Modeling Firewalls Using Hierarchical Colored Petri Nets

Christoph L. Schuba
christoph.schuba@sun.com

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303–4900
United States of America

Eugene H. Spafford
spaf@cs.purdue.edu

COAST/CERIAS and
Department of Computer Sciences
Purdue University
1398 Computer Science Building
West Lafayette, IN 47907–1398
United States of America

Abstract

This paper concentrates on one technological aspect of providing communications security, *firewall technology*. It introduces a formalism called *Hierarchical Colored Petri Nets (HCPN)* in tutorial style. The main contribution of the paper is a description of how to model firewall systems using Hierarchical Colored Petri Nets. A byproduct of this approach is a novel way of modeling audit streams in distributed systems.

HCPNs are well suited for modeling concurrent, distributed systems in which regulated flows of information are significant, such as firewall systems which enforce access control policies on network packets. The paper introduces the basics of this modeling technique. It demonstrates with several examples how firewalls can be modeled. It outlines how simulations of such models can facilitate testing, performance analysis, and interactive design exploration. Finally, the approach can serve as the basis for formal analysis techniques available through Applied Petri Net Theory.

1 Introduction

Data communications networks have become an infrastructure resource for businesses, corporations, government agencies, and academic institutions. Computer networking, however, is not without risks as Howard ([18]) illustrates in his analysis of over 4000 security incidents on the Internet between 1989 and 1995. Firewall technology is only one mechanism to protect against network based attack methods. A balanced approach to network protection must include physical security, personnel security, operations security, communication security, and social mechanisms ([20, Part II]).

Classically, firewall technology has been applied to TCP/IP (*transmission control protocol*, [36]; *internet protocol*, [37]) internetworks. Firewalls are used to guard and isolate connected segments of internetworks. “Inside” network domains are protected against “outside” untrusted networks, or parts of a network are protected against each other. Various architectures for firewalls have been published and built (see section 3).

Landwehr suggests the application of formal models of security to secure system design (see [26, §1]): by demonstrating that a design to which an implementation corresponds enforces a formal model of security, a convincing argument can be made that the system is secure. Firewall systems are often implemented through a number of mechanisms that collectively achieve the desired functionality. This paper introduces a formalism based on *Hierarchical Colored Petri Nets* (HCPN, short CPN) to describe the functionality of such mechanisms. CPNs are a formalism well suited for modeling systems in which synchronization, concurrency, composition, and activities on regulated flows of information are significant ([24]). It can be used for the representation, combination, simulation, and analysis of firewall components and firewall systems. The introduction of this design approach is the main contribution of this paper.

The paper is organized as follows: Section 2 defines terminology used throughout this paper. Section 3 provides a brief overview of firewall mechanisms. Section 4 introduces *Colored Petri Nets* (CPN) and *Hierarchical CPNs* (HCPN) as formalisms for modeling firewall mechanisms and firewall systems. Section 5 describes the modeling of an example firewall that combines two firewall mechanisms, an IP packet filter and an IPSEC (IP security working group) authentication header (AH) module ([2]). Section 6 describes how simulation can be used to facilitate the generation of various results about modeled systems, such as performance results and functionality assurance through testing. Section 7 discusses some of the approaches available by means of Applied Petri Net Theory to formally analyze modeled systems. The paper closes with section 8, summarizing its contributions and presenting ideas for future research.

2 Terminology

This section defines terminology used throughout the paper and gives a working definition of the term *firewall technology*. Technical terms not defined in this section are used according to their definitions in [6, 11, 27]. Definitions in this section are based in [44, 6, 11, 27] but extended to fit our needs.

We define *communication traffic* to be the transmission of information over a network. We denote the set of all possible transmissions by $\mathbf{T}$. Any instance of communication traffic, called *a transmission unit*, is a tuple $(ctrl, data) = t \in \mathbf{T}$ consisting of control information ($ctrl$) and data ($data$) either of which may be empty, but not both. The interpretation of what amount of information comprises a transmission unit depends on the protocol layer of observation. For example, in a popular instance of network layer functionality (see ISO model [13]), the Internet Protocol ([36]), transmission units are called *datagrams*, or *packets*.

Attribute $t.ctrl$ may contain information, such as source ($t.ctrl.src$) and destination ($t.ctrl.dst$) addresses, reliability and flow control information, access request information ($t.ctrl.acc$), and quality of service parameters ($t.ctrl.qos$). Attribute $t.data$ may contain application-specific payload or a payload that, at a higher layer of abstraction, can be interpreted as a transmission unit in itself. Transmission units do not need to contain all fields of t. For example, some fields may not be necessary at all, such as $t.data$ in control messages; others may be available through established state, such as $t.ctrl.qos$ in an existing connection.

A *security policy* is the definition of the security requirements for a given system. It can be defined as a set of standards, rules, or practices. We define a *network domain security policy* $\mathbb{P}$ as a subset of a security policy, addressing requirements for authenticity and integrity of communication traffic $t \in \mathbf{T}$, authorization requirements for access requests $req(t.ctrl.src, t.ctrl.dst, t.ctrl.acc) \ \forall \ t \in \mathbf{T}$, and auditing requirements.

A *network policy domain* $\mathbb{D}$ is a set of interconnected networks, gateways, and hosts offering services which are governed by a network domain security policy $\mathbb{P}$.

Using the above defined terminology and a study of firewall systems as described in section 3 we arrive at the following working definition of the term *firewall technology*.

Firewall technology is a set of mechanisms that can enforce a network domain security policy $\mathbb{P}$ on communication traffic $\mathbf{T}$ entering or leaving a network policy domain $\mathbb{D}$ in a fashion transparent to the user. A *firewall system*, or *firewall*, is an instantiation of firewall technology.

3 Firewall Mechanisms

In recent years leading up to mid 1997, a number of firewall architectures have been proposed and implemented. A variety of security mechanisms were developed and used, such as packet filtering, packet labeling, network address translation, or proxy forwarding. Several research papers and some text books describe the different approaches (see e.g., [16, §21], [6], [10], [47], [39], [41], [40], [42], [4], [3], and [1]).

4 Formalism for Firewall Mechanisms: Hierarchical Colored Petri Nets

This section introduces *Colored Petri Nets* (CPN) and *Hierarchical CPNs* (HCPN) as formalisms for firewall mechanisms and firewall systems. The section begins by arguing why we chose CPNs as a formalism. It then introduces the graphical representations of CPNs and HCPNs, explains their modeling equivalence and presents some limitations of CPNs.

A *Petri Net* ([33]) is a model expressed as a network of interconnected locations and activities with rules that determine when an activity can occur. It also specifies how an activity changes the states of the locations associated with it. Petri Nets have been used for the modeling and analysis of systems ([32]). A considerable body of theory exists ([34]) dealing with Petri Net properties, such as liveness and reliability. Petri Nets have been developed over the years from a simple yet universally applicable paradigm to various high-level and more complex but far more convenient methodologies: one such example is Hierarchical Colored Petri Nets. Their formal definition can be found in [22] and [24].

The following paragraphs are a summary of features that CPNs possess. These features make CPNs appropriate and fitting for modeling firewall mechanisms and systems. The summary is compiled from [23, 24]. CPNs promote problem-oriented structuring of a system and make it possible to formulate and prove system characteristics. They offer hierarchical descriptions and are suited for modeling systems of distributed control with multiple processes executing concurrently in time.

These characteristics support the modeling of firewalls that are distributed systems consisting of several interacting mechanisms. CPNs are asynchronous in nature without an inherent measure of time although a measure of time has been added in various extensions. The lack of time reflects a philosophy that states that the only important property of time, from a logical point of view, is in defining a partial ordering of the occurrence of events. There are a large number of formal analysis methods by which properties of instances of CPNs can be proved. Computer support for complex analysis methods makes it possible to obtain results that are impractical to be achieved manually.

There are other formalisms that are at least equivalent in computational power to CPNs, but not in regard to convenience. Similar arguments apply as in the choice of the right programming language to solve a given problem.

4.1 Colored Petri Nets (CPN)

The CPNs presented in this paper use the following notation (cf. figure 1). They contain *places* (ellipses) and *transitions* (rectangles). Places (a.k.a. *states*) represent conditions while transitions represent actions. Places can contain instantiations, called *tokens*, of structured data types, called *colors* (italic names next to places), hence the name Colored Petri Nets. The distribution of tokens at places is called a *marking*. The initial marking is determined by an initialization expression (font helvetica expressions next to places; figure 2). The marking (boldface font helvetica expression next to places; figure 2) for each place is a multi-set (cf. [24, §2.1] for a definition of multi-sets) over the place's color set.

Places and transitions can be connected by directed *arcs* (arrows). Transitions are *enabled* if there are tokens in all places associated with incident arcs. An enabled transition can *fire*, if token values are *bound* according to relevant *arc expressions* (typewriter expressions next to arcs) and the *guard* (typewriter expressions enclosed in square brackets next to transitions) associated with the transition evaluates to true. A transition fires by removing the required tokens from all places connected through incident edges and by adding tokens to all places connected through emanating edges. Arc and guard expressions may have a set of variables associated with them. The substitution of values for variables can lead to their unification if they have common instances. In CPNs the binding of values to these variables is equivalent to their unification ([45, §8.2]).

We use an extension ([25] and [29, Part 3]) of the programming language ML (*Meta Language*; [30, 48]) to define colors, arc expressions, guards, and code sections. ML is a strongly typed, high-level functional programming language optimized for abstract data structure specification and manipulation. The strong typing forces designers to be specific about the data types of represented information and ensures unambiguous interface specifications for the combination of CPNs. For the manipulation and simulation of CPNs we use the Design/CPN software from the University of Aarhus ([29]). It uses ML as the specification language of choice.

CPNs can be specified formally without a graphical representation: as a tuple consisting of a number of sets (color, place, transition, and arc sets) and functions (node, color, guard, arc expression, and initialization functions), as in [24, definition 2.5]. This method of specification of CPNs is necessary for a number of formal analysis methods by which properties of CPNs can be proven. We chose a graphical representation of CPNs over its set theoretic representation to graphically express the structure of modeled systems.

4.2 Hierarchical Colored Petri Nets (HCPN)

Figures 1 and 4 illustrate separate mechanisms that are used to build firewalls. Firewalls consist of a set of mechanisms that collectively provide network access control. Furthermore, they use external functions, such as authentication header verification, and external state, such as TCP connection state. For practical reasons it is not desirable to create a single large CPN that specifies a given firewall system in a flat structure.

The concept of Hierarchical CPNs allows a designer to construct large CPNs by combining a number of smaller CPNs. They are beneficial for the modular composition of CPNs. HCPNs are defined in [19]. HCPNs can be constructed top-down, bottom-up, or by a mixture of these two strategies. HCPNs make it possible to relate a number of individual CPNs to each other in a formal way, and thus allow their formal analysis ([24]).

In a top-down design one starts with a simple high-level description of a system without consideration for internal details. A specification of detailed behavior of the CPN is developed through stepwise refinement ([49]). Stepwise refinement is achieved through the application of a construct called *substitution transition*, where a more complex CPN takes the place of a transition. The CPN must conform to the interface of the replaced transition and relate identically to its surrounding arcs. Transitions Packet Filter and Authentication Header in figure 2 are examples of substitution transitions.

In a bottom-up design CPNs are combined into a larger net through *fusion places*. A fusion place is a set of places that are considered to be identical. Even if they are drawn as individual places they represent a

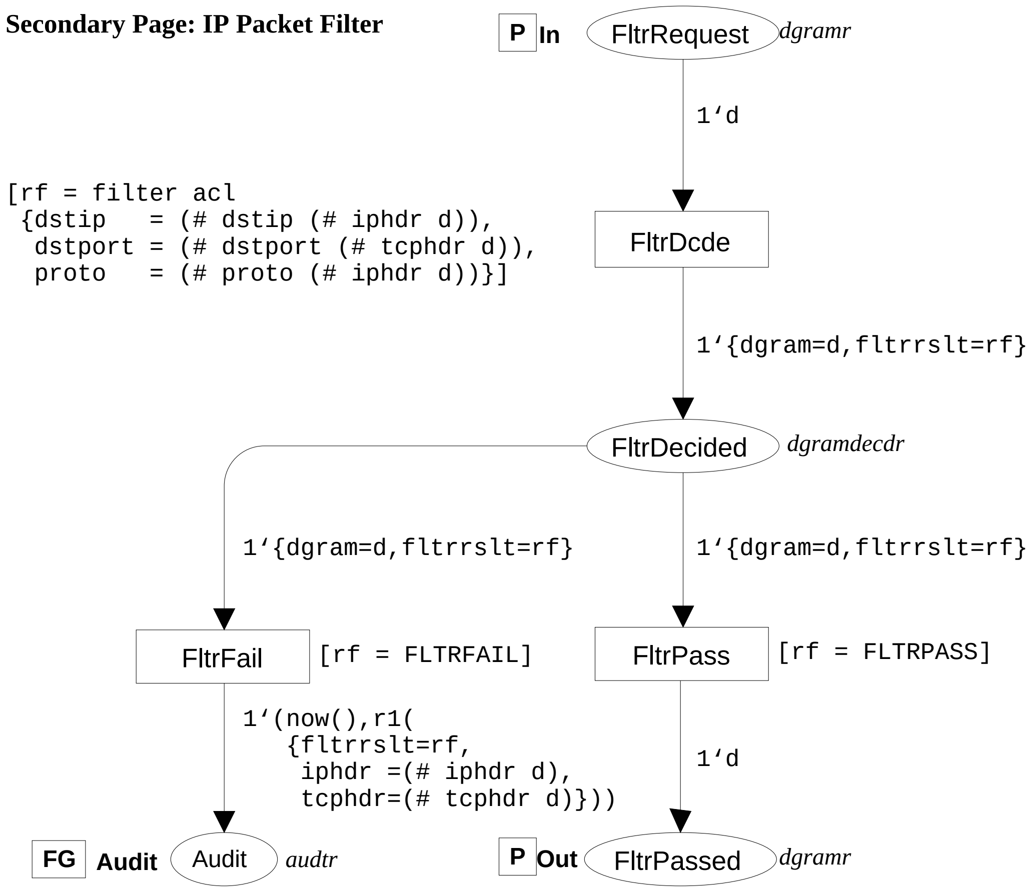


Figure 1: Example of a Colored Petri Net for IP packet filtering

single conceptual place. For each token that is added (removed) at one of the places, an identical token is added (removed) at all others. Places FltrRequest in figure 1 and P1 in figure 2 are a fusion place, for example.

A non-hierarchical CPN is called a *page*. Figures 1, 2, and 4 contain pages. A page that contains a substitution transition is called a *superpage* (e.g., figure 2); a page that contains the detailed description of the activity modeled by the corresponding substitution transition is called *subpage* (e.g., figure 1). A substitution transition is also called a *supernode*. Note that the places connected to a substitution transition by a single arc (called *socket nodes*) and their counterparts on the subpage (called *port nodes*) are fusion places. The interface between a superpage and a subpage is defined through *port assignments* where socket nodes are related to port nodes.

4.3 Equivalence of CPNs and HCPNs

Any HCPN can be translated into a behaviorally equivalent non-hierarchical CPN by replacing each substitution node with a copy of its subpage. This replacement process may need to be applied recursively. The recursion is guaranteed to terminate because a strictly hierarchical relationship between pages is enforced during construction. HCPNs are equivalent to CPNs, which means the theoretical modeling power of the two classes are identical. However, they have different properties from a practical point of view: HCPNs allow a designer to cope with large systems because of their facilities for structuring and abstraction.

4.4 Limitations

The original model of Petri Nets has several limitations that since have been addressed by extensions to the basic model. For example, in Petri Nets there is no way to test if zero tokens are in an unbounded place (cf. [32]). Although Petri Nets can be used for modeling systems at different levels of abstraction, in their original form they can be difficult to comprehend by humans, even when a system is expressed at a high-level. Hierarchical Colored Petri Nets are one example of an extended model which already deals with a subset of the original model's limitations.

Many known algorithms that operate on Petri Nets have high computational complexities ([24, Ch.4,5]). As long as CPNs are only used for their expressive power this limitation is not relevant. But several interesting properties of CPNs, such as boundedness, or the absence of deadlocks, require the application of algorithms with high computational complexity. High computational complexity, however, can still be acceptable if it is sufficient to verify the properties in question infrequently and outside of performance critical paths, such as at the time of design validation ([12, Ch.2]). Because of the high computational complexity of formal verification of properties, high-level formalisms have a disadvantage compared to low-level formalisms, such as the originally defined Petri Nets ([50]). Section 7 gives details on the computational complexity and analyses methods for CPNs. In general, low-level formalisms are a better choice for the formal analysis of systems ([50]). These capabilities are a trade-off for a greater expressiveness in high-level formalisms.

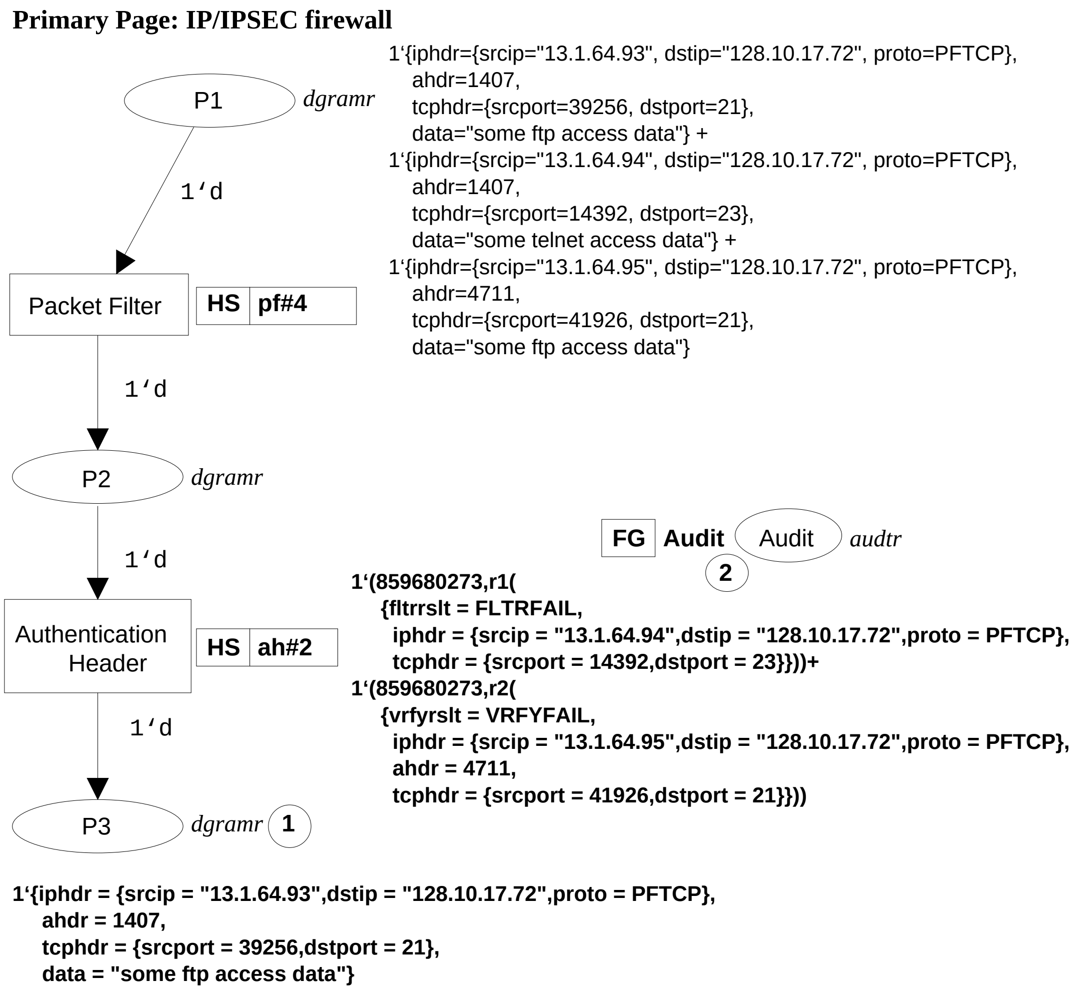
5 Example: HCPN for a simple IP Firewall

This section describes an example firewall that combines two firewall mechanisms, an IP packet filter and an IPSEC (IP security working group) authentication header module. We structure the description top-down, starting with the superpage.

The firewall system modeled in figure 2 is a superpage consisting of two components: an IP packet filter, and an AH module. Places P1, P2, and P3 contain tokens of color *dgramr* that represent IP datagrams. The two components are represented as substitution transitions, with the packet filter from figure 1 being applied first. Each instantiation d of color *dgramr* in place P1 represents a datagram d that arrives at the firewall. Note that d is a transmission unit as defined in section 2 and $d \in \mathbf{T}$. Once substitution transition Packet Filter fires, d is removed from place P1. It is only added to place P2 if d is added to place FltrPassed (figure 1), a fusion place of P2, within the subpage.

Thus, only datagrams that pass the transition Packet Filter successfully can be input to the transition Authentication Header representing the IPSEC AH firewall component. All datagrams that are added to place P3 therefore have passed both firewall components successfully and can be forwarded to their destination. Figure 2 depicts place Audit which models an audit function collecting audit events.

Remark. The meaning of arcs around substitution transitions, such as Packet Filter, differs from the meaning of arcs around regular transitions. The set of arcs around a substitution transition describes an interface of the substituted CPN rather than a unification of common instances that must occur. It means that datagrams that are removed from place P1 because transition Packet Filter fires need not be added to place P2. They are only added if they appear in the fusion place corresponding to place P2.



```
1‘{iphdr={srcip="13.1.64.93", dstip="128.10.17.72", proto=PFTCP},
    ahdr=1407,
    tcphdr={srcport=39256, dstport=21},
    data="some ftp access data"} +
1‘{iphdr={srcip="13.1.64.94", dstip="128.10.17.72", proto=PFTCP},
    ahdr=1407,
    tcphdr={srcport=14392, dstport=23},
    data="some telnet access data"} +
1‘{iphdr={srcip="13.1.64.95", dstip="128.10.17.72", proto=PFTCP},
    ahdr=4711,
    tcphdr={srcport=41926, dstport=21},
    data="some ftp access data"}
```

```
1‘(859680273,r1(
    {fltrrslt = FLTRFAIL,
      iphdr = {srcip = "13.1.64.94",dstip = "128.10.17.72",proto = PFTCP},
      tcphdr = {srcport = 14392,dstport = 23}}))+
1‘(859680273,r2(
    {vrfyrslt = VRFYFAIL,
      iphdr = {srcip = "13.1.64.95",dstip = "128.10.17.72",proto = PFTCP},
      ahdr = 4711,
      tcphdr = {srcport = 41926,dstport = 21}}))
```

```
1‘{iphdr = {srcip = "13.1.64.93",dstip = "128.10.17.72",proto = PFTCP},
    ahdr = 1407,
    tcphdr = {srcport = 39256,dstport = 21},
    data = "some ftp access data"}
```

Figure 2: Hierarchical Colored Petri Net for a simple IP firewall consisting of an IP packet filter and IPSEC authentication header module.

5.1 IP Packet Filtering

In a TCP/IP packet filtering firewall each datagram that arrives at the firewall router is passed to a packet filtering mechanism. The filter discards or forwards packets according to specified rules based on the fields of the TCP/IP packet header, e.g., source and destination addresses and port numbers. The rules operate exclusively on the contents of the datagram, because no context is maintained across datagrams that belong to the same connection.

In current packet filtering routers, security policies are translated into lists of rules (see [5], [9], [14]). Each rule allows or denies data through the firewall based on some semantic interpretation of the data contents. Rules may interact with each other, e.g., through their order. If no rule is applicable a default action is performed, e.g., "discard packet."

Although a packet filter offers the opportunity to handle and verify all data passing through it, the lack of end-to-end context prevents a security association from being established. Packet filtering does not provide integrity and authenticity control of the examined packets. The application of filtering rules to each datagram introduces some delay because their processing takes time. It may introduce jitter because the calculation of filtering results can introduce different amounts of delay for different packets. Although efforts have been made to automate (and improve the quality of) the generation of the filtering rule set (see e.g., [8]), expressing high level security policies in this low level mechanism is still a practical challenge.

Figure 1 gives a CPN specification of a typical IP packet filter. It models the invocation, filtering decision, and decision enforcement of a packet filter.

Each datagram that arrives at the packet filter is represented by a token of color *dgramr* in place FltrRequest. In this example, a datagram (type *dgramr*) consists of several possible types of headers (types *iphdrr*, *ahdrr*, *tcphdrr*) and a data portion (cf. figure 3 for the ML declaration of colors in this example). The header contains a subset of the TCP/IP header fields. It does not contain all header fields as defined in [36], but rather those that are necessary and sufficient to perform the packet filtering operation in this simple example. The header fields are used by the packet filter to decide if the datagram is to be forwarded or discarded.

The transition FltrDcde is enabled whenever the marking of place FltrRequest contains at least one token, i.e., whenever a datagram arrives at the packet filter. Variable d is then bound to the datagram values, which unifies all occurrences of d to this instance. The guard associated with FltrDcde uses function `filter` to apply the access control list defined in `acl` (cf. figure 3) against d and assigns the result to `rf` (`FLTRFAIL` or `FLTRPASS`). Function `filter` takes two arguments: a list of tuples containing patterns and their corresponding results (`acl`), and a pattern. It returns the result corresponding to the pattern if found in the access control list, and the default safe value `FLTRFAIL` otherwise. The access control policy for IP packets is not encoded in the CPN model, but in `acl`. This CPN model merely describes a mechanism for enforcing whatever policy is encoded.

Once transition FltrDcde is fired, d is removed from place FltrRequest. Datagram d and its filtering result `rf` are combined into a token of color *dgramdecdr*, a record type, and added to place FltrDecided. Depending on the value of variable `rf` exactly one of the two transitions FltrFail and FltrPass is enabled because both are guarded by mutually exclusive but collectively exhaustive expressions. Note that a guard expression, such as `[rf = FLTRFAIL]`, is no assignment but rather a test for equality: after the first assignment to variable `rf` in the guard of transition FltrDcde all occurrences of `rf` are unified and assignment and test for equality are denoted by the same symbol (=) although they are different operations. Therefore, guards in CPNs are predicates with side effects.

In case transition FltrFail is enabled and consequently fired, figure 1 models information about datagram d being added to place Audit. This process can be interpreted as the datagram being discarded and details about the denied access being logged. The place is included to be able to collect information about discarded packets for auditing purposes as well as the validation of the behavior of the packet filter itself. If transition FltrPass is enabled, then d is added to place FltrPassed. Place FltrPassed is the final place in this CPN. Each datagram in a marking of FltrPassed can now be forwarded towards its destination.

5.2 Modeling the IPSEC Authentication Header Module

This section serves three purposes. It gives a second example of a CPN firewall mechanism (the IP Authentication Header as defined in [2]), it demonstrates how to build a CPN model for it, and it demonstrates how the model interacts with its environment in an abstract manner (e.g., through use of external state or execution of external functionality).

Section 4 of the IETF (*Internet Engineering Task Force*) standard document for the IP authentication header ([2]) specifies the procedure a module has to perform to verify the authentication header in a received IP packet[1]:

[1]Note: SPI stands for *security parameter index*, an end-to-end se-

```
color  datat      = string;
color  ipt        = string;
color  portt      = int;
color  protot     = with PFUDP | PFTCP;
color  spit       = int;
color  ait        = int;
color  timet      = int;
color  iphdrr     = record srcip:ipt * dstip:ipt * proto:protot;
color  ahdrr      = spit;
color  tcphdrr    = record srcport:portt * dstport:portt;
color  dgramr     = record iphdr:iphdrr * ahdr:ahdrr * tcphdr:tcphdrr * data:datat;
color  fltrrsltt  = with FLTRFAIL | FLTRPASS;
color  dgramdecdr = record dgram:dgramr * fltrrslt:fltrrsltt;
color  spir       = record spidx:spit   * dstip:ipt * ai:ait;
color  dgramspir  = record dgram:dgramr * spi:spir;
color  vrfyrsltt  = with VRFYFAIL | VRFYPASS;
color  dgramvrfyr = record dgram:dgramr * vrfyrslt:vrfyrsltt;
color  fltrfailar = record fltrrslt:fltrrsltt * iphdr:iphdrr * tcphdr:tcphdrr;
color  vrfyfailar = record vrfyrslt:vrfyrsltt * iphdr:iphdrr * ahdr:ahdrr *
                           tcphdr:tcphdrr;
color  audtu      = union r1:fltrfailar + r2:vrfyfailar;
color  audtr      = product timet * audtu;

(*--------------------------------------------------*)
(*filter = fn : (''a * fltrrsltt) list -> ''a -> fltrrsltt*)
fun filter acl dgram =
  lookup dgram acl handle exlookup => FLTRFAIL;

(*verify = fn : spir -> dgramr -> vrfyrsltt*)
fun verify (s:spir) (d:dgramr) =
  case (# ai s) of
    42 => VRFYPASS |
    _  => VRFYFAIL;

(*now = fn : unit -> int*)
fun now () = tod ();

(*--------------------------------------------------*)
val acl = nil;
val acl = insert {dstip="128.10.17.72", dstport=21, proto=PFTCP} FLTRPASS acl;

(*--------------------------------------------------*)
var d  : dgramr;
var s  : spir;
var rv : vrfyrsltt;
var rf : fltrrsltt;
```

Figure 3: Example declaration of colors for the Colored Petri Net model of IP packet filtering and specification of an access control list for IP packet filtering. Access to host 128.10.17.72 is granted for TCP on service port 21 (ftp). All other accesses are denied.

```
‘‘Upon receipt of a packet con-
taining an IP Authentication
Header, the receiver first uses
the Destination Address and SPI
value to locate the correct Se-
curity Association.  The receiver
then independently verifies that
the Authentication Data field
and the received data packet are
consistent.  [..]

[..]  If the processing of the
authentication algorithm indi-
cates the datagram is valid, then
it is accepted.  If the algo-
rithm determines that the data and
the Authentication Header do not
match, then the receiver SHOULD
discard the received IP datagram
as invalid and MUST record the
authentication failure in the sys-
tem log or audit log.  If such a
failure occurs, the recorded log
data MUST include the SPI value,
date/time received, clear-text
Sending Address, clear-text Desti-
nation Address, and (if it exists)
the clear-text Flow ID. The log
data MAY also include other infor-
mation about the failed packet.’’
```

This procedure can be divided into four steps as follows:

1. Receipt of packet
2. Location of security association
3. Verification of authentication data field
4. Enforcement of authentication verification result

Figure 4 depicts the CPN for the AH mechanism. Step 1 is modeled through an instantiation of color *dgramr* in initial place AhRequest. Place SpiDb models external state: the set of established security associations. The lookup (step 2) of a security association is achieved through the matching of the security parameter index field present in the authentication header of datagram d against the spi index values in members of the marking of place SpiDb. It is reasonable to model the repository of *security parameter indices* (SPI) in this fashion because SPIs are established by external procedures, such as manual configuration or a key management protocol. Key management mechanisms are used to negotiate parameters other than keys to manage security associations.

curity association.

The verification of the authentication data field (Step 3) takes place in the guard of transition AhVrfy similar to the way we modeled filtering in figure 1. Our model contains a stub routine for the authentication header verification (see function `verify` in figure 4). The datagram d is then assigned to place AhPassed if the outcome of the verification is positive. Subsequently, the enforcement of the result takes place (step 4).

In case the result is VRFYPASS, d will be added to place AhPassed and continue on its path through the system. In case the result is `VRFYFAIL`, certain information from d will be augmented by further data fields, such as a time stamp, and added to place Audit. This models the audit requirement as specified in [2, §4].

The marking of place SpiDb in figure 4 contains two examples of security associations. The expression above the place is the initialization expression for the state; the one below is its current marking. The particular values of these markings were used in a simulation and are not of specific interest because they were chosen arbitrarily. Note that in this model the marking of place SpiDb will always be identical to the initial marking because a token that is unified to s for the matching that takes place in the guard for transition SpiLkup is returned to place SpiDb after transition SpiLkup is fired.

5.3 Interpretation of Simulation Results for the Example Firewall

We used the designCPN ([29]) CPN software for simulations of this firewall model. Figures 1, 2, and 4 display the markings of the model when no more transitions are enabled, i.e., after the end of a simulation. In figure 2 place P3 has instances of color *dgramr* in its final marking, and place Audit has instances of color *audtr* in its final marking.

The tokens that are part of the marking in place Audit recorded events where datagrams did not pass the firewall. The first token contains one audit record describing denied access enforced by the packet filter because the access policy encoded in the access control list `acl` allowed access only to the ftp port 21 (see figure 5).

The second token in place Audit represents a datagram that did not pass the authentication verification because we simulated an authentication failure for security parameters index 4711 and its bound authentication information 43. Only authentication information 42 leads to a positive verification result in function `verify` in figure 6. Finally, the token in place P3 passed both the packet filtering and the authentication verification and reached the final state of the CPN.

The color of the audit place is a product type containing a time stamp and a union type. The union type depends upon the type of the logged information. Events

Secondary Page: IPSEC Authentication Header Module

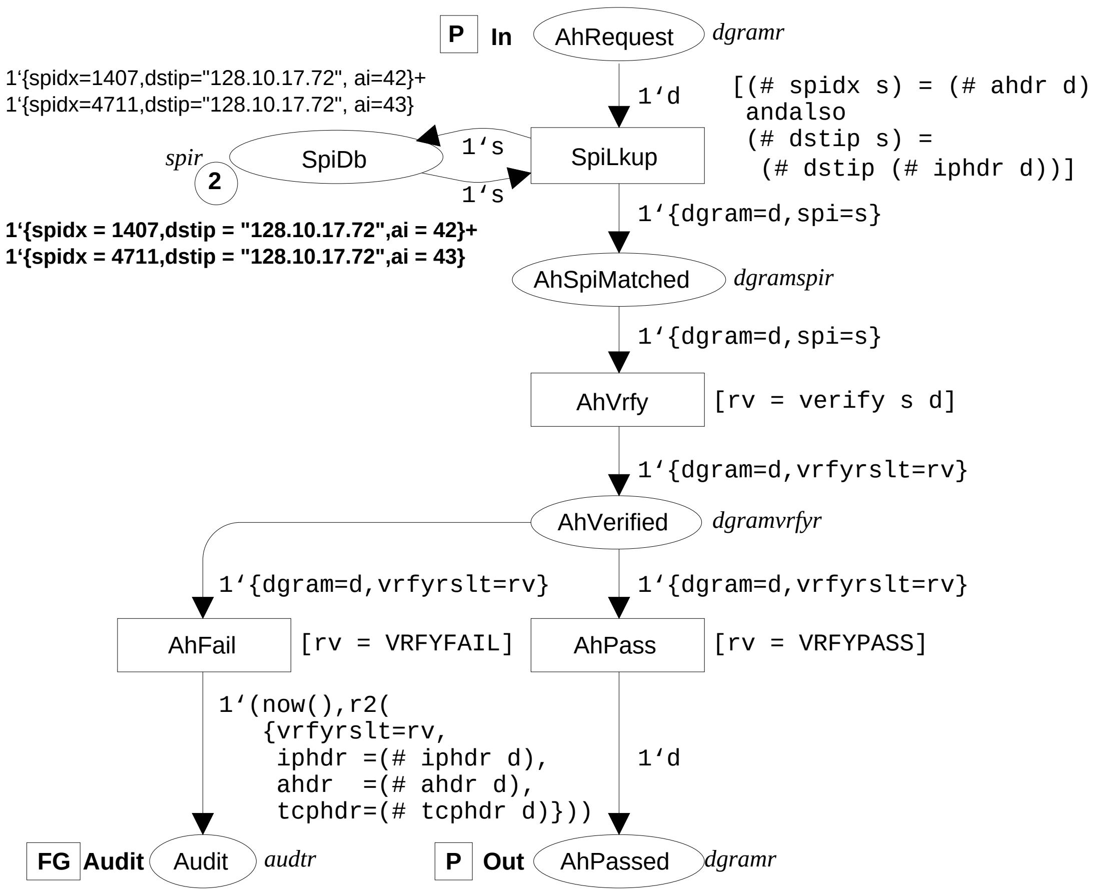


Figure 4: Example IPSEC authentication header

```
val acl = nil;
val acl = insert {dstip="128.10.17.72", dstport=21, proto=PFTCP} FLTRPASS acl;
```

Figure 5: Example access control list. Access is only allowed to the ftp port (21) on the host with IP address 128.10.17.72. (Excerpt from previous figure 3.)

```
(*verify = fn :  spir -> dgramr -> vrfyrsltt*)
fun verify (s:spir) (d:dgramr) =
  case (# ai s) of
    42 => VRFYPASS |
    _ => VRFYFAIL;
```

Figure 6: Example verification function implementing the policy that only authentication information 42 is acceptable. More realistically, this function would be replaced by cryptographic authentication code as implemented in our prototype. (Excerpt from previous figure 3.)

are logged by adding an arc from the transition representing the event's action to an audit place. In our example we used a fusion place for modeling the audit so that all logged events are collected in the same place. Information present at a transition as well as global state can be logged. Information (in contrast to events) does not need to be logged at the earliest transition because it is represented in a natural way in this model: instantiations of colors. It could be carried through the execution of a CPN as part of a token. We do not postulate which information is logged at what place in the net but we provide a simple method for modeling audit mechanisms. The research of others addresses the question of content (see e.g., [38]).

5.4 Challenges of Modeling

We gave an example of how to model firewall mechanisms in section 5.2.

Creating CPNs is a task that requires human expertise and experience, similar to other modeling techniques. Jensen provides a number of guidelines in [24, $\S1.5$] that can help modelers develop CPNs.

Designers need to understand the behavior of a mechanism before they can formalize it, which represents a problem if a given firewall mechanism is offered as a closed platform only. Designers can infer its behavior only through observation, marketing brochures, and possibly reverse engineering. Possible mismatches between the real firewall mechanism and its functional description can be difficult to detect.

One can imagine that vendors will provide formal descriptions of the behavior of their products as a service to their (potential) customers. Using the tool, guided by the formalism of the HCPNs, and using a library of generic firewalls and specific CPNs for firewall components, we can provide a beneficial design environment. Silva and Valette ([46]) argues that catalogs of well tested subnets allow component reusability which in turn leads to significant reductions in the modeling effort. The availability of such a library should encourage the adoption of HCPN technology by firewall designers, who would not need to create the models from scratch. This library would enable designers to explore various firewall designs using the available product formalizations in a simulation environment.

6 Simulation

Once a firewall mechanism or even a complete firewall system are modeled, our approach allows the simulation of nets. Statistics of simulations can provide insights about characteristics, such as timing constraints and capacity requirements and simulation enables the exploration of various designs.

6.1 Testing

Simulation enables firewall testing: for example, recorded sequences of datagrams can be played back as input to a CPN simulator modeling the behavior of a firewall design under consideration. Sequences of datagrams representing attacks can be constructed to determine how a firewall design can handle them. Furthermore, the testing of security policies is possible: an HCPN model would serve as a framework for the call to the function that calculates the policy decision (e.g., function `filter` with policy representation `acl` in figure 3). Observed behavior of the policy decision module can then be examined against expected behavior.

This approach is not sufficient to prove correctness of a system, but can at best reveal errors, similar to testing in software engineering ([22]).

6.2 Performance Analysis

The performance of firewall systems can be investigated through the use of *Timed Colored Petri Nets*. Several extensions to CPNs to introduce time are possible. Jensen's Timed CPNs are CPNs where places and transitions *consume* time and tokens are augmented by a time stamp. Time stamps contain the time after which a token is ready to be consumed by a transition. A global clock (discrete or continuous) keeps track of the simulation time. Simulations in timed CPNs are run analogously to discrete event simulations.

Values, such as "the average number of tokens in a given place" or "the average waiting time of a token in a given place," can be determined by simulations in timed CPNs. In our previous examples, tokens represented datagrams traversing a firewall system. The calculated values would therefore give designers profiling information, such as datagram delays in certain firewall functions. Estimates or measurements in a real world system must precede the simulation. The approach is useful in cases where analytical solutions through other formal approaches, such as Markov chains, cannot be obtained because their equation systems become too complex to solve ([23]). The introduction of timing information can result in infinite occurrence graphs (see section 7.1) for systems that have finite occurrence graphs otherwise ([23]). By specifying equivalence classes over the time domain one can limit these infinite occurrence graphs to finite subgraphs for which the dynamic properties and performance characteristics can still be determined ([23]).

6.3 Design Tool

Exploration of various designs is desirable because critical aspects of systems, such as single points of failure, can be determined. A design tool for firewall systems is expected to be beneficial in the early stages of firewall design: compared to prototyping, system simulation is a low cost alternative for design exploration. Furthermore, this approach is beneficial over a white paper evaluation because dynamic properties of components can be explored. Note that this approach is not specific to firewall design, but system design in general. McLendon and Vidale describe a similar approach in their research on modeling and analysis of an ADA system in [28].

7 Formal Analysis of CPNs

The early detection of design errors saves design time and costs ([7]). Jensen ([24, Ch.4,5]) lists a number of possible properties of Colored Petri Nets that can be analyzed by informal or formal analysis methods. The properties are divided into static and dynamic properties. Static (or structural) properties characterize CPNs without consideration of possible occurrence sequences while dynamic (or behavioral) properties characterize the behavior of instantiated CPNs. In general, dynamic properties are more difficult to verify than static properties, especially if one relies on informal methods. Formal analysis methods for dynamic properties often are of high computational complexity because they need to explore large combinatorial spaces.

7.1 Occurrence Graphs as the Basis for Analysis

Occurrence graphs are directed acyclic graphs (see [12, §5.4]). Their nodes represent the reachable markings of CPNs, and their arcs represent variable bindings between nodes. Their construction is a partially decidable problem. An algorithm exists that halts if and only if the occurrence graph is finite. Otherwise the algorithm does not terminate ([23, Prop. 1.4]).

The possible state explosion of occurrence graphs is a known problem (cf. [21, 28]). One can apply ad hoc reductions of occurrence graphs. However, those reductions usually do not preserve the behavior and properties of the original model. Jensen discusses in [22, §4.2] a variety of approaches for the reduction of occurrence graphs, such as by means of covering markings, by ignoring some of the occurrence sequences that are identical, by means of symmetries, or by expressing states as symbolic expressions.

Even if the occurrence graph is finite, its construction may still take a long time because occurrence graphs are generally large. The size of the graphs is dependent on several factors, such as the modeled problem or the required color sets and their domains. For example, the number of nodes in the occurrence graph for the dining philosophers problem as modeled in [23, §1.6] grows as a Fibonacci series, i.e., $N(n) = N(n-1) + N(n-2)$, where $N(2) = 3$ and $N(3) = 4$. The growth of Fibonacci numbers is exponential ([12, §2.2]).

The construction of the occurrence graphs is the dominant cost in the analysis of dynamic properties of CPNs. Many algorithms of interest to us, such as those described in [23] that operate on occurrence graphs, are of at most polynomial complexity. Therefore, the smaller the occurrence graph the lower the requirements for computation time. There are methods which reduce the size of occurrence graphs by exploiting symmetry and equivalence relations (see [23, Ch. 2,3]).

7.2 Invariants

Consider predicates which may be applied to the states of a system. A predicate is called an invariant if and only if it is valid in each state. Jensen explains the theory and use of invariants in [23, Ch. 4]. In CPNs there are place and transition invariants, and they are applied in the following way: First, equations are formulated which are postulated to be always satisfied. Second, it is proven that the equations are indeed satisfied. Third, the equations are then used to prove some of the dynamic properties of the modeled system (e.g., reachability, boundedness, home, liveness, and fairness). Place invariants are interpreted as sums of tokens which remain constant with the firing of transitions. Transition invariants deal with repetitive firing sequences.

Invariant analysis can prove structural properties of a CPN independent of its marking. Invariants have an advantage over occurrence graphs insofar as they avoid the possible state explosion.

Invariants over the number of datagrams in a net can be used to answer questions about firewall mechanisms, such as these:

- Did all datagrams that reach the final acceptance states originate in an authorized start state? A verified invariant to that extent assures that no transmission units that reach final states can get introduced into the firewall through means other than placement in initial states. In other words, firewall controls cannot be bypassed.

- Do certain attributes of transmission units adhere to a desired functional relationship? A first simple example of such a functional relationship is the identity function. It can be used to ensure that attributes, such as destination machine address and port numbers, do not change during net execution. A second example is a function mapping internal addresses to externally visible addresses, such as in *network address translation* (NAT) firewall mechanisms ([15]).

- Do all transmission units reach one of the defined final states representing acceptance or rejection? An invariant to that extent that holds would assure that no transmission units can get lost in the firewall implementation. Such a loss would be interpreted by the outside world as a possibly wrongful rejection of the datagram.

7.3 Static Analysis

Jensen defines in [24, §4.1] a set of static properties on arc expressions and transitions. A static analysis of the type of CPN models which are generated by our approach (e.g., in figures 1, 2, and 4) reveals that all arc expressions are *uniform* with *multiplicity* 1, all transitions are *uniform*, all transitions are *conservative*, all transitions, except transition SpiLkup have the *state machine property*, the primary page net in figure 2 is *open* because it has places as border nodes, and all secondary page nets are *closed* because they have transitions as border nodes.

The previous properties determine that the CPNs *as constructed* have a simple structure. Most transitions are conservative with the state machine property because they represent actions on single transmission units (for example datagrams). The transitions output single data items (for example a transmission unit augmented to a compound data structure by a result of the action taken: record *dgramvrfyr* as in transition AhVrfy).

Such a simple structure is preferable over a more complex structure because it adds less complexity to the occurrence graph. As we argued in section 7.1, smaller occurrence graphs are an advantage during the formal analysis of dynamic properties.

7.4 Dynamic Analysis

The dynamic analysis of CPNs explores properties, such as *boundedness*, *liveness*, *home markings*, *conservation*, *reachability*, *coverability*, *firing sequences*, *equivalence problems*, and *subset problems*. Definitions for these properties can be found in [32] and [24]. In the following we examine two of these properties in more detail: boundedness and liveness.

Safety properties stipulate some *bad* condition never occurs during the execution of a net. Examples for safety properties are boundedness, reachability, mutual exclusion, and freedom from deadlock.

Bad conditions can be represented by an assertion, P_{bad}, which is mapped to $true$ in exactly those states in which the condition is true. Therefore, if the safety property is true of a net, no occurrence sequence can contain such a state. Hence, the bad condition happens at some particular point in the execution. For a safety property to be true of a net, $\neg P_{bad}$ must be a net invariant. One way to demonstrate a safety property is to find a true program invariant I, so that $I \Rightarrow \neg P_{bad}$. Another way to express this idea is through the use of temporal logic ([35]). Temporal logic introduces two temporal operators on assertions: $\Box$ (always) and $\Diamond$ (eventually). A safety property can be expressed as: $\Box\neg P_{bad}$.

The *liveness* property stipulates eventually some *good* condition Q_{good} will occur during the execution of the net: $\Diamond Q_{good}$. Owicki and Lamport ([31]) presents a formal proof method based on temporal logic and proof lattices for deriving liveness properties of concurrent programs.

There is an interesting difference between safety and liveness: P_{bad} in a safety property must be a discrete event which occurs at some point in the execution while Q_{good} in a liveness property need not be discrete or occur at some particular point.

7.4.1 Boundedness

Upper (lower) bounds on places indicate the maximum (minimum) number of tokens of a particular color which can be in that place at a given time. The simple firewall model in sections 5 has no upper bounds imposed on its places. In particular, place Audit is not bounded.

The CPNs for the packet filter in figure 1 and the authentication header module in figure 4 are modeling mechanisms which in current implementations are effectively limited to dealing with one datagram at a time. This restriction places upper bounds of 1 on all but the initial (FltrRequest and AhRequest) and final places (FltrPassed, AhPassed, and Audit) of these CPNs. Places which represents external state, such as SpiDb, are subject to their own boundedness constraints.

One often needs to specify an event may happen only when a given condition does not hold, i.e., when the corresponding token is absent. The attempt to match a *dgramr* token to a *spir* token in transition SpiLkup in figure 4 can serve as an example. In the current model, if for a given token d no matching token is present in place SpiDb, token d cannot make progress towards places Audit or AhPassed. To solve the liveness problem in nets with unbounded places, Heuser and Richter suggest in [17] to use complementary places. The initial marking of a complementary place is the whole domain of the key of the token color to match. Tests for boundedness can discover these conditions.

Boundedness constraints are also useful to model limited resources. Bounded places imply finite capacities. Determining finite bounds of places can be used to gather information about resource requirements at those places.

7.4.2 Liveness

Liveness in a CPN means that a set of binding elements remains active. Liveness in a firewall representation can be interpreted as: every possible datagram starting out in place P1 will eventually reach a *final state* (representing acceptance or rejection of the datagram). This modeling approach implies that a datagram cannot disappear between its entry to the net and its reaching the final state.

8 Conclusion

This paper presented a method for modeling firewall components and firewall systems alike. It is based on a formalism that uses *Hierarchical Colored Petri Nets* (HCPN, short CPN) to describe the functionality of mechanisms used by firewall technology. HCPNs provide us with a theoretical framework and means of description, composition, simulation, and analysis of firewall systems.

We applied the formalism to a firewall system consisting of an IP packet filter followed by the IP authentication header module. By doing so we introduced CPN terminology and demonstrated how to model a network security mechanism given only its verbatim standard's specification. We built the model in a modular fashion and demonstrated how the hierarchical concepts of CPNs can be used to combine several mechanisms into a comprehensive firewall system. We learned how to model audit in CPN models. After we developed a basic modeling technique, we used the Design/CPN tool for the incremental building, syntax checking, and simulation of firewall models.

We discussed how the simulation of firewall models can be used for firewall and policy testing, for performance analysis, and as a basis for a design tool exploring design alternatives. Finally, we listed a number of static and dynamic (safety and liveness) properties defined for CPNs, which can be interpreted as desirable properties in the problem domain of firewall systems.

The final two sections of this paper present several areas for future research. For example, it may be beneficial to further investigate the question of which desirable properties of firewall systems can be expressed as dynamic properties, which in turn can be verified mechanically for HCPNs. We conjecture that concentrating on invariants as an analysis technique is likely to be a rewarding strategy: invariants are difficult to specify, but are more practical to verify. They avoid the problem of state explosion in occurrence graphs. Some of the properties that can be verified would allow designers and maintainers to gain additional confidence into a firewall system under investigation.

References

[1] R. Atkinson. *RFC-1825 Security Architecture for the Internet Protocol*. Network Working Group, Aug. 1995.

[2] R. Atkinson. *RFC-1826 IP Authentication Header*. Network Working Group, Aug. 1995.

[3] F. M. Avolio and M. J. Ranum. A Network Perimeter with Secure External Access. In *2nd Symposium on Network and Distributed System Security (NDSS)*, San Diego, California, Feb. 1994. Internet Society (ISOC).

[4] F. M. Avolio and M. J. Ranum. A Toolkit and Methods for Internet Firewalls. In *Technical Summer Conference*, pages 37–44, Boston, Massachusetts, June 1994. USENIX.

[5] M. L. Bailey, B. Gopal, M. A. Pagls, L. L. Peterson, and P. Sarkar. PathFinder: A Pattern-Based Packet Classifier. In *Proceedings of the 1st Symposium on Operating System Design and Implementation (OSDI)*, Monterey, California, Nov. 1994. USENIX.

[6] S. M. Bellovin and W. R. Cheswick. *Firewalls and Internet Security*. Addison-Wesley Publishing Company, Inc., 1994.

[7] F. P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley Publishing Company, Inc., second edition, 1995.

[8] C. J. Calabrese. A Tool for Building Firewall-Router Configurations. *The USENIX Association, Computing Systems*, 9(3):239–253, Summer 1996.

[9] D. B. Chapman. Network (In)Security Through IP Packet Filtering. In *Proceedings of the 3rd USENIX UNIX Security Symposium*, Baltimore, Maryland, Sept. 1992. USENIX.

[10] D. B. Chapman and E. D. Zwicky. *Building Internet Firewalls*. O'Reilley & Associates, Inc., Sebastopol, California, Sept. 1995.

[11] D. E. Comer. IP over ATM: Concept and Practice. Interop talk on IP over ATM, Mar. 1996.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[13] J. D. Day and H. Zimmermann. The OSI Reference Model. In *Proceedings of the IEEE*, volume 71, pages 1334–1340. IEEE, Dec. 1983.

[14] Digital Equipment Corporation (DEC). *Screening External Access Link (SEAL) Introductory Guide*, 1992.

[15] K. B. Egevang and P. Francis. *RFC-1631 The IP Network Address Translator (NAT)*. Network Working Group, May 1994.

[16] S. Garfinkel and G. Spafford. *Practical UNIX & Internet Security*. O'Reilley & Associates, Inc., Sebastopol, California, second edition, 1996.

[17] C. A. Heuser and G. Richter. Constructs for Modeling Information Systems with Petri Nets. In K. Jensen, editor, *13th International Conference on Application and*

Theory of Petri Nets, number 616 in Lecture Notes in Computer Science, Sheffield, UK, 1992. Springer Verlag.

[18] J. D. Howard. *An Analysis Of Security Incidents On The Internet 1989-1995*. PhD thesis, Carnegie Mellon University, Apr. 1997.

[19] P. Huber, K. Jensen, and R. M. Shapiro. Hierarchies in Coloured Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets*, number 524 in Lecture Notes in Computer Science. Springer Verlag, 1991.

[20] D. Icove, K. Seger, and W. VonStorch. *Computer Crime*. O'Reilley & Associates, Inc., Sebastopol, California, 1995.

[21] R. Janicki and M. Koutny. Optimal Simulations, Nets, and Reachability Graphs. In G. Rozenberg, editor, *Advances in Petri Nets*, number 524 in Lecture Notes in Computer Science, pages 205–226. Springer Verlag, 1991.

[22] K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets*, number 524 in Lecture Notes in Computer Science. Springer Verlag, 1991.

[23] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*, volume 2. Springer-Verlag, New York Inc., 1995.

[24] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*, volume 1. Springer-Verlag, New York Inc., second edition, 1996.

[25] K. Jensen. *Design/CPN Overview of CPN ML Syntax. Version 3.0*, 1996.

[26] C. E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3):247–278, Sept. 1981.

[27] D. Longley and M. Shain. *Data & Computer Security. Dictionary of Standards, Concepts, and Terms*. Macmillan Publishers Ltd., 1987.

[28] W. W. McLendon, Jr. and R. F. Vidale. Analysis of an Ada System Using Coloured Petri Nets and Occurrence Graphs. In K. Jensen, editor, *13th International Conference on Application and Theory of Petri Nets*, number 616 in Lecture Notes in Computer Science, pages 384–388, Sheffield, UK, 1992. Springer Verlag.

[29] Meta Software Corporation. *Design/CPN Reference Manual*. Cambridge, Massachusetts, 1993.

[30] R. Milner. The Standard ML Core Language. Technical Report CSR-168-84, Edinburgh University Internal Report, 1984.

[31] S. S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, pages 455–495, July 1982.

[32] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[33] C. A. Petri. Kommunikation mit Automaten. Technical Report 2 (Schriften des IIM), Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[34] H. Plünnecke and W. Reisig. Bibliography on Petri Nets 1990. In G. Rozenberg, editor, *Advances in Petri Nets*, number 524 in Lecture Notes in Computer Science. Springer Verlag, 1991. Over 4000 references to publications dealing with Petri Nets.

[35] A. Pnueli. The Temporal Logic of Programs. In *18th Symposium on the Foundations of Computer Science*, pages 46–57, Nov. 1977.

[36] J. Postel, editor. *RFC-791 Internet Protocol*. Information Science Institute, University of Southern California, Sept. 1981.

[37] J. Postel, editor. *RFC-792 Internet Control Message Protocol*. Information Sciences Institute, University of Southern California, Sept. 1981.

[38] K. E. Price. Host-Based Misuse Detection and Conventional Operating Systems' Audit Data Collection. Master's thesis, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, Dec. 1997.

[39] M. J. Ranum. A Network Firewall. In *Proceedings of the 1st International Conference on Systems and Network Security and Management (SANS-I)*, June 1992.

[40] M. J. Ranum. Internet Firewalls — An Overview, Oct. 1993. (unpublished).

[41] M. J. Ranum. Thinking About Firewalls. In *Proceedings of the 2nd International Conference on Systems and Network Security and Management (SANS-II)*, Apr. 1993.

[42] M. J. Ranum, A. Leibowitz, B. Chapman, and B. Boyle. Firewalls-FAQ, 1994.

[43] C. L. Schuba. *On the Modeling, Design, and Implementation of Firewall Technology*. PhD thesis, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, Dec. 1997.

[44] C. L. Schuba and E. H. Spafford. A Reference Model for Firewall Technology. In *Proceedings of the 13th Annual Computer Security Applications Conference (ACSAC)*, pages 133–145, San Diego, California, Dec. 1997. IEEE Computer Society.

[45] R. Sethi. *Programming Languages. Concepts and Constructs*. Addison-Wesley Publishing Company, Inc., 1990.

[46] M. Silva and R. Valette. Petri Nets and Flexible Manufacturing. In G. Rozenberg, editor, *Advances in Petri Nets*, Lecture Notes in Computer Science. Springer Verlag, 1989.

[47] K. Siyan and C. Hare. *Internet firewalls and network security*. New Riders Pub., Indianapolis, Indiana, 1995.

[48] Å. Wikström. *Functional Programming Using Standard ML*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[49] N. Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 14(4):221–227, Apr. 1971.

[50] M. Young. Private communication, 1997.