A Secure Programming Paradigm for Network Virtualization (Invited Paper)

Ana Milanova, Sonia Fahmy, David Musser, Bulent Yener

***Abstract*—The central paradigm of today's successful Internet is to keep the network core simple and move complexity towards the network end points. Unfortunately, this very paradigm limits network management and control capabilities, and creates opportunities for attacks such as worms, viruses, and spam that often seriously disrupt and degrade Internet and user performance. The thrust of this paper is that such problems cannot be effectively solved unless a paradigm shift is adopted. Towards a more secure and manageable Internet, we propose "virtualization" of the Internet, by carefully balancing its scalability and programmability properties. Our objective is to provide a programmable *virtual Internet* to users and to let them manage, control, and optimize it based on their individual needs.**

I. Introduction

The 20^{th} century Internet was based on keeping the network core simple, and pushing complexity to the hosts at the network edge. This was in clear contrast with the telecommunications paradigm in which the network core was complex (e.g., consider AT&T telephone switch software with millions of lines of code). Given the heterogeneity, volume, pervasiveness, and autonomy of 21st century Internet communications, a paradigm shift must balance these two extreme approaches: the core must support more sophisticated functionality than simply forwarding packets while remaining scalable. The current architecture in which network nodes perform the functions defined by the first three layers of the OSI architecture is insufficient for addressing the new applications enabled by advances in micro-electronics (pervasive, heterogeneous devices with wireless links, powerful servers) and optics (Terabit bandwidths).

An important paradigm shift to address these requirements is the *virtualization* of the Internet, providing a *programmable virtual Internet per user* and letting the user manage it (in a similar spirit to IBM VM). Current (limited) virtualization examples include overlay networks, virtual private networks, the mbone, and peer-to-peer networks. One common theme in these examples is the need to enhance the networking and computing capabilities of the current TCP/IP-based Internet. Therefore, we propose to extend virtualization to Internet users to enable them to create, use, and manage their own 21st century Internet.

Consider the following two security-related examples. In a standard firewall system, firewall filtering is performed at the application layer at a network end point. Clearly, a dropped packet has already wasted network resources and competed with other traffic to reach the firewall, only to be dropped. Similarly, spam filtering is performed at the client application layer, after the spam packets have already wasted network resources (spam accounts for 60% of Internet traffic by some accounts). Our proposed paradigm would allow a user to create, use, and manage *virtual firewalls* that are close to the origin of the packets. Virtual firewalls would require *programmability* at remote network entities that are owned and controlled by other enterprises. For example, a virtual firewall capability can be supported by a router enhanced by a `Vfirewall()` daemon, which is simply a process that monitors a predefined port. It may receive requests from users all over the network to maintain a virtual firewall. User requests may come in the form of executable programs that carry proofs of correctness. Similarly, virtualization can enable a user to define his/her own spam filter, and transfer an executable program with its proof to an edge router that is near the origin of the spam traffic. The proof must not only ensure that the program is executed on behalf of a legitimate and authorized user, but also that it does only what it is specified to do (i.e., spam filtering for that particular host in this example).

In both examples, virtualization distributes some of the functions at the client side to the source or server side by delegating these functions to edge routers. A natural outcome of this paradigm shift is the service agreement made between the client and the edge router at the server side. As a result, a remote edge router may charge a fee for maintaining a virtual firewall or spam filtering for a client, and thus can be held accountable.

Executing end-user programs at network entities must be both *secure* and *scalable*. Central to this approach towards programmability, therefore, is the development of verification technology. Within this technology, a programmable router states a *safety policy* that each individual program must conform to. Intuitively, the safety policy is a contract between the end user (also referred as the *client*) and the programmable router (also referred as the *server*) that specifies certain security and scalability constraints. Each client program must conform to these security and scalability constraints. We address the scalability problem that arises when multiple clients may program a server by limiting programmability. Programmability is also inherently constrained in our model as we allow only the edge routers to be programmed.

The primary challenge discussed in this paper is the development of technology that will verify that each client program conforms to the safety policy. We propose to utilize *proof carrying code (PCC)* [15], [32], [31] which has important advantages over related verification techniques (e.g., static analysis, model checking, interpretation). With PCC, the client program to be executed comes with a proof generated by the client, and the server need only check this proof. Thus, the majority of work on verification lies with the code-producing client rather than with the server, which is important to ensure scalability at the

– This research has been sponsored in part by NSF grant 0523249.

– Ana Milanova, David Musser, and Bulent Yener are with the Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th St., Troy, NY 12180–3590, USA. E-mail: {milanova, musser, yener}@cs.rpi.edu. Sonia Fahmy is with the Department of Computer Science, Purdue University, 305 N. University St., West Lafayette, IN 47907–2107, USA. E-mail: fahmy@purdue.edu.

router. We need to advance the current state in PCC—especially with respect to scalability—by building and exploiting a library of proofs of general lemmas, thereby permitting relatively compact proofs to be constructed and transmitted along with new code being delivered to network nodes. The following key research problems are discussed in this paper:

- **Programmability.** The first major research problem is to specify how the client can program the server. We explore the development of a *library* of packet manipulation routines to be used as building blocks of the client programs. We also discuss a simple yet expressive language that will allow the composition of these routines into client programs.
- **Verification Technology.** With PCC, the first goal is to define a safety policy that will specify appropriate security and scalability constraints on client programs. The second goal is to generate compact proofs that can be efficiently transmitted, and to provide efficient checking.
- **Client-server Protocols and Scheduling.** Another major issue is how to control multiple clients that attempt to execute code on a single server. We examine secure protocols that will allow the client to transmit the program, the proof, and a payment for execution. We also discuss the scheduling algorithm at the server side.
- **Execution Environment.** It is important to build an efficient kernel at the server that will efficiently execute programs built on top of the library routines. We also discuss how to constrain programmability: clearly, it is infeasible to allow the entire set of possible clients to program the router simultaneously.
- **Emulation Technology.** Emulation is an attractive prototyping and evaluation platform for our ideas. Router performance must be studied under different loads, in addition to robustness under threats, placement and partial deployment of programmable routers, and potential gains for applications.

The remainder of this paper is organized as follows. Section II gives an overview of our architecture and the research challenges associated with it. Section III discusses how to prototype and evaluate the architecture. Section IV summarizes related work. Finally, Section V gives brief concluding remarks.

II. Towards a Virtual Internet

An overview of the architecture of the virtual internet is given in Figure 1. To ensure scalability, we consider programmability at the edge routers only; in the future, we plan to investigate how to extend the technology to provide programmability at the network core routers as well. Consider the Virtual Internet (VInet) Client Module at the end user (referred to as client). The Middleware VInet System is an application-layer system that provides the interface to the programmable network and the view to the library routines that perform packet manipulation (these routines are described in detail in Section II-A). Generally, the user will be able to specify certain parameters in a user-friendly environment and the system will generate the appropriate program. For example, for a virtual firewall, the user will specify the IP-address(es) to be filtered out and some information to help identify the edge router that will perform the filtering. Similarly, for a virtual spam filter, she/he will specify the words that determine a high spam score and the edge router close to the source of the spam that must perform the filtering. The Middleware VInet System will be responsible for (1) obtaining the Safety Policy of the edge router, (2) generating the executable P, in bytecode form, (3) generating the proof $Pr(P)$ that the bytecode conforms to the Safety Policy of the server (this is done by the Proof Compiler in Figure 1), and (4) uploading the program P and the proof $Pr(P)$ at the server through a secure connection which also includes a payment. Note that the Middleware should be able to generate programs based on user-specified parameters, but it should also allow experienced users to write entire programs as well as to edit generated programs.

Consider the Virtual Internet (VInet) Server Module in Figure 1. At the edge router there is a Scheduler unit which controls the incoming programs. If a program P is successfully scheduled for execution, the Proof Checker checks the proof $Pr(P)$ against its Safety Policy. If successful, i.e., the program conforms to the constraints specified in the Safety Policy, the server optimizes the code and executes it using the efficient implementation of the packet manipulation procedures.

This architecture assumes that PCC is used to perform verification. If we employ static analysis, the entire burden of verification lays with the server. The Proof Compiler will be dropped from the VInet Client Module, and the Proof will be dropped from the transferable bytecode. At the VInet Server Module, the Proof Checker will be substituted with a Static Analyzer that will verify the conformance of the bytecode with the Safety Policy.

A. Programmability

Programmability is based on a library of packet manipulation routines accessible to the user at the client side. The user will build programs using these packet manipulation routines; the programs will be network-centric rather than computation-centric—that is, they will perform mainly packet manipulation and minimal computation. The language will consist of three kinds of statements: calls to the routines, text processing statements, and control-flow statements (e.g., IF-THEN-ELSE and WHILE). We describe the routines and give concrete examples of programs that perform firewalling, spam filtering, and multicasting.

Packet Manipulation Routines. A sample set of routines that will provide useful functionality from the point of view of the client is shown in Table I. The routines provide the interface to the programmable router that is provided to the end-user. The semantics of most procedures is straight-forward. For example, DISCARD(Packets) drops the packets in set Packets. GENERATE(SrcIP,DestIP,Content) forms one or more new packets with Content and SrcIP as a source IP and DestIP as a destination IP; these packets are subsequently sent to DestIP. GROUP_BY_DESTINATION(Packets) partitions Packets by DestIP—that is, set Packets(i)$\subseteq$ Packets is the set of packets sent to the i^{th} destination IP, DestIP(i). Procedure GROUP_BY_CONTENT(Packets) (needed in the multicasting example below) assumes that Packets contains multiple packets with the same content and different destinations. It extracts the destination IPs into DestIPs and the content into Content. For example, if Packets is viewed as an M×N matrix

[S,D1,C1],[S,D2,C1]...,[S,DN,C1]
[S,D1,C2],[S,D2,C2]...,[S,DN,C2]

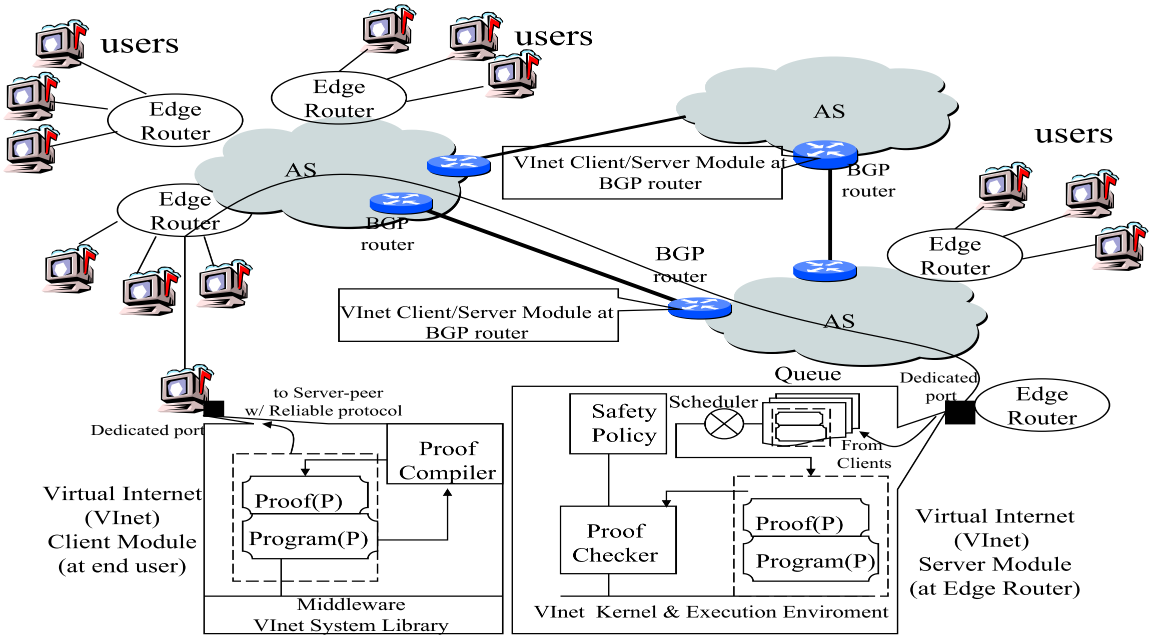

Fig. 1. The Virtual Internet Architecture.

```
...
[S,D1,CM],[S,D2,CM]...,[S,DN,CM]
```

(i.e., source S sends each content CI to destinations D1...DN). GROUP_BY_CONTENT(Packets) extracts the content: Content = {C1,C2,...CM}, and the destinations: DestIPs = {D1,D2,...,DN}.

Applications. To make our ideas more concrete, we proceed with three examples that demonstrate the kinds of programmability an end user would want. Consider a **virtual firewall** that will be installed at an edge router that is closer to the IPs whose traffic the client wants to filter out; it uses the routines in Table I.

```
LIFETIME = 2 days
MyPackets = GROUP_BY_DESTINATION(AllPackets,THIS_IP);
BadPackets = GROUP_BY_SOURCE(MyPackets,SrcIP);
DISCARD(BadPackets);
```

This code is self-explanatory. First, the client declares the desired lifetime of the program—how long should it run at the server (later the lifetime will be used to determine the cost of the program). Subsequently, the client identified by THIS_IP requests that all packets from SrcIP are dropped. The program and the proof that will be generated at the client VI Module would verify that the program manipulates only packets addressed to THIS_IP. When the client uploads the code at the router, the router must verify the authenticity of THIS_IP and check the proof against THIS_IP. Another example is a simplified **virtual spam filter** which filters out strings Word1 and Word2:

```
LIFETIME = 5 days
MyPackets = GROUP_BY_DESTINATION(AllPackets,THIS_IP);
SourceIP(i),Packets(i) = GROUP_BY_SOURCE(MyPackets);
foreach i do
    if CONTAINS_STRING(Packets(i),Word1)
and CONTAINS_STRING(Packets(i),Word2) then
        DISCARD(Packets(i));
```

Our final example using the procedures in Table I is a **multicasting** program to be executed on behalf of an edge router which is to receive a bandwidth consuming broadcast from BroadcastIP:

```
LIFETIME = 5 hrs
MyPackets = GROUP_BY_DESTINATION(AllPackets,THIS_SUBNET_IP);
Broadcast = GROUP_BY_SOURCE(MyPackets,BroadcastIP);
Destinations,Content = GROUP_BY_CONTENT(Broadcast);
GENERATE(BroadcastIP,THIS_IP,Destinations);
GENERATE(BroadcastIP,THIS_IP,Content);
DISCARD(Broadcast);
```

This simplified example first extracts the packets coming from the broadcast IP, then extracts the destinations of the multicast in Destinations, and the content into Content. Subsequently, the set of destinations is sent to the edge router by generating the new packets that contain these destinations. The content is sent *only once* in the newly generated set of packets while the old Broadcast packets are discarded.

Another application is to defend against the spread of infectious software such as Internet worms. The impact of such malicious software depends on the spreading rate which must be above the epidemic threshold. Virtualization can be used to build distributed firewalls upon detecting a worm spread within a domain. This can be done by deploying both client and server modules at BGP routers and running the firewall programs at the server side as explained above.

B. *Verification Technology*

The role of the verification technology is to ensure that a program executed at the server is secure and scalable. Therefore, we have to formally specify security and scalability constraints; informally, this is a contract between the client and the server:

(1)Procedure	(2)Functionality
DISCARD(Packets)	Discards the set of packets Packets
GENERATE(SrcIP,DestIP,Content)	Generates and sends packet(s) with content Content from SrcIP to DestIP
GROUP_BY_DESTINATION(Packets,DestIP):Packets	Extracts the packets with DestIP
GROUP_BY_DESTINATION(Packets):Packets(i),DestIP(i)	Groups the packets into groups by DestIP
GROUP_BY_SOURCE(Packets,SrcIP):Packets	Extracts the packets with SrcIP
GROUP_BY_SOURCE(Packets):Packets(i),SrcIP(i)	Groups the packets by SrcIP
GROUP_BY_CONTENT(Packets):DestIPs,Content	Groups the DestIPs for packets with content Content
CONTAINS_STRING(Packets,String):boolean	Checks content for occurrence of String

TABLE I
PACKET MANIPULATION PROCEDURES.

each client program must conform to these constraints in order to be executed at the server.

Safety Policy. The safety policy specifies the security and scalability constraints. An intuitive *security constraint* is to restrict a client, identified by its IP address, into programming only its own traffic. Let us denote the client IP by THIS_IP. The proof compiled and attached to the program will verify that the program attempts to manipulate only packets with destination THIS_IP. When the program and the proof are uploaded, the server verifies the authenticity of THIS_IP, and the Proof Checker checks the proof against THIS_IP. We informally state part of this safety policy: *A client identified by THIS_IP can discard only packets with destination IP equal to THIS_IP.* A formal specification of this safety policy using the denotational proof language of the Athena proof system (`www.cag.csail.mit.edu/∼kostas/dpls/athena/`) is given in Figure 2 (the specification is syntactically sugared for readability).

We can extend the safety constraints specified in the above policy to allow one edge router to act as the client and program another edge router to act as the server. The safety policy would state that the client may discard, group and scan only packets with destinations within its own subnet. Additionally, an end user or an edge router may generate packets only to its own IP. We have actually specified this extension in Athena but for brevity have omitted it from Figure 2.

So far, we have only considered security constraints. A major research challenge is to identify the relevant *scalability constraints*, and to specify them within the safety policy. We can address the issue of scalability by assigning *complexities* to each packet manipulation routine. The complexities reflect the cost of an individual routine in terms of server resources (e.g., processor and memory). For example, DISCARD will be relatively inexpensive, while CONTAINS_STRING and GROUP_BY_CONTENT will be relatively expensive as they require the server to store a window of packets and perform operations on them. Based on these, the complexity of the entire program P, denoted by $Complexity(P)$ will be computed by adding the complexities of the individual routines while appropriately accounting for control-flow constructs (e.g., IF-THEN-ELSE and WHILE). The total *cost* of P, denoted by $Cost(P)$ will be computed as a function of $Complexity(P)$ and the *lifetime* specified in the program. For example, we can take the product of these two components and have $f(Complexity(P), lifetime)$=$Complexity(P) \times lifetime$. The safety policy will specify upper bounds, $Complexity^{up}$ and $Cost^{up}$ on the complexity and cost of an individual program. The client must guarantee as part of the proof that his/her program has complexity $Complexity(P) \leq Complexity^{up}$ and cost $f(Complexity(P), lifetime) \leq Cost^{up}$. Assuring these bounds on individual programs would allow us to establish bounds on multiple programs running simultaneously which is essential for the development of a scalable multi-user environment.

Proof Generation and Proof Checking. Recall that the main role of the VInet Client Module is to compile the program into bytecode and to generate the proof. We continue with the firewall example from Section II-A. Using the Athena proof system, we have created a model of a simple filtering program (essentially the firewall example), and a proof that the program conforms to the Safety Policy specified in Figure 2. The model and the theorem `EndUser-IP1-observes-policy` are shown on the left-hand side in Figure 3; the proof of theorem `EndUser-IP1-observes-policy` is shown on the right-hand side in Figure 3. Subsequently, the proof is checked by the proof checker at the VInet Server Module, and, if checking is successful, the program is scheduled for execution.

An important research problem to address is how to create proofs that are compact and can be transmitted and checked efficiently. The technique we propose to solve this problem is to build a library of theorems of such general utility that they are usable as lemmas in many other proofs. Thus the proofs of these general theorems can be transmitted only once and stored in a local copy of the library in a network node for later use (whenever needed in completing proofs of the theorems that invoke them). This approach follows exactly the same principles of modularity in programming; in fact, in the Athena system proofs *are* a form of program and can be modularized in the same way as ordinary programs via a form of parameterized subroutines (called proof methods). We are following this approach in our current work on PCC in which we are developing proofs of functional correctness and safety properties of STL-like generic algorithms. We currently have a small collection of a few hundred theorems and proofs (several thousand lines of proof “code”), of which more than 90% form a reusable pyramid-like base library. Thus, in a typical scenario, only a few top-level, specialized theorems and proofs would have to be transmitted along with a new pro-

```
(domain IP)
(domain Content)
(datatype Packets (Packet IP IP Content))
(declare EndUser ((IP) → Boolean))
(declare Owns ((IP IP) → Boolean))
(declare CanDiscard ((IP Packets) → Boolean))
(declare Discard (((List-Of Packets)) → (List-Of Packets)))
(domain NetworkState)

(declare AnyFilterWith
      ((IP (List-Of Packets) (List-Of Packets))
      → NetworkState))
(define EndUser-owns-only-own-IP
  (forall ?ThisIP
    (if (EndUser ?ThisIP)
      (forall ?IP ((Owns ?ThisIP ?IP) iff (?IP = ?ThisIP))))))
(define CanDiscard-only-owned-IPs
  (forall ?ThisIP ?From ?To ?Content
    ((CanDiscard ?ThisIP (Packet ?From ?To ?Content))
      iff (Owns ?ThisIP ?To))))
(define Discard-axiom
  (forall ?ThisIP ?From ?To ?Content ?Traffic
    ((Discard (Cons (Packet ?From ?To ?Content) ?Traffic))
      = ?Traffic)))
(assert EndUser-owns-only-own-IP
   CanDiscard-only-owned-IPs
   Discard-axiom)
```

```
##########################################
A safety policy required for particular end-users
##########################################

(define If-discard-not-allowed-then-does-not-happen
  (forall ?ThisIP ?From ?To ?Content
    (if (not (CanDiscard ?ThisIP
                  (Packet ?From ?To ?Content)))
      (forall ?Incoming ?Outgoing
        ((AnyFilterWith ?ThisIP
          (Cons (Packet ?From ?To ?Content)
               ?Incoming)
          ?Outgoing)
         = (AnyFilterWith ?ThisIP
          ?Incoming
          (Cons (Packet ?From ?To ?Content)
               ?Outgoing)))))))
```

Fig. 2. Formal Requirements and Safety Policy for Packet Filtering Programs.

gram, while the rest could then be extracted and executed from the recipient's local library.

Static Analysis. An alternative verification technology is static analysis, where the client uploads the code at the server and the server verifies the security and scalability constraints using its Static Analyzer. The Static Analyzer will translate the code into Control Flow Graph (CFG) representation which is useful for data-flow analysis. Using the CFG, one can define a data-flow analysis that would check conformance to the safety policy. As a concrete example, recall the firewall program in Section II-A. The CFG of this program would be fairly simple: node 1 corresponds to the first statement (MyPackets=...), node 2 corresponds to the second statement (BadPackets=...), and node 3 corresponds to the third statement (DISCARD(...)); there are edges from node 1 to node 2 and from node 2 to node 3. We need to define an analysis on the CFG that would verify the following two facts: (1) that actual argument MyPackets in the call at node 2 contains only packets addressed to the IP of the client, and (2) that actual argument BadPackets at node 3 contains only packets addressed to the IP of the client (that is, the client manipulates only its own traffic). Assume that argument THIS_IP at node 1 is equal to the IP of the client that uploads the program (i.e., the program is safe). The verification of facts 1 and 2 can be done by classical Reaching Definitions analysis [2]. The only definition that reaches node 2 is (MyPackets,1) (i.e., the MyPackets that flow to node 2 are the ones produced at node 1). The analysis examines the call at node 1 and since THIS_IP=IP, it concludes that fact 1 holds; this conclusion takes into account the semantics of GROUP_BY_DESTINATION. Similarly, the definition that reaches node 3 is (BadPackets,2). The analysis examines the call at node 2 and since MyPackets contains only appropriate packets, it concludes fact 2 as well; again, the conclusion takes into account the semantics of GROUP_BY_SOURCE. One of the main research problems would be to construct analyses that would determine the cost of the program and verify the scalability constraint in the safety policy. Note that with static analysis, one can use the verification passes over the CFG to perform program optimizations as well—this cannot be done with PCC.

C. Client-Server Protocols and Scheduling

A server node will simultaneously execute multiple programs by multiple clients. Therefore, a protocol is required for communication between the client and the server, as well as a scheduling algorithm that will choose a client program for execution.

Client-Server Protocols. Communication is done over a reliable and secure connection between specific port numbers of network interface cards (NICs) between the client and server. We assume for now that there is a single client/server module in order to avoid race conditions and deadlocks. The client end-user and the server edge router will establish a session key (e.g., by using the Diffie-Hellman (DH) algorithm over a reliable connection). Once the shared key is constructed, all the communication between the client and the server will be in encrypted form. The shared keys can be stored at the servers once they are generated. We emphasize that one of our future research goals is to investigate the tradeoffs between scalability and performance. One can reduce the amount of state information at edge routers while increasing the computational cost. For example, it is possible to store the shared key between a pair of client-server nodes at the client side to reduce the complexity at the servers as follows: Upon establishing a shared key $K_{i,j}$ with a client i, server node j encrypts $K_{i,j}$ and a sequence number (initially set to zero at both parties) using his public key K_{P_j}, and transmits encrypted message m to the client node. The client encrypts the consecutive messages and increments the sequence number by the shared key $K_{i,j}$, and appends m to the message M sent to the server node. Upon receiving message M, server j removes m and decrypts it using its private key K_{R_j} to obtain the shared key $K_{i,j}$. The server checks the sequence numbers to prevent re-

```
(declare IP1 IP)
(assert (EndUser IP1))
(declare Filter1
  (((List-Of Packets) (List-Of Packets)) → NetworkState))
(define Filter1-relation-to-AnyFilterWith
  (forall ?Incoming ?Outgoing
    ((Filter1 ?Incoming ?Outgoing)
     = (AnyFilterWith IP1 ?Incoming ?Outgoing))))
(assert Filter1-relation-to-AnyFilterWith)
(define EndUserRule
  (forall ?From ?To ?Content ?Incoming ?Outgoing
    (if (not (?To = IP1))
      ((Filter1 (Cons (Packet ?From ?To ?Content)
                      ?Incoming)
                ?Outgoing)
       = (Filter1 ?Incoming
                  (Cons (Packet ?From ?To ?Content)
                        ?Outgoing))))))
(assert EndUserRule)
##########################################
(define EndUser-IP1-observes-policy
  (forall ?From ?To ?Content
    (if (not (CanDiscard IP1 (Packet ?From ?To ?Content)))
      (forall ?Incoming ?Outgoing
        ((AnyFilterWith IP1
            (Cons (Packet ?From ?To ?Content)
                  ?Incoming)
            ?Outgoing)
         = (AnyFilterWith IP1
             ?Incoming
             (Cons (Packet ?From ?To ?Content)
                   ?Outgoing)))))))
```

```
(!(conclude EndUser-IP1-observes-policy)
   (pick-any From To Content
    (assume (not (CanDiscard IP1 (Packet From To Content)))
      (pick-any In Out
        (!cases
          (assume (To = IP1)
           (!by-contradiction
             (assume (not ((AnyFilterWith IP1
                              (Cons (Packet From To Content) In) Out)
                           = (AnyFilterWith IP1 In
                               (Cons (Packet From To Content) Out))))
               (dseq
                 (!(conclude (Owns IP1 To))
                  (!right-instance
                     (!mp (!uspec* EndUser-owns-only-own-IP [IP1])
                          (EndUser IP1)) [To]))
                  (!absurd
                    (!right-instance CanDiscard-only-owned-IPs
                                     [IP1 From To Content])
                    (not (CanDiscard IP1 (Packet From To Content)))))))))
          (assume (not (To = IP1))
            (dseq
              (!setup left (AnyFilterWith IP1
                              (Cons (Packet From To Content) In) Out))
              (!expand left (Filter1 (Cons (Packet From To Content)
                                           In) Out)
                    Filter1-relation-to-AnyFilterWith)
              (!reduce left (Filter1 In (Cons (Packet From To Content)
                                              Out)) EndUserRule)
              (!reduce left (AnyFilterWith IP1
                              In (Cons (Packet From To Content) Out))
                    Filter1-relation-to-AnyFilterWith)))))))))
```

Fig. 3. A model of a filtering program and the proof of the Safety Policy in Figure 2.

play attacks. Clearly, with this scheme the server does not need to store keys, but it needs to perform additional decryption on each message.

Scheduler. We assume that the Virtual Internet Server module maintains a data structure for the *classes* of programs that can be executed at the edge router. The class of a program is determined by its cost. Recall that the cost of the program should be verified and specified as part of the proof, and it should not exceed the predefined complexity and cost bounds (i.e., the scalability constraint in the safety policy). Furthermore, each virtualization program carries an indicator to announce to the scheduler which class it belongs to—that is, it announces its total cost and its complexity. Note that the client would not have an incentive to lie to the scheduler because the verifier (i.e., the Proof Checker or the Static Analyzer depending on the verification technique) will check the cost and will discard the program without execution in case of discrepancy.

Virtualization requests from different network interfaces can be processed at the NICs as follows: Each request will be masked to determine its class and inserted into a dedicated queue of that class with FIFO discipline. This will be done by time-stamping the incoming request. The queues of the same class at different interfaces are merged at the server module to maintain the FIFO property (since the requests are time-stamped, this can be done easily).

There is a priority assignment over the queues of different classes. The simplest prioritization scheme would be to assign a payment to a request for program execution where the payment would be proportional to the program cost; the highest payment/cost programs will have the highest priority. Thus, the scheduler at the server module would implement a single server class-based priority queuing service. In order to achieve scalability, we require that the server edge router has a predefined *capacity* (reflecting router resources such as processor and memory). The scheduler may schedule a program only if program complexity (i.e., the contribution to the load of the currently running programs) is no greater than the difference between the router capacity and the current load. We are extending our previous work in [21] to design online algorithms that take advantage of lookahead operations in the queues.

D. The Execution Environment

The library of packet manipulation routines must be implemented in a low-level efficient manner and extensively verified. Intuitively, each program uploaded at the server edge router manipulates the connection from a SourceIP (controlled by that edge router) to the DestinationIP of the client. Note that since we aim to achieve virtualization *per user*, maintaining state information per client IP is an inherent “lower bound” on the size of the execution environment. An important research question therefore is how to combine the ideas of individual program constraints and total router capacity which restrict programmability, with *adaptive optimizations* that combine similar programs, which would help optimize resource usage and broaden programmability.

Constraining Programmability. The architecture of the environment can use a dynamic table indexed by SourceIP and then by DestinationIP as shown in Figure 4. SourceIP is drawn from the set of all IPs controlled by the edge router; note that the number of such IPs is substantially smaller than the total number of Internet IPs However, DestinationIP is drawn from the entire set of Internet IPs. We propose to use hash tables due to the fact that checks for keys are typically fast (i.e., it will typically take constant time to see if a pair ⟨SrcIP,DestIP⟩ is in the table); however, we emphasize that the choice of the right data structure is an important research problem that must be investigated thoroughly both theoretically and empirically.

Suppose that the program currently being scheduled manipulates the connection from a given SrcIP to a given DestIP. Cell `sources.get(SrcIP).get(DestIP)`, which will be created if needed, will be set to refer to that program. Subsequently, the total cost of the programs that are running at the moment will be increased by $Complexity(P)$. Recall from Section II-C that P is chosen by the scheduler only if $Complexity(P)$ does not exceed the difference between the router capacity and the total cost of the simultaneously running programs. Further, when a program at cell ⟨SrcIP,DestIP⟩ finishes execution, the cell will be freed and the total cost of the simultaneously running programs will be decreased by $Complexity(P)$.

Adaptive Optimizations. Constraints on individual programs and on all simultaneously running programs (driven by router capacity) restrict programmability. To mitigate this problem, we propose to use adaptive optimization that will allow individual programs to share resources. When a program is scheduled, it is analyzed and if there is a program that has the same functionality already running on behalf of another user, the new cell is simply set to refer to that program. With this scheme, the cost of the currently running programs will not be incremented. For example, suppose that a virtual firewall that filters out SrcIP1 runs on behalf of DestIP1. When the router schedules a virtual firewall that filters out SrcIP1 on behalf of DestIP2, cell ⟨SrcIP1,DestIP2⟩ will be set to refer to the program at cell ⟨SrcIP1,DestIP1⟩ and no additional processes will be taken.

To further increase flexibility and scalability, the programs may be grouped by functionality classes for each SourceIP, and the router may assign *quotas* for each functionality class—that is, at most N DestinationIPs may upload programs of class C that manipulate traffic from a given SourceIP. For example, we will define a class *Virtual Filtering* which will include all programs that DISCARD packets—that is, the class will include virtual spam filters and virtual firewalls. Clearly, if a large number of DestinationIPs are willing to pay to turn a SourceIP off, this is an indication that SourceIP is sending malicious packets. It is likely that the computer was actually "highjacked" and was part of a bot network. The router will shut SourceIP off for everyone and alert its user who will have to check the computer in order to get back on the net. Note that this adaptive scheme designed to help scalability will help combat distributed attacks as well.

III. Experimental Validation

A prototype virtual Internet system including both the client middleware and the programmable router is being designed. We are considering the MIT Click router [27], [8] as a possible basis for building our programmable routers. We will systematically test our proposed system for performance, as well as security and scalability under certain threat (malicious user behavior) models, against specified safety policies. We will also evaluate the gains for the example applications we discussed above.

In order to validate our results in a flexible and yet high fidelity setting, we can utilize the DETER (www.isi.deterlab.net) and Emulab (www.emulab.net) testbeds. Emulab is a universally-available time- and space-shared network emulator located at the University of Utah. The system is comprised of hundreds of linked computers (PCs) that can be connected in almost any specified topology, and a suite of software tools that manage them. The Cyber Defense Technology Experimental Research Network (DETER) is an experimental testbed – based on Emulab – that allows researchers to test and evaluate Internet cyber security technologies in a realistic, but safe environment. This environment can be accessed remotely, but is *quarantined* from the Internet. The Evaluation Methods for Internet Security Technology (EMIST) project, in which we are participating, is a companion project that designs testing methodologies and benchmarks for the DETER testbed [7].

We believe that emulation on DETER will be ideal for experiments with the virtual Internet, as it is more flexible than a hardware testbed, yet more realistic than a simulator. On the one hand, using an emulator is significantly more flexible than building a hardware testbed in our labs, since almost arbitrary topologies can be specified without rewiring machines, and the testbed is maintained by professional and experienced staff. On the other hand, an emulation environment affords much higher fidelity than a simulator, and this can expose unforeseen vulnerabilities, interactions, and performance problems. This is because an emulation testbed uses a real dedicated computer with limited resources, and a real operating system running on it, to represent each host in an experiment. Hence, any protocol implementation error or resource vulnerability can be exposed, since these are *not abstracted* by a model. Our preliminary experiments on the Emulab or DETER testbeds have exposed interesting bottlenecks and software problems that highlight differences between simulation, emulation, and testbeds including hardware routers [11]. For example, we found that synchronization effects in ns-2 simulations amplified the impact of certain attacks against TCP congestion control. Further, since ns-2 [38] does not accurately model the switching/queuing fabric, the processor, buses, or interrupt handling, no packet losses can occur as a result of bottlenecks such as head of the line blocking in a switch fabric, or interrupt livelock [17], [28] in an operating system, in contrast to real systems which exhibit such bottlenecks.

A. Router Emulation

As mentioned above, the MIT Click router [27], [8] is a possible basis for constructing our programmable routers. Developing modular but efficient software routers has been the subject of significant research since the 1990s, e.g., [17], [28]. In these studies, polling is used as an alternative to packet receive interrupts to eliminate interrupt livelock at high packet rates. This is because interrupts can consume much of the CPU and bus capacity of mid-range machines (i.e., Pentium III and below) at 100 Mbps+ speeds. In the Click router [27], programmed I/O (PIO) interaction with the Ethernet controllers is eliminated using Direct Memory Access (DMA). In our own experiments on the Wisconsin Advanced Internet Laboratory (WAIL) at http://www.schooner.wail.wisc.edu/, we have found that livelock can occur on Cisco 3600 series routers, limiting

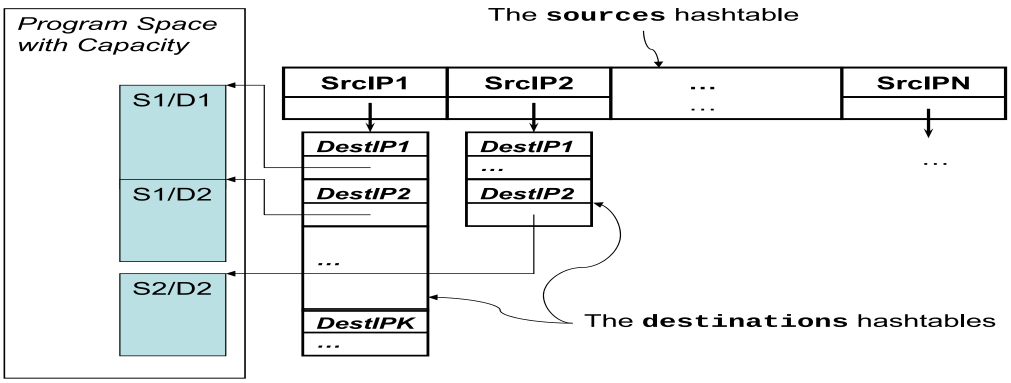

Fig. 4. The Dynamic Execution Environment.

the forwarding performance.

In Click, the entire packet path is easily described, and one can easily configure a simple IP router that bypasses the OS IP stack. Simplification of the packet path yields a performance boost, making the PC router less vulnerable to overload under high packet flows. When the router is configured, each affected network device has to be included into the configuration. It should be straightforward to change the queuing discipline and the queue depth for the queue at each output port, according to our virtualization programs. Our packet manipulation library can also be incorporated into this environment.

Using Click entails a number of challenges. For example, in our prior work, we have found that it is insufficient to change the Click *Queue* element depth [11]. This is because Click (or any software system for that matter) has to go through a device driver when it accepts or outputs a packet. Like any Linux network device driver, the driver for the Intel Pro card has internal transmit (TX) and receive (RX) buffers, whose sizes must be adjusted. The Click *Queue* elements only serve as intermediaries between these. We will have to take such subtle implementation issues into consideration when building and validating our virtual Internet prototype.

B. Experimental Methodology

In order to conduct realistic experiments on DETER that mimic the Internet, we will leverage topology and traffic generation tools and datasets that we developed/used for our prior work in the EMIST project [11]. We will also leverage topology, delay, and bandwidth information through the application of inference (Internet tomography) techniques, e.g., [41], [22]. One challenge we are facing is the need to develop tools to scale down output from topology generators to fit within testbed constraints. Since the number of testbed machines is limited, and the machines have a physical limit on how many network interface cards they have and their total bandwidth, arbitrary network topologies cannot be used. We need to be able to map a topology to an equivalent DETER-compliant topology.

Our experiments will investigate the following five key questions:

1. *Programmable router performance under different loads.* Example questions to investigate include: what are the fundamental limits on proof checking and installing different virtualization programs with and without the optimizations previously discussed? What packet rates can the router handle with different virtualization programs? How efficiently we can schedule and execute typical workloads of virtualization programs?
2. *Robustness under threats.* As we learned in the past from proposals such as active networks, potential misuse can hinder the adoption of programmable networking paradigms in the global Internet. Therefore, a key component of our study will be investigating the robustness of the execution environment at the router under malicious attacks on its security and scalability.
3. *Placement of programmable routers*. The effect of partial deployment of our programmable routers must be quantified. The tools we plan to build will enable us and other researchers to conveniently explore the "deployment space."
4. *Potential gains from different applications.* Our proposed paradigm will allow new applications, such as the virtual firewall, spam filter, and multicasting, that can have a profound positive impact on the performance perceived by the users writing the virtualization programs, as well as Internet performance at large. We plan to conduct experiments to quantify this impact. For example, we can easily emulate an environment with many spammers, and measure the improved performance with our spam filter application.
5. *Impact of network properties such as traffic, topology, and configuration.* Network properties impact network load, and hence will impact our programmable routers and our example applications. *Background traffic characteristics*, e.g., mix of http, ftp, and peer-to-peer traffic, and duration of the flows in our experiments should be representative of Internet traffic characteristics. *Experimental topology characteristics*, e.g., the "small-world" phenomenon, can also be critical since they impact how traffic is aggregated. Infrastructure parameters in the network, e.g., BGP routing policies and OSPF link weights, should also be realistic in our experiments.

Observe that it is critical to isolate effects we might observe in DETER experiments into key observations that will also hold on the Internet, versus emulation model artifacts, e.g., caused by delay emulation on DETER [11]. In order to identify and isolate such artifacts, a careful sensitivity analysis is crucial.

C. Testbed Experiment Control

A natural approach for describing tasks that must be performed on the testbed nodes involves event scripts, much like events in an event-driven simulator [38]. Emulab software implements a few event types such as link failures; however, most of the interaction with the nodes has to be done via a secure shell (SSH) session. We plan to design a *flexible middleware* to control all test machines from a central location, since manually using each computer is impossible, especially when timed events are involved. We have developed a preliminary tool, that we call a *Scriptable Event System*, to parse a script of timed events and execute it on the test machines. Our system is capable of receiving callbacks such that event synchronization can be achieved [11]. Our system needs to be extended to significantly enrich the script language, and provide an integrated instrumentation, visualization, and analysis tool.

IV. Related Work

Active Networks. Clearly, our ideas are related to research on Active Networks [37], [10] (http://nms.lcs.mit.edu/activeware/). The Liquid Software project (http://www.cs.arizona.edu/liquid/) is closely related to our ideas. In that work, it suggested that network nodes should be enhanced with complex execution environments and support the execution of large and complex software systems. The main goals of the Liquid Software were to enhance the performance of the Internet and distributed Internet applications. The security and scalability issues that arise with network programmability were not addressed. In contrast, our ideas are more focused (e.g., programmability is limited to well-defined packet operations and code can be uploaded only at edge routers). We aim to provide useful programmability while carefully balancing security and scalability. Our goal is to attack security problems abundant in today's Internet that we believe cannot be effectively addressed without a paradigm shift towards network programmability.

Other projects related to Active Networks include ANTS [40] and PLAN [23]. ANTS is a toolkit for deploying network protocols—that is, it is not concerned with programs. On the other hand, PLAN is a restricted functional language that unlike ours allows building arbitrarily complex programs (through a mechanism similar to "system" calls in UNIX shell scripts). Thus, constraining (and even understanding) the safety issues for such programs is hard and remains an open problem [3]. In addition, PLAN is dynamically checked, which may considerably slow down the execution of a PLAN program at a network node and thus slow down the network node. In contrast, we will restrict programs to compositions of well-defined packet manipulation routines and as a result one can define a relatively simple safety policy as outlined in Section II-B. Also, in our model programs will be verified statically (either by static analysis or PCC).

Verification. Model checking is a verification technique that has been applied to protocol checking [24], and more recently to software checking [18], [16], [9], [14]. Unlike PCC, it places the responsibility for checking with the server and is relatively expensive. In contrast, we propose to make use of PCC and advance its state by using a library of reusable lemmas which will help produce compact proofs. In our approach to PCC, we also work in a higher level programming language than one finds being used in most PCC work. Previous research such as Necula and Lee's [31], [33] and Appel's [1], [42] has been carried out mainly with very low level languages (machine code or Java byte code). The proof checking system we are using, Athena [5], [6], provides both a Scheme-like programming language in which programs are typically expressed as collections of recursive function definitions, and a structurally similar language for expressing deductive methods whose executions carry out proofs using both primitive inference methods of first order logic (of which there are about a dozen, such as modus ponens, existential generalization, universal specialization, etc.) and "calls" of other deductive methods previously defined by the user or imported from a deductive-method library. Both of these languages are high level by most programming language standards, offering, for example, higher-order functions (and methods)—the ability to pass functions/methods to a function/method or return them as results.

Moreover, we are experimenting with the use of these programming and deductive language facilities at a substantially higher level of abstraction than in most programming activity. We have already found elegant ways to use higher-order functions and methods in Athena to express generic requirements specifications much like the *theory* specifications of research languages such as Isabelle [39], Imps [20], or Maude [13], or the *concept descriptions* of Tecton [26], [29], [30]. There are two major benefits that stem from expressing proofs at this high abstraction level. First, once a proof of a theorem is written at this level, the functions and methods defining it can be called in many different ways, which means that the proof does not have to be redeveloped when its conclusion is needed in a different setting. Second, one can use such high-level theory or concept specifications to specify *generic* software components—ones that have a single source code expression but which can be instantiated in many different ways to produce different useful specific versions by plugging in other suitable components. Thus, the substantial effort required in constructing such proofs can be amortized over the many repeated uses of both proofs and the generic software components that can be extracted from them.

Overlay networks. A multitude of overlay network designs for resilient routing, multicasting, quality of service, content distribution, storage, and object location have been recently proposed. Overlay networks offer several attractive features, including ease of deployment, flexibility, adaptivity, and an infrastructure for collaboration among hosts. For example, overlay networks that detect performance degradation of current routing paths and re-route through other hosts include Detour [35] and RON [4]. Multicast and peer-to-peer systems have also been successful services on overlay networks. A number of overlay multicast algorithms have been proposed over the last four years. End System Multicast (or Narada) [12] was one of the earliest and most tested approaches. Overcast [25] provides scalable and reliable single-source overlay multicast. Gnutella, KaZaA, and BitTorrent are commercial peer-to-peer file systems for music and video sharing. For better scalability, several recent peer-to-peer systems, including Chord [36] and Pastry [34], use efficient distributed hash table (DHT) lookup algorithms over overlay networks. Despite their attractive features, a user must explicitly download software to join an overlay network, and different networks have their own varying membership policies and procedures. In fact, many overlay networks have limited deployment since they suffer from scalability problems. Furthermore, since they are purely application-layer, they incur a performance penalty over network-level solutions. For example, we have studied one aspect of the performance penalty of application-level multicast in [19]. Finally, again by virtue of being application-layer mechanisms, they cannot control network-level functions, which our proposed approach can control.

V. Conclusions

This paper has outlined an ambitious research agenda for virtualizing the Internet. The new virtual Internet offers significant flexibility, making it more secure and manageable. We have adapted techniques from the extensive literature on program analysis and verification, operating systems, network protocol design, and network emulation to enable the new virtual Internet. Many challenges remain in order to realize the full potential of this new paradigm, and to allow flexible security and scalability constraints to be specified and verified.

References

[1] A. Ahmed, A. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic, 2002.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Prinicples, Techniques, and Tools*. Addison Wesley, 1986.

[3] D. Alexander, W. Arbaugh, A. Keromytis, and J. Smith. Safety and security of programmable network infrastructures. *IEEE Communications Magazine, special issue on Programmable Networks*, 1998.

[4] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. of ACM SOSP*, October 2001.

[5] K. Arkoudas. *Denotational Proof Languages*. PhD thesis, MIT, 2000.

[6] K. Arkoudas. Certified computation, 2001. citeseer.nj.nec.com/arkoudas01certified.html.

[7] R. Bajcsy, S. Fahmy, and . other authors. Cyber defense technology networking and evaluation. *Communications of the ACM*, 47(3):58–61, March 2004.

[8] A. Bianco, R. Birke, D. Bolognesi, J. Finochietto, G. Galante, M. Mellia, M. Prashant, and F. Neri. Click vs. linux: Two efficient open-source IP network stacks for software routers. In *IEEE Workshop on High Performance Switching and Routing*, May 2005.

[9] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs, 2000.

[10] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. *IEEE Communications Magazine, Special Issue on Programmable Networks*, 1998.

[11] R. Chertov, S. Fahmy, and N. B. Shroff. Emulation versus simulation: A case study of TCP-Targeted denial of service attacks. In *Proceedings of 2nd International IEEE/CreateNet Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, March 2006.

[12] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay multicast architecture. In *Proceedings of the ACM SIGCOMM*, August 2001.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[14] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1), 2002.

[15] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *ACM Conference on Programming Languages Design and Implementation (PLDI'00)*, 2000.

[16] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[17] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the ACM SIGCOMM Conference*, pages 2–13, August 1994.

[18] D. Engler. Static analysis versus model checking for bug finding. In *Concurrency Theory, 16th International Conference, CONCUR 2005*, 2005.

[19] S. Fahmy and M. Kwon. Characterizing overlay multicast networks. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)*, pages 61–70, November 2003.

[20] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. In *Conference on Automated Deduction*, pages 653–654, 1990. citeseer.nj.nec.com/farmer93imps.html.

[21] J. Garay, S. Naor, B. Yener, and P. Zhao. On-line admission control and scheduling with interleaving. In *IEEE INFOCOM*, 2002.

[22] A. Habib, S. Fahmy, S. R. Avasarala, V. Prabhakar, and B. Bhargava. On detecting service violations and bandwidth theft in QoS network domains. *Computer Communications, Special Issue on Performance evaluation of IP networks and services*, 26(8):861–871, May 2003. http://www.cs.purdue.edu/homes/fahmy/papers/sla.pdf.

[23] M. W. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *International Conference on Functional Programming*, pages 86–93, 1998.

[24] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[25] J. Jannotti, D. Gifford, K. Johnson, F. M. Kaashoek, and J. O. Jr. Overcast: Reliable multicasting with an overlay network. In *Proc. of OSDI*, October 2000.

[26] D. Kapur and D. R. Musser. Tecton: A language for specifying generic system components. Technical Report 92-20, Rensselaer Polytechnic Institute Computer Science Department, July 1992.

[27] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[28] J. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[29] D. R. Musser. Tecton description of STL container and iterator concepts. http://www.cs.rpi.edu/~musser/gp/tecton/container.ps.gz, August 1998.

[30] D. R. Musser, S. Schupp, C. Schwarzweller, and R. Loos. Tecton Concept Library. Technical Report WSI-99-2, Fakultät für Informatik, Universität Tübingen, January 1999.

[31] G. C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.

[32] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)*, 1996.

[33] G. C. Necula and P. Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, 1998. IEEE Computer Society Press.

[34] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.

[35] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: a case for informed internet routing and transport. *IEEE Micro*, 1(19):50–59, January 1999.

[36] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM*, August 2001.

[37] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, 1997.

[38] UCB/LBNL/VINT groups. ns-2 Network Simulator. http://www.isi.edu/nsnam/ns/.

[39] M. Wenzel. The Isabelle/Isar Reference Manual, 2001. citeseer.nj.nec.com/article/wenzel01isabelleisar.html.

[40] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of IEEE OPENARCH*, 1998.

[41] O. Younis and S. Fahmy. Flowmate: Scalable on-line flow clustering. *IEEE/ACM Transactions on Networking*, 13(2), April 2005.

[42] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. citeseer.nj.nec.com/554698.html.