In this chapter, we propose to use an intermediate layer, which we call system view, to provide a unified representation of the information provided by different types of systems.

A system view serves two purposes. First, it can hide the differences between heterogeneous systems. For example, most of the platforms (e.g., UNIX and Windows NT) have login process to authenticate users. Though the status of the login processes on different platforms may be recorded in different formats (e.g., BSM on Solaris and Windows Event Logger on Windows), we can transform such information into a unified representation on a common system view that abstracts the common features of login processes. Certainly, the correspondence between the same kind of information on different platforms may be more complex than simple one-to-one mappings between audit records. However, compared with the possible attacks that we may discover and describe, the correspondence between the information on different platforms is relatively static, and it is fairly easy to write simple programs to transform information on heterogeneous platforms to a unified representation.

Second, a system view can hide the detailed information provided by an autonomous system. As we discussed earlier, an autonomous system that is willing to collaborate with other systems may have its own concerns about the privacy and confidentiality of its local information. It may want to provide information to a certain degree instead of giving out all kinds of information. System view then serves as an interface that specifies what kind of information is exported from the autonomous system. The information on the autonomous system may be cleansed and/or aggregated before being provided through the system view.

In the remainder of this chapter, we clarify how we formalize the concept of system view.
1. System View and Event History

Intuitively, a system view provides an abstract representation of a system. The system underlying a system view may be one single host, a network segment, or a distributed system consisting of several hosts.

Both event information and relationships among system entities are provided through a system view. Events represent what have happened or are happening in the system, while relationships among system entities represent the system state at certain times. For instance, the fact that two files are owned by the same user can be represented by a relationship same_owner between them.

The time when an event occurs is intrinsic to the event. In distributed environments, the intrusion detection related events are usually not instantaneous in terms of time. (For example, a TCP connection could span several hours.) To accommodate such events, we consider that each event has a duration and associate an interval-based timestamp with it. Notation-wise, each timestamp is denoted in the form of $[begin\_time, \ end\_time]$, representing the starting and the ending points of the time interval, respectively.

We use a predicate to represent a relationship among system entities. A predicate for a particular relationship takes the entities as arguments, and returns True if the relationship is satisfied, and False otherwise. For example, the predicate $\text{same\_owner}(\text{var\_file}_x, \text{var\_file}_y)$ determines whether the owners of $\text{var\_file}_x$ and $\text{var\_file}_y$ are the same user. It returns True if a single user owns both files, and False otherwise.

The relationships among system entities may be dynamic, i.e., the relationships may change over time. For example, suppose a user is the owner of both $\text{file}_x$ and $\text{file}_y$ at a certain time, but later the superuser changes the owner of $\text{file}_x$ to another user. Then $\text{same\_owner}(\text{file}_x, \text{file}_y)$ returns True before the change of owner, while returns False afterwards. Thus, notation-wise, we insert symbols “$[\var\_time]$” between the predicate name and its arguments, indicating that time is one necessary argument of the predicate. We call such predicates dynamic predicates. Hence the dynamic predicate for determining file owners can be denoted $\text{same\_owner}[\var\_time](\text{var\_file}_x, \text{var\_file}_y)$, which becomes a regular predicate when “$\var\_time$” is replaced with a constant time point. For example, $\text{same\_owner}[t](\text{file}_x, \text{file}_y)$ is True if and only if the owners of $\text{file}_x$ and $\text{file}_y$ are the same at time $t$. Note that “static”, or regular predicates are special cases of dynamic predicates.

The notion of system view is formally stated as follows.