Real-Time Protocol Analysis for Detecting Link-State Routing Protocol Attacks*

Ho-Yen Chang, S.F. Wu

{*hchang2, wu*}*@eos.ncsu.edu*
North Carolina State University
Raleigh, NC 27695

Y. Frank Jou

{*jou*}*@mcnc.org*
Advanced Networking Research, MCNC
RTP, NC 27709

March 16, 1999

Abstract

A real-time knowledge-based network intrusion detection model for link-state routing protocol is presented to detect different attacks for *OSPF* protocol. This model includes three layers: a *data process layer* to parse packets and dispatch data, an *event abstractor* to abstract predefined real-time events for link-state routing protocol, and an *extended timed finite state machine* to express the real-time behavior of the protocol engine and to detect the intrusions by pattern matching. The timed FSM called *JiNao Finite State Machine (JFSM)* was extended from the conventional FSM with timed states, multiple timers, and time constraints on state transitions. The construct of JFSM was implemented as a generator which can create any FSM by constructing only the configuration file. The results showed that this approach is very effective with expressive power for real-time intrusion detections. The model of our approach can be used for other network protocol intrusion detections, especially for those known attacks.

Keyword: real-time misuse intrusion detection, knowledge-based IDS, real-time network protocol analysis, link-state routing protocol security, OSPF attacks, event correlation, timed automata, timed Finite State Machine

1 Introduction

In today's Internet, since routing (e.g., RIPv2, EIGRP, BGP, and OSPFv2) and network management (e.g., SNMPv3/ng) protocols form the very heart of the network infrastructure upon which secure Internet deeply relies, security of routing protocol is of the utmost importance. Until recently, the security of these protocols has not been fully emphasized. However, there is a growing awareness of the potential consequences of attacks aimed at the infrastructure, particularly the routing protocols [11, 4, 42, 5].

Two classes of routing protocols are used on the current Internet: *distance-vector* and *link-state*. Link state routing has been considered as more secure than distance-vector protocols in many ways [18, 15], so in this paper, we are only concerned about link-state routing protocols. In link state routing, every node keeps a "map" of the entire network which is used to compute shortest paths to all destinations. Each node contributes to this global view by periodically distributing (via flooding) link state updates(LSUs); and LSU reflects the current status of all incident links of a given node. That is, a router will periodically announce to the whole world its relationship with all its neighbors.

Many insider attacks on link state routing protocols have been mentioned or discovered [33]. Here,"insider" means a trusted entity participating in the routing information exchange process or an outsider with the capability to intercept and modify the information exchange channels. For example, in [4], a new OSPF insider attack has been discovered and implemented, which allows the attacker to control the network topology for one hour by injecting a small number of bad OSPF PDUs. This attack is due to an implementation bug on many commercial routers. One particular major router vender has recently responded to our discovery by giving us a new version of their router software which makes the attack less effective.

Some of these insider attacks can be *prevented* by digitally signing the exchanged routing information. For instance, in [33], the *link state advertisements (LSA)* are signed by the LSA originator to prevent a

*This research project is supported by DARPA/ITO.

vicious intermediate router from tampering with the link state information. The objective of the preventive approach is to guarantee the *integrity* and *authenticity* of the link state information. However, the preventive approach has been rejected by the IETF OSPF working group for both technical and political reasons. These reasons include: complexity, high overheads, backward compatibility, and political concerns among vendors.

The JiNao project [21] at MCNC/NCSU is focusing on defending attacks. It takes an intrusion detection approach to defend routers against insider attacks, which may be relatively more acceptable to the industry as it requires no changes to the routing protocols themselves.

1.1 Intrusion Detection System

Because building and maintaining a perfectly secure system could be both technically hard and economically costly, network intrusion detection devices are used to detect and call attention to odd and suspicious behavior. An *intrusion detection system (IDS)* detects attacks or anomalies against computer systems and networks, through the task of monitoring the usage or traffics of the systems.

In 1987, Denning [8] proposed an intrusion-detection model based on the hypothesis that security violations can be detected by monitoring a system's audit records for abnormal patterns of system usage. Since his work, many IDS prototypes have been created. Several surveys have already been published [24, 25, 32, 12, 6] , and a partial bibliography of current IDS can also be found in [38].

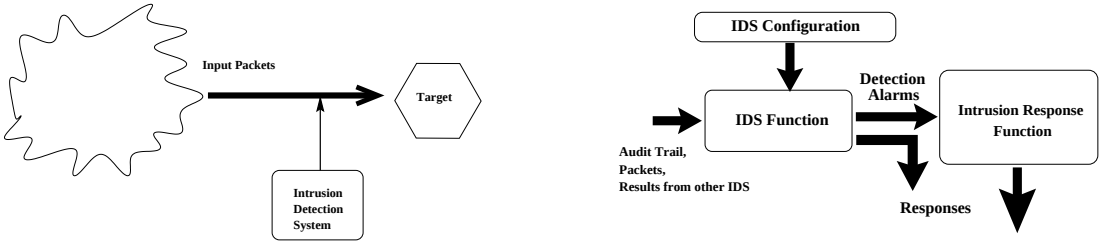

Figure 1: A General Intrusion Detection System

A general intrusion detection system (IDS) is depicted in Figure 1. Depending on the technique used, the IDS function investigates the monitored objects, either by off-line after-fact (from audit trail) detection or by online real-time (from network packets) detection. Once IDS detects intrusion, it can either show the warning message or directly inform the intrusion response system (IRS) to react to this intrusion. Basically, IDS can be adjusted by configuration to the monitored environment. Through the configuration, IDS can be tuned into better performance or sensitivity against different intrusion techniques. A good taxonomy of IDS has been presented in [7].

There are five measures usually used to evaluate the efficiency of an intrusion detection system : *accuracy*, *completeness*, *performance*, *fault tolerance*, *timeliness* [7]. *Accuracy* indicates the rate of correctness in detection results. Inaccuracy occurs when an IDS flags a legitimate action in the environment as anomalous or intrusive, which is also called a false positive. *Completeness* indicates the sensitivity of an IDS. Not all the attacks may be detectable. Incompleteness occurs when IDS fails to detect an attack, which is also called a false negative. Good IDSs will have as lower false positive and false negative as possible. *Performance* indicates the rate at which audit events are processed. Poor performance lacks the ability of real-time detection. Moreover, to remain effective anytime, an IDS should be *fault tolerant* itself to resist attacks, particularly denial of service attacks. *Timeliness* implies that the IDS's response or the reaction to an attack should be the sooner, the better. If an IDS cannot process and propagate the information of the attacks as quickly as possible, and enable the security officer to react before damage has been done, the attackers could have a chance to subvert the audit source or even the IDS itself.

Current approaches of intrusion detections can be broadly classified into two trends. *Anomaly detection systems*, or so called *Behavior-based IDS*s, and *Misuse detection systems*, or so called *Knowledge-based IDS*s. *Behavior-based intrusion detection systems* monitor and build a reference profile of the normal behavior of the information system by using statistical methods and try to detect activity that deviates from this normal behavior profile [8, 20, 3, 9]. Anything that does not correspond to previously learned behaviors is considered intrusive. The main advantage of this approach is that it can detect attempts to exploit new and unforeseen vulnerabilities without *a priori knowledge* of the security flaw of the target system. That is, it could automatically discover new potential attacks.

However, a Behavior-based IDS might be complete, but its accuracy is a question. The high false positive alarm is generally cited as the main drawback of the behavior-based IDS, because the entire scope of the behavior of an information system may not be covered during the learning phase. Also, behavior may change over time. In order to maintain the accuracy of IDS, the behavior profile needs periodically online "retraining", resulting either in unavailability of the IDS or additional false alarms. For example, the information system can undergo attacks at the same time the IDS is learning the behavior, and as a result, the behavior profile may contain intrusive behavior, which is not detected as anomalous but considered as normal behavior, resulting to a false negative.

*Knowledge-based IDS*s, accumulate the knowledge about the attacks, examine traffic, and try to identify a pattern that they can compare to the signatures or scenarios known to be dangerous or suspicious. This approach can be applied only against known attack patterns and needs to update the knowledge base frequently, for example, a virus checker, which needs to keep periodically updating the new signature as new attacks are detected and characterized. Any action that is not explicitly recognized as an attack is considered acceptable. Thus, A Knowledge-based IDS might be accurate, but its complete detectability is a difficult issue. However, a majority of break-ins or intrusions is the result of a small number of known attacks, as evidenced by reports from CERT. knowledge-based IDSs with *signatures* techniques are attractive to today's implementations, especially in commercial products because of their very low false alarm rates and very high accuracy. Also, the clear expressiveness of this technique makes maintaining the knowledge base easier for the security officer. Several techniques have been proposed in Knowledge-based IDS, including *Expert systems*, *Model-Based reasoning intrusion detection* [13], *State Transition Analysis* [19], *keystroke monitoring*, *Color Petri Nets* [23].

Furthermore, in a distributed environment, users hop from one machine to another, possibly changing identities during their moves and launching their attacks on several systems, making the task of IDS even harder. Thus, to manage diverse attacks across networks and time, local intrusion detection systems may need to cooperate with other network IDS by exchanging information with its peers. However, "cooperation" of homogeneous components requires some way to co-manage intrusion detection and response systems, such as standardizing formats, protocols and architectures, so *Common Intrusion Detection Framework* (CIDF) was formed recently to deal with these issue [39, 22, 41]. *Artificial Intelligence* techniques could be used in IDS [12], such as learning, induction, and information reduction to improve the performance of the intrusion detection.

None of the intrusion-detection approaches described above is sufficient alone to catch all intrusions. A successful intrusion-detection should incorporate several different approaches. In particular, in order to increase the difficulty of penetration, the JiNao project [21] at MCNC/NCSU which employs a statistical profile approach and a knowledge-based approach promises to be an effective combination.

1.2 JiNao Intrusion Detection System

The JiNao distributed intrusion detection system is comprised of security management agents (*i.e.,* JiNaos[1]) that collect and analyze traffic measurements and additional agents that perform higher-order analysis functions. The communication interface among these two types of JiNao's is SNMP (Simple Network Management Protocol) for interoperability. Each agent offers a set of abstract MIB (Management Information Base) variables for global and higher-level network management applications. By coupling local and global observations, the collection of agents and management applications can detect and react to correlated attacks on the network infrastructure.

The architecture of JiNao is shown in Figure 2, which defines an integrated framework. Three modules, an administrative-rule based approach, a statistical-based approach, and a protocol-based approach play complementary roles to achieve intrusion detection. The *Prevention Module*, with the administrative rules (like firewall) checking , will detect events that are in clear violation of certain network security guidelines and are deemed too dangerous to let pass through to protocol engines. The *Statistical Module* will be able to detect any intrusions that have significant impact on certain statistical signatures of the network infrastructure, even those that are not anticipated by the protocol-based mechanisms. The statistical-based mechanisms may also provide information on the locations of the potential "hot spots" for intrusion where installation or enhancement of the logical analysis mechanisms should be considered. The *Protocol Analysis Module* will be able to detect the targeted intrusions with relatively low latency. It also provides additional information for source identification of the intrusions, and some security guidelines for the Prevention Module to adjust its administrative rules. In addition, the protocol-based mechanisms will

[1] Jinao means a "chicken brain" in Chinese.

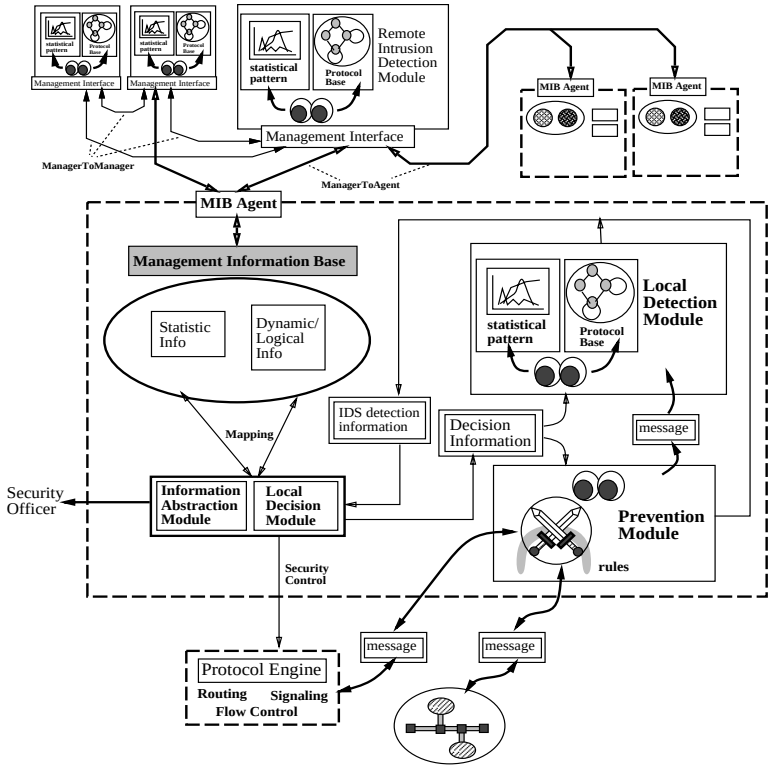

Figure 2: The JiNao IDS Architecture

provide "hints" on the expected changes in network to help the Statistical Module to adjust to new conditions. This paper focuses only on the presentation of the *Protocol Analysis Module* in detail. For the details of other system components, please refer to [21].

The network faults or malicious attacks typically result in a number of timed events. In this protocol-based analysis module, we are interested in correlating these ordered events to determine whether the protocol's behavior is normal or abnormal. This problem, referred to as "event-correlation," is a fundamental issue in many network security and fault tolerant systems. *JiNao Protocol Analysis Module (JPAM)* applied and extended the finite state machine model [17] to capture both good and bad behavior of a link-state routing protocol engine. The extended FSM model, as will be discussed later, is called *JFSM (JiNao Finite State Machine)*, which has been defined and implemented to make the FSM model more expressive in dealing with the timing issue. Independently, we implemented three OSPF insider attacks (*maxseq, maxage,* and *seq++*), which were all tested on both NCSU's and MCNC's routing testbeds and demoed in Air Force Base Rome Laboratory in Nov, 1998.

As will be presented later, the results showed that this proposed methodology is effective in detecting these real-time routing protocol attacks. Additionally, the system described here may serve as the basis for an extended study of real-time intrusion detection modeling for other network protocols.

2 Link-state Routing Protocols and Autonomous OSPF Routing

Routing protocols are used by routers to create a *map* of the network, so that the routers can tell how to get from one point to another. Link-state routing protocols create the *map* in three distinct phases. First, each router meets its neighbors and learns about its local neighborhood. Second, routers share that information with all other routers on the network by exchanging information. During this phase, a router learn about neighborhoods other than its local one. Finally, each router combines the information about each individual neighbor to get a "*map*" of the entire network (as depicted in Figure 4), from which it calculates routes.

OSPF [30] (Open Shortest Path First), based on link-state and SPF (Shortest Path First) technology, is a routing protocol which is classified as an *Interior Gateway Protocol (IGP)*. This means that it distributes

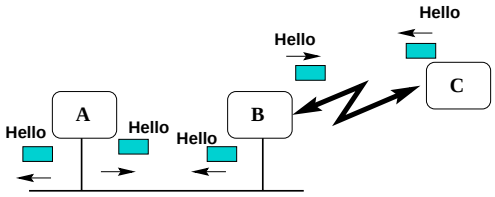

Figure 3: Hello protocol

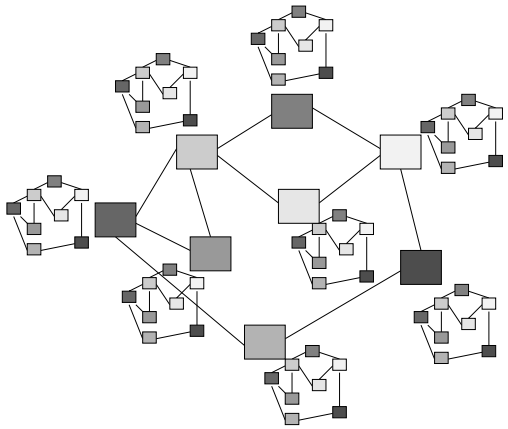

Figure 4: Link State

routing information between routers belonging to a single *Autonomous System* (AS) in which a collection of many computer systems, routers, and other network devices share a single administrative entity.

Here are three stages of OSPF,

1. **Meeting the Neighbors**
   The first step is the creation of *adjacencies*. A *Hello Protocol* is defined to establish and maintain the neighbor relationship. That is, every OSPF router periodically sending *Hello packets* to discover its neighboring routers (as shown in Figure 3). Three components in the Hello packet header keep information about the status of routers. The *hello interval* indicates how frequently the sender should retransmit its hello packets; the *router dead interval* tells how long it takes to declare a router unavailable, and a list describes the neighbors that the sender has already met. Once neighboring routers have "met" via Hello Protocol, they go through a "Database Exchange Process" to synchronize their databases.

2. **Share the information by flooding LSA**
   In the second step, the information about a router's local neighborhood is assembled into a *Link-State Advertisement* (LSA) and is broadcast (via a reliable intelligent *flooding* scheme) to all other routers. The combined information makes up the link state (LS) database for the network. In a broadcasting network, like Ethernet, a *designated router* may be elected to advertise for the whole network.

3. **Calculate Shortest Routes based on LS Database**
   Once all systems have an up-to-date link state (LS) database, each router can use the Dijkstra algorithm to calculate a shortest path tree with the router itself as the root node and then form a complete picture of routing in the network.

In a dynamic unstable network, routers may restart, network cost metric may change, and even links may fail. So all three phases happen in parallel, and they all take place continuously to maintain a well-functional routing environment.

To reduce routing traffic and the size of the topology database the protocol requires, OSPF proposes a two-level hierarchical routing scheme within an *Autonomous System* (AS). The hierarchies include *backbone* and *areas*. All systems within an area must be connected. An area is a collection of networks, hosts, and routers. Backbone serves as the hub of the AS, and all other areas in the AS must connect to the backbone. Each area runs a separate copy of the basic link-state routing algorithm. This means that each area has its own link-state database. The topology details of an area are hidden from the outside of the area. Conversely, routers internal to a given area know nothing about the detailed topology external to the area. This isolation helps to reduce routing traffic as compared to treating the entire AS as a single link-state domain. One thing to be mentioned here is that an attack which occurs inside an area may not disturb the outside world. Conversely, an internal router in an area may not be influenced by an attack which occurs outside of the area.

There are five types of OSPF packets. *Hello* packet is used by two adjacent routers to maintain a neighborhood relationship. *Database Description* and *Link State Request* packets are used to synchronize two routers' databases when an adjacency is being initialized. *Link State Update* and *Link State*

Acknowledge packets are used to broadcast LSAs (Link State Advertisement). All OSPF routing protocol exchanges may be authenticated.

2.1 Link-State Advertisement (LSA)

Once a router meets its neighbors by exchanging Hello packets, it distributes that information to the rest of the network. To do this, it floods LSA throughout the network. All LSAs share a generic header, which is shown in Figure 5.

2.1.1 LSA format

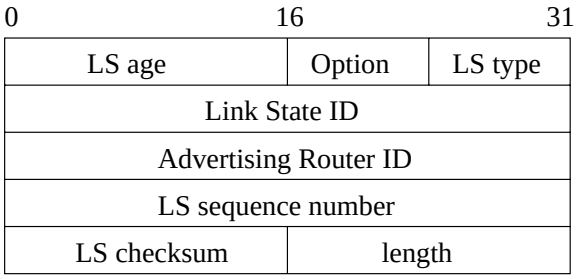

Figure 5: LSA header format

The first 16 bits store an *LSA's age*. The age starts at zero and is incremented on every hop of the flooding procedure. LSAs are also aged as they are held in each router's database. When an LSA's age reaches 3,600 (one hour), it is considered out of date, and should be purged from a router's database. If one router decides to flush an out-of-date LSA from its database, it also re-floods it as a signal for other routers to remove it so that every router will have a consistent view about the network topology. *Option* field indicates which optional capabilities are associated with an LSA.

LS type field dictates the format and function of an LSA. LSAs of different types have different names (e.g., router-LSAs or network-LSAs). *Link State ID* field identifies the piece of the routing domain that is being described by the LSA. *Advertising Router ID* field specifies the OSPF Router ID of an LSA's originator. This 32-bit number uniquely identifies each router within an AS. It can be assigned to one of the router's IP address.

LS sequence number field is a signed 32-bit integer. It is used to detect old and duplicate LSAs. It starts at `0x80000001` (`0x80000000` is reserved by OSPF) and is incremented each time a router originates a new instance of the LSA. When an attempt is made to increment the sequence number past the maximum value, i.e., `0x7fffffff`, the max sequence LSA must be flushed from the routing domain. This is done by prematurely aging the max sequence LSA to 3600, which is max age, and re-flooding it. As soon as this flood has been acknowledged by all adjacent neighbors, a new instance can be originated with sequence number `0x80000001`. As shown in Section 3, some routers fail to comply with the protocol due to implementation bugs. An intruder can block routing updates for up to one hour by simply injecting one bad LSA. *LS checksum* field contains the checksum of the complete contents of the LSA, except the *LS age* field. The *LS age* field is excluded so that an LSA's age can be incremented without updating the checksum. Fletcher checksum, which is also used by ISO connectionless datagrams, is used for this field.

Here a distinction must be made between an *LSA* and an *LSA instance*. An *LSA* is associated with a particular link or network. For example, there is a link connecting router A and B. Router A is responsible for originating *an LSA* to tell other routers that it has a link to router B, while router B will use another *LSA* to tell others that it has a link to router A. An *LSA instance* gives the state of a particular LSA at a particular time. For example, router A at time t_1 may broadcast an *LSA instance* saying the cost for its link to router B is 10. After a while, say at time t_2, the status of the link changes. Router A should broadcast a new *LSA instance* telling the new cost for this link. In a network, there may exist more than one instance for a particular LSA. *LS type*, *Link State ID* and *Advertising Router ID* uniquely identify an *LSA*, while *LS sequence*, *LS age* and *LS checksum* fields must also be considered when talking about an *LSA instance*. Unless it will cause confusion, the single word *LSA* is used for both *LSA* and *LSA instance*, since most of the time, the meaning can be easily derived from context.

2.1.2 Exchanging LSA

A router should originate an LSA whenever the status of its direct links changes (like link up/down, new neighbor, etc.). Also after a predefined timeout, a router should re-advertise the links to keep their "freshness".

An interesting issue to mention here is the "fight back" phenomenon in link state routing information exchange. Assume an intruder injects a bad LSA into a routing domain to impersonate a good router (assume the authentication option has been turned off). The other routers will compare the bad LSA with the good one in their database. If the bad LSA is older, it will be discarded in the first place, causing no harm at all. Otherwise, the good LSA will be purged out, while the bad LSA is put into the database. To make sure all other routers share the same view of the topology, the router, which has accepted the bad LSA, will forward it to all its neighbors (except the one from which it received the bad LSA). So the bad LSA will spread all over the routing domain. Finally this LSA's legal originator (victim) will receive the bad LSA! Now the originator of this LSA finds something wrong; it will increase the sequence number, reset the age, and broadcast a new LSA back into the network. This is called "fight back". "Fight back" will continue until the attacker quits.

OSPF Hello

Constant	Default Value	Action of OSPF Router
HelloInterval	10 secs	*For how frequently sender re-transmit Hello pkts.*
RouterDeadInterval	40 secs	*For how long it takes to declare a router unavailable.*

LS Age

Constant	Default Value	Action of OSPF Router
MinLSArrival	1 second	*Maximum rate at which a router will accept updates of any given LSA via flooding.*
MinLSInterval	5 seconds	*Maximum rate at which a router can update an LSA. (not allow to update LSA less than 5 secs)*
CheckAge	5 minutes	*Rate at which a router verifies the checksum of an LSA contained in its database.*
MaxAgeDiff	15 minutes	*When two LSA instances differ by more than 15 minutes, they are considered to be separate instances, and the one with the smaller LS Age field is accepted as more recent.*
LSRefreshTime	30 minutes	*A router must refresh any self-originated LSA whose age reaches the value of 30 minutes*
MaxAge	1 hour	*When the age of an LSA reaches 1 hour, the LSA is removed from the database.*

LS SequenceNumber

Constant	Define Value
InitialSequenceNumber	0x80000001
MaxSequenceNumber	0x7fffffff

Table 1: Some rules and constants in OSPF for LSA

According to OSPF standard [30, 31], Table 1 are some rules and constants in OSPF protocol.

2.2 Good Fault-Tolerant Properties of OSPF Protocol

In general, protocols can be made fault-tolerant by three "standard" techniques [14]: *the addition of sequence numbers to messages, discarding received erroneous messages, and timeouts. Sequence numbers* can be used to detect and correct all occurrences of message reorder. *Checksum* can be used to transform each occurrence of message corruption to an occurrence of message loss. *Timeout* can be used to detect and correct all occurrences of message loss (provided that there is an upper bound on the time-to-live for every sent message). The direct approach to the timeout scheme is to provide some kind of "global clock" for synchronization. However, this approach complicates the protocols and is usually very expensive. OSPF uses *hello* protocol to maintain "aliveness" information and *"Age"* to provide "freshness" information.

By combining "age" and "sequence number" protocol, OSPF can achieve a good property so called "*Self-stabilization*", in which a system can automatically recover from arbitrary transient faults in a

bounded period of time without any external intervention [10]. This property forces attacks to be persistent to be possibly successful. Recently self-stabilization has become an important issue for the fault-tolerant distributed system, surveys and papers can be found in [37, 16].

Through flooding and LS database synchronization, OSPF can have *topology-independence* detectability. That is, the router can observe all the LSAs' behavior in the same area, even from other originating routers. This property helps IDS sit on only one location and detect a whole area's intrusion status at a certain rate of accuracy.

Although OSPF is considered as a better secure routing protocol, several vulnerabilities have been found because of software engineering flaws [4] which will be discussed in the next Section.

3 OSPF Attacks: Mechanism and Implementation

3.1 Attack Implementation on FreeBSD

In order to validate the proposed approach, three OSPF insider attacks were implemented for the FreeBSD platform [4]. FreeBSD provides a mechanism called `divert socket` which can be used to intercept OSPF packets from the kernel and divert them to a user process. The attacks themselves are implemented in that user process, and the tampered OSPF packets are then re-injected into the kernel, which will deliver these packets to either the routing daemon running on the same machine (incoming direction) or its neighbors (outgoing direction). The architecture of the attack implementation is depicted in Figure 6.

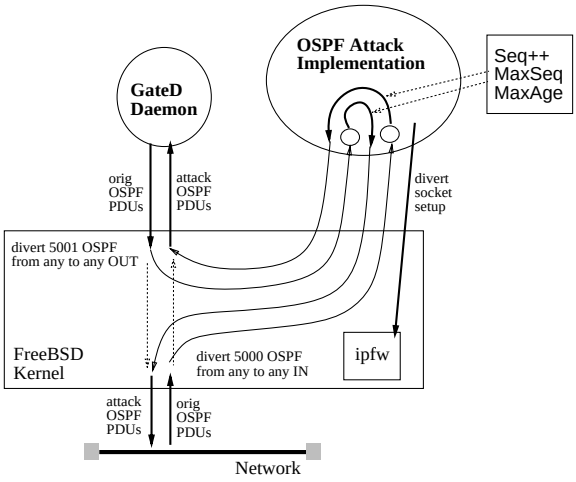

Figure 6: OSPF Attack Implementation

3.2 Attack 1: Seq++ Attack – Simple Modification

When the attacker receives a LSA instance, it can modify the link state metric and increase the LSA sequence number by 1 (i.e., Seq++). The attacker also needs to re-compute both the LSA and OSPF checksums before the tampered LSA is re-injected into the system. This attacking LSA, because it has a larger sequence number, will be considered "fresher" by other routers. And, eventually it will be propagated to the originator of this particular LSA. The originator, according to the OSPFv2 specification, will "fight back" with a new LSA carrying correct link status information and an even fresher sequence number.

The effect of this attack is an unstable network topology if the attacker keeps generating "Seq++" LSAs. For example, all routers at one point will think the link cost is infinity (the link is down), but then the fight-back LSA from the originator will tell them the cost metric is much smaller (e.g., 2).

3.3 Attack 2: Max Age

When the attacker receives a LSA instance, it can modify the LSA age to `MaxAge` (i.e., 1 hour), and re-inject it into the system. This attacking LSA, with the same sequence number but `MaxAge`, will cause all

routers to purge the corresponding LSA from their topology database. Eventually, the originator of this purged LSA will also receive the MaxAge LSA. The originator, according to the OSPFv2 specification, will "fight-back" with a new LSA carrying correct link status information and a fresher sequence number.

The effect of this attack is also an unstable network topology if the attacker keeps generating "MaxAge" LSAs. For example, all routers at one point will think the link is not available, but then the fight-back LSA from the originator will tell them the link is actually there.

3.4 Attack 3: Max Sequence Number

When the attacker receives a LSA instance, it can modify the link state metric and set the LSA sequence number to `0x7FFFFFFF` (i.e., MaxSequenceNumber). The attacker also needs to re-compute both the LSA and OSPF checksums before the tampered LSA is re-injected into the system. This attacking LSA, because it has the maximum LSA sequence number, will be considered the "freshest" by other routers. And, eventually it will be propagated to the originator of this particular LSA. The originator, according to the OSPFv2 specification, "should" first purge the LSA (setting MaxAge) and then flood a new LSA carrying correct link status information and the smallest sequence number: `0x80000001`.

It is discovered that the effect of this attack depends on the implementation of the OSPF protocol. If the protocol is indeed implemented correctly, then it is similar to `Seq++`. On the other hand, many routers did not implement the `MaxSeq` LSA handling correctly: the purging of the MaxSeq LSA is not implemented. This implies that the MaxSeq LSA will stay in every router's topology database for one hour before it reaches its MaxAge. In other words, the attacker can control the network topology database for all the routers for one hour.

4 JiNao Real-time Protocol Analysis Module (JPAM)

Although the Behavior-based IDS, such as the statistical module in the JiNao project, has complete detectability, it has costs in terms of many false alarms and longer learning period. *JiNao Protocol Analysis Module (JPAM)*, which is a Knowledge-based IDS, was made to overcome these deficiencies and to complement the JiNao statistical module.

The intuition behind the JPAM is that malicious attacks or faults often result in a set of possible alarm event sequences or some special events which can be described as input strings in the finite state machine. Good behaviors are often "predictable" so are known bad behaviors (there are already several attacks we knew). According to the order of packet events, which can be identified as good or bad behaviors based on protocol specification, we in general observe the sequence of events to tell what kind of attacks happen or the status of the router's behavior.

Figure 7 shows the concept of our approach. The upper part represents how the model of normal

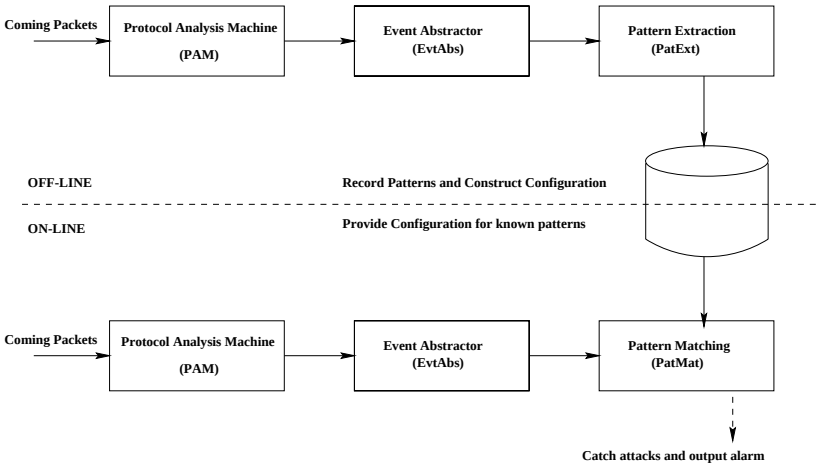

Figure 7: IDS by pattern matching of FSM

behavior and known intrusion patterns could be created off-line. The on-line detection process of the lower part relies on the assumption that when an attack exploits vulnerabilities in the system, either a new subsequence of events which deviate the normal event sequence will appear or some known intrusive

pattern will be matched. The protocol analysis machine (PAM) obtains the network packet from the interception module, doing the filtering to eliminate irrelevant packets that are not related to the protocol we monitor and then dispatch the packets to Event Abstractor(EvtAbs). EvtAbs basically will do the translation job to abstract the predefined event from a given packet. Based on the sequence of events it observed, Pattern Extraction(PatExt) will record the patterns into the IDS database, which can be a reference for constructing FSM by the system security officer for on-line intrusion detection. After constructing FSMs, each of them can do the pattern matching for different known attacks simultaneously and output the alarm information when something bad happens.

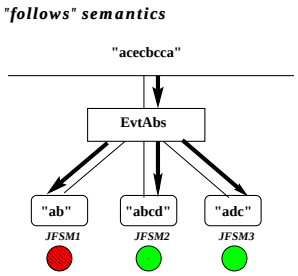

Figure 8: "follows" semantics of pattern matching

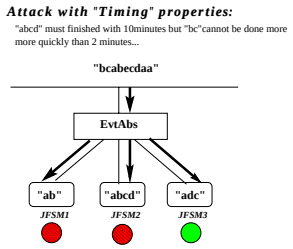

Figure 9: "timing" property in pattern matching

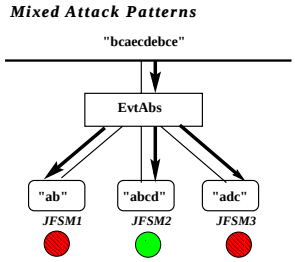

Figure 10: "mixed" attack pattern matching

Pattern matching in intrusion detection must be done with the *"follows"* semantics rather than the *"immediately follows"* semantics [23]. That is, any two adjacent sub patterns within a pattern could be implicitly separated by an arbitrary number (possibly zero) of events of any type (*i.e.*, ".∗" in Unix regular expression syntax). For example, pattern "*ab*" specifies that event "*a*" is followed by event "*b*" with some (or none) intervening events ("$a.*b$"). In other words, JFSM was designed to detect pattern "*ab*" not only from "*bcabaa*" but also from "*acecbcca*".

Many attacks come with timed properties; for instance, in order to attack successfully, a special sequence "*abcd*" must be finished within 10 minutes but the "*bc*" sub sequence cannot be done more quickly than 2 minutes (*i.e.*, "*bc*" cannot happen too soon) which may violate some protocol rule. Otherwise, this attack will fail. JFSM extended from a timed automata is able to deal with the real-time information related to these kinds of attacks. Furthermore, there could be multiple intrusion patterns mixed in a long event sequence string. For example, if "*ab*" and "*adc*" are two known attack patterns of event sequences, there could be a sequence "*bcaecdebce*" containing these mixed attack patterns. JPAM was designed to be able to detect all of them at the same time.

In terms of detectability, although its completeness depends on the regular update of knowledge about attacks, JPAM uses signature and the FSM approach which has clear expressiveness, making it easier for security officers dynamically tuning up the IDS to take preventive or corrective action. In terms of accuracy, JPAM has a potentially very low false alarm rate, and thus is an attractive approach to many applications.

4.1 Overview of JPAM Architecture

JPAM uses routing traffic and knowledge about the protocol engine to detect when an intruder is attempting an attack. When the JPAM detects such an attack, it sends an alarm message to the Local Decision Module describing the attack and containing the sequence of messages (events) used in determining that the attack took place.

The generic architecture of JPAM can be viewed as the following three basic abstractions showed in Figure 11:

The Data Process Layer incorporated with the interception module provides a low-level data interface to monitored computer or network system. It contains *packet parsers* which can translate each packet to correct protocol data structures. Based on the protocol-dependent input packet, this layer dispatches each output in a protocol-dependent format, such like packet header, with timestamp to the corresponding event machine for abstracting events.

The Event abstraction Layer provides the representation of events and generates the general domain of the inputs for JFSM. It contains an *universal event generator* which can generate the event

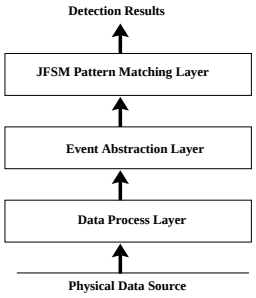

Figure 11: Three abstraction layers of JPAM

abstractor machines for different protocols based on different configuration files. All event definitions are put into these files such that dynamic adjustment is possible. Each event machine may generate a different sequence of events based on the system data or network packets provided from the lower data information layer.

The JFSM Pattern Matching Layer detects the matching patterns based on the sequence of events or signatures from its corresponding event machine. It contains a *JFSM generator* to generate different finite state machines based on different configuration files.

This kind of abstraction enables the ability for adapting alternative solutions of each layer without changing the interfaces between the layers of this model. The JPAM maintains a collection of *Event Abstractors* for each protocol JiNao is capable of monitoring. Each event abstractor can have several JFSMs to detect multiple intrusion patterns at the same time. Each JFSM is used to detect either one kind of intrusion or the good, expected behavior. Each JFSM could be constructed off-line and stored in a human-readable file. The main benefits of this model are its extensibility, portability, and flexibility by dynamic configuration. The generic JFSM generator can create any FSM based on the transition table in its configuration file. By dynamically changing the configuration files of event abstractors and JFSMs, this model could potentially be applied to other systems or even different mixed protocols.

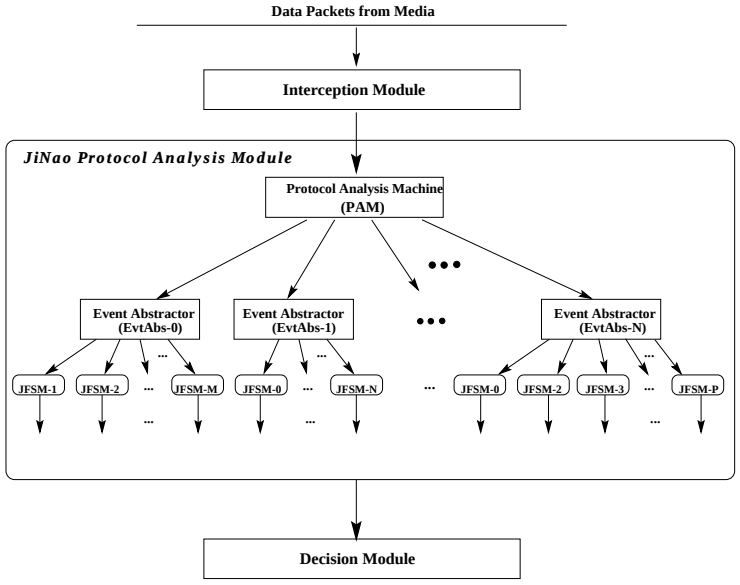

Figure 12: Architecture of JiNao Protocol Analysis Model

The architecture for the current implementation of the JiNao protocol analysis module for OSPF/LSA is described in Figure 12. The upper part, "*Protocol Analysis Machine (PAM)*" can be configured by a configuration file which contains all the information about the construction of "*Event Abstractors(EvtAbs)*" and "*JFSMs*". The file indicates which LSAs are interested by this module and which JFSMs are used for analyzing each LSA's behavior.

Each EvtAbs is responsible for one particular type of LSAs and has several JFSMs to parallelly analyze this LSA's events. Every JFSM is associated with a configuration file, which contains all the states and transitions to construct this JFSM. Each JFSM could observe one specific behavior or trying to match one known pattern. All FSMs of the same EvtAbs must work simultaneously to catch all the suspicious patterns. To analyze an LSA, one EvtAbs does not necessarily have identical JFSMs with others. EvtAbs' are constructed independently and could have different number of JFSMs to analyze their target LSAs. The predefined event types of LSA will be described in Section 4.2 in more detail.

Conceptually, when an OSPF packet comes from the interception module, PAM records the time-stamp of the local machine time, and dispatches each LSA to appropriate EvtAbs. When an EvtAbs gets the LSA, it first analyzes three fields, LS-checksum, LS-age, and LS-seq to determine if predefined events "InvalidLSA", "MaxAgeRefreshLSA" or "MaxSeqLSA" happens. Then it will check "BigJumpSeqLSA" event. If no event mentioned above happens, EvtAbs will regard it as a normal LSA update and input the "UpdateLSA" event to JFSMs. Otherwise, it sends the corresponding event to its JFSMs and then continues to process the next low-level LSA from PAM.

As shown in Figure 12, there is a basic FSM for good or normal behavior, and other FSMs for each known attack pattern (i.e., some event sequence) as well. JPAM is trying to catch attack by the historical patterns which happened before. These FSMs will work together to determine the actual status of the routers' behavior and to detect the intrusion occurrence accurately. Starting from an initial state, each JFSM will record a "currentState" variable and decide to which state it will advance based on the event string it received and the transition table in its configuration file. If a known attack happens, at least one of the JFSMs should raise alarm, and report to the higher decision module or security management module to decide the action for this attack. If JFSM cannot handle the input event from the current state, it will send out a message indicating the execution status of this JFSM. If all the JFSMs fail, JPAM will conclude an unknown attack pattern occurred and report the entire input event sequence of this attack.

4.2 JiNao Event Abstractor for OSPF LSA analysis

In order to catch events from OSPF packets, some abstract events were predefined for the value change in several fields of LSA header based on the protocol specification, as shown in Table 2. Each event is timestamped such that we can identify the order of the events and the duration between different events.

Furthermore, two categories of events were in our implementation: *Incoming events (i_events)* from other neighbor routers and *Outgoing events (o_events)* from the router itself. JFSM model is a variant of the *Timed I/O Automata* [28]. Each current and the previous ones of incoming and outgoing LSAs are saved as variables: $i\_prevLSA$, $i\_currentLSA$, $o\_prevLSA$, and $o\_currentLSA$, such that comparison and analysis between them can be made. Incoming indicates that the LSA is received from neighborhood, which is a external event to the system. Outgoing indicates that LSA is sent out by local router, which could be an action response to external world. This kind of hierarchical event abstraction and category could make the intrusion sensitivity adjustable when setting up the JFSM.

We assume that the event sequences of good behavior and bad behavior are distinguishable. In other words, if $s_0 s_1 s_2 ... s_n$ is defined as a bad sequence, there is no other correct operation with the same sequence. JiNao finite state machines were then built up based on good behavior according to OSPF standard. Each known attack pattern was also modeled in a finite state machine. Normally, routers should behave correctly by OSPF protocol specification and issue the updated LSA each "LSRefresh-Time"(default 30 minutes). The correct behaviors should comply with the rules listed in the Table 1. The event sequences resulting from the bad behaviors are abnormal and should not happen based on correct protocol implementation. The most possible reasons for these bad behaviors are attacks happen, software implementation bugs, or router system faults happen.

The event abstractor should produce all the events which form the input domain of JFSM, and JFSM does the pattern matching by processing its state transitions. This paper provides only the related concepts in JFSM components. More details on automata theory can be found in [17, 28, 27], and timed automata for real-time systems can be found in [36, 35, 1, 2, 26, 29]. A general framework and algorithms to solve the alarm correlation problem based on a probabilistic finite state machine has also been proposed in [34].

5 JiNao Finite State Machine

One of the chief goals of the JiNao project is to provide extensibility: it should be easy to adjust to the kinds of intrusions that are tracked. This requirement has the following implications for the JPAM.

(InvalidLSA) i_InvalidLSA o_InvalidLSA	indicates that LS checksum of received LSA (incoming or outgoing) is wrong and contents in LSA may have been corrupted during flooding. The corrupted LSA will be discarded by the router, in hope that the retransmitted LSA from neighbor will be uncorrupted.
(MaxAgeWithMaxSeqLSA) i_MaxAgeMaxSeq o_MaxAgeMaxSeq	indicates that router has received LSA (incoming or outgoing) with MaxAge and MaxSequenceNumber. It is a purging LSA for "fightback" purpose if from originator
(MaxAgeRefreshLSA) i_MaxAgeSameOutSeq o_MaxAgeSamePrevOutSeq	indicates that router has received the LSA(incoming or outgoing) with MaxAge and the same sequence number of the current outgoing LSA. It is going to refresh the current LSA it originated if it is outgoing.
i_MaxAgeRefresh o_MaxAgeRefresh	Incoming or outgoing LSA with MaxAge but without meeting any criteria of the above "MaxAge related events."
(MaxSeqLSA) i_MaxSeq o_MaxSeq	indicates that router has received LSA with MaxSequenceNumber but without MaxAge. It could be naturally reaching the maximum sequence number, suspicious attack or error by software flaws.
(InitSeq) i_InitSeq o_InitSeq	indicates that router has received LSA (incoming or outgoing) with minimal sequence number (0x80000001).
(BigJumpSeqLSA)	indicates that router has received the LSA with big different LS Sequence Number from previous one. Say if the difference are greater than MaxJumpSeq, an user-defined threshold value. MaxJumpSeq is defined as 0x00000010 in our model.
i_BigJumpSeqIncr o_BigJumpSeqIncr	Incoming or outgoing LSA with a very large increment on sequence number comparing to previous outgoing (originated) LSA.
(SeqIncrLSA) i_SeqIncr o_SeqIncr	indicates that router has received the LSA (incoming or outgoing) with larger sequence number than previous outgoing (originated) LSA.
o_UnNormSeqDecr	Outgoing LSA with a abnormal decreasing sequence number comparing to previous outgoing (originated) LSA.
(UpdateLSA) i_Update o_Update	indicates that router has received regular Link-State Update packet with normal LSA. (i.e. LSAs without MaxAge & MaxSeq & incorrect checksum & BigJumpSeq . . .)
HelloClockTick	an internal event always happen when receive any hello packet which has a newer timestamp than current time. It is a mechanism for FSM to update its current Timer.
i_HelloClockTick	an event happen when receive any incoming Hello packet which is at least 1 sec later then previous Hello packet.
Reset	an *internal event* always forces to initial state. This is a mechanism for FSM reset itself to initial state.

Table 2: The Abstract Events for LSA

1. The JPAM should be reconfigurable at run-time: in particular, users should be able to add (and remove) JFSMs as new types of intrusions become of concern (and old types cease to be).

2. Adding JFSMs should not require recompilations of the JPAM.

To accommodate these concerns, we use a table-driven implementation of FSMs together with a generic driver routine. A new JFSM is added simply by defining a table for it in a file and then loading it by reading this file into JPAM; the driver routine (which will be compiled into JPAM) would then handle the "execution" of the JFSM.

Moreover, for modeling temperal relations among events, timing information is introduced in JFSM's state transition diagram. A state transition will depend not only on the event identity itself but also on the occurrence time of that event. In order to catch unexpected behavior within time limit, JFSM also contains several timers to identify two special categories of JFSM internal events, "Alarm" and "Deadline." *Alarm* means that something has happened within a time constraint, *i.e.,* it probably happen too soon. *Deadline* means that something has not happened within a time constraint or even not happened yet, *i.e.,* it probably happened too late. This section describes each of these concepts in turn.

5.1 A Simple Conventional FSM

Definition 5.1 (Conventional FSM Generator [17]) *A Finite State Machine is a* $5-$*tuple:*

$$M = (Q, \Sigma, \delta, q_0, F),$$

where Q *is a finite set of states,* Σ *is the alphabet of events or input symbols,* $\delta \subset Q \times \Sigma \times Q$ *is the transition function mapping* $Q \times \Sigma$ *to* Q*. That is,* $\delta(q, a)$ *is a state for each state* q *and input symbol* a*.* $q_0 \subseteq Q$ *is the set of initial states and* $F \subseteq Q$ *is the set of final states.*

Occasionally we will omit the set of final states; this means that any state can be a final state, *i.e.,* $F = Q$.

The basic concept of a FSM [17] is simple: the machine contains a specific number of distinct internal states, of which one represents the possible active state. A finite set of input symbols is mapped to a finite set of output symbols by each state; an input symbol is given to the FSM, which returns an output symbol before making a possible transition to a new state.

In a traditional tabular representation of an FSM, each row in the table represents a state, and each column represents an input. If the $(i, j)^{th}$ element of such a table is k, that condition means that if the FSM is in state i and input j arrives, the new current state should be k. States are usually encoded as integers in the range $0, \ldots, N-1$, where N is the total number of states.

For example, the FSM in Figure 13 which has three states and six transitions can be represented as the table in Figure 14. Each element contains an output symbol and a new state in the form O/S. In fact, the FSM also can be constructed by the transition table such as in Figure 15.

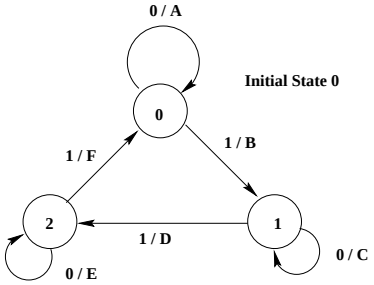

Figure 13: Simple FSM example

States\inputs	Input 0	Input 1
state 0	A / 0	B / 1
state 1	C / 1	D / 2
state 2	E / 2	F / 0

Figure 14: FSM as the Table

Current States	Input Symbol	Output Symbol	Next State
0	**0**	**A**	**0**
0	**1**	**B**	**1**
1	**0**	**C**	**1**
1	**1**	**D**	**2**
2	**0**	**E**	**2**
2	**1**	**F**	**0**

Figure 15: FSM as the Transition Table

When processing a new input symbol, the FSM checks the transiton table and looks for a suitable transiton in the currently active state, which can handle this input, and then advance itself to next state. If the example FSM gets input '1', it will advance to next state; otherwise, it will stay in the current state. In this report, we use the second approach to construct a FSM. For example, if the initial state is 0, the input sequence string "110101" will be mapped to the output string "$BDEFAB$", and the final state is 1.

5.2 JiNao Real-Time FSM Specification

In real-time modeling, a timed automaton accepts *timed words*; each of them is associated with an occurrence time. The most common way to introduce timing information in a process model is by associating lower and upper bounds with transitions. Examples of these include timed Petri nets [40], timed I/O automata [26], timed transition systems, and timed assertional proof system [35] which also showed a simple proof system by just introducing state variables of the *last event occurrence time* and *event deadline*

time but did not use a "current time" variable as other models. Real-time properties are stated using traditional assertions based on the temporal logic.

Most of these systems deal with the "discrete-time model" instead of "continuous dense-time model". [2, 1] proposed a timed finite automata to annotate state-transition graphs with time constraints using a finite number of real-valued clocks for the dense-time model. Unlike other approaches, a bound on the time taken to traverse a path in the automaton, not just the time interval between the successive transitions, can be directly expressed.

The JiNao Real-Time FSM (JFSM) model is a variant of the *Timed Automota* model [1] by associating each real-time state with two time variables, T_{in} and T_{last}, and associating each real-time transition with a conditional guard with time constraints. T_{in} of each state indicates the time when JFSM enters this state, and T_{last} of each state indicates when the very last transition happens in this state. $T_{current}$ is used in JFSM to record the current event time. By this design, JFSM can easily inspect different time constraints between states and between transitions to comply with the protocol specification.

Definition 5.2 (JFSM M) *A JFSM (JiNao Finite State Machine), M^J, consists of 8 tuples:*

$$M^J = (Q, \Sigma, \delta, q_0, F, FSMid, ReportInfo, EffExtQue),$$

where Q is a finite set of real-time states, Σ is the alphabet of events or input symbols, $\delta \subset Q \times \Sigma \times Q$ is the transition function mapping $Q \times \Sigma$ to Q. That is, $\delta(q, a)$ is a real-time state for each real-time state q and input symbol a. $q_0 \subseteq Q$ is the set of real-time initial states and $F \subseteq Q$ is the set of real-time final states. Furthermore, the construct of M^J also contains three more components related to the intrusion detection: string $FSMid$ to identify M^J, $ReportInfo$ to represent the message which will be sent out when an intrusion is detected (i.e., reaches a **critical transition***), and a queue $EffExtQue$ for recording the* **effective execution**.

Definition 5.3 (Input Events of JFSM) *The input event e_i is a 2-tuple:*

$$e_i = (ename, etime),$$

where etime is the timestamp for the occurrence of e_i, ename is the event identity as defined in Table 2.

We did not differentiate the "Input" and "Output" actions as in I/O automata [28], but in our implementation, the prefix *i_* and *o_* of *ename* were used to identify whether it was an "outgoing" or "incoming" LSA event. Furthermore, events can be sorted in partial order by their values of *etime* fields.

Definition 5.4 (Real-Time Timer Function) *A real-Time timer t_i is a 2-tuple:*

$$t_i = (val, \omega),$$

where $val \in \mathcal{N}^+$ represents the value of t_i, ω is the timer function, $\omega : \mathcal{N}^+ \to \mathcal{N}^+$, which indicates how to calculate the timer value val.

Because JFSM is dealing with the discrete event model, the time space here is in the integer-valued space. Of course, in physical processes, events do not always happen at integer-valued times, and continuous time must be approximated by choosing some fixed quantum, which could limit the accuracy with which physical systems can be modeled [2]. However, this model could be extended to the continuous time space (i.e., $\mathcal{R}^+$-valued) without too many modifications. Furthermore, timed JFSM is event-driven, so a global variable $T_{current}$ is kept updated to the newest event time whenever an event happens.

Definition 5.5 (Real-Time State of JFSM) *Each real-time state RtS_s in JFSM is a 5-tuple:*

$$RtS_s = (sID, T_{in}, T_{last}, \tau, RtT),$$

where

- *sID is RtS_s's identity in string,*
- *T_{in} is the time when JFSM was entering RtS_s from another state RtS_p such that $RtS_s \neq RtS_p$,*
- *T_{last} keeps the very last time the event happens in the current state RtS_s,*

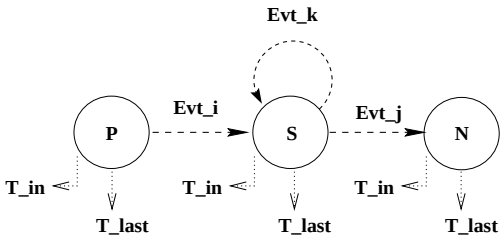

Figure 16: state of JFSM

- *τ is the set of timers for this state RtS_s (*i.e.*, any $t_i \in \tau$),*
- *RtT is the set of available real-time transitions for this state RtS_s.*

According to the above definition, when JFSM is entering to a state RtS_s from another state RtS_p ($RtS_s \neq RtS_p$) because of the input event e_i, $RtS_s.T_{in}$ and $T_{current}$ will be updated to the timestamp of e_i, $e_i.etime$. If a new transition occurs from a RtS_s to RtS_s itself, then $RtS_s.T_{in}$ will remain the same. That is, as long as JFSM stays in the current state (*e.g.,* looping around the current state), $RtS_s.T_{in}$ will not be changed; only $RtS_s.T_{last}$ and $T_{current}$ will be updated. The elapsed time during which JFSM has stayed in the "current state" can be computed by $(T_{current} - RtS_s.T_{in})$. In this way, local timeout for this specific state can be defined. Furthermore, the inter-LSA arrival time is $(T_{current} - RtS_s.T_{last})$, and the elapsed time since the specific pattern began from the initial state is $(T_{current} - RtS_{init}.T_{last})$, given that RtS_{init} is the initial state.

On the other hand, if a new transition is from a RtS_s directly to a different state RtS_n (*i.e.,* $RtS_s \neq RtS_n$) because of e_j, $RtS_n.T_{in}$ will be updated to $e_j.etime$, while $RtS_s.T_{in}$ will still remain the same. Therefore, $(RtS_n.T_{in} - RtS_s.T_{in})$ represents how much time JFSM spent in RtS_s. In other words, there is no time elapsed for transitions, and the time JFSM exits the current state RtS_s is equal to the time JFSM enters the next state RtS_n (*i.e.,* $RtS_s.T_{last} = RtS_n.T_{in}$.)

In our current implementation, three timers T_1, T_2, and T_3 were calculated by the functions,

$$T_1 = T_{current} - RtS_s.T_{last} \ ;$$

$$T_2 = T_{current} - RtS_s.T_{in} \ ;$$

$$T_3 = T_{current} - RtS_{init}.T_{last} \ ;$$

where $T_{current}$ is the current JFSM time; RtS_{init} is the initial state, and RtS_s is the current state. T_1 is the elapsed time from the very last event (i.e., event inter-arrival time) in the current state; T_2 is the elapsed time since entering this state, and T_3 is the elapsed time since this pattern began from the initial state.

Whenever an input event occurs, JFSM takes that input and checks if there is a transition available for handling this input and determines whether it can advance to the next state or not. Transitions in JFSM can be conditional. One input event may have more than one transition that can handle it, and JFSM will process the first one which satisfies the given conditions (e.g., time constraints).

Definition 5.6 (Conditional transition) *A conditional transition is as a 5-tuple,*

$$\pi_i = (RtS_{from}, e_{in}, \gamma, RtS_{to}, Cond_i),$$

where

- *RtS_{from} is the "from" state in this transition,*
- *e_{in} is the input event which triggers this transition,*
- *γ is the output result accompanied with this transition,*
- *RtS_{to} is the "to" state after this transition,*
- *$Cond_i$ is a* Condition *or* Guard *for this transition. When an input event occurs, only if the condition $Cond_i$ is true, can JFSM execute this transition to advance itself from RtS_{from} to the next state RtS_{to}.*

Condition $Cond_i$ can be represented in a complex way by a temporal logic language. However, the set of JFSM transitions is one kind of the conditional transition, which only considers the conditions related to time constraints (*e.g.,* several timeouts in a protocol) with simple semantics to describe it. When a transition π_i is "*available*" in the current state, the current input event $e_{current}$ happens such that $e_{current} = \pi_i.e_{in}$, and all the conditions are valid. That is, all the time constraints on this transition are satisfied in this state at this specific time.

Definition 5.7 (Time Constraints of JFSM) *Each time constraint for a timer t_i is specified by a 2-tuple, (T_{min}, T_{max}), where T_{min} and T_{max} represent lower and upper bounds for the value of the specific timer t_i*

Definition 5.8 (Real-Time Transition of JFSM, RtT) *Each real-time transition RtT_i in JFSM is a 5-tuple:*

$$RtT_i = (RtS_{from}, e_{in}, \gamma, RtS_{to}, \Theta),$$

where

- *RtS_{from} is the "from" state,*
- *e_{in} can be either a real input event triggering RtT_i , or a special* **NULL** *event e_ϕ,*
- *γ is the output result,*
- *RtS_{to} is the "to" state,*
- *Θ is the set of timer constraints for this transition RtT_i.*

Three timers used in current JFSM are $T1$, $T2$ and $T3$. $T1$ is to count the inter-arrival time between current and previous events, which can make sure that, in the same state, transitions must happen neither too soon nor too late. $\theta_1 = (T1_{min}, T1_{max})$ represents lower and upper bounds (integers in seconds) for the value of $T1 = (T_{current} - RtS_{from}.T_{last})$, and $\theta_2 = (T2_{min}, T2_{max})$ represents lower and upper bounds (integers in seconds), respectively, for $T2 = (T_{current} - RtS_{from}.T_{in})$. In other words, for allowing RtT_i to occur, the time that JFSM spent in RtS_{from} must be higher than $T2_{min}$ and lower than $T2_{max}$. Furthermore, $T3 = (T_{current} - RtS_{init}.T_{last})$ could specify that the whole pattern must happen within the constraint $\theta_3 = (T3_{min}, T3_{max})$.

For simplicity and to limit confusions, we will represent each transition as an 10-tuple

$$RtT_i = (RtS_{from}, e_{in}, \gamma, RtS_{to}, T1_{min}, T1_{max}, T2_{min}, T2_{max}, T3_{min}, T3_{max}),$$

in the later description. Note that in our approach, JFSM time constraints are associated with transitions, and timers are associated with each state. There is a unified clock in whole JFSM, but there could be several timers in each state. It is not necessary to reset timers because they will be recalculated while still in the current state.

Furthermore, a transition RtT_i is **effective** if $RtT_i.RtS_{from} \neq RtT_i.RtS_{to}$. That is, it can change JFSM to a different state. If the pair value of (Ti_{min}, Ti_{max}) is $(0, \infty)$, then timer T_i does not matter, and JFSM will just ignore it.

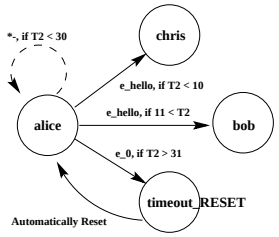

Figure 17: An example of JFSM Transitions

In JFSM, we can have two different transitions from the same state on the same input event. For example(as shown in Figure 17), we have:

$$RtT_1 = (RtS_{alice}, e_{Hello}, \gamma_1, RtS_{chris}, 0, \infty, 0, 10, 0, \infty),$$

and,

$$RtT_2 = (RtS_{alice}, e_{Hello}, \gamma_2, RtS_{bob}, 0, \infty, 11, \infty, 0, \infty).$$

Here, we do not care about the frequency of two successive event transitions and the total elapsed time since the pattern began from the initial state. We only consider time for staying in the current state RtS_{alice}. If the e_{Hello} event occurs less than 11 seconds after JFSM entered RtS_{alice}, then RtT_1 will happen and JFSM will advance to RtS_{chris}. Otherwise, RtT_2, a effective transition, will be triggered and JFSM will enter RtS_{bob}.

Please note that a special case, such as the following effective transition:

$$RtT_3 = (RtS_{alice}, e_{\phi}, \gamma_3, RtS_{timeout}, 0, \infty, 31, \infty, 0, \infty)$$

represents that, even if NO real events occur, JFSM will only stay in RtS_{alice} for at most 30 seconds and advance to state $RtS_{timeout}$. JFSM uses a time called "Event Driven Time". That is, it only updates its local time to the newest event time whenever an event happens. In case no event happened within a period of time, we also assume that there is a never-stop clock engine generating e_{tick} regularly every short (finite) period. This period is not necessary to be a fixed interval because of the discrete event driven property. For instance, e_{tick} could be the *HelloClockTick* event in OSPF.

If no transition is available for current events, an internal transition will trigger to a special "**fail state**" RtS_{fail} and output this new sequence of events from the initial state as well. This situation happens when JFSM finds a new pattern that it cannot handle. This sequence of events, serving as a new learned pattern, can be deposited to the knowledge-base. JPAM can either create or enhance a JFSM to handle it. In RtS_{fail}, JFSM will automatically reset itself to the initial state to keep monitoring later intrusions.

To eliminate unwanted event transitions, JFSM can use a wildcard "$*-$" to represent all other events except those in the available transitions. For example,

$$RtT_4 = (RtS_{alice}, *^{-}, \gamma_4, RtS_{alice}, 0, \infty, 0, 30, 0, \infty)$$

represents that, if any event occurs without matching the other available transitions (RtT_i, $i = 1, 2, 3$), and T_2 is still within 30 seconds, JFSM will stay in RtS_{alice}.

For reducing false alarms and monitoring continuous intrusions, a special "**reset state**" can also be used. If a state name has a "_RESET_" suffix, this state is considered as a "reset state" which will immediately reset to the initial state after JFSM reaches it automatically. For example, "N_RESET_" could represent the "Normal RESET" in which JFSM just does a valid reset, while "A_RESET_" represents the "Alarm RESET" in which JFSM has detected an attack and restarts itself to the initial state to keep monitoring later intrusions.

Definition 5.9 (Critical Transition of JFSM) *A critical transition Ct_i is the transition by which JFSM should print a warning message and report the related information to some responsible module. This critical transition is used, for instance, when JFSM detects an intrusion pattern; it should trigger M^J to report a detection message to the decision module, which contains $FSMid$, $ReportInfo$, and $EffExtQue$.*

For example, if $(\pi_1, \pi_2, \pi_3, ..., \pi_n)$ is the sequence of transitions in JFSM for a specific intrusion pattern, the last transition π_n should be defined as a *Critical Transition*.

JFSM with all the transitions is configured by a special configuration file. A critical transition can be also defined in this configuration file by a format in which its output string has a prefix '~'. An example of JFSM specification for OSPF/LSA intrusion detection is shown in Appendix A.2. The algorithm for the transition function of JFSM can be found in Appendix A.1.

Definition 5.10 (Execution of JFSM) *An execution fragment for a JFSM M is a finite or infinite sequence $RtS_0, RtT_1, RtS_1, RtT_2, RtS_3, ...$ of alternating states and transitions. An execution is an execution fragment beginning with a start state.*

Given an input string, the behavior of M can be represented as a sequence of transitions such as $(RtT_1, RtT_2, ...)$. The **effective execution** of M is the sequence of transitions RtT_i, such that in each of them two successive states are different.($\forall i$, $RtT_i.RsT_{from} \neq RtT_i.RsT_{to}$). That is, it does not include execution fragments looping around the same state and only contains those transitions which can make JFSM change its state.

5.3 JFSMs for Known OSPF attacks

To detect all the attacks mentioned in Section 3, three JFSMs for different attacks were designed: seq++ attack in Figure 18, maxage attack in Figure 20, and maxseq attack in Figure 21. The definitions of the events are in Table 2. Contents of the configuration file for these JFSMs can be found in Appendix A.2.

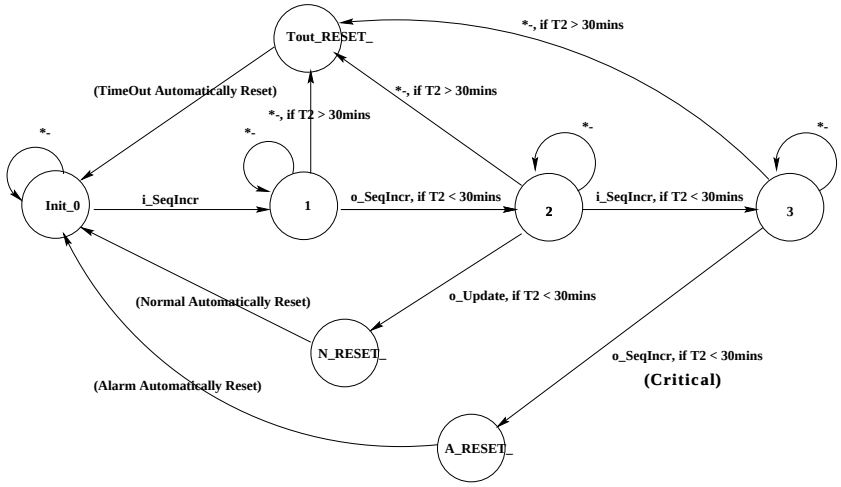

Figure 18: FSM for LSA Seq++ Attack

As an example, a JFSM to detect the `Seq++` attack is depicted in Figures 18 and 19. In this example, JiNao is running on the **originator** of the monitored LSAs. If JFSM twice receives alternate incoming and outgoing LSA with increasing sequence numbers, it will raise an alarm. The "o_SeqIncr" event by outgoing LSA from the originator is intended to "fight back" the previous malicious one.

```
FSMId: seq++FSM
ReportInfo: For_seq++
logToFileName: seq++AttTest
maxLogQueSize: 300
Transitions:
# from event d output to MinI MaxI MinS MaxS MinP MaxP
# ------------------------------------------------------
# critical transition is prefixed by '~'

01 init0 i_SeqIncr i_SeqIncr       1            0 inf 0    inf  0 inf
02 init0 *-        StayAt_0        init0        0 inf 0    inf  0 inf
03 1     o_SeqIncr oFightBack      2            0 inf 0    1800 0 inf
04 1     o_Update  NotFtBackOnTime Tout_RESET_  0 inf 1800 inf  0 inf
05 1     *-        StayAt_1        1            0 inf 0    inf  0 inf
06 2     i_SeqIncr iSeqIncrAgain   3            0 inf 0    1800 0 inf
07 2     o_Update  GoBk_init       N_RESET_     0 inf 0    1800 0 inf
08 2     *-        NotFtBackOnTime Tout_RESET_  0 inf 1800 inf  0 inf
09 2     *-        StayAt_2        2            0 inf 0    inf  0 inf
10 3     o_SeqIncr ~oFightBackAtt  A_RESET_     0 inf 0    1800 0 inf
11 3     *-        NotFtBackOnTime Tout_RESET_  0 inf 1800 inf  0 inf
12 3     *-        StayAt_3        3            0 inf 0    inf  0 inf
```

Figure 19: a JFSM Configure File for LSA `Seq++` Attack

Lines 1-2 in Figure 19 describe two transitions available for the initial state. Line 1, specially, describes that, if the new **incoming** LSA's sequence number is bigger than the previous **outgoing** LSA (with the same *LSAid*), an `i_SeqIncr` event will trigger the transition and the JFSM will then be in State 1. In the attack scenario, this transition represents that the LSA's originator has received its own LSA instance

with an abnormal sequence number — normally the sequence number should be the same as or less than the one it just sent out.

However, this unusual event by itself is not enough to raise a red flag for the `Seq++` attack. In OSPF, if a router crashes, its LSA instances will still be kept by other routers for about 30 minute (or 1800 seconds). Therefore, when this router restarts within the 30 minutes limit, it can still receive "old" LSA instances with a bigger sequence number. Under such a case, the LSA's originator will issue another LSA with a even bigger LSA to "clean-up" the old copies out there. Line 3 exactly represents this "clean-up" or "fight-back" scenario, and it is an `o_SeqIncr` as the originator will increment the LSA by at least one. Please note that at this state, the originator could receive more than one `i_SeqIncr` **before** the `o_SeqIncr` is out. The rationale is that, if the originator has more than one neighbor, it could receive more than one copy of `i_SeqIncr`. However, after the `o_SeqIncr` is out, the current outgoing LSA is updated. It is impossible for an old copy of the LSA instance to raise another `i_SeqIncr` as the most recent outgoing sequence number has been updated. Finally, if the originator itself has crashed, then it will not perform the "fight-back" within the 30 minute limit. Therefore, it will trigger the Line 4 transition and go back to the initial state.

Lines 6-9 in Figure 19 describes whether this originator receives another unusual `i_SeqIncr` from one of its neighbors after it delivered an `o_SeqIncr`. If this is indeed a `Seq++` attack, the attacker will keep increasing the sequence number and therefore, in Line 6, within thirty minutes, we will receive another `i_SeqIncr`. The transition in Line 10 is **critical**, as after another fight-back from the originator is observed, the JFSM will raise a red alarm about `Seq++` attack to the decision module.

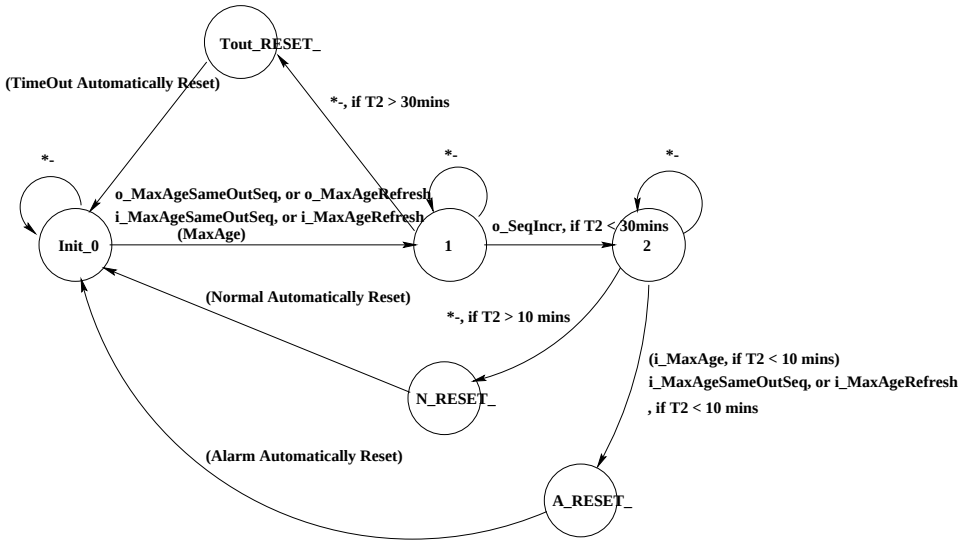

Figure 20: FSM for LSA Maxage Attack

In Figure 20 for Maxage attack, JFSM starts from initial state *Init*_0. If JPAM receives incoming or outgoing (originated) LSA with the same sequence number of previous outgoing LSA, it will be considered as the start of a suspicious premature process for current LSA. JFSM will first advance itself to state 1. From state 1, within 30 minutes, JFSM should receive an outgoing LSA Update which has a bigger sequence number than the previous one (usually increased by one) to "fight back" the last LSA, and advance itself to state 2. From state 2, if it receives another incoming maxage event within 10 minutes, it will advance to an "alarm state," send an alarm message to Local Decision Module, and immediately reset to the initial state in order to monitor the next maxage attack pattern. In order to differentiate the real premature aging and attack, we define a threshold of two maxage events. JFSM can be adjusted to a certain level of sensitivity to eliminate all false alarms by similar concepts.

In Figure 21 for Maxseq attack, JFSM differentiates good or bad implementation by two patterns: {MaxSeq, MaxAgeMaxSeq, initSeq, Update} and {MaxSeq, initSeq}. If OSPF router is good, when receiving incoming LSA with a maximum sequence number, it should issue an outgoing LSA with a maximum age and a maximum sequence to purge the previous LSA first, and then issue a LSA with an initial (minimal) sequence number. After that, the normal update LSAs will happen within every 30 minutes. If the OSPF router has a wrong implementation when receiving incoming LSA with a maximum sequence number, it will directly issue LSA with an initial sequence number without purging the previous

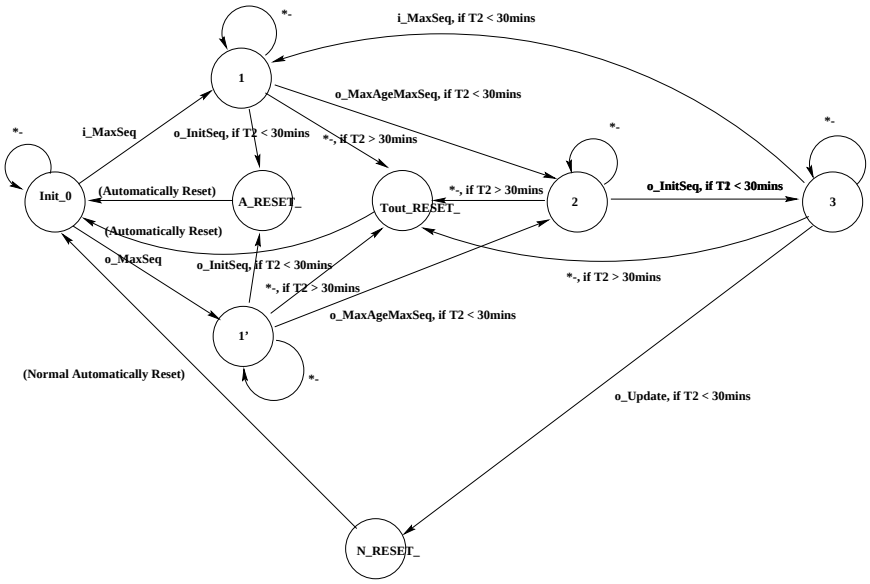

Figure 21: FSM for LSA Maxseq attack

one first, and cause a not-stop fight back cycle. Once this situation is detected, JFSM will report to Local Decision Module to re-act this attack by wrapping it with JiNao wrapper, which will help this router by issuing an extra maxage LSA for purging purpose.

6 Implementation and Experimental Results

We have developed an JFSM generator and a general JFSM object class which is dynamically configurable and can produce any JFSM by setting the transition table in the JFSM configuration file. According to Appendix A.1, transition functions are represented as a transition table in that file. The running state of an JFSM is advanced by the table lookup mechanism. Each state is associated with several transitions. Because JPAM is configurable, the configuration files need to be constructed first, and then JPAM will run itself online. This section explains the formats of two kinds of configuration files, one for JPAM and one for JFSM, in more detail.

6.1 Configurable JPAM

Configuration File of JPAM

The format of the configuration file of JPAM is as follows. Each line specifies several fields; the fields are separated by white space.

```
<RouterId> <LSID> <LSType> <CkOutF> <logFile> <NumOfFSM> <FSMconfigfile> ...
```

For instance, here is an example of the configuration file.

```
192.0.1.1 192.0.1.1 1 1 event192.0.1.1 0
192.0.1.2 192.0.1.2 1 1 NULL           1 JFSM0
192.0.1.4 192.0.1.4 1 0 event192.0.1.4 3 JFSM0 JFSM1 JFSM2
192.0.2.1 192.0.2.1 1 0 NULL           0
192.0.2.3 192.0.2.3 1 0 event192.0.2.3 1 JFSM3
192.0.2.5 192.0.2.5 1 0 event192.0.2.5 0
```

The first three fields specify the LSA object needing to be monitored. As mentioned in Section 2, LSA can be uniquely identified by these three fields. The fourth field decides whether to consider "outgoing" LSA or not. Although JiNao can observe all LSA generated by all the routers in the same AS (because of the flooding), it only can detect the outgoing LSA of its own originator router. The differentiation of

"incoming" and "outgoing" may change the construct of JFSMs later. The fifth field is to specify the name for log file. Note that if the file name is "NULL" then JPAM will not do the log file. That is, it only processes the events of this LSA online, and when the program finishes, there will be no event file logged for after-fact analysis. The sixth field represents the number of JFSM for analyzing this LSA, and the fields after that specify the configuration file names for each JFSM.

For above example, the timestamped events for the first LSA (192.0.1.1 192.0.1.1 1) are logged into file event192.0.1.1, and no JFSM applies to them. So the event sequence can be seen in the event log file. The events of the second LSA(192.0.1.2 192.0.1.2 1) are not logged into the file but are processed directly to JFSM0. The events for the third LSA are logged to file event192.0.1.4 and also are processed by three JFSMs: JFSM0, JFSM1, and JFSM2. Note that the event sequence of each LSA can be processed by different JFSMs at the same time. The fourth LSA has no log and no JFSM, so JPAM just acts as an online packet parser.

Configuration File of JFSM

Each JFSM is a timed nondeterministic automata, which can be constructed by the following format in a file.

```
FSMId: <FSMname>
ReportInfo: <InfoString>
logToFileName: <ResultOutputFileName>
maxLogQueSize <theSizeKeepInLogQueue>
Transitions:
<FromState> <Input> <Output> <ToState> <T1min> <T1max> <T2min> <T2max> <T3min> <T3max>
...
```

$< FSMname >$ will automatically prefix with the EvtAbs ID string which uniquely identifies which JFSM is reporting the detection results to the Local decision module when it detects attack. $< InfoString >$ is the string representing the report information when it detects a positive alarm. If $< ResultOutputFile >$ is not equal to "NULL", it will output the result to this file to provide after-fact information, as well as to the standard output device. $< theSizeKeepInLogQueue >$ is a number which indicates the maximal size for each log event sequence. Generally, it is impossible to record everything in memory. A queue will always keep the last several events up to the size of queue. In order to detect a known attack pattern, the queue size must be at least larger than the size of this attack pattern.

After "Transition:", each line specifies 10 fields, and the fields are separated by white space. These 10 fields represent one transition of this finite state machine. JFSM can have as many transitions as it needs, and each transition is provided one line in this format.

The first 4 fields represent the transition of a conventional FSM model. The fifth to tenth fields represent the lower ad upper bounds for three timers as mentioned before. In JFSM model, all input events and states are associated with time-stamps. All the implementation complies with the definitions mentioned in Sections 5.2

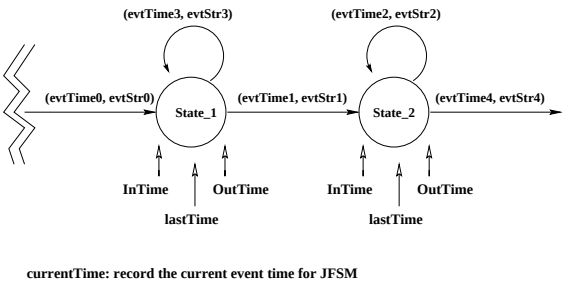

Figure 22: simple view of JFSM

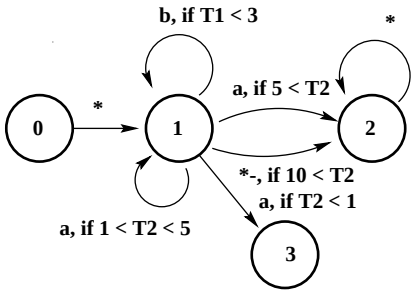

Figure 23: An example for the JFSM configuration file

Three timer values have been added in JFSM State class, *InTime*, *lastTime* and *OutTime*. InTime is the time for the transition to first advance JFSM into this current state; lastTime is the time the very last transition (including in, out or stay-Around) happening in this current state. OutTime is the time JFSM stepping out of this state. So the total time JFSM stays in this state is $(OutTime - InTime)$. That is,

InTime will be equal to the event time at which JFSM gets into this state, and OutTime is equal to the event time at which JFSM exits this state. There is a global variable *currentTime* of JFSM to record the current event time. When JFSM transitions stay in the current state, $(currentTime - InTime)$ is the time for JFSM spending at the current state so far. $(currentTime - InitialState.lastTime)$ is the time elapsed from the initial state. It can be used to specify the time requirement for a successful attack pattern. Figure 22 is a simple view of JFSM.

For example (as shown in Figure 23), state_2 is a dead State(in which JFSM will stay there forever unless you stop it) for JFSM transitions and the transitions are as

```
0 *  always1   1 0 inf 0  inf 0 inf
1 b  receive_b 1 0 3   0  inf 0 inf
1 a  tooSoon_a 3 0 inf 0  1   0 inf
1 a  output_a  1 0 inf 1  5   0 inf
1 a  tooLate_a 2 0 inf 5  inf 0 inf
1 *- timeout_a 2 0 inf 10 inf 0 inf
2 *  deadState 2 0 inf 0  inf 0 inf
```

Note that "*" is the wild-cast string stands for any input string.

From initial state 0, no matter what input, JFSM always goes to state 1. We are modeling that in state 1, if "b" comes in and happens within 3 seconds (too soon) from previous events, JFSM will stay in state 1 again, and update the lastTime as b's *evtTime*, but *InTime* will still be unchanged. In state 1, if "a" happens within 1 second (too soon), then JFSM will become state 3. In state 1, if "a" happens after 5 seconds (too late), then it will become state 2. After enters state 2, it will ignore any input thereafter and stay in state 2 until JFSM stops. The sixth transition is to prevent JFSM from waiting for "a" forever.

6.2 Testbed and Experimental Results

The testbed for our demo in the Air Force Base laboratory is as in Figure 24. Machines are referred to as $R1$, $R2$, $R3$, $Middle$, $Cisco$, and $R6$. R1 is our victim router, and R2 is the attacker.

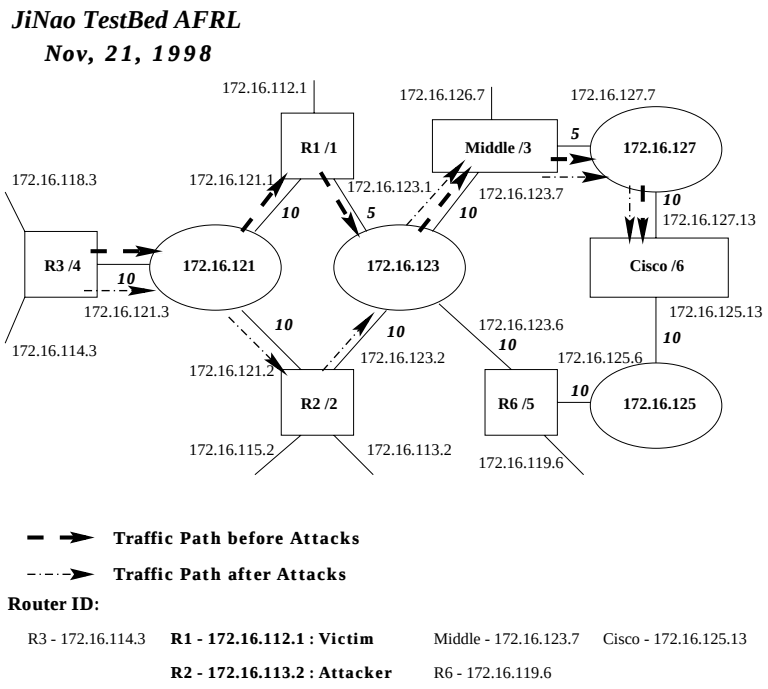

Figure 24: Demo TestBed in Air Force Rome Lab

Before launching any attack, we run traceroute from R3 to Cisco, and observed that the traffic path was {R3, R1, Middle, Cisco}. Attacks not only changed the fields of sequence number, age, checksum in LSA, but also changed the cost metrics. After we launched attack from R2 to R1 with LSA(172.16.121.1 172.16.121.1 1) , the path was changed to {R3, R2, Middle, Cisco}. That is, after the attack R2 could

get the traffic it could not get before the attack. Because the attacks were persistent, JPAM can catch them right after the second attack LSA happened and report to the decision module for reaction. For a malicious MaxSeq attack, after detecting it, JPAM sent a message to the Prevention module which performed a correct purge action to remove the MaxSeq attack effect.

Results showed out that all the known attacks are correctly detected. JPAM had 100% accuracy in the demo. JiNao can even run after the attacks have taken place, and detect them right after the next attack pattern happens. So as long as the attack is persistent, JPAM can detect it at least after the second run. OSPF's self-stabilization property makes attacks must-be-persistent and this approach becomes much more effective.

7 Conclusions, Summary and Future Work

This paper presented the approach taken in the JiNao project for the design and implementation of the protocol analysis module for detecting attacks to OSPF routing protocol. We proposed a frame work for protocol-based intrusion detection, which could be extended to other protocol analysis. Our design of JPAM/JFSM is object-oriented such that the implementation of the JPAM is modulized.

Furthermore, routing protocols usually need to deal with several time-related events, and the "*follow*" semantics with intervening events is difficult to conduct by a conventional Finite State Machine model. Therefore, an event abstractor (EvtAbs) was developed to catch the important abstract events according to OSPF protocol specification, and a extended timed automata, JiNao Finite State Machine (JFSM), was developed to handle timed events and time constraints such as alarms (happen too soon) and deadlines (happen too late). With such an extension, it is thus possible to analyze the time-related behavior of the routing protocols and do the pattern matching.

The model formalized above is effective in terms of the false alarm rate and the response latency. We have experimented our implementation against event streams collected at both MCNC's and NCSU's routing testbeds. We have also presented the DEMO for the DARPA in the Air Force Base laboratory. Both normal event sequences and three attacks were tested. The JPAM module can successfully detect all three significant attacks and can be extended to other attack scenarios.

The advantages of the protocol analysis approach for intrusion detection are

1. best for known attacks.
2. can catch unknown attacks if the good/expected behavior JFSM is specified.
3. can catch real-time hit-and-run attacks.
4. can reflect the real-time network events quickly.
5. can monitor routers' behavior closely.
6. can verify the implementations of the protocol.
7. can identify the weakness of the protocol.

We realized that an effective event engine is one of the key points for the success of JPAM. Future research should extend our approach to a *"Universal Event Abstractor"* which can be configurable to any protocol just by several configuration files. For example, protocol headers can be described in the file, and predefined events can be put on the file as well in a special language, such that this Universal Event Abstractor with a suitable protocol packet parser can generate any event needed for any different protocol.

A JFSM Configuration Examples

A.1 Transition Function of JFSM

JFSM can be constructed by the transition table in a file, in which each row specifies a transition $\pi_i \in \delta$ as 10-tuple,

$$\pi_i = (q_1, e_{in}, \gamma, q_2, T1_{min}, T1_{max}, T2_{min}, T2_{max}, T3_{min}, T3_{max}),$$

where q_1 is the from-state in this transition, $e_{in} \in \Sigma$ is the input event which triggers this transition and γ is the output resulted from this transition. q_2 is the next state after this transition. Ti_{min} and Ti_{max} are lower and upper bounds for timers Ti, $i = 1, 2, 3$. T_1, T_2, and T_3 are defined as,

$$T_1 = T_{current} - RtS_{current}.T_{last} \ ;$$

$$T_2 = T_{current} - RtS_{current}.T_{in} \ ;$$

$$T_3 = T_{current} - RtS_{init}.T_{last} \ ,$$

where RtS_{init} stands for the initial state of this JFSM; $RtS_{current}$ represents JFSM's current state; $T_{current}$ is the current time of JFSM; T_1 computes the inter-arrival time between the current and previous events; T_2 calculates the time JFSM has spent in the current state since it entered this state, and T_3 computes the elapsed time since the intrusion pattern has begun from the initial state.

Assume that q_1 is the current state of JFSM. When an input event $e_{current}$ occurs, the transition path from q_1 to q_2 is possible, if and only if,

$$\pi_i = (q_1, e_{current}, \gamma, q_2, T1_{min}, T1_{max}, T2_{min}, T2_{max}, T3_{min}, T3_{max}) \in \delta$$

and exists in the table for this current state, and the conditions are valid in the current state q_1 at this moment.

The transition function can be computed by the following algorithm.

Algorithm 1 JFSM Transition Function

1: $T_{current} \Leftarrow e_{cuurent}.etime$
2: $T_1 \Leftarrow T_{current} - RtS_{current}.T_{last}$
3: $T_2 \Leftarrow T_{current} - RtS_{current}.T_{in}$
4: $T_3 \Leftarrow T_{current} - RtS_{init}.T_{last}$
5: $bState_{Available} \Leftarrow false$
6: **for all** available transitions δ_i in the current state && $bState_{AvailableFound} \neq true$ **do**
7: **if** $\delta_i.T3_{min} \leq T_3 \leq \delta_i.T3_{max}$ && $\delta_i.T2_{min} \leq T_2 \leq \delta_i.T2_{max}$ && $\delta_i.T1_{min} \leq T_1 \leq \delta_i.T1_{max}$ **then**
8: $RtS_{next} \Leftarrow \delta_i.RtS_{to}$
9: $RtT_{current} \Leftarrow \delta_i$ {Record the current transition}
10: $bState_{Available} \Leftarrow true$
11: **if** $RtS_{next} \neq RtS_{current}$ **then** {Advance to the next different state}
12: $RtS_{current} \Leftarrow RtS_{next}$ {Update the current State}
13: $RtS_{current}.T_{in} \Leftarrow T_{current}$
14: $RtS_{current}.T_{last} \Leftarrow T_{current}$
15: **else** {Stay at the same current state}
16: $RtS_{current}.T_{last} \Leftarrow T_{current}$
17: **end if**
18: **end if**
19: **end for**
20: **if** $bState_{Available} == false$ **then** {No available transition}
21: $RtS_{current}.T_{last} \Leftarrow T_{current}$
22: $RtS_{next} \Leftarrow RtS_{fail}$ {Advance to the "fail" state}
23: $RtT_{current} \Leftarrow \delta_{internal}$
24: Output the unknown sequence from the initial state as a new pattern.
25: **end if**

A.2 A JFSM Configuration File Example for OSPF/LSA Intrusion Detection

JPAM configuration file

```
172.16.114.3 172.16.114.3 1 0 event172.16.114.3 0
172.16.118.3 172.16.118.3 1 0 event172.16.118.3 0
172.16.121.3 172.16.121.3 1 0 event172.16.121.3 0
172.16.112.1 172.16.112.1 1 1 event172.16.112.1 2 FSMmaxseq.conf FSMmaxage.conf
172.16.121.1 172.16.121.1 1 1 event172.16.121.1 0
172.16.123.1 172.16.123.1 1 1 event172.16.123.1 0
172.16.113.2 172.16.113.2 1 0 event172.16.113.2 0
172.16.115.2 172.16.115.2 1 0 event172.16.115.2 0
172.16.121.2 172.16.121.2 1 0 event172.16.121.2 0
172.16.123.2 172.16.123.2 1 0 event172.16.123.2 0
172.16.123.7 172.16.123.7 1 0 event172.16.123.7 0
172.16.126.7 172.16.126.7 1 0 event172.16.126.7 0
172.16.127.7 172.16.127.7 1 0 event172.16.127.7 0
172.16.119.6 172.16.119.6 1 0 event172.16.119.6 0
172.16.123.6 172.16.123.6 1 0 event172.16.123.6 0
172.16.125.6 172.16.125.6 1 0 event172.16.125.6 0
172.16.125.13 172.16.125.13 1 0 event172.16.125.13 0
172.16.127.13 172.16.127.13 1 0 event172.16.127.13 0
```

Seq++ JFSM configuration file

```
FSMId: seq++FSM
ReportInfo: For_seq++
logToFileName: seq++AttTest
maxLogQueSize: 100
Transitions:
init0 i_BigJumpSeqIncr iBigJumpSeqIncr 1 0 inf 0 inf 0 inf
init0 *- StayAt_0 init0 0 inf 0 inf 0 inf
1 o_BigJumpSeqIncr oFightBack 2 0 inf 0 1800 0 inf
1 o_Update ~NotFtBackOnTime Tout_RESET_ 0 inf 1800 inf 0 inf
1 *- StayAt_1 1 0 inf 0 inf 0 inf
2 i_BigJumpSeqIncr iBigJumpAgain 3 0 inf 0 1800 0 inf
2 o_Update ~GoBk_init N_RESET_ 0 inf 0 1800 0 inf
2 *- ~NotFtBackOnTime Tout_RESET_ 0 inf 1800 inf 0 inf
2 *- StayAt_2 2  0 inf 0 inf 0 inf
3 o_BigJumpSeqIncr ~oFightBackAtt A_RESET_ 0 inf 0 1800 0 inf
3 *- ~NotFtBackOnTime Tout_RESET_ 0 inf 1800 inf 0 inf
3 *- StayAt_3 3 0 inf 0 inf 0 inf
```

MaxAge JFSM configuration file

```
FSMId: maxageFSM
ReportInfo: For_maxage
logToFileName: maxageAttTest
maxLogQueSize: 100
Transitions:
init0 i_MaxAgeSameOutSeq MaxAgeOnce 1 0 inf 0 inf 0 inf
init0 i_MaxAgeRefresh MaxAgeOnce 1 0 inf 0 inf 0 inf
init0 o_MaxAgeSameOutSeq MaxAgeOnce 1 0 inf 0 inf 0 inf
init0 o_MaxAgeRefresh MaxAgeOnce 1 0 inf 0 inf 0 inf
init0 *- StayAt_0 init0 0 inf 0 inf 0 inf
1 o_SeqIncr fightBack 2 0 inf 0 1800 0 inf
1 *- NotFtBackOnTime Tout_RESET_ 0 inf 1800 inf 0 inf
1 *- StayAt_1 1 0 inf 0 inf 0 inf
2 i_MaxAgeSameOutSeq ~MaxAgeAttack A_RESET_ 0 inf 0 600 0 inf
2 i_MaxAgeRefresh MaxAgeOnce A_RESET_ 0 inf 0 600 0 inf
```

```
2 *- goBackInit0 N_RESET_ 0 inf 600 inf 0 inf
2 *- StayAt_2 2 0 inf 0 inf 0 inf
```

MaxSeq JFSM configuration file

```
FSMId: maxseqFSM
ReportInfo: For_maxseq
logToFileName: maxseqAttTest
maxLogQueSize: 100
Transitions:
init0 i_MaxSeq iMaxSeqOnce 1  0 inf 0 inf 0 inf
init0 i_MaxAgeMaxSeq iMaxAgeMaxSeq 1 0 inf 0 inf 0 inf
init0 o_MaxSeq oMaxSeqOnce 1' 0 inf 0 inf 0 inf
init0 o_MaxAgeMaxSeq oMaxAgeMaxSeq 1' 0 inf 0 inf 0 inf
init0 *- StayAt_0 init0 0 inf 0 inf 0 inf
1 o_InitSeq ~Incorrect A_RESET_ 0 inf 0 1800 0 inf
1 o_MaxAgeMaxSeq oPurgeMaxseq 2 0 inf 0 1800 0 inf
1 *- NoPurgeTout Tout_RESET_ 0 inf 1800 inf 0 inf
1 *- StayAt_1 1 0 inf 0 inf 0 inf
1' o_InitSeq ~Incorrect A_RESET_ 0 inf 0 1800 0 inf
1' o_MaxAgeMaxSeq oPurgeMaxseq 2 0 inf 0 1800 0 inf
1' *- NoPurgeTout Tout_RESET_ 0 inf 1800 inf 0 inf
1' *- StayAt_1' 1' 0 inf 0 inf 0 inf
2 o_InitSeq normalRESET 3 0 inf 0 1800 0 inf
2 *- NoInitTout Tout_RESET_ 0 inf 1800 inf 0 inf
2 *- StayAt_2 2 0 inf 0 inf 0 inf
3 o_Update NormUpdate N_RESET_ 0 inf 0 1800 0 inf
3 i_MaxSeq iMaxSeqAgain 1 0 inf 0 1800 0 inf
3 *- NormT_Out Tout_RESET_ 0 inf 1800 inf 0 inf
```

References

[1] R. Alur. Timed automata. In *In NATO-ASI Summer School on Verification of Digital and Hybrid Systems*, 1998. http://www.cis.upenn.edu/ alur/onlinepub.html.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. http://www.cis.upenn.edu/ alur/pub.html.

[3] Debra Anderson, Thane Frivold, and Alfonso Valde. Next generation intrusion detection expert system (nides): A summary. Technical report, SRI International's Computer Science Laboratory (CSL), May 1995. http://www2.csl.sri.com/nides/index5.html.

[4] S.F. Wu B. Vetter, F. Wang. An experimental study of insider attacks for the ospf routing protocol. In *5th IEEE International Conference on Network Protocols, Atlanta, GA*. IEEE press, October 1997. http://shang.csc.ncsu.edu/pubs.html.

[5] Kirk A. Bradley, Steven Cheung, Nick Puketza, Biswanath Mukherjee, and Ronald A. Olsson. Detecting disruptive routers: A distributed network monitoring approach. In *Proceedings of the 1998 IEEE Symposium on Computer Security and Privacy, Oakland, CA*. IEEE press, May 1998. http://seclab.cs.ucdavis.edu/ cheung/.

[6] James Cannady and Jay Harrell. A comparative analysis of current intrusion detection technologies. In *Proc. of the Fourth Technology for Information Security Conference'96 (TISC'96)*, May 1996.

[7] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. Technical report, IBM Zurich Research Laboratory, May 1998. http://domino.watson.ibm.com/library/cyberdig.nsf/.

[8] Dorothy E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, SE-13(2), February 1987.

[9] P. D'haeseleer, S. Forrest, and P. Helman. An Immunological Approach to Change DEtection: Algorithms, Analysis, and Implications. In *1996 IEEE Symposium on Computer Security and Privacy*. Department of Computer Science, University of New Mexico, IEEE press, 1996.

[10] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 1, March 1974.

[11] S.F. Wu F. Wang, B. Vetter. Secure routing protocols: Theory and practice. Technical report, Department of Computer Science, NC State Univ., April 1998. http://shang.csc.ncsu.edu/pubs.html.

[12] J. Frank. Artificial intelligence and intrusion detection: Current and future directions. In *Proc. of the 17th National Computer Security Conference*, October 1994. http://seclab.cs.ucdavis.edu/papers.html.

[13] T.D. Garver and Teresa F. Lunt. Model-based intrusion detection. In *Proc. of the 14th National Computer Security Conference*, October 1991.

[14] Mohamed G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, pages 969–980, 1993.

[15] Ralf Hauser, Tony Przygienda, and Gene Tsudik. Reducing the cost of security in link state routing. In *ISOC Symposium on Network and Distributed System Security*, February 1997.

[16] Ted Herman. Stabilization research at iowa. http://www.cs.uiowa.edu/ftp/selfstab/main.html.

[17] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computations*. Addison-Wesley, 1979.

[18] Christian Huitema. *Routing in the Internet*. Prentice Hall, 1995.

[19] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transaction on Software Engineering*, 21(3), March 1995. http://www.cs.ucsb.edu/ kemm/netstat.html/documents.html.

[20] Harold S. Javitz and Alfonso Valdest. The sri ides statistical anomaly detector. In *Proc. of the IEEE Symposium on Research in security and Privacy*, pages 316–326, May 1991. http://www2.csl.sri.com/nides/index5.html.

[21] Y.Frank Jou, F. Gong, C. Sargor, S.F. Wu, and W. Cleaveland. Architecture design for a scalable intrusion detection for the emerging network infrastructure. Technical report, MCNC and Dept. of Computer Science of N.C. State Univ., April 1997. http://www.mcnc.org/HTML/ITD/ANR/JiNao.html.

[22] C. Kahn, P. Porras, S. Staniford-Chen, and B. Tung. A common intrusion detection framework, July 1998. Submitted to the Journal of Computer Security; http://seclab.cs.ucdavis.edu/cidf/.

[23] Sandeep Kumar and E. H. Spafford. A pattern-matching model for instrusion detection. In *PROCEEDINGS OF THE NATIONAL COMPUTER SECURITY CONFERENCE, pp. 11-21, Baltimore, MD*, 1994. http://www.cs.purdue.edu/coast/coast-library.html.

[24] Teresa F. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *Proc. of the 11th National Computer Security Conference*, October 1988. Outstanding Paper Award; http://www2.csl.sri.com/nides/index5.html.

[25] Teresa F. Lunt. A survey of intrusion detection techniques. *Computers & Security*, 12(4):405–418, 1993.

[26] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996. Chapter 23, http://theory.lcs.mit.edu/tds/timed-aut.html.

[27] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing, PODC'87, Vancouver, British Columbia, Canada*, August 1987. Technical Memo MIT/LCS/TR-387; http://theory.lcs.mit.edu/tds/papers/Lynch/tuttle.html.

[28] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1989. Technical Memo MIT/LCS/TM-373; http://theory.lcs.mit.edu/tds/papers/Lynch/CWI89.html.

[29] Nancy Lynch and Frits Vaandrager. Forward and backward simulations, part ii: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996. Technical Memo MIT/LCS/TM-487.c., http://theory.lcs.mit.edu/tds/timed-aut.html.

[30] J. Moy. RFC 2328: OSPF Version 2, April 1998. ftp://ftp.isi.edu/in-notes/rfc2328.txt.

[31] John T. Moy. *OSPF: Anatomy of an Internet Routing Protocol.* Addison-Wesley, 1998.

[32] B. Mukherjee, L.T. Heberlein, and K.N. Levitt. Network intrusion detection. *IEEE Network*, 8(3), May-June 1994. http://seclab.cs.ucdavis.edu/papers.html.

[33] S.L. Murphy and M.R. Badger. Digital signature protection of ospf routing protocol. In *Internet Society Symposium on Network and Distributed Systems Security*, 1996.

[34] Isabelle Rouvellou and George W. Hart. Automatic alarm correlation for fault indentification. *IEEE*, 1995.

[35] A.U. Shankar. A simple assertional proof system for real-time systems. In *13th IEEE Real-Time Systems Symposium.* IEEE press, December 1992. http://www.cs.umd.edu/ shankar/.

[36] A.U. Shankar. Reasoning assertionally about real-time systems. In *Proceedings of the IEEE, special issue on Real-Time Systems.* IEEE press, January 1994. http://www.cs.umd.edu/ shankar/.

[37] Sandeep Kumar Shukla. Home page on self-stabilization. http://www.cs.albany.edu/ sandeep/README.html.

[38] Dipl.-Inf. Michael Sobirey. Intrusion detection systems bibliography, March 1997. http://www-rnks.informatik.tu-cottbus.de/ sobirey/idsbibl.html.

[39] S. Staniford-Chen, B. Tung, and D. Schnackenberg. The common intrusion detection framework; cidf, October 1998. Position paper accepted to the Information Survivability Workshop, Orlando; http://seclab.cs.ucdavis.edu/cidf/.

[40] Jiacun Wang. *Timed Petri Net.* Kluwer Academic Publishers, 1998.

[41] CIDF working group document. A common intrusion specification language, 1998. http://gost.isi.edu/projects/crisis/cidf/.

[42] S.F. Wu, F. Wang, B.M. Vetter, R. Cleaveland, Y.F. Jou, F. Gong, and C. Sargor. Intrusion detection for link-state routing protocols. In *IEEE Symposium on Security and Privacy.* IEEE press, May 1997. http://shang.csc.ncsu.edu/pubs.html.