* reconfigures the system to allow access to continue in a degraded mode while repair is being done by the *Damage Recovery Manager*. In many situations damage assessment and recovery are coupled with each other closely. For example, recovery from damage can occur during the process of identifying and assessing damage. Also, the system can be continuously reconfigured to reject accesses to newly identified, damaged data objects and to allow access to newly recovered data objects. Interested readers can refer to [AJL, LAJ00] for more details on damage confinement, damage assessment, system reconfiguration, and damage recovery mechanisms in the database context.

Intrusion confinement can significantly mitigate the overhead of damage confinement, damage assessment, system reconfiguration, and damage recovery, because substantial intrusions can be isolated before they happen; thus, they will not cause damage to the Main Data Version Store. However, we need to pay the extra cost of enforcing intrusion confinement.

Damage recovery may influence the process of intrusion confinement. For example, when a set of actions is found to be malicious and requires system recovery, some actions affected by these malicious actions may have already been marked as suspicious and isolated. At this point, removing the effects of these malicious actions from the Main Data Version Store can lead to removing the effects of these affected actions from the corresponding Suspicious Data Version Store. For simplicity and to focus on isolation itself, in the rest of the paper we assume that when a suspicious behavior is isolated, there is no damage caused in the Main Data Version Store.

### 4.2 Types of Isolation and Mergers

In the normal course of events when all users are believed to be trustworthy, each data object has only one version, namely, the main data version; any updates by a user are seen by all other users of the system and vice versa. When a suspicious user $S_i$ is detected, the following types of data flow can occur between the trustworthy actions working on the Main Data Version Store and the suspicious actions of $S_i$:

**Complete Isolation:** Updates by $S_i$'s actions are not disclosed to any trustworthy action, and any updates by a trustworthy action are not disclosed to $S_i$.

**One-way Isolation:** Updates by $S_i$'s actions are not disclosed to any trustworthy action, but all updates by a trustworthy action, if needed, can be disclosed to $S_i$. When an update is disclosed to an action, the action can read the updated value. By one-way isolation, $S_i$ can read the latest value of the data versions kept in the Main Data Version Store.

**Partial Isolation:** Some updates by $S_i$'s actions are disclosed to trustworthy actions and vice versa. This situation can happen when $S_i$'s updates on some data objects are not considered risky, while anything else $S_i$ attempts to update is confined to the Suspicious Data Version Store. By partial isolation, if some of the updates by trustworthy users are considered sensitive, they could be selectively withheld from being propagated to the Suspicious Data Version Store where they would be divulged to the suspicious user. One drawback of disclosing $S_i$'s updates to trustworthy users is that after $S_i$ is revealed as malicious, some trustworthy actions affected by $S_i$'s updates may need to be backed out.

Data flows among suspicious users can also be grouped into the above three types. However, we have not found many situations where it is useful to disclose updates among Suspicious Data Version Stores, except when the SSO finds that several suspicious users cooperate with each other to do something. At this point, it is helpful that these suspicious users are isolated within one Suspicious Data Version Store because if they are malicious users who collude to do intrusions and to protect themselves from being detected, then isolating each user within a separate Suspicious Data Version Store can alert them to the fact that something is wrong. If they are innocent users, then isolating each user in a separate Suspicious Data Version Store can prevent them from communicating. However, since collusions are usually difficult to detect, and it is usually a complicated and computing intensive process to ensure the correctness of such data flows, we will not address the problem in this paper and we would like to address it in our future research.

When a user is discovered to be malicious or innocent, a decision must be made on how to accept and merge his or her updates into the Main Data Version Store. At this point, two types of mergers are possible:

**Complete Merger:** When a user is discovered to be malicious, all of his/her updates

are discarded. When a user is discovered to be innocent, all of his/her updates are considered for merging.

**Partial Merger:** Remerging of the data version stores could be partial. Even if a user were found to be a malefactor, it might be desirable to accept certain of his/her updates into the Main Data Version Store (for example, after being examined, or those on particular data items) rather than discarding all of them.

### 4.3 Identification and Resolution of Conflicts

In isolation, unless all other trustworthy or suspicious users are forbidden to update a data object that has already been updated by a suspicious user $S_i$, it is possible to have two independent updates to a data object $x$. Since we consider the restriction is unreasonable, we will take the approach that updates can be made independently by $S_i$ and other users. As a result, there can be four states associated with each data object $x$ (Let **S** denote the set of suspicious users; **G** denote the set of trustworthy users): (1) $x$ has not been updated by either **S** or **G**; (2) $x$ has been updated by **S**, but not by **G**; (3) $x$ has been updated by **G**, but not by **S**; (4) $x$ has been updated by both **S** and **G**.

Since updates can be made independently by trustworthy and suspicious users, conflicts may arise between trustworthy and suspicious actions. As a result, the Main Data Version Store and these Suspicious Data Version Stores may become inconsistent. For example, a trustworthy action and a suspicious action may update the same data object $x$; thus, neither the main version of $x$ nor the suspicious version of $x$ is correct when we decide to merge the corresponding Suspicious Data Version Store into the Main Data Version Store.

The techniques to identify and resolve these conflicts usually vary from system to system. For example, the techniques we will propose for database systems in Section 5 are quite different from the techniques we will propose for file systems in Section 6. However, we can generally classify these techniques into two categories:

**Static Resolution** allows both the main action history, i.e., the sequence of actions performed on the Main Data Version Store, and the suspicious action histories, i.e., the sequences of actions performed on the Suspicious Data Version Stores, to grow without any restrictions. Identification and resolution of conflicts are delayed until some suspicious history is designated to be merged into the main history.

**Dynamic Resolution** does not allow either the main history or the suspicious histories to grow unless the mutual consistency is guaranteed.

### 4.4 Requirements for an Isolation System

An isolation system should satisfy the following general requirements to ensure the security and correctness of intrusion confinement.

**Disclosure Property:** The isolation strategy should never cause risky data flows from Suspicious Data Version Stores to the Main Data Version Store.

**Consistency Property:** Before and after each merger, the Main Data Version Store and the Suspicious Data Version Stores should be kept consistent. The consistency of the Main Data Version Store can be relaxed when some suspicious updates are disclosed to it. The consistency of a Suspicious Data Version Store can be relaxed when some trustworthy updates are disclosed to it.

**Synchronization Property:** Suspicious histories should be isolated in a way such that the merging of one history into the main history will not make some conflicts between the merged main history and some other suspicious history unidentifiable or unresolvable.

## 5 Intrusion Confinement in Database Systems

In this section, we will present a concrete isolation protocol in the database system context to evaluate the feasibility of our general intrusion confinement solution.

### 5.1 Overview

In the protocol, a data item $x$ has only one trustworthy version and may have multiple suspicious versions. We use some specific *version numbers* to distinguish one version from another. One and only one suspicious version of $x$ is produced for each suspicious user that has updated $x$. If $x$ has never been updated by a suspicious user, it has no suspicious versions. However, all these versions are stored in the same physical database for efficiency. We conceptually break down the database into several version stores. The Main Version Store is composed of the trustworthy version of each data item in the database. We maintain a different Suspicious Version Store for every suspicious user under isolation. The Suspicious Version Store for user $S_i$ is composed of all and only the suspicious versions produced for $S_i$. A suspicious version of a data item $x$ is produced for $S_i$ when $S_i$ first updates $x$.

Users access the database through *transactions*. Isolation is achieved by controlling the access of transactions to versions. For example, under one-way isolation, trustworthy transactions can only read and update trustworthy versions; suspicious transactions can read but not update trustworthy versions. In particular, transactions of a suspicious user $S_i$ can only update the suspicious versions produced for $S_i$. When a transaction of $S_i$ wants to read a data item $x$, if a suspicious version of $x$ has not been

produced for $S_i$, then the trustworthy version of $x$ is read. Otherwise, the suspicious version is read.

When a suspicious user $S_i$ is proved malicious, the Suspicious Version Store maintained for $S_i$ will be deleted. However, if $S_i$ is proved innocent, then the updates of $S_i$'s transactions should be merged back into the Main Version Store. Since independent updates could be performed by both $S_i$ and a trustworthy user on the same data item (but different versions) after $S_i$ was isolated, the Suspicious Version Store for $S_i$ and the Main Version Store may be *inconsistent*. Hence we cannot correctly merge these two version stores by simply forwarding the updates of one version store to the other. In order to achieve a consistent merging, we use a specific graph (called a *precedence graph*), which is built from the transactions that commit within these two version stores after $S_i$ was isolated, to figure out how to back out some specific transactions from the Main Version Store or the Suspicious Version Store so that the two resulted version stores can be consistent, and update-forwarding can be used to correctly merge them.

The protocol also supports dynamic resolution. For this purpose, we need to maintain the precedence graph on-the-fly, and use the graph to dynamically maintain the consistency between these two version stores. As a result, when a merging needs to be done, the two involved version stores may have already been consistent.

## 5.2 Isolation Model

In the model, a database is specified as a collection of data *items* (objects). The database *state* is determined by the values of these data items. Data items are operated by *transactions*. A transaction is a partial order of read and write *operations* that either commits or aborts. Two operations *conflict* if one is write.

The execution of a set of transactions is modeled by a *history*, which is a partial order $(\sum, <_H)$, where $\sum$ is the set of all operations executed by these transactions, and $<_H$ indicates the execution order of those operations. Two histories are *equivalent* if (1) they are defined over the same set of transactions and have the same operations, and (2) they order conflicting operations of nonaborted transactions in the same way. A history $H$ is *serial* if, for any two transactions $T_i$ and $T_j$ that appear in $H$, either all operations of $T_i$ appear before those of $T_j$ or vice versa. A history $H$ is *serializable* if its committed projection is equivalent to a serial history. We assume that the DBMSs employ a concurrency control protocol that produces serializable histories [BHG87]. For simplicity, we assume also that the read set of a transaction always contains its write set.

We specialize the solution proposed in Section 4 as follows: (1) We assume that there are no data flows among Suspicious Data Version Stores. (2) We assume that one-way isolation is the isolating strategy chosen by the SSO. (3) We assume that complete merger is the merging strategy chosen by the SSO. Note that the isolation protocol presented next can be easily extended to incorporate other kinds of isolating and merging strategies.

In the model, a behavior of a user is a sequence of transactions submitted by the user. Containing such actions as read, write, commit and abort, each transaction causes a transition from one database state to another. When a suspicious behavior is detected, the corresponding user is marked suspicious, and the Intrusion Detection and Confinement Manager notifies the Policy Enforcement Manager to direct the subsequent transactions submitted by the user to a separate Suspicious Data Version Store, namely, a separate database. The transactions executed on this store generate a *suspicious* history. In contrast, the transactions executed on the Main Data Version Store, namely, the main database, generate a *trustworthy history*. At one point of time, there can be multiple suspicious histories but only one trustworthy history, called the *main history*. When a suspicious user turns out to be malicious, the corresponding suspicious history is discarded to protect the Main Data Version Store from harm. On the other hand, when the user turns out to be innocent, the history is merged into the main history.

For clarity, we present our isolation solution in the database context by two steps. Step 1, which is described in Section 5.3, deals only with the situations when there is only one suspicious user identified. Step 2, which is described in Section 5.4, extends Step 1 to isolate multiple suspicious users at the same time.

## 5.3 Isolating a Single Suspicious User

### 5.3.1 Isolation Protocol

When there is only one suspicious user identified, the one-way isolation strategy can be achieved by the following protocol where each data item may have one or two *versions* and each version is identified by a specific *version number*. Note that: 1) Each data item has a `MAIN` version; all these `MAIN` versions compose the Main Data Version Store. 2) Every item that has been updated by a suspicious user $S_i$ has a $t_i$ version; all these $t_i$ versions compose the Suspicious Data Version Store for $S_i$. 3) Trustworthy transactions will never read a version associated with a time stamp version number, however, a suspicious transaction can read `MAIN` versions.

**Protocol 1** Isolating a Single Suspicious User

- Before a database system starts to run transactions, each data item $x$ has only one version which is associated with the same version number `MAIN`, denoted $x$[`MAIN`].
- When a trustworthy transaction $T$ wants to read or update $x$, $x$[`MAIN`] is given to $T$.
- When a transaction submitted by a suspicious user $S_i$ wants to update $x$,
  - If $x$ has only the `MAIN` version, then first an additional version of $x$, which is associated with a unique version number, e.g., the time stamp (denoted

$t_i$) generated when $S_i$ was found suspicious, is created by copying the value of $x$[MAIN]. The $t_i$ version is then given to $S_i$ to do updates.

- Otherwise, the $t_i$ version must exist, and it is given.

- When a transaction submitted by a suspicious user $S_i$ wants to read $x$, if there is a $t_i$ version of $x$, then the $t_i$ version is given. Otherwise, the MAIN version is given.

### 5.3.2 Static Identification and Resolution of Conflicts

Since a data item can be updated by both trustworthy and suspicious users, trustworthy transactions and suspicious transactions can conflict with each other during isolation. In this section, we propose a method to identify and resolve the possible conflicts between the main history and the history on a Suspicious Data Version Store. The method is adapted from Davidson's protocol [Dav84], which is used in partitioned distributed database systems.

In distributed database systems, groups of sites can be partitioned by communication failures. Davidson's optimistic protocol [Dav84] allows transactions to be executed within each partitioned group independently. As a result, a serializable history is generated within each partition. We denote the equivalent serial history of the history generated within partition $P_i$ as $H_i$. When two partitions $P_1$ and $P_2$ are reconnected, Davidson's protocol builds a *precedence graph* using $H_1$ and $H_2$, denoted $G(H_1, H_2)$, to identify and resolve the conflicts between these two partitioned histories. Davidson proved that if there is no cycle in $G(H_1, H_2)$, then $H_1$ and $H_2$ can be merged without conflicts. The merging is done by forwarding the updates of $H_1$ to $P_2$ or by forwarding the updates of $H_2$ to $P_1$.

Viewing the suspicious history and the suffix of the main history which starts when the Suspicious Data Version Store becomes non-empty as two partitioned histories, Davidson's protocol can be directly used in complete isolation because partitions are totally isolated. However, since our protocol is a one-way isolation protocol, we need to deal with the data flows from the Main Data Version Store to the Suspicious Data Version Store, which are not addressed in [Dav84]. It should be noticed that here a specific suffix of the main history instead of the whole main history is viewed as a partitioned history. The reason is that the corresponding prefix of the main history will never conflict with the suspicious history. For brevity, we call the specific suffix the *main history* in the following presentation.

In our approach, we build a similar precedence graph, which has one more type of edge than that in [Dav84] based on the main history and the suspicious history when the suspicious history turns out to be innocent as follows:

- Let $T_i$ and $T_j$ be two suspicious transactions or two trustworthy transactions that perform conflicting operations on a data item. There is a directed edge $T_i \rightarrow T_j$ if $T_i$ precedes $T_j$.

- If an update of a trustworthy transaction $T_g$ was disclosed to a suspicious transaction $T_s$ during the isolation, then there is a directed edge $T_g \rightarrow T_s$. This type of edge is called a *read edge*. We add read edges to the traditional precedence graph to support one-way isolation.
- Let $T_g$ be a trustworthy transaction that reads a data item that has been updated by a suspicious transaction $T_s$, and there is no path from $T_g$ to $T_s$, then there is a directed edge $T_g \rightarrow T_s$.
- Let $T_s$ be a suspicious transaction that reads a data item that has been updated by a trustworthy transaction $T_g$, and there is no path from $T_g$ and $T_s$, then there is a directed edge $T_s \rightarrow T_g$.

**Example 2** *Consider the five transactions given below:*

$READSET(T_{g1}) = WRITESET(T_{g1}) = \{d_1, d_3\}$
$READSET(T_{g2}) = WRITESET(T_{g2}) = \{d_1, d_4\}$
$READSET(T_{g3}) = \{d_2, d_4, d_5\}$, $WRITESET(T_{g3}) = \{d_5\}$
$READSET(T_{s1}) = WRITESET(T_{s1}) = \{d_7\}$
$READSET(T_{s2}) = \{d_1, d_2, d_3, d_7\}$, $WRITESET(T_{s2}) = \{d_2, d_3\}$
$READSET(T_{s3}) = \{d_3, d_6\}$, $WRITESET(T_{s3}) = \{d_6\}$

Assume that the trustworthy history is $H_g = T_{g1}\ T_{g2}\ T_{g3}$ and the suspicious history is $H_s = T_{s1}\ T_{s2}\ T_{s3}$. The precedence graph $G(H_g, H_s)$ is shown in Figure 3. Since the graph has a cycle, conflicts exist among the transactions. For example, since $T_{s2}$ reads item $d_1$, which is updated by $T_{g2}$, $T_{s2}$ should precede $T_{g2}$; since $T_{g2}$ should precede $T_{g3}$, $T_{s2}$ should precede $T_{g3}$; however, since $T_{g3}$ reads item $d_2$, which is updated by $T_{s2}$, $T_{g3}$ should precede $T_{s2}$, yielding a contradiction.

This example also shows that incorporating read edges can decrease the number of cycles in the precedence graph in many situations. Thus, it can mitigate the inconsistency between the Main Data Version Store and the Suspicious Data Version Store in these situations. In the example, if no read edges are allowed and we take a complete isolation strategy, then there will be four more cycles in the graph: the first includes $T_{g1}$ and $T_{s2}$; the second includes $T_{g1}$, $T_{s2}$, and $T_{s3}$; the third includes $T_{g1}$, $T_{g2}$, $T_{g3}$, and $T_{s2}$; the fourth includes all of the five transactions.

**Example 3** *Suppose we change the read set of $T_{g3}$ to $\{d_4, d_5\}$. Suppose further that the database executes the transactions in the following order: $T_{s1}T_{g1}T_{g2}T_{s2}T_{s3}T_{g3}$.*

The resulting conflict graph, shown in Figure 4, is acyclic, denoting that there are no conflicts among these transactions. In fact, the history $H = T_{s1}\ T_{g1}\ T_{s2}\ T_{g2}\ T_{s3}\ T_{g3}$ is an equivalent merged history that can generate the same database state as the state generated by merging $H_s$ into $H_g$, i.e., forwarding the updates of $H_s$ to the Main Data Version Store. However, it should be noticed that the initial executing order of these transactions does not construct a correct merged history.

The correctness of the approach is shown in the following theorem. Since the proof is similar to that of Theorem 2.2.2 in [Dav84], it is omitted.

**Theorem 1** Given $H_g$ and $H_s$, the precedence graph $G(H_g, H_s)$ is acyclic if and only if there is an equivalent merged history $H$ that can generate the same database state as generated by merging $H_s$ into $H_g$, i.e., forwarding the updates of $H_s$ to the Main Data Version Store.

In order to build the precedence graph, we need to keep track of read and write sets of each transaction $T$ in $H_g$ or $H_s$. We can get the write set of $T$ from the logs associated with the Main Data Version Store or the Suspicious Data Version Store, where every write operation of $T$ is recorded; we can get the read set of $T$ by several ways, such as augmenting the write log with read information (read records); extracting read sets from the profiles of transactions; extracting read information from physical or logical logs, etc. The discussion on these approaches is out of the scope of the paper.

Moreover, we need to keep track of every read edge. This can be done as follows: When a suspicious transaction reads an item $x$ from the Main Data Version Store, we search the log associated with the Main Data Version Store to get the latest record where $x$ was updated. The identifier of the trustworthy transaction that did the update is maintained in the record. If the trustworthy transaction is in the main history, then a read edge from the trustworthy transaction to the suspicious transaction is captured.

After some conflicts are identified, we usually need to back out a number of committed suspicious or trustworthy transactions so that the conflicts could be resolved. For example, the conflict shown in Figure 3 can be resolved by backing out $T_{g3}$. After $T_{g3}$ is backed out, a correct merged history can be $H = T_{s1}\ T_{g1}\ T_{s2}\ T_{g2}\ T_{s3}$. The merger can then simply be done by forwarding the updates of $T_{s1}$, $T_{s2}$, and $T_{s3}$ to the Main Data Version Store.

Unfortunately, it is shown in [Dav84] that just minimizing the number of transactions backed out is NP-complete, let alone minimizing the total back-out cost, i.e., assigning each transaction a weight or back-out cost and then minimizing the total back-out cost. Although this result discourages attempts to minimize the total back-out cost, the simulation results of [Dav84] show that in many situations where the size of transactions is small and there is a relatively small number of transactions, a relatively large number of data-items in the database and large percentage of read-only transactions, several back out strategies, in particular *breaking two-cycles optimally*, can achieve good performance. Interested readers can refer to [Dav84] for more details on these back out strategies.

### 5.3.3 Dynamic Identification and Resolution of Conflicts

The static resolution approach has several drawbacks: (1) The performance of normal transaction processing can be seriously degraded during the merger since the identification and resolution can spend a substantial amount of system resources, although

after the identification, transactions accessing other data items may safely execute. (2) Minimizing the number of transactions backed out is NP-complete and thus is computation intensive. (3) In many cases, when a transaction $T$ is backed out to break a cycle in the precedence graph all the transactions that are affected, directly or indirectly by $T$, have to be backed out at the same time. For example, in Example 2 if $T_{g2}$ is chosen to be backed out, then $T_{g3}$ needs to be backed out since it reads the item $d_4$ after it is updated by $T_{g2}$.

In this section, we adopt a dynamic approach to identify and resolve conflicts. The dynamic approach works as follows. The basic idea is that neither a trustworthy nor a suspicious transaction is allowed to commit unless we can ensure that the transaction will not introduce a cycle in the precedence graph; thus, conflicts are resolved as they arise. Note that dynamic resolution cannot be enforced in [Dav84].

1. We build an online precedence graph $G(H_g, H_s)$ based on the growing $H_g$ and $H_s$ in the same way as the static approach.

2. When a suspicious transaction $T_s$ is going to commit, we first check if $T_s$ will introduce a cycle to the precedence graph. If so, then $T_s$ is rolled back; if not, then $T_s$ is committed and added to $H_s$.

3. When a trustworthy transaction $T_g$ is going to commit, we first check if $T_g$ will introduce a cycle to the precedence graph. If not, then $T_g$ is committed and appended to $H_g$; if so, then either $T_g$ needs to be backed out or a set of suspicious transactions need to be backed out. The SSO can make the choice based on the costs of these two strategies. For example,

   - Assume the set of suspicious transactions with the minimum total weights that can be backed out to remove all the new cycles caused by $T_g$ is **TS**;
   - Assume the *suspicion factor* of $H_s$, a real number between 0 and 1 which measures the extent of the suspicion that the system has on $H_s$, is $\alpha$;
   - Let $Weight(T_g)$ denote the weight of $T_g$, then if $Weight(T_g) \geq (1-\alpha)Weight(\mathbf{TS})$, that is, the cost of rollbacking $T_g$ is greater than or equal to the cost of backing out $TS$, then $TS$ is backed out; otherwise, $T_g$ is rolled back.

To illustrate, assume that in Example 2 the system executes transactions in the following order: $T_{s1}T_{g1}T_{s2}T_{g2}T_{s3}T_{g3}$, when $T_{g3}$ is going to commit, we will find that $T_{g3}$ will bring a cycle to the precedence graph (shown in Figure 3). At this point, we can either roll back $T_{g3}$ or back out $T_{s2}$ and $T_{s3}$ to remove the cycle. Assume the weight for each transaction is 1; if the suspicion factor of $H_s$ is less than 0.5, then $T_{g3}$ will be rolled back; otherwise, $T_{s2}$ and $T_{s3}$ will be backed out.

Compared with static resolution, dynamic resolution can support flexible strategies in resolving conflicts and can give users more availability and less service delay. Although dynamic resolution may introduce extra costs when $H_s$ turns out to be malicious, for example, the cost of maintaining the precedence graph for $H_s$, the merging of $H_s$ into $H_g$ can be done with almost no delay, and transactions can be processed

with almost no wait when $H_s$ turns out to be innocent. In addition, the complexity of the static approach can be dramatically decreased, since in dynamic resolution when a potential cycle is detected it can be removed in a very simple way, for example, backing out the transaction that causes the cycle.

For $H_g$ and $H_s$, the costs of back-out, i.e., the total number of transactions that need to be backed out to merge these two histories, in either static resolution or dynamic resolution, depend on the access characteristics of the transactions in $H_g$ and $H_s$ and the interleaving of the trustworthy transactions in $H_g$ and the suspicious transactions in $H_s$. For example, if the transactions that may cause cycles are executed earlier in each history, then dynamic resolution may back out fewer transactions; however, if there are many situations where several transactions are responsible for one cycle, then the static approach may achieve better results.

## 5.4 Isolating Multiple Suspicious Users

The complexity of isolating multiple suspicious users lies in two facts: (1) Different suspicious users are usually identified and isolated at different points of time. (2) As clarified in Section 5.4.1, the merging of one suspicious history into the main history, in either static or dynamic resolution, can make some other active suspicious histories invalid, if these histories are not properly tailored or synchronized. In this section, we investigate the negative impact that merging of one history may have on other suspicious histories. Corresponding algorithms to remove this impact are also presented.

### 5.4.1 Impact of Merging on Active Suspicious Histories

The negative impact of the merging of one history on other suspicious histories is twofold: First, after the merging, the main history usually contains more transactions. As a result, more conflicts can exist among the main history and the other suspicious histories after the merging. In static resolution, these conflicts can be identified and resolved in the same way as described in Section 5.3.2 when any of these suspicious histories is going to be merged. Strategies to handle these conflicts in dynamic resolution will be addressed in Section 5.4.3.

Second, the merging of one history can make some other suspicious histories invalid in two possible situations: One is that during a merging the back-out of a trustworthy transaction $T_g$ can make invalid all of the other suspicious histories that have one or more read edges from $T_g$. The reasons are that after $T_g$ is backed out, its updates will become invalid, and the suspicious transactions pointed to by these read edges can also become invalid because they have read the results of some of these invalid updates. For similar reasons, all the other suspicious transactions that have read, directly or indirectly, from these suspicious transactions can also become invalid.

Under this situation, we can tailor each invalid history as follows to make it valid again. Since backing out a trustworthy transaction during a merging can cause extra

back-outs in other suspicious histories, we must make sure that these extra costs are taken into account when we plan to back out a trustworthy transaction.

- When $T_g$ is backed out, for each active suspicious history $H_s$ which has a read edge from $T_g$, and for each such read edge, denoted $T_g \rightarrow T_s$
  - Back out $T_s$.
  - Back out each suspicious transaction in $H_s$ which is *dependent on* $T_s$, i.e., in $T_s$'s *reads-from* closure[§].
  - In dynamic resolution, $T_s$'s *reads-from* closure can be obtained directly from the precedence graph maintained on-the-fly for $H_s$. In static resolution, a *dependence graph* of $H_s$, which contains all and only the reads-from edges among transactions in $H_s$, can be instead maintained on-the-fly to enable quick identification of the closure. Although this may cause extra costs when a suspicious history turns out to be malicious, the costs are usually much less than those caused in dynamic resolution because dependence graphs are usually much simpler to maintain than precedence graphs. Moreover, dependence graphs can be used directly later to build precedence graphs when mergings are needed.

The other situation is shown in Figure 5. Assume that the main history $H_g$ began at time $t_0$, suspicious history $H_{s1}$ began and ended at times $t_1$ and $t_3$ respectively, suspicious history $H_{s2}$ began and ended at times $t_2$ and $t_4$ respectively, and both $H_{s1}$ and $H_{s2}$ were active during the period of time from $t_2$ to $t_3$. Assume that transaction $T_1$ in $H_{s1}$ was executed before $T_2$ in $H_g$, and $T_2$ was executed before $T_3$ in $H_{s2}$. Assume the read and write sets of $T_1$, $T_2$, and $T_3$ are as follows.

$$READSET(T_1) = \{d_1, d_2\},\ WRITESET(T_3) = \{d_2\}$$
$$READSET(T_2) = \{d_1\},\ WRITESET(T_2) = \{d_1\}$$
$$READSET(T_3) = \{d_1, d_2, d_3\},\ WRITESET(T_3) = \{d_3\}$$

We further assume that from $t_1$ to $t_3$, $d_2$ had only been updated by $T_1$, and $d_1$ had only been updated by $T_2$. Then after $H_{s1}$ was merged into $H_g$ at time $t_3$, $T_1$ will be merged into $H_g$ in a position preceding $T_2$ because $T_1$ read $x$ before $x$ was updated by $T_2$ (here we assume neither $T_1$ nor $T_2$ will be backed out). At this point, $H_g$ between $t_1$ and $t_2$ is changed. And from the viewpoint of the new $H_g$, $T_3$ should have read $d_2$ from $T_1$ according to one-way isolation because $T_3$ had read $d_1$ from $T_2$ and $T_1$ was serialized before $T_2$. However, $T_3$ had not read $d_2$ from $T_1$ because $T_1$ was not in $H_g$ when $T_3$ read $d_2$. This makes $T_3$ invalid in terms of one-way isolation. This situation is also denoted as the *phantom* problem. See that after $H_{s1}$ is merged, from the viewpoint of the system (the main history), $T_3$ is a phantom that should have not been executed.

[§]Within a single history, a transaction $T_i$ reads $x$ from $T_j$ if $T_j$ reads $x$ after $T_i$ has updated $x$ and there are no transactions that update $x$ between the time $T_i$ updates $x$ and $T_j$ reads $x$. $T_i$ reads from $T_j$ if $T_i$ reads an item from $T_j$.

The characteristics of phantoms (also denoted phantom transactions) are specified by the following definition.

**Definition 3** A transaction $T_s$ in a suspicious history $H_s$ is a *phantom* with respect to the main history $H_g$ if and only if there is a transaction $T_g$ in $H_g$ such that

1. $T_s$ is dependent on a transaction $T_{sk}$ in $H_s$ to which an update of $T_g$ was disclosed; or $T_s$ itself is $T_{sk}$.
2. $T_s$ had read an item $x$ that had been updated by another transaction $T_{gk}$ in $H_g$, which is serialized before $T_g$, but the update of $T_{gk}$ on $x$ was not disclosed to $T_s$.

Fortunately, we found that the phantom problem can be tackled in both static and dynamic resolution, as illustrated in Sections 5.4.2 and 5.4.3, respectively.

### 5.4.2 Extension for Static Resolution

We found that phantom transactions can be detected and removed using the precedence graph. For the example shown in Figure 5, after $H_{s1}$ is merged into $H_g$, the part of the precedence graph $G(H_g, H_{s2})$ that contains $T_1$, $T_2$, and $T_3$ can be as shown in Figure 6. We build the precedence graph in the same way as specified in Section 5.3.2. Note that there is no read edge from $T_1$ to $T_3$ because $T_3$ had not read $d_2$ from $T_1$. It is easy to observe that the precedence graph has a cycle involving a read edge. This is special because when isolating a single suspicious user no precedence graph will have such cycles. This also raises the question of whether every phantom transaction is included in a cycle containing read edges, which is answered by the following theorem.

**Theorem 2** When a suspicious history $H_s$ is merged into the main history $H_g$, a transaction $T_s$ in $H_s$ is a phantom transaction if and only if in the precedence graph $G(H_g, H_s)$

- $T_s$ is included in a cycle containing a single read edge (Assume $T_{gi}$ is the source transaction of the read edge), and
- Within the cycle, there is an edge from $T_s$ to a transaction in $H_g$, and there is a path from the transaction to $T_{gi}$ which includes only transactions in $H_g$.

**Proof:** *Only If:* According to Definition 3 and the way we build the precedence graph, since there must be a trustworthy transaction $T_g$ such that an update of $T_g$ is disclosed to $T_s$ or a transaction on which $T_s$ is dependent, there must be a path from $T_g$ to $T_s$ in $G(H_g, H_s)$. It is clear that the path can contain only one read edge, denoted $(T_{gi}, T_{sj})$, and no cycles. Note that $T_{gi}$ can be $T_g$ or a transaction in $H_g$ serialized after $T_g$; $T_{sj}$ can be $T_s$ or a transaction in $H_s$ serialized before $T_s$. Since $T_s$ must have read an item $x$ which had been updated by a transaction $T_{gk}$ in $H_g$ which is serialized

before $T_g$ but the update of $T_{gk}$ on $x$ was not disclosed to $T_s$, there must be an edge from $T_s$ to $T_{gk}$, and a path from $T_{gk}$ to $T_g$, which contains only transactions in $H_g$. These three paths compose the cycle where $T_s$ is included.

*If:* We denote the read edge contained in the cycle where $T_s$ is included as $(T_{gi}, T_{sj})$. We denote the specific edge from $T_s$ to $H_g$ as $(T_s, T_{gl})$. According to Definition 3, we need to find three specific transactions $T_g$, $T_{sk}$, and $T_{gk}$. Here we show that $T_{gi}$, $T_{sj}$, and $T_{gl}$ can just be the three transactions, respectively. Since there is an edge from $T_s$ to $T_{gl}$, $T_s$ had read an item which had been updated by $T_{gl}$, and $T_s$ did not read the item from $T_{gl}$. Since there is a path from $T_{gl}$ to $T_{gi}$ which includes only transactions in $H_g$, $T_{gl}$ is serialized before $T_{gi}$. Therefore, condition 2 of Definition 3 holds. In order to show that condition 1 of Definition 3 holds, we need to show that $T_s$ is dependent on $T_{sj}$ or $T_s$ itself is $T_{sj}$ (See that since there is a read edge from $T_{gi}$ to $T_{sj}$, an update of $T_{gi}$ is disclosed to $T_{sj}$). If $T_s$ is $T_{sj}$, then the proof is done. Otherwise, showing that $T_s$ is serialized after $T_{sj}$ can also complete the proof. Since $T_s$ and $T_{sj}$ are in the cycle, a serial order must exist between them. Suppose $T_s$ is serialized before $T_{sj}$, then in the cycle $T_s$ has two paths to $T_{sj}$: one is through $T_{gl}$ and $T_{gi}$, and the other is through some transactions in $H_s$. This contradicts the definition of cycles. □

Theorem 2 justifies our observation and provides a way to detect phantom transactions. The following theorem goes further, showing us during a merging how to remove the detected phantom transactions and resolve the conflicts between the main history and histories containing phantom transactions. Since the proof is similar to that of Theorem 2.2.2 in [Dav84], it is omitted.

**Theorem 3** When a suspicious history $H_s$ is merged into the main history $H_g$,

- The precedence graph $G(H_g, H_s)$ is acyclic only if $H_s$ does not contain any phantom transactions.
- $G(H_g, H_s)$ is acyclic if and only if there is an equivalent merged history $H$ which can generate the same database state as generated by merging $H_s$ into $H_g$, i.e., forwarding the updates of $H_s$ to the Main Data Version Store.

### 5.4.3 Extension for Dynamic Resolution

In dynamic resolution, the merging of one suspicious history $H_{s1}$ can cause some transactions in another suspicious history $H_{s2}$ to become phantoms. In the example shown in Figure 5, before $H_{s1}$ is merged at time $t_3$, $T_3$ is not a phantom. However, after $H_{s1}$ is merged, $T_3$ becomes a phantom. As shown in Theorem 2, these phantoms will introduce cycles to $G(H_g, H_{s2})$, which is maintained on-the-fly. However, more new cycles can be introduced to $G(H_g, H_{s2})$ by the normal conflicts (between $H_g$ and $H_{s2}$) introduced by the merging. To deal with the two types of cycles, several strategies can be taken after $H_{s1}$ is merged:

1. Break all the cycles introduced to $G(H_g, H_{s2})$ before any new transaction is processed. It should be noticed that cycles usually cannot be broken concurrently for multiple precedence graphs, since different sets of transactions in $H_g$ can be backed out in breaking the cycles of different precedences graphs. However, in order to enable concurrent and quick cycle breaking, we can allow no trustworthy transactions in $H_g$ to be backed out during cycle breaking, although this may cause more back-out costs.

2. When some cycles are introduced to $G(H_g, H_{s2})$, leave them alone until the time when $H_{s2}$ must be merged or discarded.

3. When some cycles are introduced to $G(H_g, H_{s2})$, remove only part of the cycles.

Strategy 1 may suspend normal transaction processing when a merging is done; however, it can ensure that every merging can be done with almost no delay. Strategy 2 may allow transactions to be executed continuously after a merging is done; however, at the time when a suspicious history is merged into the main history, there can be cycles in the precedence graph, and breaking these cycles may cause significant service delay. Strategy 3 is a trade-off between strategies 1 and 2.

Compared with static resolution, in dynamic resolution a merging seems to have more serious impact on other suspicious histories. However, dynamic methods may still be able to enhance the performance of static methods to some extent, since usually most of the cycles in a precedence graph will be removed during normal processing of transactions.

### 5.5 Implementation Issues

The isolation protocol we have presented above can be easily implemented by adapting a multi-version DBMS. We assume the multi-version DBMS has a classic three module structure [BHG87] where (1) a *Transaction Manager* performs any required preprocessing of database and transaction operations it receives from transactions; (2) a *Scheduler* controls the relative order in which database and transaction operations are executed; (3) a *Data Manager (DM)* manages the cache and uses a transaction log to ensure atomicity and durability.

The multi-version DBMS can be adapted as follows to support intrusion confinement:

- The Scheduler is informed by the Policy Enforcement Manager a list of current suspicious users, including the times when these users were isolated (the list is refreshed whenever a user is identified suspicious and whenever a suspicious user is proved malicious or innocent). The Scheduler also maintains for each Suspicious Version Store a list of the data items that have a version in the store.

- Instead of producing a new *version* of a data item $x$ for each Write on $x$, the DM produces a new version of a data item $x$ only for the first Write of a suspicious

user on $x$. For each Write on $x$, the Scheduler not only decides when to send the Write to the DM, but it also tells the DM whether to produce a new version or not. If the answer is `YES`, the Scheduler tells the DM the corresponding version number. If the answer is `NO`, the Scheduler tells the DM which one of the versions of $x$ to write. Similarly, for each Read on x, the Scheduler decides both when to send the Read to the DM and which one of the versions of $x$ the DM should read. It should be easy to see that the Scheduler can correctly make these decisions based on the concurrency control algorithm and the lists it maintains.

- The Scheduler needs not to achieve *one-copy serializability* because (1) each data item $x$ has only one trustworthy version; and (2) each suspicious user has a separate history. Instead, the Scheduler needs only a concurrency control algorithm to achieve serializability for the main history and each suspicious history.

- An *Isolation Manager (IM)* is added for two purposes: (1) When a suspicious user $S_i$ is proved malicious, the IM constructs a specific transaction and submits it to the DBMS to delete all the suspicious versions produced for $S_i$; (2) When a suspicious user $S_i$ is proved innocent, the IM merges the updates of $S_i$ back into the Main Version Store after it identifies and resolves all the conflicts between $S_i$'s transactions and the main history based on a specific precedence graph. In static resolution, the graph is built by the IM based on (1) the transaction log where both the main history and each suspicious history are recorded; and (2) the read edges maintained by the Scheduler. While in dynamic resolution the Scheduler is responsible for maintaining the online precedence graph. When a set of transactions is determined to be backed out to resolve these conflicts, the IM constructs a set of specific transactions and submits them to the DBMS to semantically remove the effects of these transactions (Note that these transactions cannot be directly rolled back because they are already committed). Finally, in order to merge the two resulted consistent version stores, the IM constructs a set of specific transactions and submits them to the DBMS to forward updates. Note that these transactions are executed by the DBMS just as normal trustworthy transactions.

  When multiple suspicious users are isolated at the same time, the IM is also responsible for (1) identifying and removing phantom transactions; and (2) removing all the other negative impact that the merging of $S_i$'s history may have on other active suspicious histories.

# 6 Intrusion Confinement in File Systems

In this section, we will present a concrete isolation protocol in the file system context to evaluate the feasibility of our general intrusion confinement solution.

## 6.1 Isolation Protocol

Consider a file system with a set of files, denoted $f_1$, $f_2$, ..., $f_n$, that is accessed by users. These files can be of many types, such as normal files, directories, and devices. A user can access a file in various ways, such as reading, writing, modifying, renaming, deleting, copying, and moving. Each file access is modeled as an action.

In this section, we choose one-way isolation as the isolation strategy and complete merger as the merging strategy, and we forbid data flows among Suspicious Data Version Stores.

The isolation protocol specified below is adapted from [PPR+83], in which a protocol is proposed to detect and resolve mutual inconsistencies in distributed file systems. In this protocol, the isolation is processed in terms of each file. A file $f_i$ always has a main version (unless it is deleted) that is believed to be undamaged, and it may have several isolated versions depending on how many suspicious users have updated the file. When $f_i$ is modified by a suspicious user $S_i$, the modification and the possible following modifications of $S_i$ on $f_i$ will be isolated until $S_i$ proves to be malicious or innocent. To identify the conflicts between the modifications of $S_i$ on $f_i$ and the modifications of trustworthy users on $f_i$, we associate a specific *version vector* with the main version of $f_i$ and every isolated version of $f_i$. For two versions of a file $f_i$, if their version vectors are *compatible*, then they do not conflict. Otherwise, they conflict with each other, and either manual or automatic resolution is needed. Note that if a file has never been modified by a suspicious user, then it has no associated version vectors.

**Protocol 2** Isolation Protocol for a File System

- Each file $f_i$ is associated with a system-wide, unique identifier, called the *origin point* (denoted $OP(f_i)$), which is generated when $f_i$ is created. It is an immutable attribute of $f_i$, although $f_i$'s name is not immutable (indeed, $f_i$ may have different names in different versions). Thus, no number of modifications or renamings of $f_i$ will change $OP(f_i)$.

- At the very beginning when there are no suspicious users, the version vector of the main version of a file $f_i$ can be viewed as $< G : 0 >$, although such a vector does not exist until $f_i$ is updated by a suspicious user. Here, $G$ denotes the main version.

- When a file $f_i$ is first modified by a suspicious user (denoted $S_1$), an isolated version of $f_i$ (with the same $OP(f_i)$), called the $S_1$-*version* of $f_i$, is created for $S_i$ to perform the modification. As a result, two different versions of $f_i$ exist after the modification: one is the main version, with no effects of the modification on it; the other is the $S_1$-version on which the modification is performed. We generate the corresponding version vectors as follows: (1) For the main version, $< G : 0, S_1 : 0 >$ is created as the version vector. $S_1 : 0$ signifies that the version has not been modified by $S_1$. $G : 0$ signifies that the version has not been modified by trustworthy users since $f_i$ is first modified by a suspicious user

(here the user is $S_1$). (2) For the $S_1$-version, $< G : 0, S_1 : `m' >$ is created as the version vector. The $G$ dimension $(G : 0)$ is copied from the main version vector. $S_1 : `m'$ signifies that $f_i$ has been *m*odified by $S_1$. Note that $`m'$ is not a number. The $S_1$ dimension remains to be $`m'$ no matter how many times the $S_1$-version is modified.

- When a trustworthy user asks to delete a file $f_i$, if the main version has already been deleted, i.e., the main version does not exist, then the user will be informed that $f_i$ does not exist. Otherwise,
  - If the main version vector does not exist, then the main version is directly deleted.
  - If the main version vector exists, then its $G$ dimension is changed to $G : `d'$, which indicates that the main version is *d*eleted, and the main version will then be removed. However, the origin point $OP(f_i)$ associated with the main version vector will remain. We will keep the vector because at this time some suspicious versions of $f_i$ could still exist.
- When a trustworthy user asks to modify a file $f_i$, if the main version of $f_i$ is deleted, then the user will be informed that $f_i$ does not exist. Otherwise, the modification will be performed on the main version, and if the main version vector exists, it will be changed as follows: (1) after each such modification, its value in the $G$ dimension will be increased by 1, i.e., from $G : n$ to $G : n + 1$; and (2) the value in any other dimension will be unchanged.
- When a suspicious user $S_i$ asks to delete a file $f_i$,
  - If there was an $S_i$-version of $f_i$ but it has been deleted, i.e., the value of the $S_i$-version vector in the $S_i$ dimension is $`d'$, then the user will be informed that $f_i$ does not exist.
  - If there exists an $S_i$-version, then the $S_i$ dimension of the $S_i$-version vector will be changed to $S_i : `d'$, and the $S_i$-version will then be removed. However, the origin point $OP(f_i)$ associated with the $S_i$-version vector of $f_i$ will remain.
  - Otherwise, the $S_i$ dimension with the value $S_i : 0$ will be inserted into the main version vector of $f_i$, and the $S_i$-version vector of $f_i$, which will remain, will be created by first copying the main version vector and then changing its $S_i$ dimension to $S_i : `d'$. Note that here no $S_i$-versions need be deleted since they do not exist.
- When a suspicious user $S_i$ asks to modify a file $f_i$ (we assume $f_i$ has already been modified by some suspicious users),
  - If an $S_i$-version of $f_i$ exists, then the version will be given.
  - If there was an $S_i$-version of $f_i$ but it has been deleted, then the user will be informed that $f_i$ does not exist.

  - Otherwise, we will first insert into the main version vector of $f_i$ the $S_i$ dimension with the value $S_i : 0$, and then create the $S_i$-version of $f_i$ on which the modification will be performed. The $S_i$-version vector will be created by first copying the main version vector and then changing its $S_i$ dimension to $S_i : `m'$. As before, additional modifications of $S_i$ on $f_i$ will not change the $S_i$-version vector.

- When a suspicious $S_i$ asks to read a file $f_i$, if there is an $S_i$-version of $f_i$, then the version will be given. If the main version of $f_i$ exists, then the main version is given; otherwise, the user will be informed that $f_i$ does not exist.

- When a suspicious version is merged with the main version, the main version vector of $f_i$ can have more dimensions than the suspicious version vector. To identify and resolve the conflicts between these two versions, we need to pad the suspicious version vector such that the two vectors have the same set of dimensions. The padding can be done by inserting each missed dimension with the value 0 into the suspicious version vector. The techniques used to identify and resolve the conflicts are presented in the next section.

## 6.2 Identification and Resolution of Conflicts

We wish to consider two types of conflicts: *name conflicts* and *version conflicts*. A name conflict occurs when two files with different origin points have the same name. In contrast, a version conflict occurs when two versions of the same file (the same origin point) have been incompatibly modified. For example, two versions of a file generated by independent modifications of an older version of the file can introduce a version conflict. Two versions of a file are *compatible* iff there is no version conflict between them.

When a suspicious version is merged into the main data version, we identify the version conflicts on a file $f_i$ as follows:

- If both the suspicious version of $f_i$ and the main version of $f_i$ are deleted, or at least one of them is deleted, then they are compatible.

- If none of them are deleted, then the two versions are compatible if their corresponding version vectors are compatible. Two version vectors are *compatible* if one vector $\vec{v_i}$ *dominates* the other vector $\vec{v_j}$ in the value of every dimension. If so, we say $\vec{v_i}$ *dominates* $\vec{v_j}$. In our model, value domination is defined as follows: (1) A value dominates itself. (2) A number $n_1$ dominates another number $n_2$ if $n_1$ is larger than $n_2$. (3) The value $'m'$ dominates the value 0. (4) The value $`d'$ is dominated by any other values.

Name conflicts can be easily resolved by renaming files after all of the version conflicts have been resolved. Resolution of version conflicts should be accompanied by resolution of version vector conflicts because we cannot merge two versions without

merging their version vectors. When the conflicts between a suspicious $S_i$-version (with version vector $\vec{v_s}$) and a main data version (with version vector $\vec{v_g}$) of a file $f_i$ are identified, we resolve the conflicts and merge the two versions as follows. The protocol can ensure that in the merged vector: (1) $G$ : '$d'$ indicates the merged version is removed; otherwise, it exists. (2) $S_i$ : 0 indicates there is (or was) a suspicious $S_i$-version. The version may still be active or may have already been discarded. (3) $S_i$ : '$d'$ indicates there was a suspicious $S_i$-version that has been deleted by $S_i$, who is innocent. (4) $S_i$ : '$m'$ indicates there was a suspicious $S_i$-version. The version is modified by $S_i$ (who is innocent) and is merged into the main version.

**Protocol 3** Merging Protocol for a File System

- If both of the two versions are deleted, then they need not be merged; however, their version vectors need to be merged. The merging is done by taking $\vec{v_g}$ with its value in the $S_i$ dimension changed to $S_i$ : '$d'$ as the merged vector.
- Suppose that the main version is deleted, but the $S_i$-version is not deleted. If the value of $\vec{v_s}$ in the $S_i$ dimension is '$m'$, then the $S_i$ version is the merged version, and the merged vector is $\vec{v_g}$ with its value in the $G$ dimension changed to that of $\vec{v_s}$ and its value in the $S_i$ dimension changed to '$m'$. Otherwise, the deleted main version is the merged version, and the merged vector is $\vec{v_g}$[¶].
- Suppose that the $S_i$-version is deleted, but the main version is not deleted. If either of the following two conditions holds, then the main version is the merged version, and $\vec{v_g}$ with its value in the $S_i$ dimension changed to $S_i$ : '$d'$ is the merged vector.

  1. The value of $\vec{v_g}$ in the $G$ dimension is larger than that of $\vec{v_s}$, that is, $f_i$ has been modified by some trustworthy users after $f_i$ was isolated for $S_i$.
  2. There is a dimension such that the value of $\vec{v_g}$ in the dimension is '$m'$ but the value of $\vec{v_s}$ in the dimension is 0 or $\vec{v_s}$ does not include the dimension at all (Here we assume padding is not used), that is, some modification was performed on $f_i$ by some suspicious user (who was isolated earlier than $S_i$) after $f_i$ was isolated for $S_i$ and the modification has been merged into the main version, or some modification was performed on $f_i$ by some suspicious user (who was isolated later than $S_i$) and has been merged into the main version.

  Otherwise, the deleted $S_i$ version is the merged version, and $\vec{v_g}$ with its value in the $G$ dimension changed to '$d'$ and its value in the $S_i$ dimension changed to '$d'$ is the merged vector.

- If none of the two versions are deleted:

[¶]The decision is made because at this point $S_i$ has done nothing to the file while some trustworthy user has deleted the file, so the deletion should be valid.

- If $\vec{v_g}$ dominates $\vec{v_s}$, then the main version is the resolved version. $\vec{v_g}$ is the merged vector.
- If $\vec{v_s}$ dominates $\vec{v_g}$, then the suspicious version is the resolved version. $\vec{v_s}$ is the merged vector.
- If $\vec{v_g}$ and $\vec{v_s}$ are incompatible, then the resolution can be done either manually or automatically, based on the semantics of the modifications that have been applied on these versions. After the resolution, $\vec{v_g}$ with its value in the $S_i$ dimension changed to '$m$' is the merged vector. Note that here a version conflict can happen only if the value of $\vec{v_s}$ in the $S_i$ dimension is '$m$'.

**Example 4** *Assume that when the $S_2$-version is merged into the main version, the main version vector of a file $f_i$ is $\vec{v_g}(f_i) =< G:0, S_1:0, S_2:0 >$. At this point, if the $S_2$-version vector of $f_i$ is $\vec{v_s}(f_i) =< G:0, S_1:0, S_2:m >$, then since $\vec{v_s}(f_i)$ dominates $\vec{v_g}(f_i)$, there are no conflicts, the $S_2$-version is the merged version, and $\vec{v_s}(f_i)$ is the merged vector. If $\vec{v_s}(f_i) =< G:0, S_1:0, S_2:d >$, then the two versions are still compatible. The deleted $S_2$-version is the merged version, and $< G:d, S_1:0, S_2:d >$ is the merged vector.*

*Consider another scenario where $\vec{v_g}(f_i) =< G:2, S_1:0, S_2:0 >$ and $\vec{v_s}(f_i) =< G:0, S_1:0, S_2:m >$. $\vec{v_g}(f_i)$ and $\vec{v_s}(f_i)$ conflict. Note that the version of $f_i$ with the vector $< G:0, S_1:0, S_2:0 >$ has been independently modified by $S_2$ and some trustworthy actions. At this point, manual or automatic approaches need to be applied to resolve the conflicts. The version vector of the resolved version is $< G:2, S_1:0, S_2:m >$.*

To reduce the overhead of maintaining version vectors, we need to reset main version vectors. The reset for the main version vector of $f_i$ can be processed when there are no active accesses to $f_i$ and when $f_i$ has not been modified or deleted in any active suspicious versions that have not been merged or discarded. The reset can be easily done by removing the main version vector.

Although there is no general way to resolve conflicts automatically, conflict resolution can be automated in many file systems by exploiting the operational semantics. For example, reconciliation for two important types of files in LOCUS [PPR+83], directories and user mailboxes, is handled automatically. Interested readers can refer to [PPR+83] for more details on this topic.

### 6.3 Dealing with Multiple Suspicious Users

Synchronizing multiple suspicious file access histories in the file system context is much simpler than synchronizing multiple suspicious transaction histories in the database system context, since in the file system context,

- Conflicting updates are identified and resolved in terms of each data object (file) instead of each transaction or history.

- Conflict resolution of one data object (file) will not affect the values of other objects (files).
- The resolution of the conflicts on each data object (file) that has been updated independently by two histories implies the resolution of the conflicts between these two histories.

Therefore, in the file system context suspicious access actions need not be synchronized.

# 7 Related Work

A substantial body of work has been done on intrusion detection [Lun93, MHL94, LM98], based on either detecting deviations from expected statistical profiles [JV94] or pattern-matching against known methods of attack [Ilg93, GL91, PK92, IKP95, SG91, SG97, LWJ98]. In [JV94], the idea of setting multiple alert levels is proposed, where each alert level corresponds to a specific degree of anomaly and different actions are taken at each alert level. However, the issues of what actions should be taken at each level and how to enforce these actions are not addressed in [JV94]. Our isolation scheme can be viewed as a realization of the idea in which two alert levels are set. At the first alert level, we isolate users' accesses when a suspicious behavior is discovered. At the second level, we reject users' accesses when an intrusion is reported. Multi-level isolation schemes are certainly possible.

In [HL93] and [HLR92], a probabilistic model of intrusion detection is proposed in which (1) computer use is modeled as a mixture of two specific stochastic processes that generate, respectively, normal actions and misuse actions[‖]; and (2) the objective of intrusion detection is to identify the actions that are most likely to be generated by the stochastic process that generates misuse actions. Our probabilistic model can be viewed as a simplified version of the model proposed in [HL93] and [HLR92]. We focus on the performance of a detection system at a single point of time. However, our main purpose is to help the SSO to justify the necessity of intrusion confinement when he or she wants to enforce intrusion confinement in an information system, as opposed to accurately modelling an intrusion detector and measuring its effectiveness. Moreover, our model is based on behaviors which are a sequence of actions, whereas the model presented in [HL93] and [HLR92] is based on single actions.

In [JLM98], an application-level isolation protocol is proposed to cope with malicious database users. In this paper, we extend the work of [JLM98] in several aspects: (1) [JLM98] does not answer clearly such questions as "Why are there suspicious actions?", "How can these suspicious actions be detected?", "Why is isolation necessary?", and "When should isolation be enforced?". In this paper, we give clear answers to these questions by proposing, modeling, and analyzing *intrusion confinement* based on a probabilistic model presented to evaluate the effectiveness of current intrusion detection systems. (2) [JLM98] is limited to the database context. In this paper, we

[‖]In [HL93] and [HLR92], actions are called *transactions*.

extend the isolation mechanisms proposed in [JLM98] to a general solution that can be applied to many types of information systems, such as file systems and object systems. (3) In the database system context, [JLM98] achieves complete isolation; our solution achieves one-way isolation. (4) We present a novel isolation protocol for file systems that is not addressed by [JLM98].

Although the area of IW defense is new, some relevant work exists. Graubart et al. [GSM96] identify a number of aspects of the management of a DBMS that affect vulnerability with respect to IW. McDermott and Goldschlag [MG96a, MG96b] identify some techniques for defending against data jamming that, while primarily intended for detection, could also help deceive the attacker and confuse the issue of which data values are critical.

Finally, Ammann et al. [AJMB97] take a detailed look at the problem of surviving IW attacks on databases. They identify a number of phases of the IW process and describe activities that occur in each of them. To maintain precise information about the attack, they propose to mark data to reflect the severity of detected damage as well as the degree to which the damaged data has been repaired. They define an access protocol for normal transactions and show that transactions following the protocol will maintain database consistency even if part of the database is damaged.

# 8 Conclusion

In this paper, we proposed, modeled, and analyzed the problem of intrusion confinement. It is shown that intrusion confinement can effectively resolve the conflicting design goals of an intrusion detection system by achieving both a high rate of detection and a low rate of errors. It is also shown that as a second level of protection in addition to access control intrusion confinement can dramatically enhance the security (especially integrity and availability) of a system in many situations.

We proposed a general solution that is based on a specific isolation mechanism to achieve intrusion confinement. We evaluated the feasibility of the solution by presenting two concrete isolation schemes that can be enforced in the database and file system contexts, respectively. It is shown that these protocols are more flexible, economical, and efficient than fishbowling, and they can be applied to every database or file system. Finally, we should mention that the general intrusion confinement solution can be applied to many other types of information systems in addition to databases and file systems, such as object-oriented systems, distributed information systems, and workflow management systems. Developing concrete isolation protocols for these systems is a topic of our future research.

## Acknowledgments

Jajodia and McCollum were partially supported by Air Force Research Laboratory/Rome under contract F30602-97-1-0139.

## References

[AJL] P. Ammann, S. Jajodia, and P. Liu. Recovery from malicious transactions. Technical report, George Mason University, Fairfax, VA. http://www.research.umbc.edu/~pliu/papers/dynamic.ps.gz.

[AJMB97] 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.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[Dav84] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–581, September 1984.

[GL91] T. D. Garvey and T. F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, October 1991.

[GSM96] R. Graubart, L. Schlipper, and C. McCollum. Defending database management systems against information warfare attacks. Technical report, The MITRE Corporation, 1996.

[HL93] P. Helman and G. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Transactions on Software Engineering*, 19(9):886–901, 1993.

[HLR92] P. Helman, G. Liepins, and W. Richards. Foundations of intrusion detection. In *Proceedings of the 5th IEEE Computer Security Foundations Workshop*, pages 114–120, June 1992.

[IKP95] 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.

[Ilg93] 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.

[JLM98] S. Jajodia, P. Liu, and C. D. McCollum. Application-level isolation to cope with malicious database users. In *Proceedings of the 14th Annual Computer Security Application Conference*, pages 73–82, Phoenix, AZ, December 1998.

[JV94] H. S. Javitz and A. Valdes. The nides statistical component description and justification. Technical Report A010, SRI International, March 1994.

[LAJ00] P. Liu, P. Ammann, and S. Jajodia. Rewriting histories: Recovery from malicious transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[LM98] Teresa Lunt and Catherine McCollum. Intrusion detection and response research at DARPA. Technical report, The MITRE Corporation, McLean, VA, 1998.

[Lun93] T.F. Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.

[LWJ98] J. Lin, X. S. Wang, and S. Jajodia. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, Rockport, Massachusetts, June 1998.

[MG96a] J. McDermott and D. Goldschlag. Storage jamming. In D. L. Spooner, S. A. Demurjian, and J. E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 365–381. Chapman & Hall, London, 1996.

[MG96b] 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.

[MHL94] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, June 1994.

[PG99] Brajendra Panda and Joe Giordano. Reconstructing the database after electronic attacks. In Sushil Jajodia, editor, *Database Security XII: Status and Prospects*. Kluwer, Boston, 1999.

[PK92] P. A. Porras and R. A. Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *Proceedings of the 8th Annual Computer Security Applications Conference*, San Antonio, Texas, December 1992.

[PPR+83] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, 1983.

[SG91] S.-P. Shieh and V. D. Gligor. A pattern oriented intrusion detection model and its applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1991.

[SG97] S.-P. Shieh and V. D. Gligor. On a pattern-oriented model for intrusion detection. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):661–667, 1997.

| Parameter | Meaning |
|---|---|
| $D_i$ | the value of a distance such that if $d(\vec{v_s}, \vec{v_l}) \geq D_i$, then $\vec{v_s}$ is reported as an intrusion. |
| $P_i$ | the conditional probability that when a behavior is reported as an intrusion, that is, $d(\vec{v_s}, \vec{v_l}) \geq D_i$, it is really an intrusion. |
| $D_s$ | the value of a distance such that if $D_s \leq d(\vec{v_s}, \vec{v_l}) < D_i$, then $\vec{v_s}$ is reported as suspicious. |
| $P_s$ | the conditional probability that when a behavior is reported as suspicious, that is, $D_s \leq d(\vec{v_s}, \vec{v_l}) < D_i$, it is an intrusion. |
| $A_i$ | the probability that a behavior deviates from the corresponding long-term behavior with $d(\vec{v_s}, \vec{v_l}) \geq D_i$. |
| $A_s$ | the probability that a behavior deviates from the corresponding long-term behavior with $D_s \leq d(\vec{v_s}, \vec{v_l}) < D_i$. |
| $P_g$ | the conditional probability that a behavior with $d(\vec{v_s}, \vec{v_l}) < D_s$ is an intrusion. |

Table 1: Evaluation Parameters for a Statistical Profile-Based Detection System

| System No. | $P_i$ | $P_s$ | $A_i$ | $A_s$ | $P_g$ |
|---|---|---|---|---|---|
| 1 | 0.85 | 0.47 | 0.05 | 0.18 | 0.01 |
| 2 | 0.85 | 0.32 | 0.10 | 0.13 | 0.01 |

Table 2: Values of the Other Parameters

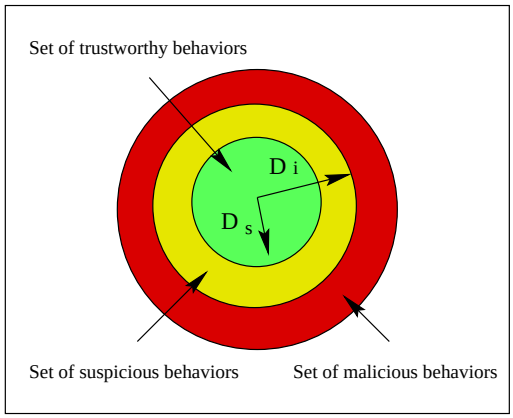


Figure 1: Classification of User Behaviors

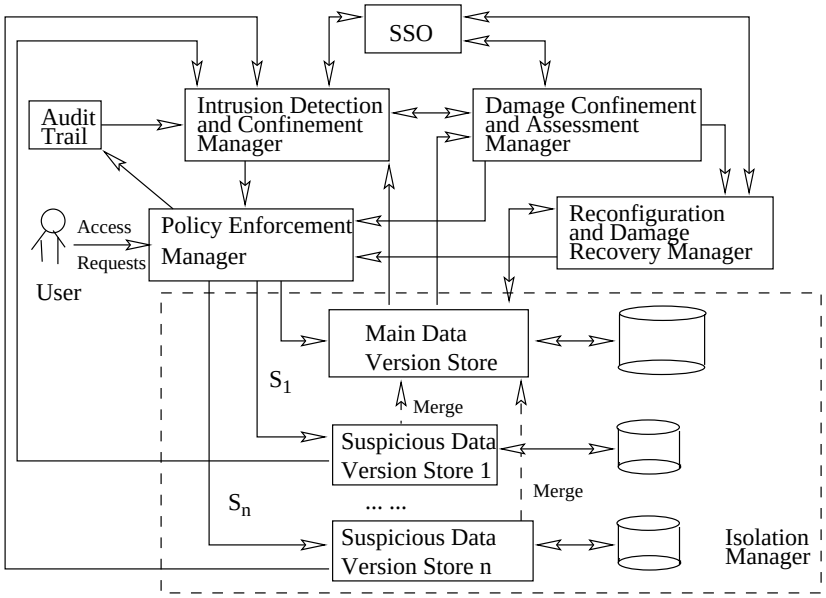


Figure 2: Architecture of the Intrusion Confinement System

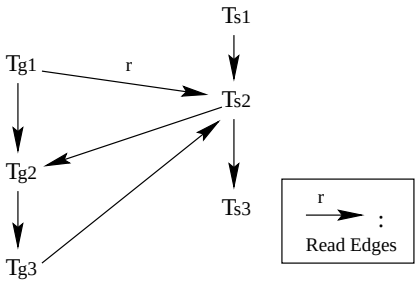


Figure 3: Precedence Graph for the History in Example 2

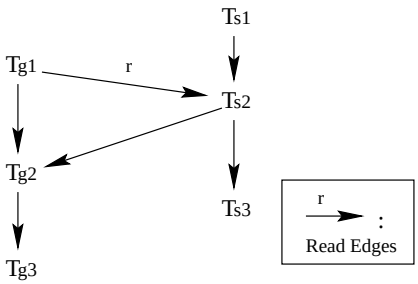


Figure 4: Precedence Graph for the History in Example 3

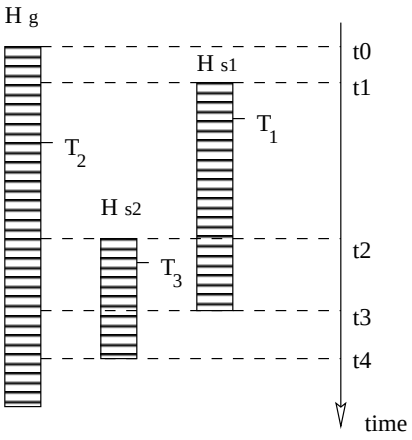


Figure 5: The Phantom Problem

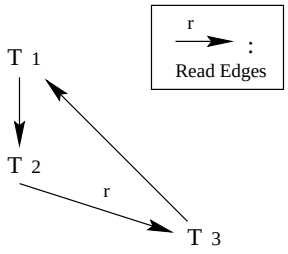


Figure 6: Detecting Phantom Transactions in Precedence Graphs