Constructing Attack Scenarios through Correlation of Intrusion Alerts [*]

Peng Ning   Yun Cui   Douglas S. Reeves
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
Emails: ning@csc.ncsu.edu, ycui4@eos.ncsu.edu, reeves@csc.ncsu.edu

Abstract

Traditional intrusion detection systems (IDSs) focus on low-level attacks or anomalies, and raise alerts independently, though there may be logical connections between them. In situations where there are intensive intrusions, not only will actual alerts be mixed with false alerts, but the amount of alerts will also become unmanageable. As a result, it is difficult for human users or intrusion response systems to understand the alerts and take appropriate actions.

This paper presents a practical technique to address this issue. The proposed approach constructs attack scenarios by correlating alerts on the basis of *prerequisites* and *consequences* of intrusions. Intuitively, the prerequisite of an intrusion is the necessary condition for the intrusion to be successful, while the consequence of an intrusion is the possible outcome of the intrusion. Based on the prerequisites and consequences of different types of attacks, our method correlates alerts by (partially) matching the consequence of some previous alerts and the prerequisite of some later ones. The contribution of this paper includes a formal framework for alert correlation, the implementation of an off-line alert correlator based on the framework, and the evaluation of our method with the 2000 DARPA intrusion detection scenario specific datasets. Our experience and experimental results have demonstrated the potential of the proposed method and its advantage over alternative methods.

1 Introduction

Intrusion detection has been studied for about twenty years since the Anderson's report [1]. Intrusion detection techniques can be roughly classified as *anomaly detection* and *misuse detection*. Anomaly detection (e.g., NIDES/STAT [10]) is based on the normal behavior of a subject (e.g., a user or a system); any action that significantly deviates from the normal behavior is considered intrusive. Misuse detection (e.g., NetSTAT [20]) detects intrusions based on the characteristics of known attacks or system vulnerabilities; any action that conforms to the pattern of a known attack or vulnerability is considered intrusive.

Traditional intrusion detection systems (IDSs) focus on low-level attacks or anomalies, and raise alerts independently, though there may be logical connections between them. In situations where there are intensive intrusions, not only will actual alerts be mixed with false alerts, but the amount of alerts will also become unmanageable. As a result, it is difficult for human users or intrusion response systems to understand the alerts and take appropriate actions. Therefore, it is necessary to develop techniques to construct *attack scenarios* (i.e., steps that attackers use in their attacks) from alerts and facilitate intrusion analysis.

[*]Results in this technical report overlap with TR-2001-13 and TR-2002-01. The intention of this technical report is to provide a full version for the paper to appear in ACM CCS 02 [14].

Several alert correlation methods have been proposed to address this problem. These methods fall into three classes. The first class (e.g., Spice [17], the probabilistic alert correlation [19], and the MIRADOR method [4]) correlates alerts based on the similarities between alert attributes. Though they are effective for correlating some alerts (e.g., alerts with the same source and destination IP addresses), they cannot fully discover the causal relationships between related alerts. The second class (e.g., LAMBDA [6] and the data mining approach [7]) bases alert correlation on attack scenarios specified by human users or learned through training datasets. Obviously, these methods are restricted to *known* attack scenarios. A variation in this class uses a consequence mechanism to specify what types of attacks may follow a given attack, partially addressing this problem [8]. The third class (e.g., JIGSAW [18]) is based on the preconditions and consequences of individual attacks; it correlates alerts if the precondition of some later alerts are satisfied by the consequences of some earlier alerts. Compared with the first two classes of methods, this class can potentially uncover the causal relationship between alerts, and is not restricted to known attack scenarios. (Please see Section 5 for more related work.)

To our best knowledge, JIGSAW [18] is the only published result that falls into the third class[1]. It was originally proposed to represent complex attacks, and the authors envisaged to apply it to correlate intrusion alerts. However, several problems make it difficult for JIGSAW to be a practical alert correlation technique. First, JIGSAW requires all the preconditions (i.e., required capabilities in [18]) of an (abstract) attack to be satisfied in order to consider its consequences. This is theoretically okay; however, it has a negative impact on alert correlation in practice. In particular, if the IDS fails to detect one of the attacks that prepare for later attacks, JIGSAW will miss the opportunity to correlate the detected attacks. Moreover, JIGSAW treats low-level attacks individually, and does not correlate an alert if it does not prepare for (or is prepared for by) other alerts, even if the alert is related to others. For example, if an attacker tries several variations of the same attack in a short period of time, JIGSAW will treat them separately, and only correlate those that prepare for (or are prepared for by) other alerts. In addition, JIGSAW ignores failed attempts of attacks even if they belong to a sequence of well planned attacks. Finally, no specific mechanism has been provided in JIGSAW to correlate alerts, though this has been speculated as an application of JIGSAW in [18]. Thus, additional work is necessary to have a practical solution for constructing attack scenarios from alerts.

In this paper, we address the limitations of JIGSAW, and develop a practical alert correlation technique that can be used to construct attack scenarios for real-life intrusion analysis. Our method can be explained easily based on the following observation: most intrusions are not isolated, but related as different stages of attacks, *with the early stages preparing for the later ones*. For example, in Distributed Denial of Service (DDOS) attacks, the attacker has to install the DDOS daemon programs in vulnerable hosts before he can instruct the daemons to launch an attack. In other words, an attacker has to (or usually does) reach a certain state before he can carry out certain attacks, and usually reaches the state by launching some other attacks.

Based on this observation, we correlate alerts using *prerequisites and consequences of intrusions*. Intuitively, the prerequisite of an intrusion is the necessary condition for the intrusion to be successful, while the consequence of an intrusion is the possible outcome of the intrusion. For example, the existence of a vulnerable service is the prerequisite of a remote buffer overflow attack against the service, and as the consequence of the attack, the attacker may gain access to the host. Accordingly, we correlate the alerts together when the attackers launch some early attacks to prepare for the prerequisites of some later ones. For example, if they use a UDP port scan to discover the vulnerable services, followed by an attack against one of the services, we can correlate the corresponding alerts together. To address the limitations of JIGSAW, our method allows alert aggregation as well as partial satisfaction of prerequisites of an intrusion. In addition, our formalism provides an intuitive representation of correlated alerts and a specific mechanism for alert correlation, which leads to our implementation of the method.

[1]Recent work by Cuppens and Miege [5] has substantial similarity to our work, which is done independently. The comparison of [5] with our work can be found in [14]

The contribution of this paper is three-fold. First, we develop a framework for alert correlation by addressing the limitations of JIGSAW. Unlike JIGSAW, our method can deal with attack attempts and correlate alerts as long as there are signs of connections between them, even if some related attacks fail or bypass the IDS. In addition, our method provides an intuitive mechanism (called hyper-alert correlation graph) to represent attack scenarios constructed through alert correlation. Second, we develop an off-line tool that implements our alert correlation method. Based on the information about different types of attacks, our tool processes the alerts reported by IDSs and generates hyper-alert correlation graphs as the output. As we will see in Section 4, these hyper-alert correlation graphs reveal the structure of series of attacks, and thus the strategy behind them. Third, we perform a series of experiments to validate our method using 2000 DARPA intrusion detection scenario specific datasets [12]. Our results show that our method not only correlates related alerts and uncovers the attack strategies, but also provides a way to differentiate between alerts.

The remainder of this paper is organized as follows. The next section presents our formal framework for correlating alerts using prerequisites and consequences of intrusions. Section 3 describes the implementation of our method. Section 4 reports our experiments with the 2000 DARPA intrusion detection scenario specific datasets. Section 5 discusses additional related work. Section 6 concludes this paper and points out future research directions. Appendices A and B present further details about the experiments.

2 A Framework for Alert Correlation

As discussed in the introduction, our method is based on the observation that in series of attacks, the component attacks are usually not isolated, but related as different stages of the attacks, with the early ones preparing for the later ones. To take advantage of this observation, we propose to correlate the alerts generated by IDSs using prerequisites and consequences of the corresponding attacks. Intuitively, the *prerequisite* of an attack is the necessary condition for the attack to be successful. For example, the existence of a vulnerable service is the prerequisite of a remote buffer overflow attack against the service. Moreover, the attacker may make progress in gaining access to the victim system (e.g., discover the vulnerable services, install a Trojan horse program) as a result of an attack. Informally, we call the possible outcome of an attack the (possible) *consequence* of the attack. In a series of attacks where the attackers launch earlier attacks to prepare for later ones, there are usually strong connections between the consequences of the earlier attacks and the prerequisites of the later ones. Indeed, if an earlier attack is to prepare for a later attack, the consequence of the earlier attack should at least partly satisfy the prerequisite of the later attack.

Accordingly, we propose to identify the prerequisites (e.g., existence of vulnerable services) and the consequences (e.g., discovery of vulnerable services) of each type of attack. These are then used to correlate alerts, which are attacks detected by IDSs, by matching the consequences of (the attacks corresponding to) some previous alerts and the prerequisites of (the attacks corresponding to) some later ones. For example, if we find a *Sadmind Ping* followed by a buffer overflow attack against the corresponding *Sadmind* service, we can correlate them to be parts of the same series of attacks. In other words, we model the knowledge (or state) of attackers in terms of individual attacks, and correlate alerts if they indicate the progress of attacks.

Note that an attacker does not *have to* perform early attacks to prepare for a later attack, even though the later attack has certain prerequisites. For example, an attacker may launch an individual buffer overflow attack against a service blindly, without knowing if the service exists. In other words, the prerequisite of an attack should not be mistaken for the necessary existence of an earlier attack. However, if the attacker does launch attacks with earlier ones preparing for later ones, our method can correlate them, provided that the attacks are detected by IDSs.

In the following subsections, we adopt a formal approach to develop our alert correlation method.

2.1 Prerequisite and Consequence of Attacks

We propose to use predicates as basic constructs to represent the prerequisites and (possible) consequences of attacks. For example, a scanning attack may discover UDP services vulnerable to a certain buffer overflow attack. We can use the predicate *UDPVulnerableToBOF* (*VictimIP, VictimPort*) to represent the attacker's discovery (i.e., the consequence of the attack) that the host having the IP address *VictimIP* runs a service (e.g., *sadmind*) at UDP port *VictimPort* and that the service is vulnerable to the buffer overflow attack. Similarly, if an attack requires a UDP service vulnerable to the buffer overflow attack, we can use the same predicate to represent the prerequisite.

Some attacks may require several conditions be satisfied at the same time in order to be successful. To represent such complex conditions, we use a logical combination of predicates to describe the prerequisite of an attack. For example, a certain network born buffer overflow attack may require that the target host have a vulnerable UDP service accessible to the attacker through the firewall. This prerequisite can be represented by *UDPVulnerableToBOF* (*VictimIP, VictimPort*) $\wedge$ *UDPAccessibleViaFirewall* (*VictimIP, VictimPort*). To simplify the discussion, we restrict the logical operators to $\wedge$ (conjunction) and $\vee$ (disjunction).

We also use a set of predicates to represent the (possible) consequence of an attack. For example, an attack may result in compromise of the root privilege as well as modification of the *.rhost* file. Thus, we may use the following to represent the corresponding consequence: $\{$*GainRootAccess* (*VictimIP*), *rhost-Modified* (*VictimIP*)$\}$. Note that the set of predicates used to represent the consequence is essentially the conjunction of these predicates and can be represented by a single logical formula. However, representing the consequence as a set rather than a long formula is more convenient and will be used here.

The consequence of an attack is indeed the *possible* result of the attack. In other words, the attack may or may not generate the stated consequence. For example, after a buffer overflow attack against a service, an attacker may or may not gain the root access, depending on if the service is vulnerable to the attack.

We use *possible* consequences instead of *actual* consequences due to the following two reasons. First, an IDS may not have enough information to decide if an attack is effective or not. For example, a network based IDS can detect certain buffer overflow attacks by matching the patterns of the attacks; however, it cannot decide whether the attempts succeed or not without more information from the related hosts. Thus, it may not be practical to correlate alerts using the actual consequence of attacks. In contrast, the possible consequence of a type of attack can be analyzed and made available for IDS. Second, even if an attack fails to prepare for the follow-up attacks, the follow-up attacks may still occur simply because, for example, the attacker wants to test the success of the previous attack, or the attacker uses a script to launch a series of attacks. Using possible consequences of attacks will lead to better opportunity to correlate such attacks.

For the sake of brevity, we refer to a *possible consequence* simply as a *consequence* throughout this paper.

2.2 Hyper-alert Type and Hyper-alert

Using predicates as the basic construct, we introduce the notion of a *hyper-alert type* to represent the prerequisite and the consequence of each type of alert.

Definition 1 A *hyper-alert type T* is a triple (*fact, prerequisite, consequence*), where (1) *fact* is a set of attribute names, each with an associated domain of values, (2) *prerequisite* is a logical combination of predicates whose free variables are all in *fact*, and (3) *consequence* is a set of predicates such that all the free variables in *consequence* are in *fact*.

Each hyper-alert type encodes the knowledge about a type of attack. The component *fact* of a hyper-alert type tells what kind of information is reported along with the alert (i.e., detected attack), *prerequisite* specifies what must be true in order for the attack to be successful, and *consequence* describes what could be true if the attack indeed succeeds. For the sake of brevity, we omit the domains associated with the attribute names when they are clear from the context.

Example 1 Consider the buffer overflow attack against the *sadmind* remote administration tool. We may have a hyper-alert type *SadmindBufferOverflow* = ({*VictimIP*, *VictimPort*}, *ExistHost* (*VictimIP*) $\wedge$ *VulnerableSadmind* (*VictimIP*), {*GainRootAccess*(*VictimIP*)}) for such attacks. Intuitively, this hyper-alert type says that such an attack is against the host at IP address *VictimIP*. (We expect the actual values of *VictimIP* are reported by an IDS.) For the attack to be successful, there must exist a host at IP address *VictimIP*, and the corresponding *sadmind* service must be vulnerable to buffer overflow attacks. The attacker may gain root privilege as a result of the attack. $\Box$

Given a hyper-alert type, a *hyper-alert instance* can be generated if the corresponding attack is detected and reported by an IDS. For example, we can generate a hyper-alert instance of type *SadmindBufferOverflow* from a corresponding alert. The notion of hyper-alert instance is formally defined as follows.

Definition 2 Given a hyper-alert type *T* = (*fact*, *prerequisite*, *consequence*), a *hyper-alert (instance) h of type T* is a finite set of tuples on *fact*, where each tuple is associated with an interval-based timestamp [*begin_time*, *end_time*]. The hyper-alert h implies that *prerequisite* must evaluate to True and all the predicates in *consequence* might evaluate to True for each of the tuples. (Notation-wise, for each tuple t in h, we use $t.begin\_time$ and $t.end\_time$ to refer to the timestamp associated with t.)

The *fact* component of a hyper-alert type is essentially a relation schema (as in relational databases), and a hyper-alert is a relation instance of this schema. One may point out that an alternative way is to represent a hyper-alert as a record, which is equivalent to a single tuple on *fact*. However, such an alternative cannot accommodate certain alerts possibly reported by an IDS. For example, an IDS may report an IPSweep attack along with multiple swept IP addresses, which cannot be represented as a single record. In addition, our current formalism allows aggregation of alerts of the same type, and is flexible in reasoning about alerts. Therefore, we believe the current notion of a hyper-alert is an appropriate choice.

A hyper-alert instantiates its *prerequisite* and *consequence* by replacing the free variables in *prerequisite* and *consequence* with its specific values. Since all free variables in *prerequisite* and *consequence* must appear in *fact* in a hyper-alert type, the instantiated prerequisite and consequence will have no free variables. Note that *prerequisite* and *consequence* can be instantiated multiple times if *fact* consists of multiple tuples. For example, if an IPSweep attack involves several IP addresses, the *prerequisite* and *consequence* of the corresponding hyper-alert type will be instantiated for each of these addresses.

In the following, we treat timestamps implicitly and omit them if they are not necessary for our discussion.

Example 2 Consider the hyper-alert type *SadmindBufferOverflow* in example 1. There may be a hyper-alert $h_{SadmindBOF}$ as follows: {(*VictimIP* = 152.1.19.5, *VictimPort* = 1235), (*VictimIP* = 152.1.19.7, *VictimPort* = 1235)}. This implies that if the attack is successful, the following two logical formulas must be True as the prerequisites of the attack: *ExistHost* (152.1.19.5) $\wedge$ *VulnerableSadmind* (152.1.19.5), *ExistHost* (152.1.19.7) $\wedge$ *VulnerableSadmind* (152.1.19.7). Moreover, as possible consequences of the attack, the following might be True: *GainRootAccess* (152.1.19.5), *GainRootAccess* (152.1.19.7). This hyper-alert says that there are buffer overflow attacks against *sadmind* at IP addresses 152.1.19.5 and 152.1.19.7, and the attacker may gain root access as a result of the attacks. $\Box$

A hyper-alert may correspond to one or several related alerts. If an IDS reports one alert for a certain attack and the alert has all the information needed to instantiate a hyper-alert, a hyper-alert can be generated from the alert. However, some IDSs may report a series of alerts for a single attack. For example, EMERALD may report several alerts (within the same thread) related to an attack that spreads over a period of time. In this case, a hyper-alert may correspond to the aggregation of all the related alerts. Moreover, several alerts may be reported for the same type of attack in a short period of time. Our definition of hyper-alert allows them to be treated as one hyper-alert, and thus provides flexibility in the reasoning about alerts. Certain constraints are necessary to make sure the hyper-alerts are reasonable. However, since our hyper-alert correlation method does not depend on them directly, we will discuss them after introducing our method.

Ideally, we may correlate a set of hyper-alerts with a later hyper-alert together if the consequences of the former ones imply the prerequisite of the latter one. However, such an approach may not work in reality due to several reasons. First, the attacker may not always prepare for certain attacks by launching some other attacks. For example, the attacker may learn a vulnerable *sadmind* service by talking to people who work in the organization where the system is running. Second, the current IDSs may miss some attacks, and thus affect the alert correlation if the above approach is used. Third, due to the combinatorial nature of the aforementioned approach, it is computationally expensive to examine sets of alerts to find out whether their consequences satisfy (or more precisely, imply) the prerequisite of an alert.

Having considered these issues, we adopt an alternative approach. Instead of examining if several hyper-alerts satisfy the prerequisite of a later one, we check if an earlier hyper-alert *contributes* to the prerequisite of a later one. Specifically, we decompose the prerequisite of a hyper-alert into pieces of predicates and test whether the consequence of an earlier hyper-alert makes some pieces of the prerequisite True (i.e., make the prerequisite easier to satisfy). If the result is yes, then we correlate the hyper-alerts together. This idea is specified formally through the following Definitions.

Definition 3 Consider a hyper-alert type T = (*fact*, *prerequisite*, *consequence*). The *prerequisite set (or consequence set, resp.) of* T, denoted $P(T)$ (or $C(T)$, resp.), is the set of all predicates that appear in *prerequisite* (or *consequence*, resp.). Given a hyper-alert instance h of type T, the *prerequisite set (or consequence set, resp.) of* h, denoted $P(h)$ (or $C(h)$, resp.), is the set of predicates in $P(T)$ (or $C(T)$, resp.) whose arguments are replaced with the corresponding attribute values of each tuple in h. Each element in $P(h)$ (or $C(h)$, resp.) is associated with the timestamp of the corresponding tuple in h. (Notation-wise, for each $p \in P(h)$ (or $C(h)$, resp.), we use $p.begin\_time$ and $p.end\_time$ to refer to the timestamp associated with p.)

Example 3 Consider the *Sadmind Ping* attack through which an attacker discovers possibly vulnerable *sadmind* services. The corresponding alerts can be represented by a hyper-alert type *SadmindPing* = ({*VictimIP*, *VictimPort*}, *ExistHost* (*VictimIP*), {*VulnerableSadmind* (*VictimIP*)}).

Suppose a hyper-alert instance $h_{SadmindPing}$ of type *SadmindPing* has the following tuples: {(*VictimIP* = 152.1.19.5, *VictimPort* = 1235), (*VictimIP* = 152.1.19.7, *VictimPort* = 1235), (*VictimIP* = 152.1.19.9, *VictimPort* = 1235)}. Then we have the prerequisite set $P(h_{SadmindPing})$ = {*ExistHost* (152.1.19.5), *ExistHost* (152.1.19.7), *ExistHost* (152.1.19.9)}, and the consequence set $C(h_{SadmindPing})$ = {*VulnerableSadmind* (152.1.19.5), *VulnerableSadmind* (152.1.19.7), *VulnerableSadmind* (152.1.19.9)}. □

Example 4 Consider the hyper-alert $h_{SadmindBOF}$ discussed in example 2. We have $P(h_{SadmindBOF})$ = {*ExistHost* (152.1.19.5), *ExistHost* (152.1.19.7), *VulnerableSadmind* (152.1.19.5), *VulnerableSadmind* (152.1.19.7)}, and $C(h_{SadmindBOF})$ = {*GainRootAccess* (152.1.19.5), *GainRootAccess* (152.1.19.7)}. □

Definition 4 Hyper-alert h_1 *prepares for* hyper-alert h_2, if there exist $p \in P(h_2)$ and $C \subseteq C(h_1)$ such that for all $c \in C$, $c.end\_time < p.begin\_time$ and the conjunction of all the predicates in C implies p.

The prepare-for relation is developed to capture the causal relationships between hyper-alerts. Intuitively, h_1 prepares for h_2 if some attacks represented by h_1 make the attacks represented by h_2 easier to succeed.

Example 5 Let us continue examples 3 and 4. Assume that all tuples in $h_{SadmindPing}$ have timestamps earlier than every tuple in $h_{SadmindBOF}$. By comparing the contents of $C(h_{SadmindPing})$ and $P(h_{SadmindBOF})$, it is clear that *VulnerableSadmind* (152.1.19.5) (among others) in $P(h_{SadmindBOF})$ is also in $C(h_{SadmindPing})$. Thus, $h_{SadmindPing}$ prepares for, and should be correlated with $h_{SadmindBOF}$. □

Given a sequence S of hyper-alerts, a hyper-alert h in S is a *correlated hyper-alert*, if there exists another hyper-alert h' in S such that either h prepares for h' or h' prepares for h. If no such h' exists, h is called an *isolated hyper-alert*. Our goal is to discover all pairs of hyper-alerts h_1 and h_2 in S such that h_1 prepares for h_2.

2.2.1 Temporal Constraints for Hyper-alerts

As discussed earlier, we allow multiple alerts to be aggregated into a hyper-alert, which gives some flexibility in reasoning about alerts. However, the definition of hyper-alert is overly flexible in some situations; it allows alerts that occur at arbitrary points in time to be treated as a single hyper-alert. Although some attackers usually spread their intrusive activities over time, aggregating alerts at arbitrary time points is still overly broad, and may affect the effectiveness of alert correlation.

In the following, we introduce two temporal constraints for hyper-alerts. The purpose of these temporal constraints is to restrict the alert aggregation to meaningful ones. We are particularly interested in hyper-alerts that satisfy at least one of the constraints. However, most of our discussion in this paper applies to general hyper-alerts. Thus, we will not specifically indicate the constraints if it is not necessary.

Definition 5 Given a time duration D (e.g., 100 seconds), a hyper-alert h satisfies *duration constraint of* D if $Max\{t.end\_time|\forall t \in h\} - Min\{t.begin\_time|\forall t \in h\} < D$.

Definition 6 Given a time interval I (e.g., 10 seconds), a hyper-alert h satisfies *interval constraint of* I if (1) h has only one tuple, or (2) for all t in h, there exist another t' in h such that there exist $t.begin\_time < T < t.end\_time$, $t'.begin\_time < T' < t'.end\_time$, and $|T - T'| < I$.

The temporal constraints are introduced to prevent unreasonable aggregation of alerts. However, this does not imply that alerts have to be aggregated. Indeed, in our initial experiments, we treat each alert as an individual hyper-alert. In other words, aggregation of alerts is an option provided by our model, and temporal constraints are restrictions that make the aggregated hyper-alerts meaningful. (In related work being performed by the authors, which is not reported here, the results have shown the usefulness of alert aggregation in reducing the complexity of intrusion analysis.)

There could be other meaningful constraints for hyper-alerts, depending on the semantics of the relevant contexts. We will identify such constraints when we encounter them in our research; however, they are not the current focus and will not be discussed in this paper.

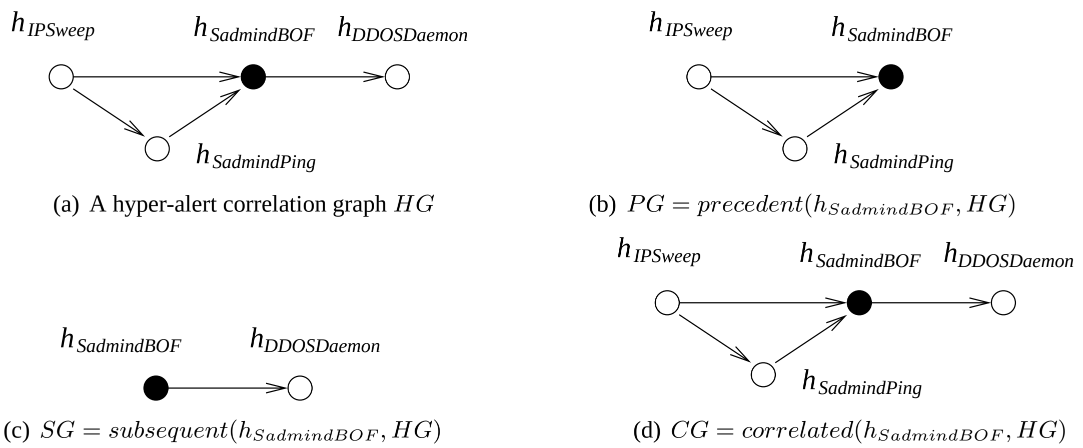
2.3 Hyper-alert Correlation Graph

The prepare-for relation between hyper-alerts provides a natural way to represent the causal relationship between correlated hyper-alerts. In the following, we introduce the notion of a *hyper-alert correlation graph* to represent attack scenarios on the basis of the prepare-for relation. As we will see, the hyper-alert correlation graph reflects the high-level strategies or logical steps behind a sequence of attacks.

Definition 7 A *hyper-alert correlation graph* HG = (N, E) is a connected DAG (directed acyclic graph), where the set N of nodes is a set of hyper-alerts, and for each pair of nodes $n_1, n_2 \in N$, there is an edge from n_1 to n_2 in E if and only if n_1 prepares for n_2.

Example 6 Suppose in a sequence of hyper-alerts we have the following ones: $h_{IPSweep}$, $h_{SadmindPing}$, $h_{SadmindBOF}$, and $h_{DDOSDaemon}$. The hyper-alerts $h_{SadmindBOF}$ and $h_{SadmindPing}$ have been explained in examples 2 and 3, respectively. Suppose $h_{IPSweep}$ represents an IP Sweep attack, and $h_{DDOSDaemon}$ represents the activity of a DDOS daemon program. Assume we have: $h_{IPSweep}$ prepares for $h_{SadmindPing}$ and $h_{SadmindBOF}$, respectively, $h_{SadmindPing}$ prepares for $h_{SadmindBOF}$, and $h_{SadmindBOF}$ prepares for $h_{DDOSDaemon}$. These are intuitively shown in a hyper-alert correlation graph in Figure 1(a). □

The hyper-alert correlation graph provides an intuitive representation of correlated hyper-alerts. With this



(a) A hyper-alert correlation graph HG

(b) $PG = precedent(h_{SadmindBOF}, HG)$

(c) $SG = subsequent(h_{SadmindBOF}, HG)$

(d) $CG = correlated(h_{SadmindBOF}, HG)$

Figure 1: Hyper-alerts correlation graphs

notion, the goal of alert correlation can be rephrased as the discovery of hyper-alert correlation graphs that have maximal number of nodes from a sequence of hyper-alerts.

In addition to getting all the correlated hyper-alerts, it is often desirable to discover those that are directly or indirectly correlated to one particular hyper-alert. For example, if an IDS detects a DDOS daemon running on a host, it would be helpful to inform the administrator how this happened, that is, report all the alerts that directly or indirectly prepare for the DDOS daemon. Therefore, we define the following operations on hyper-alert correlation graphs.

Definition 8 Given a hyper-alert correlation graph HG = (N, E) and a hyper-alert n in N, $precedent$ (n, HG) is an operation that returns the maximal sub-graph PG = (N', E') of HG that satisfies the following conditions: (1) $n \in N'$, (2) for each $n' \in N'$ other than n, there is a directed path from n' to n, and (3) each edge $e \in E'$ is in a path from a node n' in N' to n. The resulting graph PG is called the *precedent graph of* n *w.r.t.* HG.

Definition 9 Given a hyper-alert correlation graph HG = (N, E) and a hyper-alert n in N, $subsequent$ (n, HG) is an operation that returns the maximum sub-graph SG = (N', E') of HG that satisfies the following conditions: (1) $n \in N'$, (2) for each $n' \in N'$ other than n, there is a directed path from n to n', and (3) each edge $e \in E'$ is in a path from n to a node n' in N'. The resulting graph SG is called the *subsequent graph of* n *w.r.t.* HG.

Definition 10 Given a hyper-alert correlation graph HG = (N, E) and a hyper-alert n in N, $correlated$ (n, HG) is an operation that returns the maximal sub-graph CG = (N', E') of HG that satisfies the following conditions: (1) $n \in N'$, (2) for each $n' \in N'$ other than n, there is either a path from n to n', or a path from n' to n, and (3) each edge $e \in E'$ is either in a path from a node in N' to n, or in a path from n to a node in N'. The resulting graph CG is called the *correlated graph of* n *w.r.t.* HG.

Intuitively, the precedent graph of n w.r.t. HG describes all the hyper-alerts in HG that prepare for n directly or indirectly, the subsequent graph of n w.r.t. HG describes all the hyper-alerts in HG for which n prepares directly or indirectly, and the correlated graph of n w.r.t. HG includes all the hyper-alerts in HG that are correlated to n directly or indirectly. It is easy to see that $correlated(n, HG) = precedent(n, HG) \cup subsequent(n, HG)$.

Assuming the black node $h_{SadmindBOF}$ in figure 1(a) is the hyper-alert of concern, figures 1(b) to 1(d) display the precedent graph, subsequent graph, and correlated graph of $h_{SadmindBOF}$ w.r.t. the hyper-alert correlation graph in figure 1(a), respectively. Note that figure 1(d) is the same as figure 1(a). This is because all the hyper-alerts in figure 1(a) are related to $h_{SadmindBOF}$ via the prepare-for relation. In reality, it is certainly possible that not all hyper-alerts are related to the hyper-alert of concern. In this case the correlated graph only reveals those directly or indirectly correlated to that hyper-alert.

The hyper-alert correlation graph is not only an intuitive way to represent attack scenarios constructed through alert correlation, but also reveals opportunities to improve intrusion detection. First, the hyper-alert correlation graph can potentially reveal the intrusion strategies behind the attacks, and thus lead to better understanding of the attacker's intention. Second, assuming some attackers exhibit patterns in their attack strategies, we can use the hyper-alert correlation graph to profile previous attacks and thus identify on-going attacks by matching to the profiles. A partial match to a profile may indicate attacks possibly missed by the IDSs, and thus lead to human investigation and improvement of the IDSs. Nevertheless, additional research is necessary to demonstrate the usefulness of hyper-alert correlation graphs for this purpose.

2.4 Discussion

Our method has several advantages. First, our approach provides a high-level representation of correlated alerts that reveals the causal relationships between them. As we will see in Section 4, the hyper-alert correlation graphs generated by our implementation clearly show the strategies behind these attacks. Second, our approach can potentially reduce the impact caused by false alerts by providing a way to differentiate alerts. While true alerts are more likely to be correlated with other alerts, false alerts, which do not correspond to any actual attacks, tend to be more random than the true alerts, and are less likely to be correlated to others.

Our method does not depend on predefined attack scenarios to discover sequences of related attacks. This is a common feature that our method shares with JIGSAW [18]. However, unlike JIGSAW, our method can correlate alerts as long as there are signs of connections between them, even if some alerts correspond to failed attack attempts, or the IDSs fail to detect some related attacks. In addition, our method provides an intuitive representation (i.e., hyper-alert correlation graph) of correlated alerts, which reveals the high-level strategy behind the attacks.

Our decision certainly has its pros and cons. On the positive side, our method is simple and yet able to correlate related alerts even when the IDSs miss certain attacks. However, on the negative side, our method may correlate alerts incorrectly when one attack seemingly prepares for another. In other words, our method has both a higher true correlation rate and a higher false correlation rate than JIGSAW. Nevertheless, Section 4 shows experimentally that our method has a low false correlation rate.

Our approach has several limitations. First, our method depends on the underlying IDSs to provide alerts. Though our reasoning process can compensate for undetected attacks, the attacks missed by the IDSs certainly have a negative effect on the results. If the IDS misses a critical attack that links two stages of a series of attacks, the related hyper-alerts may be split into two separate hyper-alert correlation graphs. In the extreme case where the underlying IDSs missed all the attacks, our approach cannot do anything.

Second, our approach is not fully effective for alerts between which there is no prepare-for relationship, even if they may be related. For example, an attacker may launch concurrent *Smurf* and *SYN flooding* attacks against the same target; however, our approach will not correlate the corresponding alerts, though there are connections between them (i.e., same time and same target). Therefore, our method should be used along with other, complementary techniques (e.g., the probabilistic alert correlation [19]).

3 Implementation

We have implemented an off-line intrusion alert correlator using the method discussed in section 2. Figure 2 shows the architecture. It consists of a knowledge base, an alert preprocessor, a correlation engine, a

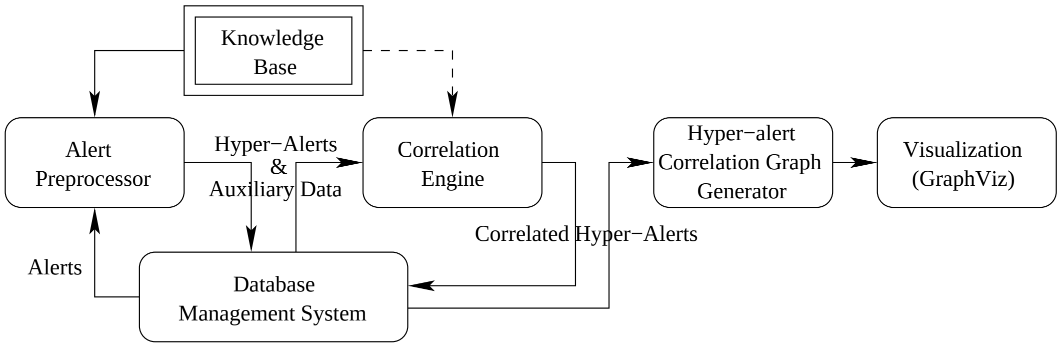


Figure 2: The architecture of the intrusion alert correlator

hyper-alert correlation graph generator, and a visualization component. All these components except for the visualization component interact with a DBMS, which provides persistent storage for the intermediate data as well as the correlated alerts. The program was written in Java, with JDBC to access the database. To save development effort, we use the GraphViz package [2] as the visualization component.

The knowledge base contains the necessary information about hyper-alert types as well as implication relationships between predicates. In our current implementation, the hyper-alert types and the relationship between predicates are specified in an XML file. When the alert correlator is initialized, it reads the XML file, and then converts and stores the information in the knowledge base.

Our current implementation assumes the alerts reported by IDSs are stored in the database. Using the information in the knowledge base, the alert preprocessor generates hyper-alerts as well as auxiliary data from the original alerts. The correlation engine then performs the actual correlation task using the hyper-alerts and the auxiliary data. After alert correlation, the hyper-alert correlation graph generator extracts the correlated alerts from the database, and generates the graph files in the format accepted by GraphViz. As the final step of alert correlation, GraphViz is used to visualize the hyper-alert correlation graphs.

3.1 Preprocessing and Correlation of Hyper-alerts

Preprocessing and correlation of hyper-alerts are the major tasks of the alert correlator. These tasks are performed using the hyper-alert type information stored in the knowledge base. As discussed earlier, the knowledge base stores two types of information: the implication relationships between predicates and the hyper-alert type information. When the alert correlator reads in the hype-alert types, it generates the prerequisite and consequence sets of each hyper-alert type. In addition, it expands the consequence set of each hyper-alert type by including all the predicates in the knowledge base implied by the consequence set. We call the result the *expanded consequence set* of the corresponding hyper-alert type. (Similar to the consequence set of a hyper-alert type, we may instantiate the expanded consequence set with a hyper-alert instance and get the expanded consequence set of the hyper-alert.)

To simplify preprocess and correlation of hyper-alerts, we make the following assumptions.

Assumption 1 *Given a set $P = \{p_1(x_{11}, ..., x_{1k_1}), ..., p_m(x_{m1}, ..., x_{mk_m})\}$ of predicates, for any set of instantiations of the variables $x_{11}, ..., x_{1k_1}, ..., x_{m1}, ..., x_{mk_m}$, deriving all predicates implied by P followed by instantiating all the variables leads to the same result as instantiating all the variables and then deriving all the predicates implied by the instantiated predicates.*

Assumption 2 *All predicates are uniquely identified by their names and the special characters "(", ")", and "," do not appear in predicate names and arguments.*

The major preparation for alert correlation is performed at the preprocessing phase. The alert preprocessor generates hyper-alerts from the alerts reported by IDSs and instantiates the prerequisite and expanded consequence sets of each hyper-alert. The current implementation generates one hyper-alert from each alert, though our method allows aggregating multiple alerts into one hyper-alert (see Section 2). Note that having the expanded consequence set of a hyper-alert, we can test if some predicates in the consequence set imply a predicate by checking whether the latter predicate is included in the expanded consequence set.

We encode instantiated predicates as strings, and thus further transform the alert correlation problem to a string matching problem. Specifically, each instantiated predicate is encoded as the predicate name followed by the character "(", followed by the sequence of arguments separated with the character ",", and finally followed by the character ")". Thus, under assumption 2, comparing instantiated predicates is equivalent to comparing the encoded strings.

We store the encoded prerequisite and expanded consequence sets in two tables, *PrereqSet* and *ExpandedConseqSet*, along with the corresponding hyper-alert ID and timestamp, assuming that each hyper-alert is uniquely identified by its ID. (Note that one hyper-alert may have multiple tuples in these tables.) Both tables have attributes *HyperAlertID*, *EncodedPredicate*, *begin_time*, and *end_time*, with meanings as indicated by their names. As a result, alert correlation can be performed using the following SQL statement.

```
SELECT DISTINCT c.HyperAlertID, p.HyperAlertID
FROM PrereqSet p, ExpandedConseqSet c
WHERE p.EncodedPredicate = c.EncodedPredicate AND c.end_time < p.begin_time
```

The correctness of our implementation method is guaranteed by the following theorem.

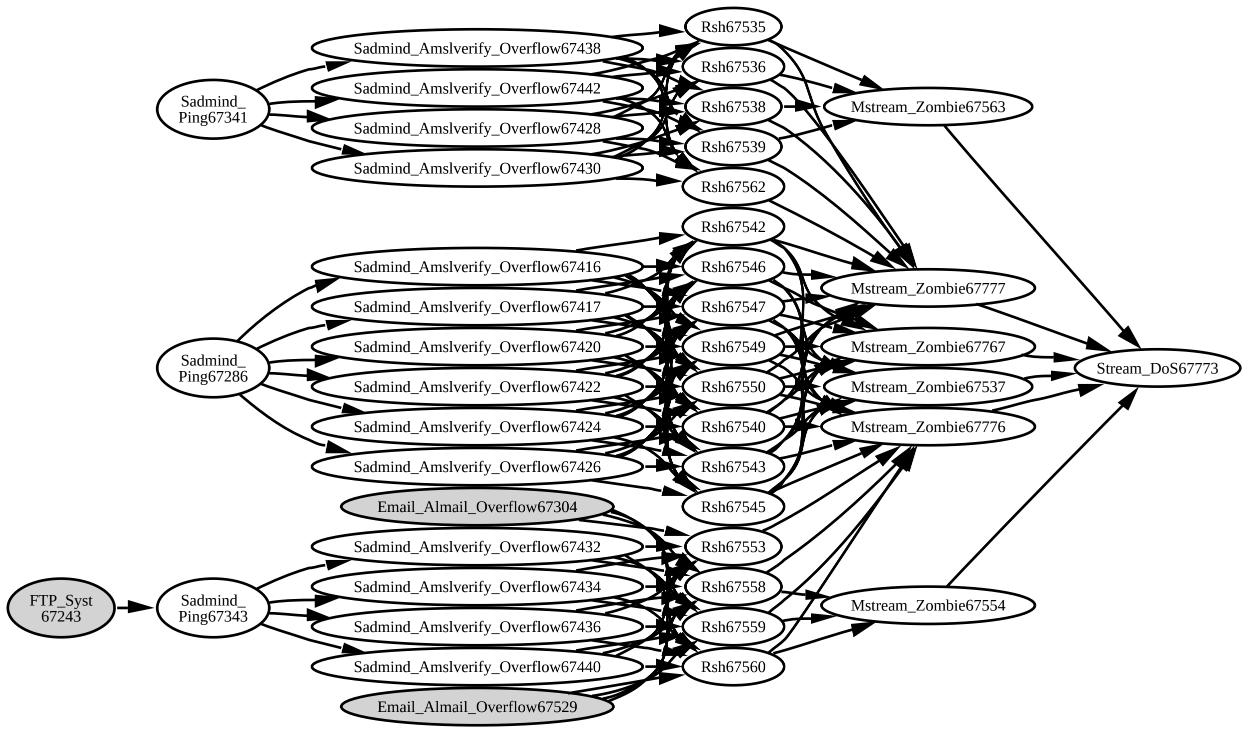
Theorem 1 *Under assumptions 1 and 2, our implementation method discovers all and only the hyper-alert pairs such that the first one of the pair prepares for the second one.*

Proof Sketch: Consider a pair of hyper-alerts h_1 and h_2 such that h_1 prepares for h_2. By Definition 4, there exists $p \in P(h_2)$ and $C \subseteq C(h_1)$ such that for all $c \in C$, $c.end\_time < p.begin\_time$ and the conjunction of all predicates in C implies p. By assumption 1, p should be in the expanded consequence set of h_1 (but associated with a different timestamp). Thus, both $PrereqSet$ and $ExpandedConseqSet$ have a tuple that has the same encoded predicate along with the appropriate hyper-alert ID and timestamp. As a result, the SQL statement will output that h_1 prepares for h_2.

Suppose the SQL statement outputs that h_1 prepares for h_2. Then there exist t_1 in $ExpandedConseqSet$ and t_2 in $PrereqSet$ such that $t_1.EncodedPredicate = t_2.EncodedPredicate$ and $t_1.end\_time < t_2.begin\_time$. By assumption 2, $t_1.EncodedPredicate$ and $t_2.EncodedPredicate$ must be the same instantiated predicate. Let us refer to this predicate as p_1 when it is in the expanded consequence set of h_1, and as p_2 when it is in $P(h_2)$. Thus, $p_1 = p_2$ and $p_1.end\_time < p_2.begin\_time$. If p_1 is also in $C(h_1)$, let $C = \{p_1\}$. Otherwise, let C be the set of predicates in $C(h_1)$ that are instantiated by the same tuple in h_1 as p_1. By the way in which an expanded consequence set is constructed and assumption 1, p_1 is implied by the predicates in C. This is to say that p_2 is implied by $C \subseteq C(h_1)$ such that for all $c \in C$, $c.end\_time < p_2.begin\_time$. Thus, h_1 prepares for h_2 by Definition 4. $\square$

4 Experimental Results

To evaluate the effectiveness of our method in constructing attack scenarios and its ability to differentiate true and false alerts, we performed a series of experiments using the 2000 DARPA intrusion detection scenario specific datasets (LLDOS 1.0 and LLDOS 2.0.2) [12]. LLDOS 1.0 contains a series of attacks in which



Rsh67535
Sadmind_Amslverify_Overflow67438
Rsh67536
Sadmind_Amslverify_Overflow67442
Sadmind_ Ping67341
Rsh67538
Mstream_Zombie67563
Sadmind_Amslverify_Overflow67428
Rsh67539
Sadmind_Amslverify_Overflow67430
Rsh67562
Rsh67542
Sadmind_Amslverify_Overflow67416
Rsh67546
Mstream_Zombie67777
Sadmind_Amslverify_Overflow67417
Rsh67547
Sadmind_Amslverify_Overflow67420
Rsh67549
Mstream_Zombie67767
Sadmind_ Ping67286
Stream_DoS67773
Sadmind_Amslverify_Overflow67422
Rsh67550
Mstream_Zombie67537
Sadmind_Amslverify_Overflow67424
Rsh67540
Mstream_Zombie67776
Sadmind_Amslverify_Overflow67426
Rsh67543
Email_Almail_Overflow67304
Rsh67545
Sadmind_Amslverify_Overflow67432
Rsh67553
Sadmind_Amslverify_Overflow67434
Rsh67558
FTP_Syst 67243
Sadmind_ Ping67343
Mstream_Zombie67554
Sadmind_Amslverify_Overflow67436
Rsh67559
Sadmind_Amslverify_Overflow67440
Rsh67560
Email_Almail_Overflow67529

Figure 3: The (only) hyper-alert correlation graph discovered in the inside network traffic of LLDOS 1.0.

an attacker probes, breaks-in, installs the components necessary to launch a Distributed Denial of Service (DDOS) attack, and actually launches a DDOS attack against an off-site server. LLDOS 2.0.2 includes a similar sequence of attacks run by an attacker who is a bit more sophisticated than the first one.

Each dataset includes the network traffic collected from both the DMZ and the inside part of the evaluation network. We have performed four sets of experiments, each with either the DMZ or the inside network traffic of one dataset. In each experiment, we replayed the selected network traffic in an isolated network monitored by a RealSecure Network Sensor 6.0 [9]. RealSecure was chosen because it has an extensive set of well documented attack signatures. In all the experiments, the Network Sensor was configured to use the *Maximum_Coverage* policy with a slight change, which forced the Network Sensor to save all the reported alerts. Our alert correlator was then used to process the alerts generated by RealSecure.

We mapped each alert type reported by RealSecure to a hyper-alert type with the same name. The prerequisite and consequence of each hyper-alert type were specified according to the descriptions of the attack signatures provided with the RealSecure Network Sensor 6.0. The hyper-alert types (as well as the implication relationships between predicates) are given in Appendix A. Note that our method is based on the knowledge about attacks. Thus, it does not need any training data set.

4.1 Effectiveness of Alert Correlation

Our first goal of these experiments is to evaluate the effectiveness of our method in constructing attack scenarios from alerts. Before discussing the quantitative measures, let us first look at one of the hyper-alert correlation graphs generated by the alert correlator.

Figure 3 shows the (only) hyper-alert correlation graph discovered from the inside network traffic in

LLDOS 1.0. Each node in Figure 3 represents a hyper-alert. The text inside the node is the name of the hyper-alert type followed by the hyper-alert ID. (We will follow this convention for all the hyper-alert correlation graphs.)

There are 44 hyper-alerts in this graph, including 3 false alerts, which are shown in gray. We will discuss the false alerts later. The true hyper-alerts can be divided into five stages horizontally. The first stage consists of three *Sadmind_Ping* alerts, which the attacker used to find out the vulnerable *Sadmind* services. The three alerts are from source IP address 202.077.162.213, and to destination IP addresses 172.016.112.010, 172.016.115.020, and 172.016.112.050, respectively. The second stage consists of fourteen *Sadmind_Amslverify_Overflow* alerts. According to the description of the attack scenario, the attacker tried three different stack pointers and two commands in *Sadmind_Amslverify_Overflow* attacks for each victim host until one attempt succeeded. All the above three hosts were successfully broken into. The third stage consists of some *Rsh* alerts, with which the attacker installed and started the *mstream* daemon and master programs. The fourth stage consists of alerts corresponding to the communications between the DDOS master and daemon programs. Finally, the last stage consists of the DDOS attack.

We can see clearly that the hyper-alert correlation graph reveals the structure as well as the high-level strategy of the sequence of attacks. The other hyper-alert correlation graphs and the corresponding analysis are included in Appendix B.

This hyper-alert correlation graph is still not perfect. The two *Email_Almail_Overflow* hyper-alerts (shown in gray in Figure 3) are false alerts, and are mis-correlated with the *Rsh* alerts, though it is possible that an attacker uses these attacks to gain access to the victim system and then copy the DDOS program with *Rsh*. The *FTP_Syst* hyper-alert is also a false one; it is correlated with one of the *Sadmind_Ping*s, because an attacker may use *FTP_Syst* to gain the OS information and then launch an *Sadmind_Ping* attack. Moreover, the attacker used a *telnet* as a part of the sequence of attacks, but this graph does not include the corresponding hyper-alert.

Another interesting issue is that we correlated alerts that are not attacks. In both DMZ and inside traffic of LLDOS 2.0.2, we correlated an *Email_Ehlo* with an *Email_Turn* from 135.013.216.191 to 172.016.113.105. Our further analysis indicated that these were normal and related activities between email servers.

To better understand the effectiveness of our method, we examine the *completeness* and the *soundness* of alert correlation. The completeness of alert correlation assesses how well we can correlate related alerts together, while the soundness evaluates how correctly the alerts are correlated. We introduce two simple measures, R_c and R_s, to quantitatively evaluate completeness and soundness, respectively:

$$R_c = \frac{\#correctly\ correlated\ alerts}{\#related\ alerts}, \quad R_s = \frac{\#correctly\ correlated\ alerts}{\#correlated\ alerts}.$$

Counting the numbers in R_c and R_s is easy, given the description of the attacks in the DARPA datasets. However, RealSecure generated duplicate alerts for several attacks. In our experiments, we counted the duplicate alerts as different ones. False alerts are counted (as incorrectly correlated alerts) so long as they are correlated. Though non-intrusive alerts (e.g., the above *Email_Ehlo* and *Email_Turn*) are not attacks, if they are related activities, we counted them as correctly correlated ones.

Table 1 shows the results about completeness and soundness of the alert correlation for the two datasets. As shown by the values of R_s, most of the hyper-alerts are correlated correctly. The completeness measures (R_c) are satisfactory for LLDOS 1.0; however, they are only 62.5% and 66.7% for the DMZ and inside traffic in LLDOS 2.0.2. Our further analysis reveals that all the hyper-alerts missed are those triggered by the *telnet*s that the attacker used to access a victim host. Each *telnet* triggered three alerts, *TelnetEnvAll*, *TelnetXDisplay* and *TelnetTerminalType*. According to RealSecure's description, these alerts are about attacks that are launched using environmental variables (*TelnetEnvAll*) in a telnet session, including XDisplay (*TelnetXDisplay*) and TerminalType (*TelnetTerminalType*). However, according to the description of the

Table 1: Completeness and soundness of alert correlation.

	LLDOS 1.0		LLDOS 2.0.2	
	DMZ	Inside	DMZ	Inside
# correctly correlated alerts	54	41	5	12
# related alerts	57	44	8	18
# correlated alerts	57	44	5	13
completeness measure R_c	94.74%	93.18%	62.5%	66.7%
soundness measure R_s	94.74%	93.18%	100%	92.3%

Table 2: Ability to differentiate true and false alerts

Dataset		# observable attacks	Tool	# alerts	# detected attacks	Detection rate	# true alerts	False alert rate
LLDOS 1.0	DMZ	89	RealSecure	891	51	57.30%	57	93.60%
			Our method	57	50	56.18%	54	5.26%
	Inside	60	RealSecure	922	37	61.67%	44	95.23%
			Our method	44	36	60%	41	6.82%
LLDOS 2.0.2	DMZ	7	RealSecure	425	4	57.14%	6	98.59%
			Our method	5	3	42.86%	3	40%
	Inside	15	RealSecure	489	12	80.00%	16	96.73%
			Our method	13	10	66.67%	10	23.08%

datasets, the attacker did not launch these attacks, though he did telnet to one victim host after gaining access to it. Nevertheless, to be conservative, we consider them as related alerts in our evaluation. Considering these facts, we can conclude that our method is effective for these datasets.

4.2 Ability to Differentiate Alerts

Our second goal of these experiments is to see how well alert correlation can be used to differentiate false alerts and true alerts. As we conjectured in Section 2, false alerts, which do not correspond to any real attacks, tend to be more random than the actual alerts, and are less likely to be correlated to others. If this conjecture is true, we can divert more resources to deal with correlated alerts, and thus improve the effectiveness of intrusion response.

To understand this issue, we deliberately drop the uncorrelated alerts and then compare the resulting detection rate and false alert rate with the original ones of RealSecure.

We counted the number of actual attacks and false alerts according to the description included in the datasets. False alerts can be identified easily by comparing the alerts with the attack description provided with the datasets; however, counting the number of attacks is subjective, since the number of attacks depends on how one views the attacks. Having different views of the attacks may result in different numbers.

We adopted the following way to count the number of attacks in our experiments. The initial phase of the attacks involved an IP Sweep attack. Though many packets were involved, we counted them as a single attack. Similarly, the final phase had a DDOS attack, which generated many packets but was also counted as one attack. For the rest of the attacks, we counted each action (e.g., *telnet*, *Sadmind_Ping*) initiated by the attacker as one attack. The numbers of attacks observable in these datasets are shown in Table 2. Note that some activities such as *telnet* are not usually considered as attacks; however, we counted them here if the attacker used them as part of the attacks.

RealSecure Network Sensor 6.0 generated duplicate alerts for certain attacks. For example, the same *rsh* connection that the attacker used to access the compromised host triggered two alerts. As a result, the number of true alerts (i.e., the alerts corresponding to actual attacks) is greater than the number of detected attacks. The detection rates were calculated as $\frac{\#detected\ attacks}{\#observable\ attacks}$, while the false alert rates were computed as $(1 - \frac{\#true\ alerts}{\#alerts})$.

Table 2 summarizes the results of these experiments. For the DMZ network traffic in LLDOS 1.0, RealSecure generated 891 alerts. According to the description of the data set, 57 out of the 891 alerts are true alerts, 51 attacks are detected, and 38 attacks are missed. Thus, the detection rate of RealSecure Network Sensor is 57.30%, and the false alert rate is 93.60%.[2] Our intrusion alert correlator processed the alerts generated by the RealSecure Network Sensor. As shown in Table 2, 57 alerts remain after correlation, 54 of them are true alerts, and 50 attacks are covered by these alerts. Thus, the detection rate and the false alert rate after alert correlation are 56.18% and 5.26%, respectively. The results for the other datasets are also shown in Table 2.

The experimental results in Table 2 show that discarding uncorrelated alerts reduces the false alert rates greatly without sacrificing the detection rate too much. Thus, it is reasonable to treat correlated alerts more seriously than uncorrelated ones. However, simply discarding uncorrelated alerts is dangerous, since some of them may be true alerts, which correspond to individual attacks or attacks our method fails to correlate.

5 Related Work

Intrusion detection has been studied for about twenty years, since Anderson's report [1]. A survey of the early work on intrusion detection is given in [13], and an excellent overview of current intrusion detection techniques and related issues can be found in a recent book [3]. The introduction has discussed the other alert correlation methods; we do not repeat them here.

Several techniques have used prerequisite and consequence of attacks for vulnerability analysis purposes. In [15], model checking technique was applied to analyze network vulnerabilities on the basis of prerequisites and results (i.e., consequences) of exploits (i.e., attacks) along with hosts and network connectivity information. In [16] and [11], the technique in [15] was further extended to generate and analyze all possible attacks against a vulnerable networked system. These techniques are focused on analyzing what attacks *may happen* to a given system. In contrast, our purpose is to reconstruct what *have happened* to a given system according to the alerts reported by IDSs, and our technique has to deal with the inaccuracy of IDSs (i.e., false alerts and undetected attacks). We consider our method as complementary to these vulnerability analysis techniques.

6 Conclusion and Future Work

This paper presented a practical method for constructing attack scenarios through alert correlation, using prerequisites and consequences of intrusions. The approach was based on the observation that in series of attacks, component attacks were usually not isolated, but related as different stages, with the earlier stages preparing for the later ones. This paper proposed a formal framework to represent alerts along with their prerequisites and consequences, and developed a method to correlate related hyper-alerts together, including an intuitive representation of correlated alerts that reveals the attack scenario of the corresponding attacks. We also developed an off-line tool on the basis of the formal framework. Our initial experiments have demonstrated the potential of our method in correlating alerts and differentiating false and true alerts.

[2]Choosing less aggressive policies than Maximum_Coverage can reduce the false alert rate; however, we may also lose the opportunity to detect some attacks.

Several issues are worth future research. In particular, we plan to develop better ways to specify hyper-alert types, especially how to represent predicates to be included in their prerequisite and consequence sets to get the best performance for alert correlation. In addition, we will continue to study the effectiveness of our approach and improve the correlation method if possible.

Acknowledgement

This work is partially supported by the U.S. Army Research Office under grant DAAD19-02-1-0219, by the National Science Foundation under grant 0207297, and by NCSU Center for Advanced Computing & Communication.

References

[1] J. P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, 1980.

[2] AT & T Research Labs. Graphviz - open source graph layout and drawing software. http://www.research.att.com/sw/tools/graphviz/.

[3] R.G. Bace. *Intrusion Detection*. Macmillan Technology Publishing, 2000.

[4] F. Cuppens. Managing alerts in a multi-intrusion detection environment. In *Proceedings of the 17th Annual Computer Security Applications Conference*, December 2001.

[5] F. Cuppens and A. Miege. Alert correlation in a cooperative intrusion detection framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, May 2002.

[6] F. Cuppens and R. Ortalo. LAMBDA: A language to model a database for detection of attacks. In *Proc. of Recent Advances in Intrusion Detection (RAID 2000)*, pages 197–216, September 2000.

[7] O. Dain and R.K. Cunningham. Fusing a heterogeneous alert stream into scenarios. In *Proceedings of the 2001 ACM Workshop on Data Mining for Security Applications*, pages 1–13, November 2001.

[8] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection*, LNCS 2212, pages 85 – 103, 2001.

[9] ISS, Inc. RealSecure intrusion detection system. http://www.iss.net.

[10] H.S. Javits and A. Valdes. The NIDES statistical component: Description and justification. Technical report, SRI International, Computer Science Laboratory, 1993.

[11] S. Jha, O. Sheyner, and J.M. Wing. Two formal analyses of attack graphs. In *Proceedings of the 15th Computer Security Foundation Workshop (To appear)*, June 2002.

[12] MIT Lincoln Lab. 2000 DARPA intrusion detection scenario specific datasets. http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html, 2000.

[13] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May 1994.

[14] P. Ning, Y. Cui, and D. S Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (to appear)*, Washington, D.C., November 2002. Available at http://infosec.csc.ncsu.edu/pubs/ccs02.pdf.

[15] R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 156–165, May 2000.

[16] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2002.

[17] S. Staniford, J.A. Hoagland, and J.M. McAlerney. Practical automated detection of stealthy portscans. To appear in Journal of Computer Security, 2002.

[18] S. Templeton and K. Levit. A requires/provides model for computer attacks. In *Proceedings of New Security Paradigms Workshop*, pages 31 – 38. ACM Press, September 2000.

[19] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, pages 54–68, 2001.

[20] G. Vigna and R. A. Kemmerer. NetSTAT: A network-based intrusion detection system. *Journal of Computer Security*, 7(1):37–71, 1999.

A Hyper-alert Types Used in Our Experiments

This appendix describes the hyper-alert types used in our experiments, which are listed in Table 3. The hyper-alert types and predicates are self-explanatory. Detailed description of the corresponding attacks can be found in the document provided along with RealSecure Network Sensor 6.0 [9].

We did not specify the prerequisite and consequence of *Admind*. This is equivalent to excluding all *Admind* hyper-alerts from consideration. The reason is that *Admind* is too generic. RealSecure generates an *Admind* alert each time when there is an event involving the *sadmind* service. Examples of these alerts include *Sadmind_Ping* with which an attacker gains information about the *sadmind* service, and *Sadmind_Amslverify_Overflow* which is a buffer overflow attack against the *sadmind* service. These alerts are so different that we cannot have reasonably specific prerequisites and consequences that work for all of them. Fortunately, there is always a more specific alert (e.g., *Sadmind_Ping* and *Sadmind_Amslverify_Overflow*) that occurs along with each *Admind* alert in the experiments. Thus, we discard the *Admind* alerts while using the more specific ones.

As discussed in the paper, choosing predicates to specify prerequisite and consequence of a hyper-alert type is a subjective process. Indeed, predicates are the basis of alert correlation in our method. To maximize the performance of alert correlation, we may not want to include all necessary conditions in the prerequisite or all possible results in the consequence. For example, though the prerequisite of *FTP_Syst* requires that the user has access to the ftp server, most of *FTP_Syst* alerts are triggered by certain legitimate ftp clients which use the SYST command after connecting to a ftp server; it is very unlikely that an attacker uses *FTP_Syst* to discover the OS information after he gains access to the host. Thus, we do not include *GainAccess (DestIP)* in the prerequisite of *FTP_Syst*. (Indeed, we would have more falsely correlated hyper-alerts if we were to do it.) However, we include *GainAccess (DestIP)* in the prerequisite of *FTP_Put*, since an attacker may use ftp to copy something to the victim host after he gains access to it. In other words, we include a predicate in the prerequisite of a hyper-alert type, if it is necessary for the corresponding attack to succeed *and* an attacker may launch this attack on the basis of it. Similarly, we include a predicate in the consequence of a hyper-alert type, if it is a possible result of the attack *and* an attacker may launch a follow-up attack on the

Table 3: Hyper-alert types used in the experiments.
The *fact* component of all these hyper-alert types is {SrcIP, SrcPort, DestIP, DestPort}.

Hyper-alert Type	Prerequisite	Consequence
Admind		
Email_Almail_Overflow	ExistService(DestIP, DestPort) $\wedge$ VulnerableMailServer(DestIP)	{GainAccess(DestIP)}
Email_Debug	ExistService(DestIP, DestPort) $\wedge$ SendmailInDebugMode(DestIP)	{GainAccess(DestIP)}
Email_Ehlo	ExistService(DestIP, DestPort) $\wedge$ SMTPSupportEhlo(DestIP)	{GainSMTPInfo(SrcIP,DestIP)}
Email_Turn	ExistService(DestIP, DestPort) $\wedge$ SMTPSupportTurn(SrcIP, DestIP)	{MailLeakage(DestIP)}
FTP_Pass	ExistService(DestIP,DestPort)	
FTP_Put	ExistService(DestIP,DestPort) $\wedge$ GainAccess(DestIP)	{SystemCompromised (DestIP)}
FTP_Syst	ExistService(DestIP,DestPort)	{GainOSInfo(DestIP)}
FTP_User	ExistService(DestIP,DestPort)	
HTTP_ActiveX	ExistActiveXControl(SrcIP)	{SystemCompromised(SrcIP)}
HTTP_Cisco_Catalyst_Exec	ExistService (DestIP, DestPort) $\wedge$CiscoCatalyst3500XL(DestIP)	{GainAccess(DestIP)}
HTTP_Java	JavaEnabledBrowser(SrcIP)	{SystemCompromised(SrcIP)}
HTTP_Shells	ExistService(DestIP, DestPort) $\wedge$ VulnerableCGIBin(DestIP) $\wedge$ OSUNIX(DestIP)	{GainAccess(DestIP)}
Mstream_Zombie	SystemCompromised(SrcIP) $\wedge$ SystemCompromised(DestIP)	{ReadyToLaunchDOSAttack}
Port_Scan		{ExistService(DestIP,DestPort)}
RIPAdd		
RIPExpire		
Rsh	GainAccess(SrcIP) $\wedge$ GainAccess(DestIP)	{SystemCompromised(SrcIP), SystemCompromised(DestIP)}
Sadmind_Amslverify_Overflow	VulnerableSadmind(DestIP) $\wedge$ OSSolaris(DestIP)	{GainAccess (DestIP)}
Sadmind_Ping	OSSolaris(DestIP)	{VulnerableSadmind(DestIP)}
SSH_Detected		
Stream_DoS	ReadyToLaunchDOSAttack	{DOSAttackLaunched}
TCP_Urgent_Data		{SystemAttacked(DestIP)}
TelnetEnvAll		{SystemAttacked(DestIP)}
TelnetTerminaltype	ExistService(DestIP, DestPort)	{GainTerminalType(DestIP)}
TelnetXdisplay	ExistService(DestIP, DestPort)	{SystemAttacked(DestIP)}
UDP_Port_Scan	ExistService(DestIP, DestPort)	{ExistService(DestIP,DestPort)}

basis of it. This leads to another research problem: How do we decide systematically what to include and what not to include in the prerequisites of hyper-alert types to get the best performance for alert correlation? We will address this problem in our future research.

There are certain implication relationships between predicates, which are used by the alert correlator in reasoning about the hyper-alerts. Some of these implications are intuitive. For example, *OSSolaris (DestIP)* implies *OSUNIX (DestIP)*. However, some implications are not always true. For example, we have *GainOSInfo (DestIP)* implying *OSUNIX (DestIP)*, which means that if an attacker gains the OS information about *DestIP*, he knows that *DestIP* is a UNIX machine. This implication is true only when the OS of *DestIP*

Table 4: (Partial) Implication relationships between predicates.

Predicate	Implied Predicate
ExistService(DestIP,DestPort)	GainInformation(DestIP)
GainAccess(DestIP)	SystemCompromised(DestIP)
GainOSInfo(DestIP)	GainInformation(DestIP)
SystemCompromised(DestIP)	SystemAttacked(DestIP)
OSSolaris(DestIP)	OSUNIX(DestIP)
GainOSInfo(DestIP)	OSSolaris(DestIP)
GainSMTPInfo(SrcIP,DestIP)	SMTPSupportTurn(SrcIP,DestIP)

is UNIX. We call such implications *phantom implications*. We argue that we should consider phantom implications, since if we don't, we lose the opportunity to correlate alerts when the attacker uses some information not seen by the IDSs. Part of implications between predicates are shown in Table 4. The others are omitted due to space reasons.

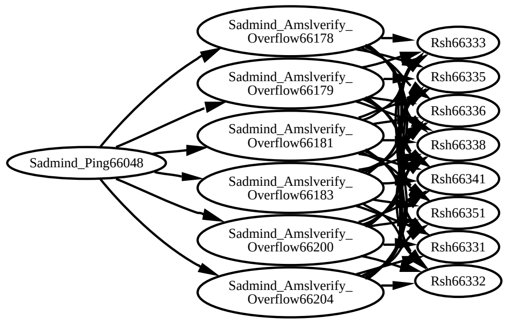
B Hyper-alert Correlation Graphs in Our Experiments

This appendix presents the experimental results with the two 2000 DARPA intrusion detection scenario specific datasets [12]. In both datasets, the attacker takes advantage of the vulnerability of *Sadmind* RPC service and launches buffer overflow attacks against the vulnerable hosts. The attacker installs the mstream DDOS software after he breaks into the hosts successfully. As the last step, the attacker launches DDOS attacks from the victim hosts. The difference between these two scenarios lie in two aspects: First, the attacker uses *IPSweep* and *Sadmind_Ping* in LLDOS 1.0, while *DNS_HInfo* in LLDOS2.0.2, to find out the vulnerable hosts. Second, the attacker attacks each host separately in LLDOS1.0, while in LLDOS2.0.2, he breaks into one host first and then fans out from there.

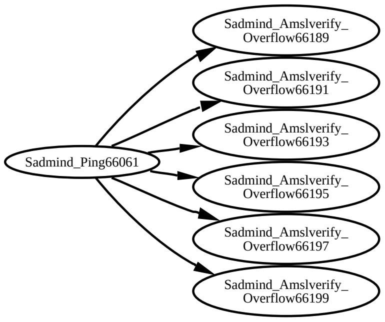
Each dataset includes the network traffic collected from both DMZ and inside part of the evaluation network. We have performed four sets of experiments, each with either the DMZ or the inside network traffic of one dataset. The result for the inside traffic in LLDOS 1.0 has been shown in Figure 3 and discussed in the paper. Figures 4, 5 and 6 show the hyper-alert correlation graphs for the other experiments.

Figure 4 includes six hyper-alert correlation graphs discovered in the DMZ traffic in LLDOS 1.0. Each graph is for a different targeted host. Similar to the inside traffic, two *Email_Almail_Overflow* alerts are mis-correlated. The *Mstream_Zombie* alerts (i.e., communication between the DDOS programs) are not observable in the DMZ traffic, and thus are not shown in Figure 4. Similarly, *Stream_DoS* is not included, either. These six hyper-alert correlation graphs reflect the attacker's strategy against the victim hosts: probe *Sadmind* service with *Sadmind_Ping*, attack the service with (up to six variations of) *Sadmind_Amslverify_Overflow*, and copy DDOS program to the victim host if one of the attacks is successful. As indicated by these graphs, three out of the six attempts are successful. Moreover, Figure 4(e) includes three falsely correlated hyper-alerts, which are shown in gray.

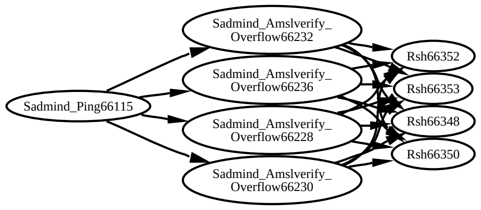
Figure 5 shows the hyper-alert correlation graphs for the inside traffic in LLDOS 2.0.2. It is easy to see that there are similarities between Figure 5(a) and Figure 3. In LLDOS 2.0.2, the attacker uses *DNS_HInfo* to discover the *Sadmind* service. However, RealSecure missed this attack. Thus, we do not have such an alert in Figure 5(a). In addition, the attacker uses *ftp* to copy the DDOS program. This is captured by the hyper-alert correlation graph, which includes two *FTP_Put*s as the step after the *Sadmind_Amslverify_Overflow* hyper-alerts. Note that the alert correlator didn't find the prepare-for relation between *Mstream_Zombie* with ID 64167 and the *Stream_DoS*, which is shown in dotted line in Figure 5(a). This is because RealSecure generated the same timestamp for both alerts. Figure 5(b) consists of two related alerts not included in the


Sadmind_Ping66048
Sadmind_Amslverify_ Overflow66178
Sadmind_Amslverify_ Overflow66179
Sadmind_Amslverify_ Overflow66181
Sadmind_Amslverify_ Overflow66183
Sadmind_Amslverify_ Overflow66200
Sadmind_Amslverify_ Overflow66204
Rsh66333
Rsh66335
Rsh66336
Rsh66338
Rsh66341
Rsh66351
Rsh66331
Rsh66332

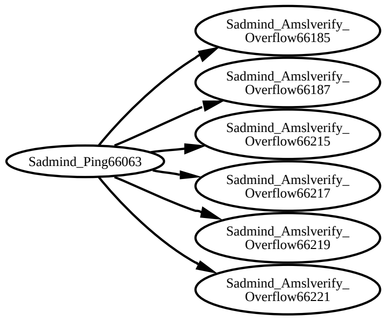
(a) Destination IP=172.016.115.020


Sadmind_Ping66061
Sadmind_Amslverify_ Overflow66189
Sadmind_Amslverify_ Overflow66191
Sadmind_Amslverify_ Overflow66193
Sadmind_Amslverify_ Overflow66195
Sadmind_Amslverify_ Overflow66197
Sadmind_Amslverify_ Overflow66199

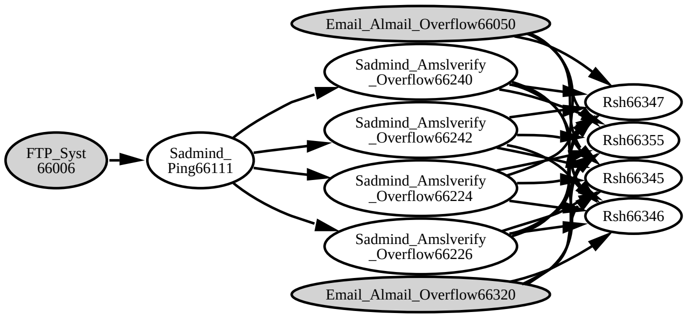
(b) Destination IP=172.016.114.010


Sadmind_Ping66115
Sadmind_Amslverify_ Overflow66232
Sadmind_Amslverify_ Overflow66236
Sadmind_Amslverify_ Overflow66228
Sadmind_Amslverify_ Overflow66230
Rsh66352
Rsh66353
Rsh66348
Rsh66350

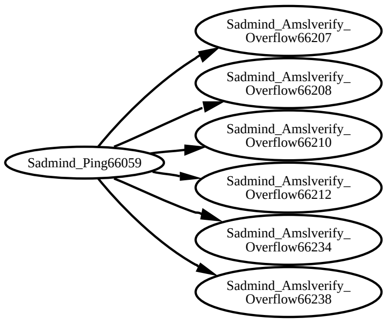
(c) Destination IP=172.016.112.010


Sadmind_Ping66063
Sadmind_Amslverify_ Overflow66185
Sadmind_Amslverify_ Overflow66187
Sadmind_Amslverify_ Overflow66215
Sadmind_Amslverify_ Overflow66217
Sadmind_Amslverify_ Overflow66219
Sadmind_Amslverify_ Overflow66221

(d) Destination IP=172.016.114.020

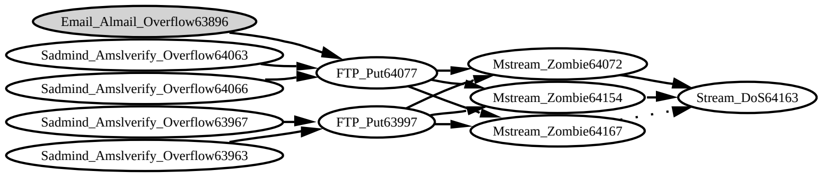

Email_Almail_Overflow66050
FTP_Syst 66006
Sadmind_ Ping66111
Sadmind_Amslverify _Overflow66240
Sadmind_Amslverify _Overflow66242
Sadmind_Amslverify _Overflow66224
Sadmind_Amslverify _Overflow66226
Email_Almail_Overflow66320
Rsh66347
Rsh66355
Rsh66345
Rsh66346

(e) Destination IP=172.016.112.050

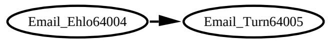

Sadmind_Ping66059
Sadmind_Amslverify_ Overflow66207
Sadmind_Amslverify_ Overflow66208
Sadmind_Amslverify_ Overflow66210
Sadmind_Amslverify_ Overflow66212
Sadmind_Amslverify_ Overflow66234
Sadmind_Amslverify_ Overflow66238

(f) Destination IP=172.016.114.030

Figure 4: The hyper-alert correlation graphs for the DMZ traffic in LLDOS1.0.


Email_Almail_Overflow63896
Sadmind_Amslverify_Overflow64063
Sadmind_Amslverify_Overflow64066
Sadmind_Amslverify_Overflow63967
Sadmind_Amslverify_Overflow63963
FTP_Put64077
FTP_Put63997
Mstream_Zombie64072
Mstream_Zombie64154
Mstream_Zombie64167
Stream_DoS64163

(a) The graph related to the attacks.

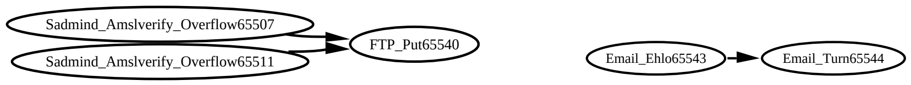


(b) The graph related to normal activities.

Figure 5: The hyper-alert correlation graphs for inside traffic in LLDOS2.0.2.

attacks. Our analysis indicates that they are normal but related email activities (an *Email_Ehlo* followed by an *Email_Turn*) from 135.013.216.191 to 172.016.113.105.

Figure 6 shows the hyper-alert correlation graphs for DMZ traffic in LLDOS2.0.2. Since the attacker


Sadmind_Amslverify_Overflow65507
Sadmind_Amslverify_Overflow65511
FTP_Put65540
Email_Ehlo65543
Email_Turn65544

(a) The graph related to the attacks. (b) The graph related to normal activities.

Figure 6: The hyper-alert correlation graph for the DMZ traffic in LLDOS2.0.2.

attacks other victim hosts from the first compromised host inside the private network, only the attacks against the first victim host are observable in the DMZ traffic. Figure 6(a) indicates that the attacker first launches two instances of *Sadmind_Amslverify_Overflow* to attack the victim host, and then uses *ftp* to copy something (i.e., the DDOS program) to the victim host. Figure 6(b) is the same as Figure 5(b).