Global Infrastructure Protection System*

Sabrina De Capitani di Vimercati[1] **Patrick Lincoln**[2] **Livio Ricciulli**[3] **Pierangela Samarati**[4]

(1) Dip. di Elettronica per l'Automazione - Università di Brescia - 25123 Brescia, Italy

(2) Computer Science Laboratory - SRI International - Menlo Park, CA 94025, USA

(3) Metanetworks Inc. - 647 N. SantaCruz Ave, Los Gatos, CA 95032

(4) Dip. di Tecnologie dell'Informazione - Università di Milano - 20163 Crema, Italy

decapita@ing.unibs.it lincoln@csl.sri.com livio@metanetworks.org samarati@dsi.unimi.it

Abstract

The development of new classes of distributed applications such as telephony, remote video, and virtual reality introduces new quality requirements that make inadequate even the current best-effort service model provided by international networks. Recent research activity has proposed new approaches to accommodate these new application requirements. While exploiting the resource needs of different applications to use the network resources efficiently, these approaches require the maintenance of a considerable amount of state information at the network nodes. Correctness and availability of such information are the basic requirements for the proper working of the network.

In this paper we present a system, called *Global Infrastructure Protection System* (GIPS), to control improper modifications to this state information. We illustrate the GIPS's architecture and identify topology conditions to guarantee the distributed fault tolerant detection of anomalies. The system is based on the use of a hierarchical structure to organize and maintain the information at each node. Improper network states are described through rules that characterize state information updates that may result anomalous (or uncommon) with respect to the network status, past events occurred, or statistical measures. We introduce a notation to represent state information coming from heterogeneous protocols, and statistical operators to examine the history of state updates accumulated during operation. Finally, we present some examples using our notation to express heuristical rules detecting anomalous operations in a network.

*A preliminary version of this paper appeared under the title "PGRIP: PNNI Global Routing Infrastructure Protection" in *The Internet Society 1999 Network and Distributed System Security Symposium (NDSS'99)*, San Diego, CA, February, 1999.

1 Introduction

Global networks require novel security mechanisms to protect control information spanning multiple untrusted administrative domains. There are two main reasons for which it is important to guarantee proper protection of the global data structures necessary for the operation of large distributed networks. First, compromises of the control information can have catastrophic effects on the network working, thus being a good target for denial-of-service attacks. Second, user-level security mechanisms can be made less effective by compromising the underlying data transport layers. For instance, by preempting the updates of the cryptographic keys, one could force a user to use less secure, expired keys.

The design of IP networks [14] was specifically targeted to minimize the amount of state information kept into the network nodes (nodes, for short) and therefore allow for fast, robust, and dynamic reconfiguration of network resources in case of failures. In IP networks, failure or misconfiguration of a node can be easily tolerated by the network in a way completely transparent to the end-user application. This tolerance is due to the fact that nodes do not contain per-flow information, but rather characterize all traffic as being the same and simply provide best-effort data delivery. This simple approach has proven to be extremely scalable and robust and is the basis of today's Internet. It can be argued, however, that best-effort data delivery is not enough since it cannot guarantee a minimum level of quality of service necessary for certain important applications like telephony, teleconferencing, or desirable business and economic models for selling Internet access. To satisfy these requirements, the best-effort approach needs to be augmented with the capability of properly scheduling communication resources.

Quality of Service (QoS) routing and resource reservation protocols [2, 4, 12] have been designed to allow the efficient reservation of communication resources. Several such standards [6, 7, 21] are actively being pursued to provide network protocols for creating state information into the network nodes to guarantee proper level of services for end-users. The major risk in introducing state information into the network to provide individualized access to network resources is due to the possible difficulty in managing and controlling such a huge amount of critical information. Improper or inconsistent initialization or modification of this state information is likely to lead to unusable systems that would not be able to provide the promised level of service because of their intrinsic complexity and lack of scalability.

In this paper we provide a solution to this dilemma providing for the design of stateful networks that, while allowing the dynamic reconfiguration of complex data structures in the network, provides strong guarantees on the preservation of the correct global state. These guarantees assume arbitrary fault modalities and therefore also cover malicious modifications or corruption of network control information. Our proposal departs from previous proposals approaching the problem to secure the global network state with designs based on cryptography [10, 22, 26, 27, 29]. These designs typically rely on a key management infrastructure that must be as large as the network itself and that is resilient to faults. However, the key management problem has not been completely solved even for relatively small networks; and therefore, performing secure and fault tolerant key management for global networks can be considered a substantial obstacle. Another shortcoming of cryptographically secure routing protection designs is that they focus on protection against malicious faults, and do not directly address spontaneous accidental failures. In fact, most designs have no direct handling of nonmalicious faults, and leave these simpler fault tolerance issues to other network protocols. These limitations result in possible conflicts between the built-in mechanisms that reconfigure the system upon failure and the security mechanisms that overlay the routing protocols. For these reasons, other methods of achieving global network integrity are required. Our design overcomes these limitations by protecting the network infrastructure against both malicious and nonmalicious faults in a unified manner by replicating network resources and using Byzantine fault tolerant protocols to identify failures.

Another limitation of cryptographic protection comes from the fact that it is preventive: the nodes protect the integrity of the system by preventing unauthorized modification of the state through encryption. This approach, if correctly implemented, offers very strong guarantees that no integrity is lost because of malicious activities. This guarantee, however, may come at a very high price in performance for having to continuously encrypt and decrypt a large amount of redundant state information.[1] Our approach is reactive: we leave the protocols that affect the global state in the clear, thus releasing the burden of encryption, and replicate enough resources to guarantee that, if an anomaly can be detected by a heuristical rule present in the system, it will be detected shortly after it happens. In our reactive approach, once an anomaly is detected and correctly diagnosed, predefined protocols (perhaps cryptographically secure) intervene in resolving the anomaly. The advantage of a reactive approach is that it treats the common case more efficiently since the topology information propagates unencrypted, and introduces policing actions only in the rare cases when they are needed.[2]

By replicating both processing and communication resources according to rigorous and well-understood rules derived from fault tolerant distributed system theory [17], our *Global Infrastructure Protection System* (GIPS) is able to detect and, in some cases, tolerate large classes of fault or attack scenarios, either intentionally or unintentionally provoked, relying on cryptography only when absolutely necessary.

The basic assumption of our approach is that the security requirements of large networks can be met by maintaining the integrity of the collective state information. Monitoring and ultimately diagnosing and resolving inconsistencies in the state of the network can be enough to supervise the operation of heterogeneous networks without having to impose protocol-specific security requirements. Our abstractions are not targeted to secure and/or detect compromises of the network protocols but instead target the integrity of the state of the system. We argue that this higher level of abstraction is crucial to the interoperability and ultimately to the effectiveness of large-scale infrastructure protection mechanisms. Furthermore, in this paper we refine some of the ideas used in the design of the anomaly detection module of the system presented in [8].

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 gives preliminary concepts. Section 4 presents conditions that must be satisfied in the design of GIPS's architecture to provide correct identification of anomalies. Section 5 illustrates the GIPS node-level architecture. Section 6 describes the Anomaly Detection Language used to describe anomaly behaviors. Section 7 provides some examples that show how the state information managed by two specific routing protocols, namely PNNI and BGP, can be abstracted according to our model. Section 8 illustrates an example of how the Anomaly Detection Language can be used to detect anomalous behaviors within PNNI [11] and BGP [24]. Finally, Section 9 gives our conclusions.

---

[1] In most cases, to protect against reply attacks, identical information needs to be made unique through the use of time-stamps.

[2] This argument assumes a routing system to be stable and therefore not to exhibit a large number of anomalies. We do not consider unstable routing systems because they would not be viable and would not be worth protecting.

2 Related work

Most routing infrastructure protection mechanisms that have been proposed (see [10, 13, 22, 26, 27]) differ from our approach because they are preventive in that they use cryptographic services to secure the routing protocols. Hauser et al. [13], present two techniques for efficient and secure generations and processing of updates in link state routing protocols. The first technique is appropriate for stable network environments, that is, networks where links and nodes go up and down infrequently. The second one is designed for more dynamic routing environments. In summary, these techniques are based on the work in [22] and allow to reduce the cost of cryptographic protection by optimizing the authentication of the routing messages that carry redundant information though hash chains. The drawback of the proposed techniques is that they are target for a specific kind of protocols and are not suitable for the worst case scenario of arbitrary failures. Sirois et al. [26] present an approach to secure the routing system Nimrod from denial of service attacks. The countermeasures proposed for Nimrod include different sequence integrity mechanisms and a protocol for the establishment of a shared secret between two network nodes. However, these countermeasures are still vulnerable to Byzantine attacks. Smith et al. [27] analyze the security requirements of distance-vector routing protocols. The countermeasures they propose rely on the digital signature of routing updates, the addition of sequence numbers, and on the use of the predecessor information. The predecessor information is used to verify the integrity of a given update entry. The work closest to ours is the one by Wu et al. [15]. They designed an intrusion detection system that can detect anomalies in the OSPF routing protocol. In their design, properly modified SNMP agents and protocol analyzers are distributed throughout the OSPF routing hierarchy to detect anomalous behavior. Although the idea is in principle similar to our GIPS proposal, there several crucial differences. The most relevant difference is that GIPS does not analyze the protocol packets but operates at a higher abstraction level. This allows GIPS to be decoupled from the actual routing transport mechanism and therefore to be more portable. In particular, GIPS can transparently coexist with different security measures employed at the protocol level without modification. Another important differences are that Wu et al. concentrate on intrusions; they do not seem to allow the cooperation of different distributed nodes in the diagnosis process, and their design does not seem to handle spontaneous failures. Moreover, their detection module allows the real-time interception and manipulation of routing packets, thus introducing additional complexity in the verification of the routing protocols thus compromising their resilience.

A preliminary version of our work appeared in [8]. However, while our previous work was centered on the PNNI routing protocol [11], GIPS's design abstracts the underlying protocols and uses those abstractions to implement integrity protection in a protocol-independent way. We envision that this feature will be extremely useful in real commercial settings where Internet service providers may use different internal protocols. In this heterogeneous environment, global cooperation among disparate administrative domains will greatly benefit from abstractions such as the ones provided by GIPS.

3 Preliminaries

We describe the network abstraction on which our system is based and give preliminary concepts used throughout the paper.

Our system is based on a network abstraction that provides a high-level description of the network

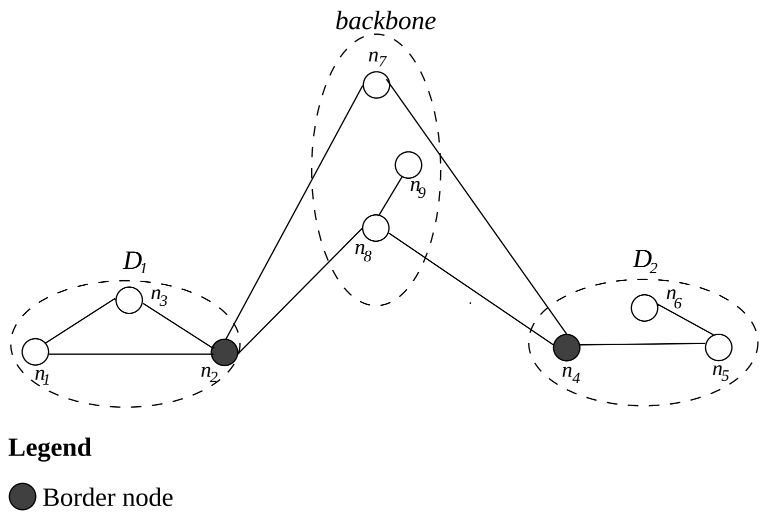

Figure 1: An example of two-level hierarchy

and state information. The use of this high-level abstraction allows us to uniformly handle security requirements of current routing protocols (e.g., PNNI [11]) and adjunct services (e.g., RSVP [2]) and provides two very important advantages. First, it allows the concurrent analysis of multiple protocols and therefore enables the correlation of anomalous behaviors from events occurring through the operations of different protocols. For instance, in the specification of an anomaly, we can combine state changes occurring in the operation of RSVP [2] with those in the underlying routing protocol thus naturally supporting an inter-protocol correlation capability. Second, the production of protocol-specific middleware to derive these abstractions enables our system to be portable and to inter-operate across different protocols, thus enabling the correlation of events across heterogenous environments. For instance, a network using RIP [18] will be able to communicate anomaly reports to a network using OSPF [19].

The network abstraction is based on a hierarchical organization of the network nodes by which a topology is broken down into several layers of hierarchy, which reduces the routing overhead while providing good scaling capabilities [16]. This hierarchical organization may be natively provided by the routing protocol standard itself (like for PNNI [11]) or can be abstracted by our system to superimpose hierarchy levels across existing nonhierarchical protocols (like for RIP [18] and BGP [24]).

In the abstraction of the network, nodes are grouped into *domains* which are then organized hierarchically. Although in very large networks several levels of the hierarchy can be defined (as in PNNI), typically, a two-level hierarchy is used (as in OSPF [19] and IS-IS [23]). At the first level the network is split into a set of independent domains connected by a "backbone"; at the second level nodes in the backbone domain are used for data transmission across different domains. Some nodes in a domain, called *border nodes*, can communicate with nodes in other domains, while others can communicate only with nodes within their domain. There must be at least one border node in each domain, connecting the domain to the "external world". Figure 1 illustrates an example of two-level hierarchy. At the lower level there are six nodes organized into domains D_1 and D_2. At the higher level there are three nodes, namely n_7, n_8, and n_9, all belonging to the backbone domain. Connection between levels is carried out through border nodes n_2 and n_4.

Each node of a domain stores state information regarding the node itself and all the nodes belonging to its domain. Requiring members of the domain to store information of all other domain members clearly implies that, to limit the amount of information stored at each node, the number of members in the domain should be relatively small. The domain information is made known to other domains through summarization and communication to the higher level.

The information stored at a node reflects the node's view of the network. Modifications to the network structure are communicated by transmission of *advertisements*, combined in packets which we call *state information packets*, grouping together information of the same type. Our concept of state information packets correspond to routing update messages in RIP and BGP, and to PNNI Topology State Elements (PTSEs) in PNNI.

State information stored at a node form the *state database* of the node. A state database consists of *simple* and *complex* properties. Simple properties contain a value from a pre-defined type, such as `integer`, `real`, and `string`. Complex properties contain simple and/or complex properties. For instance, the IP address of a node is a simple property and the routing table of a node is a complex property. Changes to the state information can be enforced via manual administrative requests or via protocol operations such as database synchronization of the exchange protocol in PNNI or QoS reservations in RSVP.

4 Conditions on the GIPS's architecture

We now discuss the conditions that our system imposes on the architecture. While requiring every domain to enforce the GIPS control, we do not require all the nodes in the domain to be enabled for that; a subset suffice. We therefore require that a subset of the nodes, called *Core Group* (CG), in each domain be equipped with GIPS. The CG nodes must include a leader, called *group leader* (GL), that aggregates and summarizes local information and make it available to other domains. The CG nodes must meet specific conditions to guarantee the possibility of detecting and resolving anomalies. We now both motivate and characterize these necessary and sufficient conditions with regard to the domains' topology, and the amount of replication of GIPS's resources. All these conditions are assumed to be recursively applied at all levels of the hierarchy.

4.1 Domain topology

Nodes belonging to the same domain share a common view of the state of the network (link states and reachability information) by periodically flooding the domain with messages that synchronize their state database. Whenever a node receives a packet that changes information in its own state database, the node automatically relays the change to all the other nodes directly connected to it (minus the node from which the change was received and nodes residing in other domains). GIPS's anomaly detection and diagnosis requirements map very naturally to this mechanism and can fully exploit this inherent feature. As we assume Byzantine faults (faults can be arbitrarily bad), faulty nodes may be able to drop, modify, or forge malicious packets that travel throughout a domain: there is no guarantee that legitimate messages traversing a domain can reach every node in the domain. To guarantee that messages flooding a domain reach every node in the domain, even in the presence of

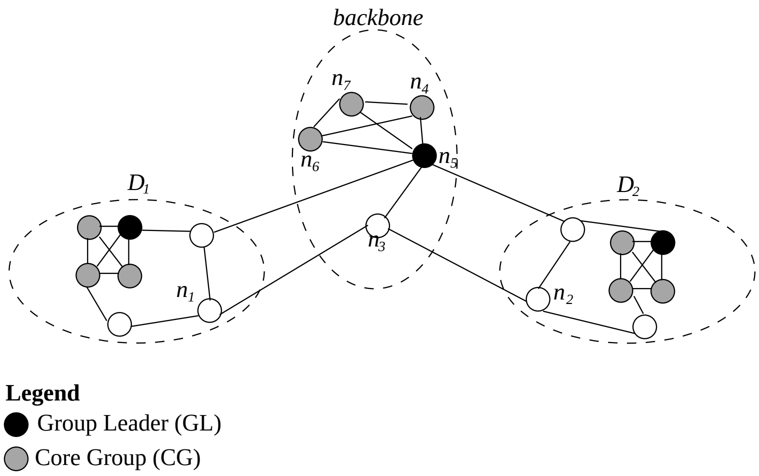

Figure 2: An example of network equipped with GIPS

multiple failures (i.e., to prevent m faulty nodes from partitioning the domain), we formulate the first condition under which GIPS must operate.

Condition 1 *Given a maximum of m simultaneous faulty nodes, domains must be $(m+1)$-connected, that is, any two nodes in any domain must have at least $m+1$ distinct paths, to communicate with each other, that do not traverse any other node twice.*

This condition, analogous to the one proposed by Perlman [22], guarantees that, even if the m faults block m routes, there exists at least one other route through which legitimate message transmissions can take place. Notice that the $(m+1)$-connected requirement only imposes a precise and rigorous amount of redundancy in the communication links, which can be easily introduced at the planning stage of the network topology. For instance, the network illustrated in Figure 2, where nodes are 2-connected, satisfies this condition for $m = 1$ at all levels of the hierarchy.

4.2 Inter-domain topology

The previous condition guarantees that a messages transmitted within a domain reaches each node in the domain. A similar guarantee must be provided for nodes belonging to different domains. Indeed, as we will see in the next sections, to correctly detect anomalies, in some cases, messages origined from nodes of a specific domain must be flooded to other domains. Due to Byzantine faults, any node (border nodes included) can drop, modify, or forge packets. For this reason, there should be enough link redundancy for nodes of one domain to flood messages to another domain.

Condition 2 *Given a maximum of m simultaneous faulty nodes, each domain must have at least $m+1$ distinct links to communicate with any other domain that have to be connected with it. These links have to start from at least $m+1$ distinct border nodes.*

This condition, combined with Condition 1, guarantees that even if m border nodes are faulty, there exists at least one route through which it is possible to flood messages to another domain. For instance, consider the network in Figure 2 and suppose that domains D_1 and D_2 have to be connected only with the backbone domain. Under this assumption, the network satisfies Condition 2 for $m = 1$, meaning that there exists at least two different paths from D_1 and D_2 to the backbone domain starting from two different boarding nodes. Condition 2 is however not sufficient to guarantee the correct transmission of messages among different domains. In fact, the group leader node of each domain has to be itself resilient to faults because, as better discussed in the remainder of the paper, GL nodes are responsible to take a final decision when an anomalous situation has been detected, and to flood alarm messages among different domains.

4.3 Byzantine agreement

By judiciously picking a subset of the nodes (CG) to independently carry out fault detection tasks, one can tolerate a failure in any node in the system, including one or more of the nodes performing the diagnosis. The theoretical results arising from the formulation of Byzantine agreement algorithms (first appearing in [17] and later refined in [28]) can be applied to provide necessary and sufficient conditions for determining how many CG nodes must be employed in each domain.

Condition 3 *To tolerate the correct diagnosis of m arbitrary (including Byzantine) faults, there must exist at least $3m + 1$ nodes, in the system, which independently perform the diagnosis.*

By employing at least $3m + 1$ nodes of the domain in performing the diagnostic functions, we ensure that the nodes will reach a consensus on the diagnosis[3] even if m or fewer CG nodes conspire in an attempt to confuse the diagnosis.

To achieve Byzantine agreement with standard algorithms, CG nodes must have additional redundancy in the connectivity among them. In fact, another well-known result from [17] translates to the following:

Condition 4 *If the $3m + 1$ CG nodes are connected through point-to-point connections, the topology of the $3m + 1$ nodes must be at least $3m$-connected.*

This condition prevents malicious CG nodes from affecting the consensus by intercepting and changing messages while the nodes perform the diagnosis agreement algorithm. For instance, the topology illustrated in Figure 2 is tolerant to one fault ($m = 1$) and satisfies both Condition 3 and Condition 4. At any level of the hierarchy, four completely connected CG nodes are responsible for reaching a consensus on fault diagnosis.

5 GIPS's node-level architecture

We describe the GIPS's node-level architecture we have designed in order to increase the preservation of the correct global state. As shown in Figure 3, the GIPS's node-level architecture is composed of a protocol-specific Middleware and four protocol-independent modules.

- The *Middleware* interfaces with different protocols to provide the necessary abstractions for the operation of the rest of the system.

[3]Note that it is also possible to reach a consensus on the impossibility of a correct diagnosis.

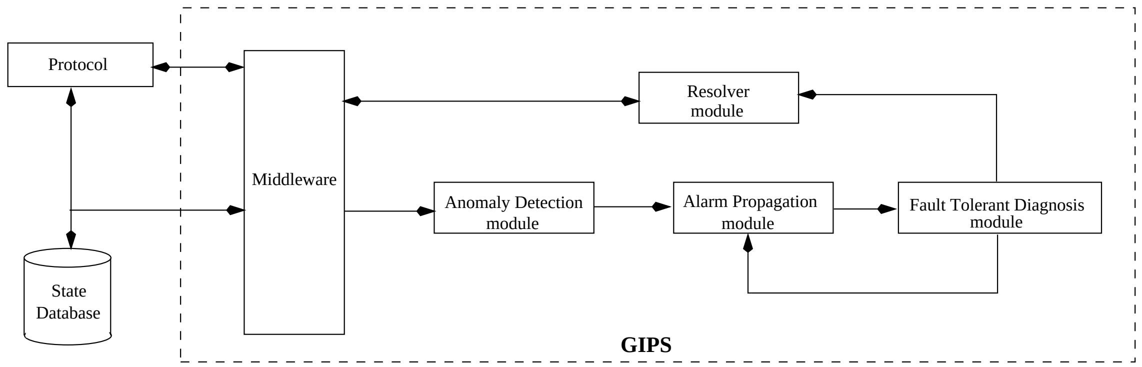

Figure 3: GIPS's node-level architecture

- The *Anomaly Detection* module monitors all changes to the state database to determine suspicious or anomalous state modifications. Based on *anomaly detection rules* (see Section 6.4), the Anomaly Detection module can generate alarms that characterize anomalous changes to the database.

- The *Alarm Propagation* module receives alarms, generated by the anomaly detection rule evaluation, from either the local or remote nodes. It propagates the alarm to its local *Fault Tolerant Diagnosis* module and then, according to the diagnosis, can further propagate the alarm to other remote nodes.

- The *Fault Tolerant Diagnosis* module uses interactive consistency agreement protocols of the kind proposed in [17, 28] to reach consensus on the actions to take. The most basic form of action is to simply log a diagnosis for further review by an operator. Other possible actions are to further propagate the alarm through the Alarm Propagation module or to feed the diagnosis to the Resolver module.

- The *Resolver* module logs results coming from the Fault Tolerant diagnosis module and in some extreme cases initiates a specialized protocol to remedy the diagnosed fault.

In the remainder of this section we briefly describe each of the main components of GIPS's node-level architecture and show how their design can be integrated into existing standards.

5.1 Middleware

The Middleware is a key component in providing inter-operation and enabling the reuse of GIPS's core technology. The Middleware will be developed to be portable to the same platforms of the corresponding routing protocols and will be fully integrated in the protocol's distributions. The main activities of the Middleware can be summarized as follows. First, the Middleware translates the node's state database to the representation we will describe in Section 6; Section 7 gives examples of how state databases can be easily encoded according to this representation. Second, updates to the state database are interpreted and propagated to the Anomaly Detection module. For instance, with respect to the hierarchy in Figure 2, suppose that node n_4 sends to its neighbor n_5 a packet including data about the link between them. Upon reception of the packet, n_5 enters the data contained in the packet in the state database, possibly overwriting existing older information (in this case the operation is interpreted as an update). Third, the Middleware interfaces to the node's transport protocols (TCP or AAL layer) to provide data transport mechanisms to both the Alarm Propagation module and the Fault Tolerant Diagnosis module. Finally, it implements re-mediation mechanisms (specific to each routing protocol) to be used by the Resolver module upon the diagnosis of a faulty node.

5.2 Anomaly Detection Module

The Anomaly Detection module monitors all changes to the state database stored at a node to detect possible anomalous updates. The evaluation is based on a set of appropriate anomaly detection rules that will be illustrated in Section 6.4. Intuitively, each of these rules characterize an anomalous state or state update and define the alarm to be raised whenever the rule evaluates to TRUE. The Anomaly Detection module can thus detect all the anomalous behaviors that have been represented in the form of rules. The limitation of this control is that it allows the discovery only of anomalous scenarios that can be described by the rules. This problem is partially solved (see next section) by exploiting the statistical knowledge accumulated during the operation of the nodes to detect database operations that are unexpected. A statistical component allows the definition of rules that capture operations that are "largely deviant" from the expected behavior. For instance, some types of denial-of-service attacks (including saturation, delay, and so on) can easily be detected by analyzing the current network performance to discover if it appears consistent or not with historical observations (see Section 8.1).

5.3 Alarm Propagation Module

As a result of applying anomaly detection heuristics, nodes can generate alarms that are propagated throughout the network infrastructure.

As discussed in Section 3, routing architectures are hierarchical and the nodes are arranged in domains that share common views of the state of the network (link states and reachability information) by constantly flooding the domain with messages that synchronize the different nodes' databases. In addition, GL nodes collect, aggregate, and built data and make it available to other domains. GIPS's alarm propagation requirements map very naturally to this hierarchical data flow organization and can fully exploit it. After the distributed diagnosis phase executed among the CG nodes (see Section 5.4), the GL uses the Alarm Propagation module to take one or more of two actions: (1) log the diagnosis locally and take no further action or (2) use higher-level binding information to flood the alarm to other domains. Action 1 should always be followed, modulo some filtering to avoid redundant log. Action 2 results in delegating higher-level CG nodes to perform a more global analysis of the anomaly. Each time the alarm goes up a level in the hierarchy, following action 2, the GL of each higher level possesses more and more global information and perhaps can correlate alarms coming from different lower-level sources to make more informed decisions. At the same time, this mechanism removes traditional bottlenecks arising from the aggregation of several alarms by a unique alarm correlation service (as it is commonly done today). Propagation of alarms upward in the hierarchy results in good distribution of the alarm correlation, transmission, and processing load.

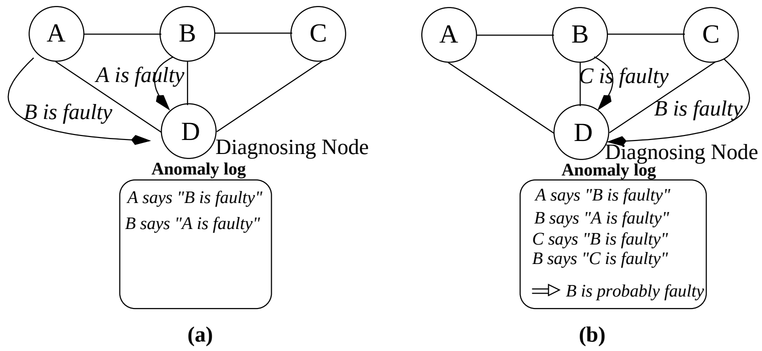

Figure 4: Diagnosis of Byzantine Fault

5.4 Fault Tolerant Diagnosis Module

The Fault Tolerant Diagnosis module should satisfy two basic requirements. First, it should be able to correctly interpret anomaly reports (alarms) so that appropriate actions can be taken in case of significant failures. In particular, it should avoid false positive and false negative diagnoses. Second, it should be itself resilient to faults. The first objective is more challenging. The second objective can be obtained by using well-known results derived from fault tolerant theory (see Section 4).

Fault diagnosis Researchers in the field of distributed network management have long been investigating techniques for performing diagnosis of network malfunctions through alarm correlation (for an overview see, for example, [25]). The rationale is that, given a set of symptoms represented by a variety of distinct alarm messages, an expert system should be able to correlate the symptoms and diagnose the underlying problem.

A major assumption existing in current alarm correlation software is that the faults are nonmalicious. Fault diagnosis in a Byzantine environment is much harder because in some situations a malicious node can be smart enough to generate anomaly reports that inhibit the correlation functions. Although the limitation of fault diagnosis in a Byzantine environment has been long recognized, recent work by Lincoln et al. [28] has demonstrated that, under reasonable fault modeling assumptions, by recording the history of anomalous events, one can construct an algorithm that converges to satisfactorily high levels of accuracy of diagnosis, even for Byzantine faults.

The idea is illustrated in Figure 4. Suppose that node A tells D that B generated an anomaly and that node B tells D that A generated an anomaly (see Figure 4(a)). Node D can detect that there is a fault but it does not have enough information to determine which node is lying. Node D records the anomaly and does not take any corrective action. Suppose now that, some time later, node C tells D that B generated the anomaly and node B makes a counter claim as before (see Figure 4(b)). This time, node D can use its history and deduce that it is unlikely that nodes A and C are both conspiring against B, and therefore heuristically declare node B faulty on the basis of multiple correlated anomaly reports. Although algorithms such as the one described in [28] could be used for improving GIPS's fault diagnosis behavior in the Byzantine environment, the fundamental limitation in diagnosis of Byzantine faults must be recognized and acknowledged in the design to accommodate the case in which anomaly reports cannot resolve the source of a fault. For this reason, the Alarm Propagation module can simply log an undecidable diagnosis without taking further action.

It is important to realize that the extent of this limitation is dependent on the accuracy of the Anomaly Detection module and its ability to correlate anomalous behavior with past events, and it is therefore not easily quantifiable. A more common and more easily solvable case (which is directly addressed by standard alarm correlation software) is that anomaly correlation can be performed on the basis of multiple concurrent anomaly reports coming simultaneously from different nodes in the network. In this case, by gathering enough global information, the diagnosis system can immediately and decisively identify faulty nodes and pass the information to the Resolver module.

5.5 Resolver Module

The Resolver module is activated in the group leader node after the core group nodes agree on a diagnosis. The Resolver module should answer, with very specific countermeasures, only those threats that are particularly severe. The Resolver module should be used very carefully because it can affect the network's operation. If misused, it can introduce instability or side effects that may be worse than the original fault.

We see the Resolver module as being capable of offering additional protocols to routing standards so that (1) state information can be verified, thus exploiting the redundancy and replication of information in the network hierarchy and (2) the CG can preempt some nodes by cutting them out of the network hierarchy until an operator can assess and remedy potential integrity compromises. Both mechanisms could be implemented, as can the rest of GIPS, without cryptography but, as we will see, their effectiveness would be greatly increased through the use of cryptographic signatures. For instance, in PNNI v2.0 and OSPF v2.0 the routers use cryptography to authenticate the routing updates, thus providing a convenient hook for this feature.

Database verification In the routing standards, nodes that have established connectivity synchronize their state databases by directly exchanging topology information. In the state database verification mechanism, the protocol used by nodes that are directly connected is used also by nodes that are indirectly connected when they suspect that a directly connected neighbor is faulty. In this mechanism, as a result of detecting a faulty node, a GL can establish a special connection to a third-party node and explicitly request topology information. This mechanism may reuse existing database synchronization protocols, and it permits an additional level of verification mechanism without much standardization. The mechanism does not require cryptography, but relies exclusively on redundancy. Cryptography, if available, could make this verification mechanism even more effective when the topology does not satisfy Condition 1 and the suspected faulty node can mount a man-in-the-middle attack and affect the state database synchronization.

Node preemption The preemption mechanism allows the GL to effectively eliminate a node from the domain databases so that it cannot adversely affect the domain. The defective node may affect the preemption process and still cause problems during this process.

As in the case of database verification, cryptographically securing the node preemption algorithm would render it much more effective. Once a node is diagnosed to be faulty, properly signed messages can be broadcast by the GL node to implement the preemption throughout the entire domain. In this case a malicious node could not affect the preemption process; thus, a resolution of the fault could be performed in a much more expeditious manner.

6 The anomaly detection language

Since we are interested in detecting suspicious or anomalous modifications on the state database, we need a flexible way to specify anomalous behaviors. To this purpose, we introduce a rule-based *Anomaly Detection Language*. Each rule characterizes modifications that appear anomalous with respect to the current network status, events occurred, or statistical measures on them, and that could therefore compromise the integrity of the state information. The specification and evaluation of anomaly detection rules require a means of addressing state information stored at a node, characterizing operations, and expressing anomalies. Their treatment is the focus of this section.

6.1 State information representation

Consider the general problem of representing and querying state information stored in a state database. Although state information may have some structure, it is in general too varied, irregular, and changeable to easily map to a fixed schema. Moreover, the structure and the type of state information used and transmitted within different protocols may differ in many aspects. As an example, consider the RIP [18] and the BGP [24] routing protocols. Both BGP and RIP nodes maintain a table, called *peer table*, containing information about the connections with BGP and RIP peers, respectively. However, the BGP peer table includes some information that the RIP peer table does not contain and vice versa. In addition, even within the same protocol, the structure of a particular kind of state information is not always fixed. Some data can be optional (e.g., in PNNI metrics and attribute on the systems directly reachable from a node are optional); other data can appear several times (e.g., in OSPF a network advertisement contains a variable number of node IDs whose cardinality depends on the number of nodes attached to the network that is being described). Finally, because some protocols are still under standardization, other kinds of state information might be added. To allow future extensibility, it is important that the chosen representation be simple enough, and yet flexible and powerful enough, to describe state information coming from heterogeneous protocols without modifying the database structure. To provide such expressiveness and flexibility, by borrowing some ideas from graph rewriting theory [9] and, more recently, semistructured data management [1, 3, 20], we abstract the state information maintained at each node by viewing it as a tree, called the *state tree*. The state tree is a labeled tree, where a vertex corresponds either to the *name* of or the *value* of a property in the state database, or the name of an additional property introduced for data structuring. Properties are represented as circle nodes and base values as square nodes. Every instance of a state tree has a root called `Node`. The root `Node` has one child for each protocol for which information is stored. The parent-child relationship of the (sub)tree rooted at a given protocol vertex reflects the structural organization of the protocol information. The correspondence between a state database and its state tree is established by the following rules.

Simple property A simple property p and its value v are represented as a tree including the nodes labeled p and v, rooted at node p. The edge between them denotes the property-value relationship.

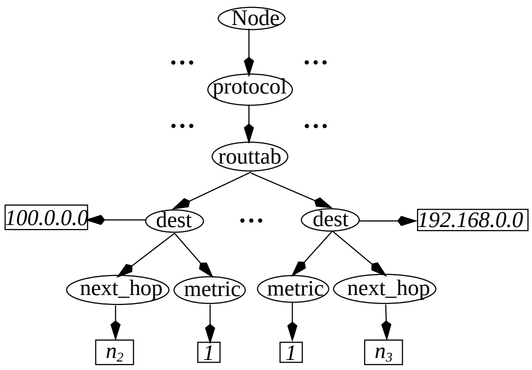

Figure 5: A simple example of state tree representing the routing table of a node

Complex property A complex property p which contains properties $p_1, \dots, p_N$ is represented as a tree rooted at node p whose children are $p_1, \dots, p_N$. Each p_i can in turn be a simple property, a complex property, or an aggregation of related properties, which we call *aggregate* property.

Aggregate property An aggregate property $(p_1, \dots, p_N)$ is represented as a tree rooted at node p_i whose children are $p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_N$. Property p_i is an identifier for the aggregate. If there is no property p_i in the set that can be used as an identifier for the aggregate, then ad-hoc identifier is used.

Each property in a state database is represented according to the above rules.

Example 1 Consider a generic routing protocol. Usually each node maintains a *routing table* stating, for each possible destination, the neighbor node to which packets for that destination should be sent. A routing table can then be seen as a complex property that includes a set of triples of the form ⟨`dest`,`next_hop`,`metric`⟩, where `dest` is the destination, `next_hop` is the next hop, and `metric` is a measure used to determine the optimal path to a destination. Such triples are aggregate (sub)properties of the routing table and therefore this table can be represented by applying the second and third rules discussed above. The routing table can then be represented as a state tree as described in Figure 5. According to this routing table, packets addressed to network `100.0.0.0` are forwarded to node n_2 and packets addressed to `192.168.0.0` are forwarded to node n_3.

For simplicity, assume that the routing table contains one and only one entry for each destination so that the destination field can be used to identify each entry of the routing table.

Given a state tree, every property/value maintained at a node can be referenced by means of the corresponding *path expression*. A path expression is a dot-separated sequence of labels of the form $\texttt{Node.label}_1\texttt{.label}_2\ldots\texttt{label}_\texttt{n}$, where `Node` is the root of the tree and $\texttt{label}_i$, $i = 1, \dots, n$, is the name of labels associated with nodes representing the path from the root `Node` to vertices labeled $\texttt{label}_\mathrm{n}$ passing trough vertices $\texttt{label}_1$, $\texttt{label}_2$, …, $\texttt{label}_{(\texttt{n}-1)}$. For instance, with reference to Figure 5, path expression `Node.protocol.routtab.dest` denotes all the destination vertices. The notion of path expression is not new and has been previously introduced in contexts where information is modeled by means of a tree structure. For instance, in the context of semistructured data, there are query languages based on path expressions [1]. In the Web context, the *XPath* language [5] uses complex path expressions for navigating through the hierarchical structure of an XML document. By adopting ideas taken from graph rewriting theory [9], we extend path expressions to contain variables. Variables can be used to represent generic vertex labels, thus increasing the expressive power of path expressions, and to retrieve from the database data matching different values, possibly bounded by some conditions. Given a path expression $\texttt{Node.label}_\texttt{1}\ldots\texttt{label}_\texttt{n}$, if a condition on the value of a simple property $\texttt{label}_\texttt{i}$ has to be specified, $\texttt{label}_\texttt{i}$ is followed by a variable between parenthesis. A condition has the form "$X\,op\;v$", where $op \in \{=,\neq,>,<,\geq,\leq\}$ X is a variable, and v is a basic value. In addition to variables, also basic values can be specified within parenthesis; they permit to refer to specific properties. In the remainder of this paper, we use uppercase single letters to denote variables.

Example 2 Consider the state tree in Figure 5. Path expression `Node.protocol.routtab.dest.X` denotes the set of all `next_hop` and `metric` vertices of every destination `dest`. Path expression `Node.protocol.routtab.dest(192.168.0.0).metric` denotes the `metric` property associated with destination `192.168.0.0`. Finally, path expression `Node.protocol.routtab.dest(X).next_hop` with associated condition "`X = 100.167.0.0`" denotes the `next_hop` vertices of destination `100.167.0.0`.

Address patterns can appear instead of specific addresses in conditions. Address patterns are specified by using the wild card character $*$ and identify a set of addresses. For instance, `100.`$*$ denotes all the addresses with `100` as a prefix like `100.0.0.0` and `100.120.10.8`. In the following, we use the term *conditional path expression* to denote a pair $(path\_exp, C)$, where *path_exp* is a path expression and C is an optional condition bounded to the variables in *path_exp*. For instance, `(Node.protocol.routtab.dest(X).next_hop, X = 100.*)` is a conditional path expression denoting the `next_hop` property associated with all destinations matching the pattern `100.*`.

6.2 Update operations and recording

We now describe how state information is modified. In general, a change in a state database associated with a node occurs when the node receives an advertisement from its neighbors. For instance, in a distance vector routing protocol, a node will periodically receive link updates from its neighbors. An example is an advertisement communicating a change in the cost associated with a link. Upon reception of it, a node will need to recompute its distance from the destination. Evaluation of modifications at the level of whole advertisement, however, does not seem to be sufficient to enforce meaningful controls. Hence, we consider information at a finer, more semantically meaningful, granularity level. In particular, we consider state information changes referred to properties recorded in the state database. In graphical terms, changes to the state database can occur on any value node in a state tree. The three basic change operations are the following:

Add The operation `Add(`*path_exp*`)` adds property *path_exp* to the state database. For instance, operation `Add(Node.protocol.routtab.dest(149.150.0.0))` states that destination `149.150.0.0` is added in the routing table.

Update The operation `Upd(`*path_exp*`)` updates the value of property *path_exp*. For instance, operation `Upd(Node.protocol.routtab.dest(100.0.0.0).metric)` states that the metric associated with the destination network `100.0.0.0` is being modified.

Delete The operation `Del(`*path_exp*`)` removes property *path_exp* from the state database. For instance, operation `Del(Node.protocol.routtab.dest(192.168.0.0))` removes the information related to destination network `192.168.0.0` (i.e., it removes the whole subtree rooted at the corresponding `dest` vertex).

In accordance with the given protocol, our abstraction is able to consider operations as executed at different levels of granularity, simple as well as complex properties complex properties. However, it is important to note that an update operation on a complex property *cp* is always split into a set of update operations on the involved simple sub-properties of *cp*. This splitting is necessary because, as we will see in Section 6.4.1, we need to know what properties have been actually changed. In the remainder of this paper we often use the term *event* to denote the occurrence of an operation on properties identified by a conditional path expression, denoted `op(cond_path_exp)`. For instance, `Upd(Node.protocol.routtab.dest(Y).metric, Y=100.*)` denotes all the update operations executed on property `metric` associated with destination networks whose address starts with `100`. An example of update operation matching this event is `Upd(Node.protocol.routtab.dest(100.0.0.0).metric)`.

To allow the evaluation of anomaly conditions based on operations previously executed, or on the previous status of the network, each node maintains a *history log* recording all operations executed. Intuitively, the history log represents a sequence of modifications. Each log entry is a 4-tuple of the form ⟨`time`,`event`,`bf_value`,`af_value`⟩, where `time` is a timestamp, `event` = `op(cond_path_exp)` is the operation performed at time `time`, `bf_value` and `af_value` denote the values of the properties on which the operation `op` is executed *before* and *after* the operation, respectively. Fields, `bf_value` and `af_value` correspond to sets of pairs (*path_exp*,*value*), where *path_exp* is an optional path expression representing a simple property and *value* is the value of *path_exp*. These path expressions are "relative" and are considered as starting from the node identified by the path expression appearing in the `event` field. Note that the `bf_value` (`af_value` resp.) field is null in case of an insert (delete resp.) operation.

Example 3 Log entry ⟨5Jan01,`Upd(Node.protocol.routtab.dest(192.168.0.0))`,{`(metric,1)`, `(next_hop,`n_3`)`},{`(metric,2)`,`(next_hop,`n_4`)`}⟩ states that at time `5Jan01` operation `Upd(Node.protocol.routtab.dest(192.168.0.0))` was executed that updated `metric` from value 1 to value 2 and property `next_hop` from value n_3 to value n_4.

To avoid the history log to grow indefinitely, a log maintenance policy can be enforced that eliminates old entries that are not needed anymore or all entries when a maximum size of the log is specified [].

6.3 Basic operators of the language

Our language provides some basic operators that allow us to refer to summaries, aggregates, and statistical measures derived from the recorded history. Operators fall into three classes of measures: *count*, *time*, and *aggregation*.

Count measures in this class count the number of occurrences of events and include the following operators:

- `count(`*result,event,time_int*`)` returns the number *result* of events *event* executed in time interval *time_int*.

- `countcond(`*result,event,cond,time_int*`)` returns the number *result* of events *event* that satisfy condition *cond* executed in time interval *time_int*. Condition *cond* may include functions `bf_val(`*cond_path_exp*`)` and `af_val(`*cond_path_exp*`)`. These functions return the *cond_path_exp*'s value stored in the `bf_value` and `af_value` field (if any), respectively, of the history entry with an operation field matching the *event*.

Time measures in this class keep track of the time interval between two events. They include the following operators:

- `avgtime(`*result,path_exp,time_int*`)` returns the average time interval *result* between any two requests on property *path_exp* in time interval *time_int*.
- `timeint(`*result*, $event_1$, $event_2$, *time_int*`)` returns the time interval *result* between the last two events $event_1$ and $event_2$ occurring within time interval *time_int*.

Aggregation measures in this class combine a set of past events into useful abstractions to be used to decide whether the effect of an update operation appears consistent with historical observations.

- `freq(`*result,event,time_int*`)` returns the frequency value *result* with which a given *event* has occurred over a given period of time *time_int* (e.g., number of updates originated by a node per unit time in the last hour).
- `avgval(`*result,path_exp,time_int*`)` returns the average value *result* of the simple property *path_exp* over the time interval *time_int* (e.g., the average value of the bandwidth on a link).
- `max(`*result,path_exp,time_int*`)` returns the maximum value *result* assigned to the simple property *path_exp* over the time interval *time_int* (e.g., the maximum value of the bandwidth on a link).
- `min(`*result,path_exp,time_int*`)` returns the minimum value *result* assigned to the simple property *path_exp* over the time interval *time_int* (e.g., the minimum value of the bandwidth on a link).

We adopt a time constant *now* that keeps track of the current time. For instance, operator `countcond(`X, `Upd(Node.protocol.routtable.dest)`,`bf_val(metric)` $\neq$ `af_val(metric)`,[0,*now*]`)` returns the number X of update operations executed on property `metric` of any destination `dest` from time 0 to the current time.

6.4 Anomaly detection rules

An anomaly detection rule is composed of four fields.

Event It is a description of the event on the state database. It consists of an event and an optional list of path expressions. These expressions refer to the state database that would result *after* the operation is executed.

<table>
<tr><td>⟨rule_definition⟩</td><td>::=</td><td>⟨event_field⟩ ⟨state_field⟩ ⟨condition_field⟩ ⟨alarm_field⟩</td></tr>
<tr><td>⟨event_field⟩</td><td>::=</td><td>**event:** ⟨event⟩ [,⟨path_exp_list⟩]</td></tr>
<tr><td>⟨event⟩</td><td>::=</td><td>**Add(**⟨cond_path_exp⟩**)** | **Upd(**⟨cond_path_exp⟩**)** | **Del(**⟨cond_path_exp⟩**)**</td></tr>
<tr><td>⟨state_field⟩</td><td>::=</td><td>**state:** ⟨complex_state⟩</td></tr>
<tr><td>⟨complex_state⟩</td><td>::=</td><td>⟨simple_state⟩ | ⟨simple_state⟩,⟨complex_state⟩</td></tr>
<tr><td>⟨simple_state⟩</td><td>::=</td><td>⟨path_exp⟩ | **avgval(**⟨variable⟩,⟨path_exp⟩,⟨time_int⟩**)** | **freq(**⟨variable⟩,⟨event⟩,⟨time_int⟩**)** |
avgtime(⟨variable⟩,⟨path_exp⟩,⟨time_int⟩**)** | **timeint(**⟨variable⟩,⟨event⟩,⟨event⟩,⟨time_int⟩**)** |
min(⟨variable⟩,⟨path_exp⟩,⟨time_int⟩**)** | **max(**⟨variable⟩,⟨path_exp⟩,⟨time_int⟩**)**
count(⟨variable⟩,⟨event⟩,⟨time_int⟩**)** | **countcond(**⟨variable⟩,⟨event⟩,⟨complex_cond⟩,⟨time_int⟩**)**</td></tr>
<tr><td>⟨condition_field⟩</td><td>::=</td><td>**condition:** ⟨complex_cond⟩</td></tr>
<tr><td>⟨complex_cond⟩</td><td>::=</td><td>⟨simplex_cond⟩ | ⟨complex_cond⟩ ⟨bool_op⟩ ⟨complex_cond⟩</td></tr>
<tr><td>⟨simplex_cond⟩</td><td>::=</td><td>⟨variable⟩ ⟨operator⟩ ⟨variable⟩ | ⟨path_exp⟩ ⟨operator⟩ ⟨string⟩ |
⟨path_exp⟩ ⟨operator⟩ ⟨number⟩ | **bf_val(**cond_path_exp**)** | **af_val(**cond_path_exp**)**</td></tr>
<tr><td>⟨alarm_field⟩</td><td>::=</td><td>**alarm:** ⟨string⟩</td></tr>
<tr><td>⟨path_exp_list⟩</td><td>::=</td><td>⟨path_exp⟩ | ⟨path_exp⟩,⟨path_exp_list⟩</td></tr>
<tr><td>⟨cond_path_exp⟩</td><td>::=</td><td>[(]⟨path_exp⟩[, ⟨complex_cond⟩] [)]</td></tr>
<tr><td>⟨path_exp⟩</td><td>::=</td><td>⟨label⟩[(⟨var_list⟩)] | ⟨label⟩[(⟨var_list⟩)].⟨path_exp⟩</td></tr>
<tr><td>⟨bool_op⟩</td><td>::=</td><td>∧ | ∨</td></tr>
<tr><td>⟨operator⟩</td><td>::=</td><td>> | < | ≥ | ≤ | = | ≠</td></tr>
<tr><td>⟨time_int⟩</td><td>::=</td><td>[⟨variable⟩|⟨number⟩,⟨variable⟩|⟨number⟩]</td></tr>
<tr><td>⟨var_list⟩</td><td>::=</td><td>⟨variable⟩ | ⟨variable⟩,⟨var_list⟩</td></tr>
<tr><td>⟨variable⟩</td><td>::=</td><td>⟨string⟩</td></tr>
<tr><td>⟨label⟩</td><td>::=</td><td>⟨string⟩ | ⟨number⟩</td></tr>
<tr><td>⟨string⟩</td><td>::=</td><td>⟨symbol⟩ | ⟨string⟩ ⟨symbol⟩</td></tr>
<tr><td>⟨symbol⟩</td><td>::=</td><td>⟨letter⟩ | ⟨otherchar⟩</td></tr>
<tr><td>⟨number⟩</td><td>::=</td><td>⟨digit⟩ | ⟨number⟩ ⟨digit⟩</td></tr>
<tr><td>⟨digit⟩</td><td>::=</td><td>**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**</td></tr>
<tr><td>⟨letter⟩</td><td>::=</td><td>**A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** |
Y | **Z** | **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**</td></tr>
<tr><td>⟨otherchar⟩</td><td>::=</td><td>**:** | **-** | **.** | **/** | ***** | **&**</td></tr>
</table>

Figure 6: Syntax of the language to express rules

State It is a description of the state of the database *before* the event is executed. This field may contain statistical measures on past events and path expressions referring to the state database before the execution of the event.

Condition It is a logical expression of simple conditions of the form "X *math_op* Y", where *math_op* is a mathematical operation in $\{=, \neq, \leq, <, \geq, >\}$ and X and Y are variables bounded in both the event and the state fields. A condition always evaluates to either TRUE or FALSE.

Alarm It is a unique identifier for a type of anomaly to be raised whenever the condition evaluates to TRUE.

Figure 6 shows the BNF syntax of the language to express rules. Bold face strings represent terminal symbols. The main characteristic of these rules is the possibility to specify path expressions referring to the state of the database both before and after an operation is executed, thus allowing the evaluation of the effects of an operation on the database. Anomalous situations controlled by these rules can be divided into three main classes:

Algorithm 1 *Rule-matching algorithm*

MAIN
/* Input: An event $\mathtt{op}(cp)$, a set of rules $\mathcal{R}$, and a state tree $\mathcal{T}$ */
For $r \in \mathcal{R}$ **do**
  Let r:
    event: $\mathtt{op}'(cp')\ p_{e1}, \ldots, p_{en}$
    state: $p_{s1}, \ldots, p_{sm}$
    condition: *cond*
    alarm: A
  If $\exists\theta$ s.t. $\mathtt{op}'(cp'\theta)$ matches $\mathtt{op}(cp)$ **then**
    For $p_{ej} \in \{p_{e1}, \ldots, p_{en}\}$ **do**
      Let θ_i be the substitution obtained evaluating $p_{ej}\theta$ over $\mathtt{new}(\mathcal{T})$
      $\theta = \theta \cup \theta_i$
    For $p_{sk} \in \{p_{s1}, \ldots, p_{sm}\}$ **do**
      Let θ_i be the substitution obtained evaluating $p_{sj}\theta$ over $\mathtt{old}(\mathcal{T})$
      $\theta = \theta \cup \theta_i$
    If $cond\theta$ evaluates to TRUE **then** raise A

Figure 7: Rule-matching algorithm

- *Unusual modifications* Some database entries should be modified very rarely. Their modification is considered anomalous and require further investigation. For instance, a change to the status (e.g., cost) of a link may be considered suspicious.
- *Suspicious status modifications* Other anomalies may correspond to improbable (sequence of) updates. Anomalous situation can be, for instance, the transmission of "obsolete" information (i.e., information with very old timestamp). If obsolete information is accepted and transmitted, then a node could take incorrect decisions. On the other hand, simply discarding the information may cause a denial of service.
- *Consistency of the state tree* Rules can also be used to verify the consistency of the information stored in a state database. For instance, with respect to the PNNI protocol, the maximum cell rate of a node must always be greater than or equal to the corresponding available cell rate; an update not satisfying this constraint is to be considered suspicious.

More complex anomaly detection rules will be illustrate in Section 8.

6.4.1 Rule-matching algorithm

The anomaly detection rules state when the execution of an event is considered anomalous with respect to the current state database and/or past events occurred. Intuitively, the evaluation of all the rules for a given event produces an alarm whenever the event satisfies some rules. Figure 7 illustrates the **Rule-matching** algorithm enforcing the evaluation of a set $\mathcal{R}$ of rules against an event $\mathtt{op}(cp)$ performed on a state tree $\mathcal{T}$. The primary task of **main** is to check if a given rule $r \in \mathcal{R}$ is *applicable* to the considered event $\mathtt{op}(cp)$. Intuitively, rule r is applicable to an event when the event specified in the rule matches the considered event, that is, they are defined on the same kind of operation and on the same object. More precisely, let $\mathtt{op}'(cp')$ be the event specified in the event field of r. Then, we say that r is applicable to $\mathtt{op}(cp)$ if and only if $\mathtt{op} = \mathtt{op}'$ and there exists a substitution θ of variables to values such that $cp'\theta$ matches cp. Note that for a given event and rule, there may exist more than one substitution θ. This may happen when cp' is a sub-property of cp and $\mathcal{T}$ includes more than one instance of cp'. Therefore, in the general case, θ is a multiset because it may include more than one binding for the same variable. For instance, suppose that the event field of rule r is `Del(Node.protocol.routtab.dest(X).metric(Y))` and that destination `100.0.0.0` has associated two different metrics whose values are: 1 and 2.[4] Moreover, suppose that $\mathtt{op}(cp) =$ `Del(Node.protocol.routtab.dest(100.0.0.0))`. By substitution $\theta = \{\mathtt{X} := \mathtt{100.0.0.0}, \mathtt{Y} := \mathtt{1}, \mathtt{Y} := \mathtt{2}\}$, rule r is applicable to $\mathtt{op}(cp)$. If for all variables in cp' there exists at least one binding in θ for which cp' matches cp, then the algorithm proceeds with the evaluation of all the path expressions and operators possibly listed in the event and state fields of r against the state tree after ($\mathtt{new}(\mathcal{T})$) and before ($\mathtt{old}(\mathcal{T})$) the execution of the considered event. Due to this evaluation, θ will include a binding for all variables appearing in rule r. If $cond\theta$ evaluates to TRUE, that is, if for all variables in r there exists at least one binding for which the condition evaluates to TRUE, then the alarm A specified in the alarm field of r is raised.

Example 4 Consider the following rule controlling updates to property `metric` associated with destination nodes `dest` (see Figure 5).

event:	`Upd(Node.protocol.routtab.dest(X))`, `Node.protocol.routtab.dest(X).metric(A)`
state:	`Node.protocol.routtab.dest(X).metric(B)`
condition:	$\mathtt{A} \neq \mathtt{B}$
alarm:	`has_changed`

Consider the event `Upd(Node.protocol.routtab.dest(100.0.0.0).metric)` and suppose that the value of the `metric` property of destination `100.0.0.0` is 5 before the execution and 7 after the execution of the corresponding operation. The evaluation proceeds as follows.

1. Due to substitution $\theta = \{\mathtt{X} := \mathtt{100.0.0.0}\}$, the given rule is applicable to the received event `Upd(Node.protocol.routtab.dest(100.0.0.0))`.

2. Starting from $\theta = \{\mathtt{X} := \mathtt{100.0.0.0}\}$, a mapping for variables A and B is determined. The substitution θ becomes $\theta = \{\mathtt{X} := \mathtt{100.0.0.0}, \mathtt{A} := \mathtt{2}, \mathtt{B} := \mathtt{1}\}$.

3. Condition $\mathtt{A} \neq \mathtt{B}$ is evaluated with respect to θ. The condition evaluates to TRUE, and the alarm `has_change` is raised.

7 Examples of state database encoding

We now to illustrate examples of the state database encoding process performed by the Middleware, we consider two routing protocols: PNNI [11] and BGP [24], respectively. We outline the main concepts of these protocols and show how the state information maintained at each PNNI and BGP node can be represented.

7.1 PNNI basics

The Private Network-Network Interface (PNNI) is based on the link-state routing technique. The PNNI protocol uses a hierarchical organization like the one described in Section 3. Nodes are organized

[4] Note that this assumption is only for the sake of this example; it does not correspond to any real situation.

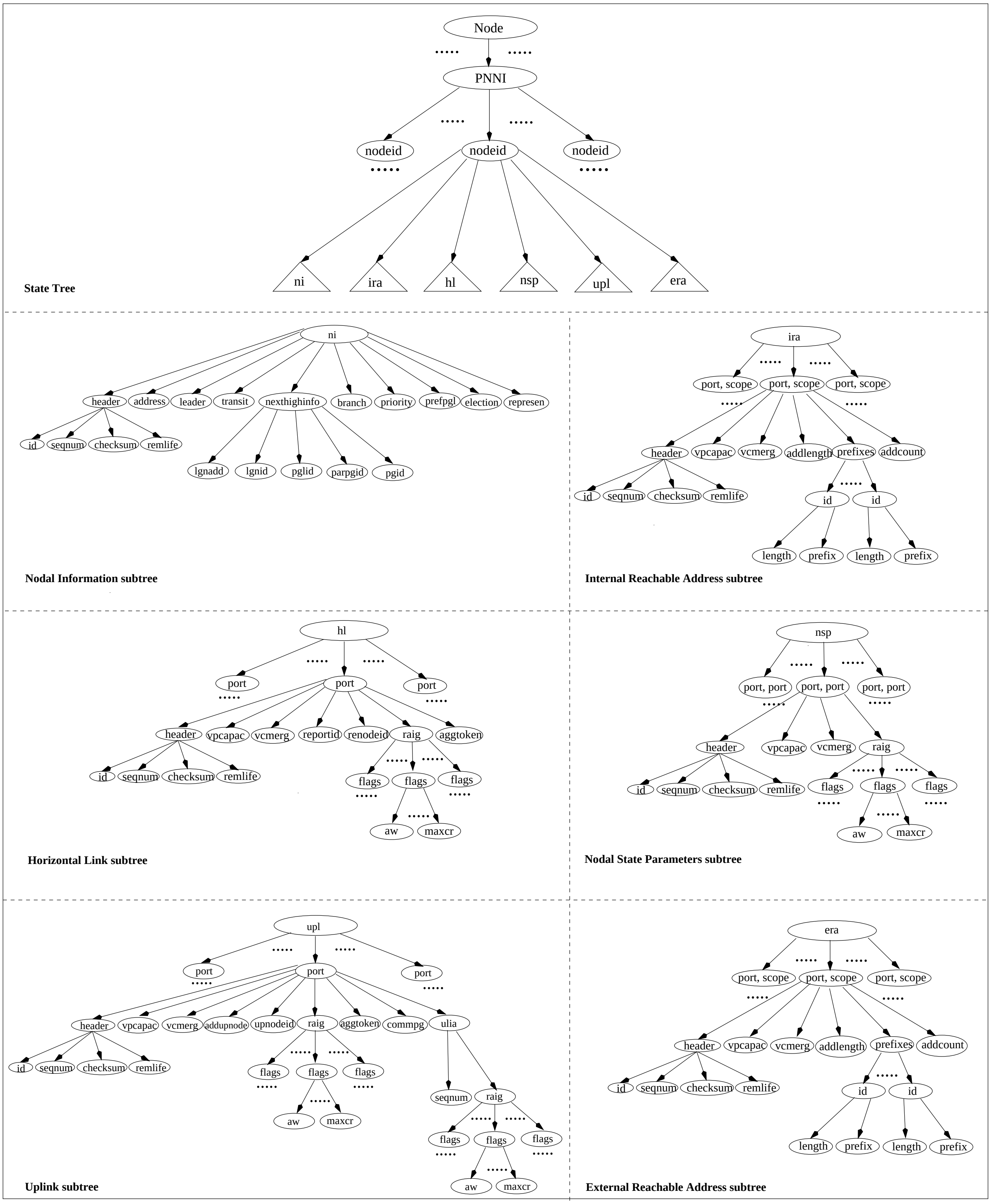

Figure 8: Graphical representation of the PNNI topology database

into peer groups, each having a unique peer group identifier (PGID). Each group has a leader (PGL) that abstracts and represents the group at the next higher level of the hierarchy. The protocol distinguishes different types of links connecting nodes: *horizontal links*, which connect nodes within the same peer group; *outside links*, which connect nodes (called border nodes) belonging to different peer groups; and *uplinks*, which are derived links that connect a border node to a node of a different peer group. Each node maintains a *topology database* containing information regarding the node itself, all the nodes belonging to its peer group, and all ancestor nodes. The set of data referred to each of such nodes is organized into *information groups* (IGs) as follows.

- *Nodal Information* (NI) describes the state of a node. It includes address, priority information, and administrative flags regulating the participation of the nodes in routing and election protocols.
- *Nodal State Parameter* (NSP) defines the characteristics of the node. It is used when the node represents its peer group at the next higher level of the hierarchy. It includes all metrics and attributes for the given input-output port pair.
- *Horizontal Link* (HL) contains information on the connections between nodes. For each link, it includes node ID and port ID of the nodes it connects, and all metrics and attributes associated with the link.
- *Uplink* (UPL) contains information on uplinks. For each uplink, it includes node ID and port ID of the border node, and all metrics and attributes associated with both directions of the link.
- *Internal and External Reachable ATM Address* (IRA and ERA) contain information on the end systems directly reachable from the node.

Metrics and attributes in the information groups described above are also organized into (sub)information groups, called *Resource Availability Information Groups* (RAIGs). Examples of metrics are the data transmission delay and the administrative weights to be used in routing decisions. Examples of attributes are the maximum and available data transmission. The information groups stored at a node reflect the node's view of the network. According to our abstraction, modifications to the network structure are communicated by transmission of collections of IGs, called PNNI Topology State Elements (PTSEs), grouping together information groups of the same type. A node's topology database consists of a collection of all PTSEs received. Every PTSE contains a header which consists of a PTSE *identifier*; a PTSE *sequence number*, used to determine which instance of a same PTSE is more recent; a PTSE *checksum*, used to verify if a PTSE has been corrupted; and a PTSE *remaining life* that is the number of seconds before the PTSE will be considered invalid. PTSEs are transmitted to nodes by means of the *flooding* and *database synchronization* protocols. The flooding protocol is a reliable transmission protocol by which packets (PTSEs) are transmitted between nodes. The *database synchronization* protocol uses flooding to efficiently exchange topology information between directly connected nodes in the same peer group. In addition to manual administrative changes, information stored at each node can be modified by other PNNI protocols, like the *Hello* protocol, by which nodes establish and control connectivity, and the *PGL election* protocol, by which nodes of the same peer group establish their leader.

Information describing a node's local topology may therefore be dispersed in different PTSEs. Following the design rules illustrated in Section 6, we provide an abstraction of the topology database that collects IGs referring to the same node. Figure 8 shows a possible representation of the information stored in a topology database. For readability, the subtrees corresponding to the different information groups are represented separately and leaf vertices containing the values of IG's properties are not reported. Each vertex corresponding to a node has six children, one for each information group referred to it. Some vertices are needed to univocally identify entities (such as nodes or links) to which information is referred. In particular, as we can seen from Figure 8, Nodal State Parameter IGs are identified by the pair ⟨input port, output port⟩; Internal and External Reachable ATM Address IGs are identified by the pair ⟨port,scope[5]⟩; Horizontal Link IGs and Uplink IGs are identified by their ports; Resource Availability IGs are identified by the flags indicating the service categories to which the metrics and attributes apply. Since multiple instances of a same PTSE and therefore of a same information group may exist simultaneously, each information group has also associated the header of the PTSE used to transmit the information group. Such header forms the subtree rooted at node *header*. Note that the proposed representation is only an example of how the information managed by this protocol can be abstracted. Due to the flexibility of the model, other representations can be chosen.

7.2 BGP basics

Border Gateway Protocol Version 4 (BGP-4) is a distance-vector protocol. Each BGP node maintains a routing table with one entry for every possible destination in the system. BGP nodes within the same domain (autonomous system, in the BGP terminology) communicate with one another to ensure that they have a consistent view of the domain and to determine which BGP node within that domain will serve as the connection point to or from others domains. The initial data exchange between two nodes is the entire BGP routing table. Incremental updates are sent as the routing tables change. The BGP metric stored in the routing table is an arbitrary unit number specifying the degree of preference of a particular path. These metrics are typically assigned by the network administrator and depend on a number of criteria, such as domain count (paths with a smaller domain count are generally better than paths with a larger domain count), or type of link. BGP nodes communicate with each other by means of four different types of messages: *open* messages, which are the first messages that two adjacent nodes sent out after that a transport protocol connection is established between them; *update* messages provide routing updates to other BGP systems; *notification* messages, which are sent when an error condition has been detected; and *keepalive* messages, which frequently are sent to ensure the liveness of the connection.

The state information within a BGP node includes information on the supported BGP protocol version, the local autonomous system number, and the BGP identifier of the node (i.e., the IP address assigned to that node). In addition, each BGP node maintains the following three tables:

- *BGP Peer* It includes one entry per BGP node. Each entry contains information about the connection with a BGP node and is composed of twenty four fields including the BGP peer connection state, the local IP address, and the remote IP address.

[5]The scope defines the highest level of the hierarchy at which the address will be visible.

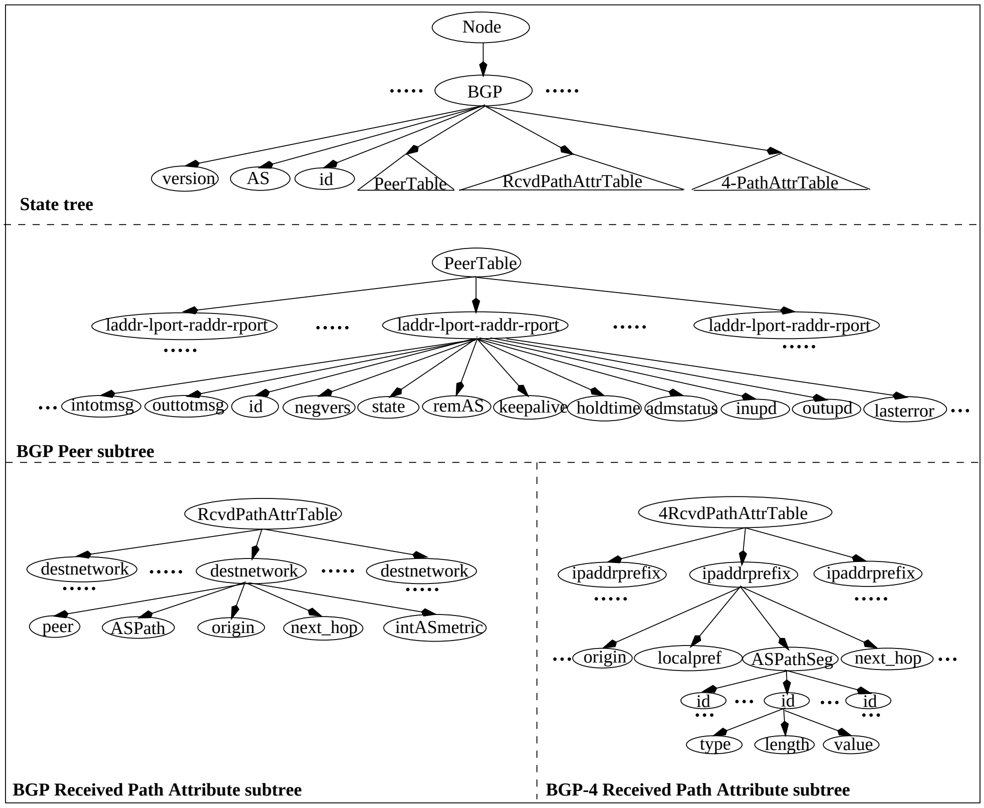

Figure 9: Graphical representation of the BGP state database

- *BGP Received Path Attribute* It includes an entry per path to a network. Each entry contains the path attributes received from all nodes running BGP version 3 or less and is composed of six fields: the IP address of the node where the path information was received; the address of the destination network; the ultimate origin of the path information; the set of systems that must be traversed to reach the network; the address of the border node; and optional metrics.

- *BGP-4 Received Path Attribute* It includes on entry per path to a network. Each entry contains path attributes received from all nodes running BGP version 4 and is composed of fourteen fields, among which the IP address of the node where the path information was received and the address of the border node.

Figure 9 illustrates the graphical representation of the BGP state database. Like for the PNNI example, the subtrees corresponding to the different tables are represented separately and value nodes are not reported. The roots of these subtrees have as many children as the entries in the tables. Every entry has one child for each property in the table. Note that, for simplicity, Figure 9 reports only some of the BGP Peer table and BGP-4 Received Path Attribute table attributes. An entry in the Peer Table is identified by the 4-tuple ⟨local addr,local port,remote addr,remote port⟩; an entry in the BGP Received Path Attribute is identified by the address of the destination network; and an entry in the BGP-4 Received Path Attribute is identified by the IP address prefix of the destination network.

Rule 1	**event:**	`Upd`(Node.PNNI.nodeid(X).ni), Node.PNNI.nodeid(X).ni.priority(P), Node.PNNI.nodeid(X).ni.leader(L)
	state:	Node.PNNI.nodeid(X).ni.priority(O), Node.PNNI.nodeid(X).ni.leader(M)
	condition:	$P \neq O \wedge L = M$
Rule 2	**event:**	`Upd`(Node.PNNI.nodeid(X).hl(Y)), Node.PNNI.nodeid(X).hl(Y).raig(Z).aw(A)
	state:	Node.PNNI.nodeid(X).hl(Y).raig(Z).aw(B)
	condition:	$A \neq B$
Rule 3	**event:**	`Upd`(Node.PNNI.nodeid(X).Y(Z)), Node.PNNI.nodeid(X).Y(Z).raig(F).avcr(C)
	state:	`avg`(V,Node.PNNI.nodeid(X).Y(Z).raig(F).avcr(W),[0,now])
	condition:	$C < V \vee C > V$
Rule 4	**event:**	`Upd`(Node.PNNI.nodeid(X).Y)
	state:	`timeint`(V,`Upd`(Node.PNNI.nodeid(X).W),`Upd`(Node.PNNI.nodeid(X).Y),[0,now])
	condition:	V < threshold
Rule 5	**event:**	`Add`(Node.PNNI.nodeid(X).hl(Y))
	state:	`timeint`(V,`Del`(Node.PNNI.nodeid(X).hl(Y)),`Add`(Node.PNNI.nodeid(X).hl(Y)),[0,now])
	condition:	V < threshold
Rule 6	**event:**	`Del`(Node.PNNI.nodeid(X).hl(Y))
	state:	`timeint`(V,`Add`(Node.PNNI.nodeid(X).hl(Y)),`Del`(Node.PNNI.nodeid(X).hl(Y)),[0,now])
	condition:	V < threshold
Rule 7	**event:**	`Upd`(Node.PNNI.nodeid(X).ni(Y)), Node.PNNI.nodeid(X).ni(Y).address(A)
	state:	Node.PNNI.nodeid(X).ni(Y).address(B)
	condition:	$A \neq B$
Rule 8	**event:**	`Upd`(Node.PNNI.nodeid(X).nsp(Y,Z)), Node.PNNI.nodeid(X).nsp(Y,Z).raig(F).avcr(A), Node.PNNI.nodeid(X).nsp(Y,Z).raig(F).maxcr(M)
	state:	
	condition:	$A > M$
Rule 9	**event:**	`Upd`(Node.PNNI.nodeid(X).hl(Q)), Node.PNNI.nodeid(X).hl(Q).raig(W).avcr(C) (R, X: remote nodes)
	state:	Node.PNNI.nodeid(X).nsp(P,Q).raig(W).maxcr(M), Node.PNNI.nodeid(X).hl(P).renodeid(R), Node.PNNI.nodeid(X).hl(P).reportid(O), Node.PNNI.nodeid(R).hl(O).raig(W).avcr(A)
	condition:	$C > M \vee A > M$ (O, P: ports connecting R and X)
Rule 10	**event:**	`Upd`(Node.PNNI.nodeid(X).hl(Z)), Node.PNNI.nodeid(X).hl(Z).raig(F).maxcr(C), Node.PNNI.nodeid(X).hl(Z).renodeid(R), Node.PNNI.nodeid(X).hl(Z).reportid(P)
	state:	Node.PNNI.nodeid(R).hl(P).raig(F).maxcr(M)
	condition:	$C \neq M$

Figure 10: Example of rules in PNNI

8 Dealing with heterogeneous protocols

One of the most interesting features of our system is the ability to correlate anomalous behaviors coming from heterogeneous protocols. In a large network, it is very unlikely that a single routing protocol will be used for the entire network. To show examples of anomaly correlation, we consider the simple two-level network illustrated in Figure 2 and assume that within domain D_1 the BGP routing protocol [24] is used and within domain D_2 and backbone domain the PNNI protocol [11][6] is used. We first give examples of anomaly detection rules for the two protocols and then describe an alarm processing scenario combing them.

8.1 Example of anomaly detection rules in PNNI

We present some examples of anomalous states or operations that can be detected by means of the evaluation of appropriate rules. Figure 10 reports these rules, where, for space reasons, the alarm field is not specified.

[6] Note that in general, to exchange routing information between networks that use different routing protocols, multiple routing protocols have to be configured in a single border node.

Unusual modifications

Rule 1 The change of the leadership priority of a node must be considered suspected if it is not accompanied by a corresponding change of the leadership status (leader to non-leader or vice versa).

Rule 2 The weight associated with a link is an example of relatively static information, which affects the PNNI routing decisions. Rule 2 then raises an alarm whenever the weight associated with a link is modified.

Suspicious status modifications

Rule 3 An anomalous situation can be a sharp increase or decrease in the bandwidth associated with a link. In particular, a sharp increase in the bandwidth can indicate malicious transmissions, such as traffic intended to cause *denials of service*. Indeed, an intruder may be able to consume all the available bandwidth on a network by generating a large number of packets directed to the network. Rule 3 raises an alarm whenever the available cell rate of a RAIG differs considerably from the mean values of the rates observed until now.

Rule 4 PNNI requires a minimum time between two requests from the same node. Thus, an anomalous behavior can be a short interval between two requests originated by the same node. Rule 4 raises an alarm whenever the time between two update requests from the same node is smaller than a specified threshold.

Rules 5-6 Another anomalous situation happens whenever a link goes up and down frequently. Rules 5-6 raise an alarm whenever there are two requests (`Add` and `Del`, or vice versa) for the same link within an interval of time smaller than a specified threshold.

Rule 7 A suspicious status modification happens when a node advertises to be part of a different peer group. Rule 7 raises an alarm whenever the peer group identifier of a node is changed.

Consistency of the state tree

Rule 8 The maximum cell rate (maxCR) of a node must always be greater than or equal to the corresponding available cell rate AvCR. Rule 8 can be used to control satisfaction of this condition.

Rule 9 maxCR must also be less than or equal to the AvCR associated with the input/output ports of the link. Rule 9 raises an alarm whenever this condition is violated.

Rule 10 maxCR associated with input and output ports with the same link should be equal. Rule 10 can be used to control satisfaction of this condition.

8.2 Example of anomaly detection rules in BGP

Like for the PNNI protocol, we present some examples of anomaly behaviors. Figure 11 reports the corresponding rules.

Rule 1	**event:**	`Upd`(Node.BGP.4RcvdPathAttrTable.idaddrprefix(X)), Node.BGP.id(Y),
		Node.BGP.4RcvdPathAttrTable.idaddrprefix(X).next_hop(A)
	state:	Node.BGP.4RcvdPathAttrTable.idaddrprefix(X).next_hop(B)
	condition:	$A \neq B \wedge A = Y$
Rule 2	**event:**	`Upd`(Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W))
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).state(A)
	state:	Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).state(B),
		`countcond`(V,`Upd`(Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W)),
		(`bf_val`(state) $\neq$ `af_val`(state) $\wedge$ `af_val`(state) = 'idle'),[0,now])
	condition:	$A \neq B \wedge A =$ 'idle' $\wedge V >$ threshold
Rule 3	**event:**	`Upd`(Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W)),
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).inupd(A)
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).intotmsg(B)
	state:	Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).inupd(C)
	condition:	$A > B \vee A \leq C$
Rule 4	**event:**	`Upd`(Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W)),
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).outupd(A)
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).outtotmsg(B)
	state:	Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).outupd(C)
	condition:	$A > B \vee A \leq C$
Rule 5	**event:**	`Upd`(Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W)),
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).keepalive(A)
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).holdtime(B),
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).keepaliveconf(C),
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).holdtimeconf(D)
	state:	
	condition:	$A \neq 0 \wedge B \neq 0 \wedge A/B \neq C/D$
Rule 6	**event:**	`Upd`(Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W)),
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).inupd(A)
		Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).inupdelaptime(B)
	state:	Node.BGP.PeerTable.laddr-lport-raddr-rport(X,Y,Z,W).inupd(C)
	condition:	$A \neq C \wedge B \neq 0$

Figure 11: Example of rules in BGP

Unusual modifications

Rule 1 The change of the next hop property must be considered suspect if the new address of the next hop coincides with the address of the node that receives this change.

Suspicious status modifications

Rule 2 An anomalous situation happens whenever a neighbor relationship is frequently terminated. Rule 2 raises an alarm whenever the number of transitions from a given connection state to 'idle' of a BGP connection is greater than a specified threshold.

Consistency of the state tree

Rules 3-4 The number of BGP update messages received (transmitted, resp.) on a given connection must always be less than or equal to the total number of messages received (transmitted, resp.) on the same connection and it must be greater than the previous value. Rule 3 (Rule 4 resp.) is used to verify satisfaction of this condition.

Rule 5 For each given node, the ratio between the value of `keepalive` and `holdtime` must be equal to the ratio between `keepaliveconfigured` and `holdtimeconfigured`. Rule 5 raises an alarm whenever this condition is violated.

Rule 6 The Peer Table includes a property, called `inupdateelapsedtime`, denoting the elapsed time in seconds since the last update message was received. Each time the property `inupdates` is incremented, the value of `inupdateelapsedtime` has to be set to zero. Rule raises an alarm whenever a change of `inupdates` is not accompanied by a corresponding change of `inupdateelapsedtime`.

8.3 Anomaly correlation

Consider nodes n_1, n_2, and n_3 in Figure 2. Suppose that node n_1 floods a suspicious status modification regarding the fact that n_1's connection to node n_3 is terminated. By evaluating **Rule 2** in Figure 11, the members of the CG of domain D_1 detect that the neighbor relationship between n_1 and n_3 is frequently terminated. The CG nodes at this point may come to a consensus that the anomaly is significant but should be resolved at a higher level. In fact, the frequent number of changes in the state connection can be a consequence of a problem in node n_1 or in node n_3. However, it is impossible to tell who is fault just from examining the anomaly. The anomaly is thus passed to the higher-level domain (backbone domain). At the same time, node n_2 floods a similar suspicious status modification regarding the fact that n_2's connection to node n_3 is terminated. The change in the state connection causes an anomaly to be reported in the domain's CG (see **Rule 6** in Figure 10). Like for domain D_1, the GL of domain D_2 floods the anomaly up to the higher-level domain. At the higher level, nodes n_4, n_5, n_6, and n_7 receive two independent alarms and can come up with a diagnosis. They can deduce that it is unlikely that nodes n_1 and n_2 are both faulty and therefore can declare n_3 faulty on the basis on these multiple concurrent anomaly reports coming from different nodes running different routing protocols.

9 Conclusions

We have presented a novel approach for securing global infrastructures and have instantiated our ideas for the design of GIPS. The core part of the GIPS's architecture is the Anomaly Detection Language, a rule-based language used to specify anomalous (or uncommon) state information updates. The GIPS design can be used to offer a high level of integrity protection without requiring significant changes to the standard, and without entirely relying on cryptography. GIPS handles both malicious and nonmalicious faults in a unified manner and can therefore be used as an additional level of assurance for the proper operation of large communication networks. We are currently planning the development of a prototype to be used in the field to log anomalous state information updates for review by an operator. We believe that this functionality alone could be very useful in a production environment to spot and resolve inconsistencies in the system configuration that could lead to integrity violations.

Our approach leaves space for further work. First, GIPS's effectiveness is intimately tied to its ability to properly detect and processing anomalies and making precise and informative diagnoses. It is therefore necessary to validate its effectiveness with hands-on experiments that can reproduce numerous fault scenarios and help in accumulating enough heuristical experience for the specification of effective anomaly detection rules. Another important issue to be addressed concerns the substantial complexity of Byzantine faults diagnosis.

10 Acknowledgments

This work was supported in part by the European Community within the FASTER Project in the Fifth (EC) Framework Programme under contract IST-1999-11791.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1), 1997.

[2] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. Technical Report RFC 2205, September 1997.

[3] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM SIGMOD International Conference*, pages 505–516, Montreal, Canada, June 1996.

[4] S. Chen and K. Nahrstedt. An Overview of Quality-of-Service Routing for the Next Generation High-Speed Networks: Problems and Solutions. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, 1998.

[5] J. Clark and S. DeRose (ed.). *XML Path Language (XPath) Version 1.0*. World Wide Web Consortium (W3C), July 1999. http://www.w3.org/1999/07/WD-xpath-19990709.

[6] L. Delgrossi and L. Berger. Internet Stream Protocol Version 2 (ST2) Protocol Specification - Version ST2+. Internet RFC 1819, August 1995.

[7] L. Delgrossi and L. Berger. Internet Stream Protocol Version 2 (ST2) Protocol Specification - Version ST2+. RFC 1819, August 1995.

[8] S. De Capitani di Vimercati, P. Lincoln, L. Ricciulli, and P. Samarati. PGRIP: PNNI Global Routing Infrastructure Protection. In *The Internet Society 1999 Network and Distributed System Security Symposium (NDSS'99)*, San Diego, CA, February 1999.

[9] S. Eker. Fast Matching in Combinations of Regular Equational Theories. In J. Meseguer, editor, *First Intl. Workshop on Rewriting Logic and Its Applications*, pages 90–108. Elsevier Science, September 1996.

[10] D. Estrin and G. Tsudik. Security Issues in Policy Routing. In *Proc. of the 1989 IEEE Symposium on Security and Privacy*, pages 183–193, Oakland, CA, May 1989.

[11] ATM Forum. Private Network-Network Interface Version 2.0 Specification. Technical Report BTD-PNNI-02.00, 1997.

[12] R. Guerin and A. Orda. QoS-based Routing in Networks with Inaccurate Information: Theory and Algorithms. In *Infocom'97*, Japan, April 1997.

[13] R. Hauser, T. Przygienda, and G. Tsudik. Lowering Security Overhead in Link State Routing. In *Computer Networks and ISDN Systems*, April 1999.

[14] C. Hunt. *TCP/IP Network Administration, 2nd Edition.* O'Reilly and Associates Sebastopol, California, 1997.

[15] Y.F. Jou and S.F. Wu. Architecture Design of a Scalable Intrusion Detection System for the Emerging Network Infrastructure. Technical Report Order Number: E296, DARPA, April 1997.

[16] L. Kleinrock and F. Kamoun. Hierarchical Routing for Large Networks: Performance Evaluation and Optimization. *Computer Networks*, 1:155–174, 1977.

[17] L. Lamport, R. Shostak, and M. Pease. The Byzantine General Problem. *ACM Transaction on Programming Languages and Systems*, 4(3):382, 1982.

[18] G. Malkin. RIP Version 2. Technical Report RFC2453, November 1998.

[19] J. Moy. OSPF Version 2. Technical Report RFC 2328, April 1998.

[20] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *IEEE International Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, March 1995.

[21] J. Pasquale, G. Polyzos, E. Anderson, and V. Kompella. The Multimedia Multicast Channel. In *Proc. of 3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1992.

[22] R. Perlman. *Network Layer Protocols with Byzantine Robustness.* MIT LCS TR-429, Massachusetts Institute of Technology, October 1988.

[23] R. Perlman. *Interconnections: Bridges and Routers.* Addison-Wesley, 1992.

[24] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). Technical Report RFC1771, March 1995.

[25] L. Ricciulli and N. Shacham. Modeling Correlated Alarms in Network Management Systems. In *Communication Networks and Distributed Systems Modeling and Simulation*, 1997.

[26] K.E. Sirios and S.T. Kent. Securing the Nimrod Routing Architecture. In *Proc. of the 1997 Symposium on Network and Distributed System Security (NDSS'97)*, pages 74–84, San Diego, CA, February 1997.

[27] B.R. Smith, S. Murthy, and J.J. Garcia-Luna-Aceves. Securing Distance-Vector Routing Protocols. In *Proc. of the 1997 Symposium on Network and Distributed System Security (NDSS'97)*, pages 85–92, San Diego, CA, February 1997.

[28] C.J. Walter, P. Lincoln, and N. Suri. Formally Verified On-Line Diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684, 1997.

[29] 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 *5 Minute Talks session at 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.

List of Figures

List of Footnotes

1. In most cases, to protect against reply attacks, identical information needs to be made unique through the use of time-stamps.

2. This argument assumes a routing system to be stable and therefore not to exhibit a large number of anomalies. We do not consider unstable routing systems because they would not be viable and would not be worth protecting.

3. Note that it is also possible to reach a consensus on the impossibility of a correct diagnosis.

4. Note that this assumption is only for the sake of this example; it does not correspond to any real situation.

5. The scope defines the highest level of the hierarchy at which the address will be visible.

6. Note that in general, to exchange routing information between networks that use different routing protocols, multiple routing protocols have to be configured in a single border node.