Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software xxx (2005) xxx–xxx
The Journal of Systems and Software
www.elsevier.com/locate/jss

OPERA: An open-source extensible router architecture for adding new network services and protocols ☆

Ben C.B. Chan, John C.F. Lau, John C.S. Lui *

Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, NT, Hong Kong

Received 8 October 2003; received in revised form 4 January 2005; accepted 7 January 2005

Abstract

In this paper, we present the design and implementation of a programmable and extensible router architecture. The proposed architecture not only provides the conventional packet forward/routing functions, but also the flexibility to integrate additional services (or extension) into a router. These extensions are dynamically loadable modules so one can easily deploy new services, such as reliability and security enhancement, onto the router in a dynamic and incremental fashion. To avoid new extensions that may monopolize system resource and degrade the performance of normal packet forwarding/routing function, we propose a novel CPU resource reservation scheme which facilitates the efficient use of resources and increases the stability of extension execution. To illustrate the "extensibility" and "effectiveness" of the proposed architecture, we present the results of a new service, namely, how to perform "Distributed Denial-of-Service (DDoS) attack traceback". In particular, we illustrate the deployment of the probabilistic marking in performing IP traceback. Note that this approach requires the collaboration of routers so that effective traceback can be performed. Currently, the programmable router platform is released as an open source[1] and we believe the system provides an ideal platform for researchers to experiment and to validate new services and protocols.

Keywords: Programmable router; Network system software

1. Introduction

The basic functionality of a network router is to determine where and how a packet should be forwarded within a network core. Majority of router implementations on market today is either hardware oriented or proprietary in nature. Hence, it is difficult for researchers to experiment with new protocols or add new services since these routers are not open for user/kernel level programming. In contrast to the limitations in hardware routers, software programmable routers have the potential to provide additional features in a dynamic and incremental fashion. Therefore, new services such as QoS supports (Li and Ravindran, 2002; Striegel and Manimaran, 2003), enhanced multicast service supports (Tsai et al., 2004) and security-related policies (Sun et al., 2004; Vaughn et al., 2002) can be easily integrated into a software programmable router. The aim of this work is to present an architecture for software programmable router that offers both flexibility and extensibility so researchers and network engineers and easily experiment with new protocols and services.

There exists software router architectures in both various research institutions (Dong et al., 2004; Kohler et al., 2000, 2002; Yau and Chen, 2001). Nevertheless, the design on the architecture of router program modules is still not very mature. In this paper, the main focus of the overall design is thus on the "***router extensions***" on an open platform. Extensions are extra modules that can be dynamically loaded and provide additional

☆ This research is supported in part by the RGC Earmarked Grant.
* Corresponding author. Tel.: +852 260 98407; fax: +852 260 35024.
E-mail address: cslui@cse.cuhk.edu.hk (J.C.S. Lui).
[1] It can be downloaded from: www.cse.cuhk.edu.hk/~cslui/ANSR-lab/software/opera

0164-1212/$ - see front matter © 2005 Elsevier Inc.. All rights reserved.
doi:10.1016/j.jss.2005.01.008

packet processing capabilities. Using these extensions, one can construct a value-added router that supports various kinds of "services".

The contribution of our work is as follows:

- To link up the existing routing and packet processing facilities with additional modules added and construct a comprehensive software programmable router architecture.
- To design a flexible architecture so one can securely add modules onto a router.
- To provide resource management for basic packet routing/forwarding elements and various added modules.
- To develop some important and illustrative applications (i.e., policies for QoS and security support) for the proposed programmable router architecture. In particular, we will present two possible approaches of performing the DDoS traceback at later section of this paper.

Currently, OPERA implementation supports basic routing facilities, CPU resource reservation and dynamic loading of both kernel and user-space extensions. Security extension loading feature is also supported so as to avoid malicious extension loading. To demonstrate the extensibility and effectiveness of the proposed programmable router architecture, we illustrate a new service which is the *Distributed Denial-of-Service (DDoS) attack traceback and detection*. The DDoS attack detection is an important service to enhancing network security feature.

The balance of this work is as follows: In Section 2, we present the architecture and implementation of the proposed programmable architecture. In Section 3, we present two new extensions of detecting DDoS attack. Experimental results of the effectiveness of the DDoS detection and traceback are presented in Section 4. Related work is given in Section 5 and Section 6 concludes.

2. System architecture of OPERA

Fig. 1 illustrates the overview of the OPERA architecture. The OPERA's kernel consists of three modules, namely, the *core module*, the *extension module* and the *security module*. The components of each module will be discussed in the following sub-sections.

2.1. Architecture overview

2.1.1. The core module

The core module provides the basic routing facilities and handles internal resource management. It has three components and their functionalities are:

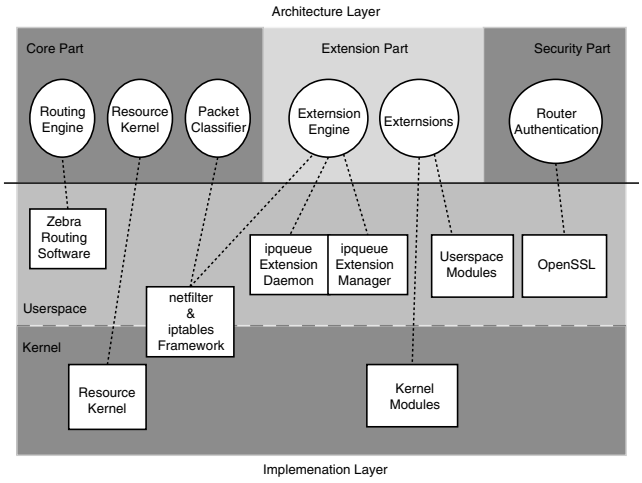

Fig. 1. An overview of the components in the OPERA architecture.

- *Routing engine*: The routing engine supports the common routing protocols such as RIP, OSPF and BGP. The duty of the engine is to help broadcasting and forwarding route advertisements as well as updating its routing table based on the advertised information. The route information will be used to determine the next hop a packet should be forwarded to.
- *Resource kernel*: The resource kernel is responsible for resource management. Since there can be multiple extensions running on the OPERA router at the same time, the extensions may compete for CPU system resources. In order to ensure all extensions operate properly without monopolizing the common CPU resource, the OPERA router is resource-award and manages the CPU resource.
- *Packet classifier*: The packet classifier serves the purpose for packet classification and directing packets for further processing. It is the entry point to the *extension engine* in the OPERA architecture. The support of a generic and comprehensive packet classification can make the work easier to identify a specific type of packet or flow and then direct the target to corresponding processing routines.

2.1.2. The extension module

The extra packet processing capabilities of the router, which we call them "***services***", are provided in the extension module. Each processing module is called an "***extension***". Several extensions can be installed on the router simultaneously to provide different services. The extension module consists of the extension engine and dynamically loadable extensions:

- *Extension engine*: The engine manages extensions in a centralized manner. It supports dynamically loading/unloading of extensions at system runtime.

- *Extensions*: The *extensions* in the proposed router basically refer to dynamically loadable programs with packet processing capabilities. One or more extensions can be loaded into the router and provide different kinds of services. An extension can be packet filter which performs a passive scanning of packets passing through the OPERA router and perform filtering, if necessary. The extension can also take an active role such as modifying packet contents or interacting with other applications (both in kernel or in the user-space) to achieve other purposes.

2.1.3. The security module

It is paramount that security is built into the OPERA architecture so as to avoid any malicious extension installation or faking of any extension's communication. Therefore, it is essential to provide the feature of *router authentication* while at the same time, maintain a certain level of convenience and flexibility. We implement an efficient authentication scheme for securing the router communication. Upon receiving any sensitive data, an OPERA router has to authenticate the sender and ensure the integrity and credibility of the data before processing any process.

2.2. System implementation

In the previous section, we described the overall architecture of the proposed router. A prototype architecture has been implemented on the Linux platform. The relationship between the architecture design and actual implementation is shown in Fig. 1. Parts of the router implementation are based on some existing facilities in Linux. We employ the *netfilter*[2] architecture inside the Linux 2.4.*x* kernel to implement packet classification and parts of the extension engine. We also use the zebra[3] as the routing engine. The implementation details of some important parts of the proposed router will be discussed in the following.

2.2.1. Implementation of the resource kernel

Under OPERA, we modify and enhance the Linux kernel to become a resource kernel. In particular, we built a *resource kernel framework* inside the Linux 2.4.18 kernel. The framework provides basic supports for resource management. In general, resource management can be applied to many resources such as CPU, memory, disk storage, I/O or network bandwidth. So far, we focus on the CPU resource sharing since it is the most critical measure of the extensions' performance.

(1) *Resource reservation*: The major duty of the *resource kernel* is to grant system resources to processes. Processes can communicate with the *resource kernel* through the system call `rk_signal` which we added to the kernel. All commands are sent to the *resource kernel* using this system call with different parameters.
When a process wants to make a reservation on any resource, it has to bind a "*resource set*" first. A resource set is used by the *resource kernel*, which is a structure holding the information of resources associated with a process. After binding a resource set, the process can make a resource reservation. As mentioned before, CPU is the only resource available for reservation at this stage. When a process terminates, all the resources reserved and the resource set binded by it will be released automatically. It can also be explicitly release the resources.

(2) *CPU Reservation and scheduling*: The policy of CPU time sharing in the proposed router is based on CPU reservation. Each process can make a request to the kernel to reserve a portion of CPU time. This enables the CPU resource can be fairly shared among processes according to their processing requirements.
In the current implementation, we define each process committing CPU reservation to be a "*client*". Each client is associated with two parameters, *share* and *weight*. *Share* is the fraction of total CPU time owned by a client, where *weight* is the relative *share* of a client compared to others. A client can join one of the two reservation classes: *absolute* (ABS) and *proportional* (PROP). The absolute reservation provides a basic support for processing requirements by some real-time clients. Once this kind of reservation is granted by the *resource kernel*, the target client will obtain a fraction of total CPU share it requested. On the other hand, the proportional reservation is to maintain a proportional share of CPU time among all PROP clients. The remaining CPU shares, excluding those owned by ABS clients, are assigned proportionally to PROP clients according to their weights.
In this work, we use a round-robin class algorithm called *Virtual Time Round-Robin* (VTRR) (Nieh et al., 2001). This algorithm uses a proportional share basis of scheduling with a coarse error bound and the scheduling overhead is relatively small, as compared with other proportional share algorithms like the EEVDF (Stoica and Abdel-Wahab, 1995; Goddard and Tang, 2000). In particular, the scheduling decision can be carried in constant time.
We have implemented the VTRR scheduling algorithm in the Linux kernel. The original process scheduling routine is not abandoned but the

[2] The netfilter/iptables Project. Available from: http://www.netfilter.org/
[3] GNU Zebra. Available from: http://www.zebra.org/

VTRR scheduler is built aside the original one. We treat all processes that are not in any reservation classes (ABS and PROP) as a virtual client. It shares all remaining CPU time after the clients in reservation class are granted for CPU resources. Within the scheduling period of the virtual client, the original process scheduling policy in the kernel is executed.

2.2.2. Implementation of the extension module

The extension part is developed based on the *netfilter* framework. In general, *netfilter* provides an efficient and scalable way for packet filtering, network address translation (NAT) and packet mangling. *netfilter* defines a set of *hooks* (Fig. 2) inside the kernel's network stack which allows kernel modules to register callback functions. We can therefore attach our packet processing routines in several intermediate points in the protocol stack. When a network packet traverses one of those hooks, the related callback functions will be invoked to process the packet.

In conjunction with *netfilter*, *iptables* is an interface to define a set of rules in a table structure. Every rule within a table consists of some classifiers *(matches)* and corresponding action *(target)*. The matches and targets are implemented as kernel modules that provides packet classification and manipulation, respectively. We employ *iptables* in packet classification and forwarding packets to different extensions.

There are two classes of extensions in our design, the kernel space and the user-space extensions:

(1) *Kernel space extensions*: The kernel space extensions are implemented as *netfilter* modules which are attached to one of the hooks defined in *netfilter*. This class of extension is suitable for services which require low-latency processing. As this class of module is directly inserted into kernel space, the number of buffer copy and the packet traveling time is minimized. It provides comparably higher processing performance over the user-space one. However, coding a kernel module is less flexible than a user-space one as there are fewer libraries supported in kernel.

(2) *User-space extensions*: The user-space extensions are implemented with the aid of *ipqueue* which is an kernel module for *netfilter* (the QUEUE target) that provides user-space packet queuing facility. With *ipqueue*, packets can be pulled out from the kernel and queued to user-space for further processing. The meta-data (nfmark, MAC address,...,etc.) of a packet and the IP payload (optional) are sent to an user-space process via the *netlink socket* (Fig. 3). Hence, one can develop customized packet handling routines using the *netfilter*-provided library for *ipqueue* called "*libipq*" without much modification to the kernel codes. This class of extension is suitable for user customized and complicated services. User-space modules can easily make use of existing libraries and interact with other user level applications. Hence it provides a convenient solution to implement various kinds of services. The trade-off to this flexibility

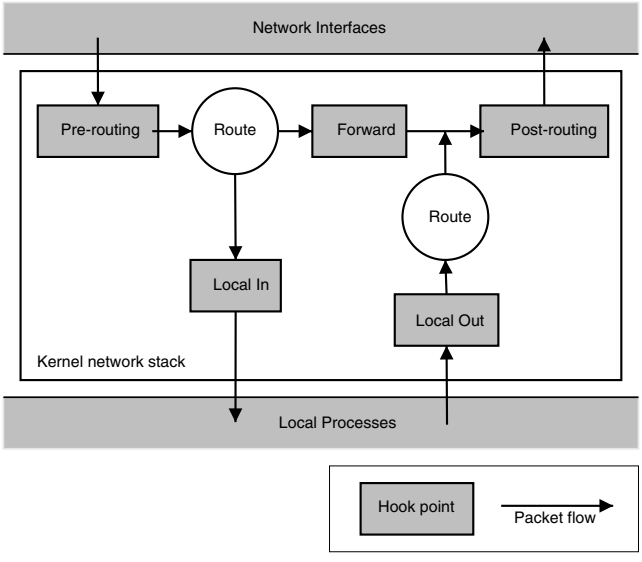

Fig. 2. Hook points for netfilter: pre-routing, forward, post-routing, local-in and local-out.

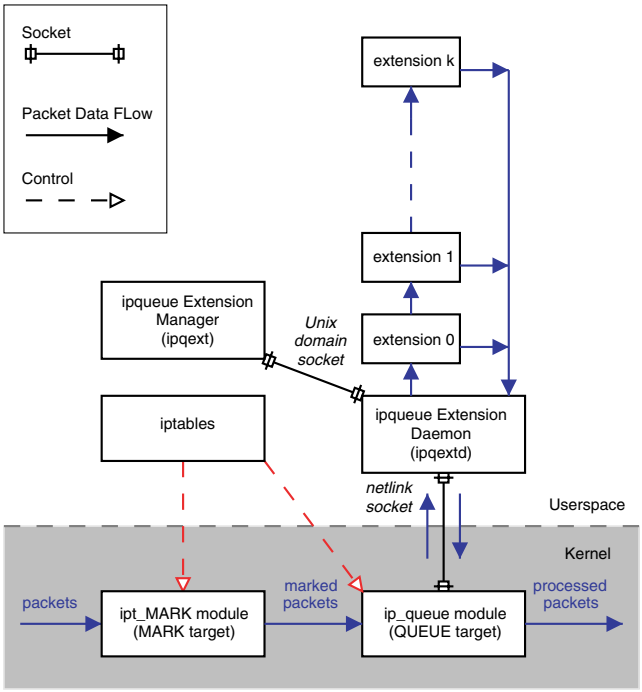

Fig. 3. Overview of the user-space extension architecture.

is the lower performance of user-space extension as compared to the kernel extension as it requires two extra buffer copies, from kernel to user-space and vice versa for communication.

Here are the components related to the user-space extension class:

- *ipqueue extension daemon* (`ipqextd`): Since currently only one user-space process is supported by *ipqueue* facility, we designed and implemented an user-space *extension daemon* to de-multiplex packets to multiple extensions. The *daemon* runs on top of *ipqueue*. It handles the registration/de-registration of user-space extensions, keeps track of loaded extensions and assigns the packets queued in *ipqueue* to corresponding extensions. For the remaining of the paper, we will simply call it *daemon* for easy presentation.
- *ipqueue extension manager* (`ipqext`): The *extension manager* provides an interface to the *daemon*. The *manager* communicates with the *daemon* through Unix domain socket. Users can control the *daemon* and manage extensions at runtime via the *manager*. The *manager* mainly supports basic controls such as loading/unloading and starting/stopping extensions. The syntaxes of the *manager* is designed similar to *iptables*[2] so that users can easily pick up the commands.
- *Dynamically loadable extensions*: User-space extensions are implemented as *dynamic libraries (modules)* which can be loaded by the *daemon* on user demand. Using dynamic modules instead of static ones gives the flexibility of integrating services into the proposed router at run time without the need to shut down or reboot the router. It also reduces the resources in the router as unused extensions can be temporarily stopped or offloaded.

In order to fit into the extension architecture, every extension module has to fulfill several criteria. There is an *extension information structure* (`ipext_t`) in each module which contains the internal status variables and function pointers. This information block enables the *daemon* to reach the functionalities and internal status of an extension.

```
/* ipqext structure */
typedef struct ipqext ipqext_t;
struct ipqext{
  struct ipqext *prev, *next;
  void* handle;
  int active;
  const char *name;
  const char *version;
  unsigned long groupmask;
  int (*proc)(ipq_packet_msg_t *m);
  int (*start)( );
  int (*stop)( );
  int (*reset)( );
  const char* (*info)( );
};
```

Each module also needs to export several standardized functions such that extensions with different functionalities can be plugged into the router in a generalized way. The standardized functions are

`_init`: initialization procedures initiated when the *daemon* loads the extension,
`_fini`: finalization or cleanup procedures executed when the *daemon* unloads the extension
`proc`: packet processing routine called by the *daemon* when a packet targeted for the extension arrives,
`start`: procedures executed when the *daemon* starts the extension,
`stop`: procedures executed when the *daemon* stops the extension,
`reset`: function to reset the internal status and variables of the extension,
`info`: function to obtain the information of internal status of the extension.

In the following, we will describe the flow of packet processing in user-space extensions as depicted in Fig. 4.

During the module initialization (`_init`), an extension registers itself to the *daemon* by passing its extension information structure. The *daemon* keeps a list of extension information structure of loaded extensions. Extensions are loaded into the *daemon* in a sequential order. Each extension is associated with an index that indicates its order of execution. Packets are processed

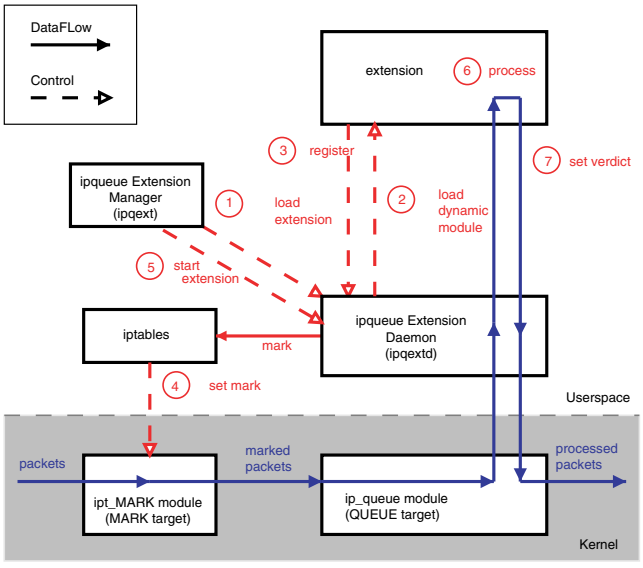

Fig. 4. Flow of packet processing under the user-space extensions.

by corresponding extensions one by one with the smallest index first. Several extensions can be simultaneously loaded into the *daemon*. An loaded extension can either be in "started" or "stopped" state which indicates whether it processes packets or not.

The `proc` function of an extension will be called when a packet targeted for the extension is ready for processing. Directing a packet or a flow to a specific extension is done by "*packet marking*". We use the MARK target provided by *netfilter* for marking packets. Once the *daemon* starts, packets or flows can be directed to it using *iptables*. Packets targeted for user-space processing are first filtered by specific rules in the "mangle" table of *iptables*. These rules pass the packets to the MARK target which alters the *netfilter mark (nfmark)* field associated with each packet. In our design, we treat the mark as a 32-bit vector with each bit representing a group of extensions. A group can consist of some extensions and an extension can belong to several group. An extension belongs to the *i*th group if the *i*th bit of its group mask is set. Each marked packet enters *ipqueue* and is delivered to corresponding extension modules by the *daemon*. The marked packet is processed by the extensions in the *i*th group if the *i*th bit of the mark is set.

Every time an extension finishes the processing on a packet, the `proc` function returns a *verdict* which determines the fate of the packet. In normal case, the function returns `NF_QUEUE` to inform the *daemon* that the packet has been processed and is ready for further processing by other extensions. Then the packet will be passed to the next extension in the chain. If the return value is `NF_ACCEPT`, the packet will immediately exit from the user-space queue and return to kernel even the packet has not passed through all user-space extensions. If `NF_DROP` is returned, the packet will skip further processing and be dropped.

2.2.3. Implementation of the security module

Sometimes system administrators may want to install/update the modules or launch some services remotely on a programmable router. In some applications, a set of programmable routers often need to collaborate together to complete an IP traceback (e.g., probabilistic marking algorithm, which will be discussed in later section). Both the module update and module communication involve exchanges of information, commands and even dynamic program modules. In the following text, we simply call these data "*sensitive data*".

Inadequate security measure in router communication can make a router become vulnerable. Attackers can make use of the transmission of sensitive data to attack, break in or take over the routers. They can bring down some router services by sending fake commands to a router. They can also install some malicious codes remotely into a router via the security holes in the module update mechanism. Therefore, security control plays an important role in the extensible router architecture. And particularly, it is essential to enable *router authentication*. In order to prohibit previously mentioned attacks while maintaining a certain level of convenience and flexibility, we need an efficient authentication scheme for securing the router communication. Upon receiving any sensitive data, a router has to authenticate the sender and ensure the integrity and credibility of the data before proceeding the process.

We use the widely used *electronic certificate* and *digital signature* technology in the *public key infrastructure (PKI) standard* (Rescorla, 2000; Viega et al., 2002) to support the router authentication. In the current implementation, every router has its own certificate issued by a common certificate authority (CA). The certificate is used to prove the identity of a router in every authentication process. During the information exchange, a router will attach a digital signature and its certificate together with the sensitive data to be sent. On receiving any sensitive data, a router will check the attached digital signature and certificate so as to ensure the credibility of the sender and the data integrity. In addition to message authentication, the sensitive data can be optionally encrypted to prevent any unauthorized access to the content.

2.3. Evaluating the effectiveness of CPU reservation

To verify the performance of the implemented VTRR scheduling algorithm, we tested the VTRR scheduler on

Table 1
Average quanta per cycle received by each client

Client	Share requested	Average number of quanta per cycle	Average relative share received (normalized w.r.t client A)
A	5	44.74898	5.00000 (1.00000)
B	10	88.76735	9.91540 (1.98308)
C	15	132.33214	14.77987 (2.95597)
D	20	177.28673	19.80266 (3.96053)

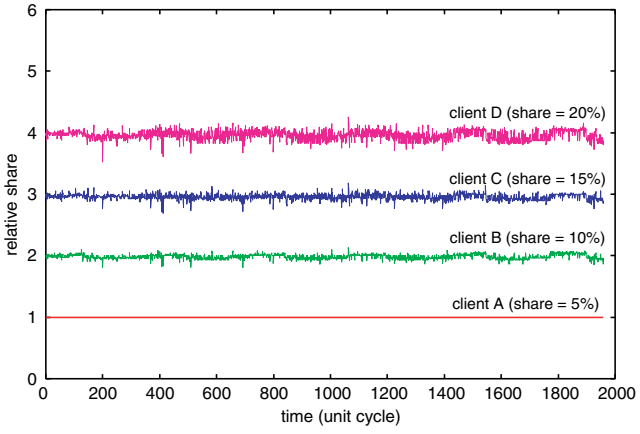

Fig. 5. Relative CPU share of all clients (normalized with respect to client A).

the Linux Opera router. In the experiment, there were four identical client processes which reserved CPU resources through the *resource kernel*. They were all absolute class clients and granted with 5%, 10%, 15% and 25% of total CPU share, respectively. In each scheduling cycle, the number of quanta consumed by each client was recorded. Table 1 shows the average number of quanta per cycle received by each client and Fig. 5 shows the related CPU share for whole measurement period of 2000 time units. The result shows that the OPERA system can *accurately* allocate CPU resources to each client according to the value they reserved.

3. New service on DDoS traceback services

DDoS traceback is an important step to tackle the flooding-based DDoS attack since it can identify the source of the attack. However, this type of service is currently not available or not supported by the network routing core. To perform effective traceback, one needs to first obtain the attack graph topology (Savage et al., 2000; Law et al., 2002, in press) and a running statistics of traffic volume from different sources destined to the victim site. With the attack graph, one can determine the suspicious sources of attack and carry out further actions, for example, limit the incoming traffic to a controllable level and maintain the availability of services to legitimate users. Another requirement for efficient traceback is to determine the location of attack on the fly during the period when a victim is under a DDoS attack.

Note that for DDoS traceback, simply examining the source address of every incoming packet at the gateway router of the victim is not effective because an attacker can easily hide itself by using spoofed source IP addresses. One way to resolve the problem requires all Internet Service Providers to enforce some mechanisms to prevent IP spoofing. However, this is difficult to achieve since it requires universal adaptation. In this work, we illustrate the *probabilistic marking* (Law et al., 2002, in press) to implement the traceback service on the proposed programmable router. Note that the methodology relies on the collaboration of a set of coordinated routers associated with a victim site. Since this is a new service, one needs to provide hookpoints and extensions to the proposed programmable routers.

Unless we state other wise, any mentioned "*traffic*" is referred to the one destined for a victim site. Each participating router carries both *transitive (forwarded)* and *local traffics* to the victim site. The transitive traffics are aggregated to a router from its upstream routers while the local traffics are generated within the *local administrative domain* of that router. These two traffics contribute to the *outgoing traffic* of the router destined for the victim. If an attack traffic is generated within the domain of one participating router while the others carry normal traffics, the local traffic rate of that router would be *significantly higher* than the others transitive traffics. By *determining* the local traffic rates of all participating routers, one can compare these values and *deduce* the approximate location of attack. Therefore, one can then narrow down the search scope to a small number of domains and continue the traceback until the source of attack is found.

In the following sections, we formally describe the traceback methodology.

3.1. DDoS attack traceback—probabilistic marking approach

This approach (Law et al., 2002, in press) extends the *probabilistic edge marking algorithm* (Savage et al., 2000) to determine the local traffic rates of participating routers.

Under the probabilistic edge marking, packet marking is a path encoding method to overcome the problem of IP spoofing. With the support of the underlying programmable routers, the path information can be recorded on a packet during its traversal along the routers. Provided that the involved routers are trusted and the mark information is authenticated, the information can be used to recover the attack path and eventually the attack graph.

Note that recording the complete traversed path information on a packet may be prohibitive because the size of mark information grows linearly with the number of routers the packet has visited. Therefore, the space needed to store the variable length marker cannot be pre-determined and thus may introduce the difficulty on storing the complete mark. Marking every packet also burdens a router since this requires too much processing. An alternative approach is probabilistic edge marking (Savage et al., 2000).

Under the probabilistic edge marking, a router decides to mark on every forwarding packets with a marking probability p. Each mark records a "*partial path information*" which corresponds to a segment (edge) along the whole route to the victim site. It consists of three static fields, *start*, *end* and *distance*, which represent the IP addresses of routers on the two ends of a segment, and the distance of that segment from the victim site. Since the size of mark does not grow with the number of routers a packet has visited, therefore the storage overhead of the edge information is reduced to minimal.

Under the probabilistic marking, if a router decides to mark a packet, it puts its address into the *start* field and initialize the *distance* field to zero. Otherwise, it first checks whether the packet is marked with a *distance* equal to zero. In this case, the router puts its IP address into the *end* field. If a router decides not to mark a

packet, the router always increments the *distance* field by one.

Under the probabilistic edge marking, every router has a probability p to mark the transitive packets and so every edge information is eventually included in the marked packets. By collecting sufficient number of marked packets, one can collect all the edge information which packet traversed and can then construct the attack graph by the following algorithm:

Algorithm. (Attack graph construction procedure at the victim site V)

```
Initialize the tree G to have a root node which is the
victim site V;
Filter out unmarked packets;
Sort the marked packets in ascending order of the
value of the distance field;
/* attach each marked edge to the tree G */
For (each marked packet w) {
   If (w.distance == 0) {
    insert edge (w.start, V, 0) into G;
}
Else {
   If ((w.end == one of the outermost node in G) and
    (w.distance == that outermost node's distance))
   insert edge (w.start, w.end, w.distance) into G;
 }
}
Extract path (R_i ... R_j) by enumerating acyclic paths
in G;
```

Using the above attack graph construction procedure, the victim can gather the information about the attack graph topology. When a router R_i sends packets to the victim V, all routers that are on the path between R_i and V can mark the packets with certain probability. So, if a suitable marking probability is chosen and there is a sufficient amount of attack packets, one can gather the marked packets with all different edges on that path and recover the attack path completely.

With the marked packets and attack graph information, the local traffic rates of different routers can be deduced mathematically based on the concept of stochastic comparison (Ross, 1996). A minimum stable time $t_{\min}$ is defined in (Law et al., 2002, in press) to determine the minimum time required for the traffics to be stable so that one can determine the local traffic rates of all participating routers. The end result is that one can *rank* all these local traffic rates, in a non-increasing order, and find out the possible sources of DDoS attacks. Note that the probabilistic marking has to be performed by participating routers. We have implemented this new service in the programmable router. The detail of determining the local traffic rates and the minimum stable time will be given in the next subsection.

3.2. Implementation details

3.2.1. Implementation of the security functions

The security part of the router is implemented with the *OpenSSL toolkit*.[4] OpenSSL is an open source toolkit provides the library for implementing *Secure Sockets Layer (SSL v2/v3)* and *Transport Layer Security (TLS v1)* protocols as well as cryptography library.

As the authentication scheme is based on electronic certificate and digital signature, every router has to equipped with its own certificate. A X.509 certificate is generated per router and signed by a certificate authority (CA). In the current scheme, the certificate is stored in PEM[5] format or PKCS#12. For a small scale deployment, one can use the OpenSSL toolkit to setup a CA for a group of routers. To simplify the certificate verification process, one level certification hierarchy can be used such that all router certificates are signed by a self-maintained root CA. The certificate chain is thus shorten.

To provide the authentication on sensitive data, a digital signature is generated per message by using hash function (SHA-1, MD5) and public key cryptography (DSA, RSA). A message digest of sensitive data is computed by a hash function. The message digest is then encrypted with the private key of a public key cryptographic algorithm. We currently use SHA-1 with RSA to implement the digital signature but it can be easily changed to different combinations with the OpenSSL toolkit. A timestamp is appended, when necessary, to sensitive data before digesting so as to prevent replay attacks.

Message authentication can only guarantee the data integrity and credibility. In order to prevent any unauthorized access to the content of sensitive data, the data can be optionally encrypted using the cryptography library provided in OpenSSL. Alternatively, a secure SSL communication channel can be established in advance for a large amount of data transfer.

3.2.2. Implementation of the DDoS traceback service: probabilistic marking approach

We implement the packet edge marking process as a kernel space extension. All the transitive and local traffic destined for the specified site are directed to the kernel module using *iptables* rules.

The implementation of probabilistic edge marking in the original paper (Savage et al., 2000) uses the identification field of IP header together with the compressed edge fragment sampling algorithm to store the mark information. This technique minimizes the storage overhead as well as preserving the robustness. Yet the com-

[4] The OpenSSL project. Available from: http://www.openssl.org/

[5] Privacy Enhanced Mail format defined in RFC1421.

pressed edge fragment sampling algorithm does involve much processing.

In the current implementation, we chose a simpler way to store the mark information by using IP options. We store the mark directly as IP options without compression. This simplifies the mark processing while the storage for mark information is still kept at an acceptable size.

The internal status of the edge marking kernel extension can be retrieved via the Linux *proc file system (procfs)*. It is provided for system administrators to keep track of the marking process.

3.2.3. Attack graph construction

The attack graph construction algorithm we use is the algorithm which we depicted in Section 3.1. Before the attack graph construction, a linked list structure is used to store the edges and their frequency in ascending order of edge distance. Then a tree data structure is constructed for the attack graph. Starting from the victim as the root node, the edges are recursively joined to the tree. If the edges cannot join any node on the tree, it will be ignored. So there is no broken path in the resulting attack graph G.

3.2.4. Determination of local traffic rates and minimum stable time

With the attack graph information, the remaining issue is to determine the local traffic rates of all participating routers. Before we introduce the methodology, let us use an example to illustrate the concept.

Fig. 6 illustrates a general topology wherein each router may be connected by several upstream and downstream routers. For example, router R_j in the figure has three upstream routers R_{j+1}, R_{j+2} and R_{j+3} and two downstream routers R_{j-1} and R_k. For this attack graph, the furthest router from the victim site V is router R_n, which has a distance of $d \geqslant 1$ hops away from V. Each router receives its local traffic from its network domain, some of these traffics are destined to the victim site V. Let λ_j denotes the average local traffic rate, in unit of packets per second, from the router R_j to the victim site V. Let λ_j^{in} denotes the total average traffic rate from all upstream routers of R_j to the victim site V and λ_j^{out} denotes the total average traffic rate from router R_j to all its downstream routers which are targeted to the victim site V. Again, the goal is to deduce the *local traffic rate* λ_j for all routers in an attack graph, then based on their local traffic intensities, we can identify the locations of the potential attackers.

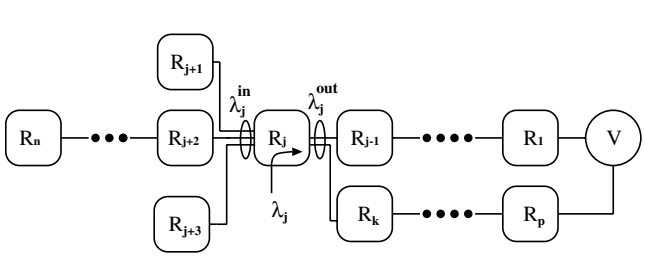

Fig. 6. A general attack graph G.

Let G denotes the attack graph based on the construction method we described in Section 3.1. We say that a router is a *leaf* router in the attack graph G if it is not connected to any upstream router. For example, in Fig. 6, router R_n is a leaf router. All other routers are called *internal* routers. Let (i,j) denotes an edge between router R_i and R_j, we define $\lambda_{(i\to j)}$ as the average traffic rate to the victim site V that passes through the edge (i,j) between router R_i and R_j.

The average local traffic rate λ_j for router $R_j \in G$ is

$$\lambda_j = \begin{cases} \lambda_j^{\text{out}} & \text{if } R_j \text{ is a leaf router,} \\ \lambda_j^{\text{out}} - \lambda_j^{\text{in}} & \text{if } R_j \text{ is an internal router,} \end{cases} \tag{1}$$

where λ_j^{in} and λ_j^{out} are the average traffic rates into and out of router R_j, respectively. These average traffic rates can be computed by

$$\lambda_j^{\text{in}} = \sum_{\forall(i,j) \text{ where } R_i \text{ is an upstream router of } R_j} \lambda_{(i\to j)}, \tag{2}$$

$$\lambda_j^{\text{out}} = \sum_{\forall(j,k) \text{ where } R_k \text{ is a downstream router of } R_j} \lambda_{(j\to k)}. \tag{3}$$

Therefore, if we can estimate the average traffic rate $\lambda_{(i\to j)}$ for all marked edges (i,j) in the attack graph G, then we can deduce the average local traffic rate of each router based on Eqs. (1)–(3). In Fig. 7, we present the procedure to estimate $\lambda_{(i\to j)}$ for each marked edge (i,j) in G.

Let $\widetilde{N}_j^{\text{out}}(t)$ be the random variable denoting the number of marked packets received by the victim site V at time t such that the `start` field is equal to router R_j. Let $\widetilde{N}_j^{\text{in}}(t)$ be the random variable denoting the number of marked packets received by the victim site V at time t such that the `end` field is equal to router R_j. After a sufficient amount of time in collecting these marked packets, we have the following stochastic relationship (Ross, 1996):

$$\widetilde{N}_j^{\text{out}}(t) \geqslant_{\text{st}} \widetilde{N}_j^{\text{in}}(t) \quad \forall R_j \in G. \tag{4}$$

The above relationship holds because

1. there is non-negative local traffic originated from router R_j to the victim site V, therefore, the number of *marked* output packets from R_j can be greater than the number of *marked* input packets to R_j in the long run;
2. the probabilistic edge marking algorithm will mark any transit packet to V from the upstream of router R_j. Therefore, the router R_j may erase any edge marking of a transit packet from its upstream routers.

The remaining issue is that to have an accurate estimation of the local traffic rate λ_j, we have to guarantee that the conditions in Eq. (4) are satisfied. Once the

Algorithm: Estimating the Average Traffic Rate $\lambda_{(i\to j)}$ for each edge (i,j) in G

Let G be an attack graph with root V;

Each marked edge (i,j) in G has $(R_i, R_j, d_{(i\to j)}, N_{(i\to j)})$

/* R_i = start address of the marked edge (i,j)

R_j = end address of the marked edge (i,j)

$d_{(i\to j)}$ = hop count between R_i and victim V

$N_{(i\to j)}$ = the number of times this marked edge (i,j) has been collected */

Let $\lambda_{(i\to j)} = 0$ for all edges (i,j) in G;

For each edge in G {

/* go through each edge in G */

add $\lambda_{(i\to j)}$ by $\frac{N_{(i\to j)}}{tp(1-p)^{d_{(i\to j)}}}$

}

output: average traffic rate $\lambda_{(i\to j)}$ for each edge (i,j) in G.

Fig. 7. Procedure to estimate $\lambda_{(i\to j)}$ for every marked edge (i,j) in G.

conditions of Eq. (4) are satisfied, we can then estimate local traffic rate λ_j based on Eqs. (1)–(3) to traceback the potential attackers. In the following, we provide an analytical method to derive the minimum stable time $t_{\min}$ such that the conditions of Eq. (4) are satisfied.

Let $N_j^{\text{out}}(t)$ be the number of marked packets received by the victim site V at time t such that the `start` field is equal to router R_j. Let $N_j^{\text{in}}(t)$ be the number of marked packets received by the victim site V at time t such that the `end` field is equal to router R_j. We have the following relationship:

$$N_j^{\text{in}}(t) = \sum_{\forall(i,j)} \lambda_{(i\to j)}(t) p(1-p)^{d_{(i\to j)}-1} t, \tag{5}$$

$$N_j^{\text{out}}(t) = \sum_{\forall(j,k)} \lambda_{(j\to k)}(t) p(1-p)^{d_{(j\to k)}-1} t, \tag{6}$$

where $\lambda_{(i\to j)}(t)$ and $\lambda_{(j\to k)}(t)$ are the traffic rate estimation of the corresponding edge (i,j) and (j,k) at time t according to the procedure given in Fig. 7, $d_{(i\to j)}$ and $d_{(j\to k)}$ are the hop count from router R_i to victim V and from router R_j to victim V, respectively.

To simplify notation, let us define $\lambda_j'^{\text{in}}(t)$ and $\lambda_j'^{\text{out}}(t)$ as

$$\begin{aligned} \lambda_j'^{\text{in}}(t) &= \sum_{\forall(i,j)} \lambda_{(i\to j)}(t) p(1-p)^{d_{(i\to j)}-1}, \\ \lambda_j'^{\text{out}}(t) &= \sum_{\forall(j,k)} \lambda_{(j\to k)}(t) p(1-p)^{d_{(j\to k)}-1}, \\ N_j^{\text{in}}(t) &= \lambda_j'^{\text{in}}(t) t \quad \text{and} \quad N_j^{\text{out}}(t) = \lambda_j'^{\text{out}}(t) t. \end{aligned} \tag{7}$$

One can re-formulate the problem of finding the minimum stable time $t_{\min}$ such that

$$\text{Prob}\left[\widetilde{N}_j^{\text{out}}(t_{\min}) \geqslant \widetilde{N}_j^{\text{in}}(t_{\min})\right] \geqslant p_{\text{threshold}} \quad \forall R_j \in G, \tag{8}$$

where $p_{\text{threshold}}$ is a large probability (e.g., $p_{\text{threshold}} = 95\%$). *The formulation implies to find the minimum time such that with a very high probability, the number of collected packets marked by the router R_j* is higher than the number of collected packets marked by the upstream routers of R_j, for all routers in the attack graph G. Assuming that the random variable is a Poisson process, we have

$$\begin{aligned} &\text{Prob}\left[\widetilde{N}_j^{\text{out}}(t_{\min}) \geqslant \widetilde{N}_j^{\text{in}}(t_{\min})\right] \\ &= \sum\nolimits_{k=0}^{\infty} \text{Prob}\left[\widetilde{N}_j^{\text{out}}(t_{\min}) \geqslant k\right] \text{Prob}\left[\widetilde{N}_j^{\text{in}}(t_{\min}) = k\right] \\ &= \sum\nolimits_{k=0}^{\infty} \left[\sum_{n=k}^{\infty} \frac{[\lambda_j'^{\text{out}}(t_{\min}) t_{\min}]^n}{n!} e^{-[\lambda_j'^{\text{out}}(t_{\min}) t_{\min}]} \right] \\ &\quad \times \frac{[\lambda_j'^{\text{in}}(t_{\min}) t_{\min}]^k}{k!} e^{-[\lambda_j'^{\text{in}}(t_{\min}) t_{\min}]} \quad \forall R_j \in G. \end{aligned} \tag{9}$$

Again, one can easily determine the minimum stable time $t_{\min}$ using some standard numerical methods.

4. DDoS experiments and results

In order to test the two extensions for DDoS traceback, we carry out two experiments to verify the implementation with the theoretical and simulation results in (Law et al., 2002, in press). The two experiments concern with the minimum stable time in the probabilistic marking approach in performing the traceback.

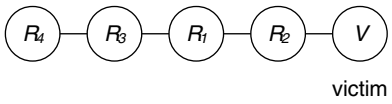

Fig. 8. The linear network topology for verifying the minimum stable time.

4.1. Verification of the theoretical minimum stable time

This set of experiments aims to verify the theoretical minimum stable time $t_{\min}$ in (Law et al., 2002, in press) under the current implementation and real network traffics. The theoretical minimum stable time $t_{\min}$ is defined as

$$P = \text{Prob}\left[\widetilde{N}_j(t_{\min}) \geqslant \widetilde{N}_{j+1}(t_{\min})\right] \geqslant p_{\text{threshold}} \quad \forall j \in \{1, \ldots, d-1\}. \tag{10}$$

In this set of experiments, we setup a small scale linear network topology as shown in Fig. 8 which consists of four routers. There is one attacker in the local administrative domain of one of the routers which generates a larger amount of traffic while the traffic from other routers is kept at 50 pkts/s.

We increment the attack traffic rates from 100 to 200 pkts/s with a increase of 50 pkts/s each time. For a particular attack traffic rate, we sample $N_j(t)$, the number of marked packets received by the victim site V at different time t (from $t = 0$ to $10t_{\min}$) such that the *start* field is equal to router R_j for all router. We take 200 samples for each attack traffic rate. Over the 200 samples, we count the number of times such that $N_j(t) \geqslant N_{j+1}(t)$ $\forall j \in \{1, \ldots, d-1\}$. This gives the probability P in Eq. (10). Theoretically, the inequality holds at $t \geqslant t_{\min}$. Therefore, we try to see at which time P is greater than or equal to $P_{\text{threshold}}$ from the experimental results. In all experiments, the marking probability of each router is set to $p = 0.25$. The value of $P_{\text{threshold}}$ is set to 0.95.

Experiment A. (Attacker at node R_4) In this experiment, the attacker resides in the local domain of the furtherest router from the victim site V and the result of the experiment is

Attack traffic rate	Probability P at different time			
	$t_{\min}$	$1.25t_{\min}$	$1.5t_{\min}$	$1.75t_{\min}$
100	0.93	0.955		
125	0.94	0.955		
150	0.915	0.95		
175	0.925	0.94	0.97	
200	0.88	0.93	0.94	0.955

From the results listed above, one can notice that the value of P is close to $P_{\text{threshold}}$ at $t = t_{\min}$. With no more than 1.75 $t_{\min}$, the inequality $P \geqslant P_{\text{threshold}}$ holds. In other words, one can determine the location of attackers within 1.75 min.

Experiment B. (Attacker at node R_1) In this experiment, the attacker resides in the local domain of the nearest router from the victim site V and the result of the experiment is

Attack traffic rate	Probability P at different time	
	$t_{\min}$	$1.25t_{\min}$
100	0.95	
125	0.94	0.98
150	0.935	0.965
175	0.93	0.96
200	0.94	0.96

From the results, we can notice that the value of P is close to $P_{\text{threshold}}$ at $t = t_{\min}$. After $t \geqslant 1.25t_{\min}$, $P \geqslant P_{\text{threshold}}$. In other words, one can determine the location of attackers within 1.25 min.

5. Related work

Authors (O'Malley and Peterson, 1992) propose the component-based synthesis of routing protocol in *x*-kernel was first reported in O'Malley and Peterson (1992). This work was solely concentrated on the routing functionalities without providing the capability of adding new service to routers. Recently, this platform has been extended to other applications and services (Descaper et al., 1998; Kohler et al., 2000; Merugu et al., 2000; Spalink et al., 2001; Yau and Chen, 2001) but these revised platforms are not available to research community. Recently, the Click architecture (Kohler et al., 2000) also supports service extensions under the push/pull data movement paradigm. The main contribution of Click is a new paradigm of 1 h "*configuration language*" and system support in construction flow service pipelines. On the other hand, our proposed architecture is targeted for easy service extension and deployment on an open architecture. Authors (Descaper et al., 1998) propose a router plug-in for programmable routers. However, these plug-in gates are fixed in the IP forwarding path and cannot be dynamically extended. Under our proposed architecture, one can extend various hook-points, either in the kernel space IP forwarding path or at the user level so that extensions can be easily added.

For the network security, Savage et al. (2000) proposed probabilistic marking for traceback without generating separate ICMP packets to the victim. Routers mark packets probabilistically and store the partial path information in the IP header. Each piece of information represents a sample edge of the attack path. Victim collects the attack packets and can reconstruct the attack path based on the partial path information. This

approach does not need the coordination among the network administrators and it does not increase the traffic flow or the storage requirement of a router. Park and Lee (2001) analyzed this marking approach and pointed out that spoofing of the marking field may impede traceback by the victim. Attackers may choose the spoofed marking value, source address to hide themselves. Dean et al. (2001) formulated the traceback problem as a polynomial reconstruction problem, They used algebraic coding theory to encode traceback information in the packet, similar to Savage approach. It also suffers the same spoofing problem and may be more vulnerable without the distance field in the marking. Song and Perrig (2001) reported that if the victim knows the map of its upstream routers, it does not need the full IP address in the packet marking. They improved Savage's marking approach by hashing so as to achieve a lower false positive rate and a lower computation overhead. They proposed efficient authentication of packet markings to filter packets with spoofed markings from the attackers. Note that approaches by Savage et al. (2000) and Song and Perrig (2001) provide the topology of the attack graph. Our approach can be view as a complementary approach to their so as to locate potential attackers in the attack graph.

6. Conclusion

In this paper, we present the design and implementation of an open architecture of a programmable and extensible router architecture. The platform is built on top of the open source kernel and it can be easily accessible by general public. The objectives are to provide an open platform for researchers to experiments with new network protocols and extension services. Our programmable router architecture not only provides the traditional packet forward/routing functions, but also provides the flexibility to integrate additional extension into a router. These extension are dynamically loadable modules so services can be deployed on the fly without shutting down the routing elements. We provide a framework for CPU resource management so that no extensions can monopolize system sources (e.g., CPU resource). We also illustrate the extensibility of the proposed architecture for implementing a new service extension that can perform DDoS traceback, namely: IP traceback via the probabilistic marking algorithm. We performed experiments to illustrate that the proposed architecture can allocate CPU resource precisely and the added extensions can achieve their objectives effectively, e.g., tracing the attackers within a short period of time. Our programmable router platform provides an ideal platform and flexibility for researchers to experiments new Internet protocols and security services.

References

Dean, D., Franklin, M., Stubblefield, A., 2001. An algebraic approach to IP traceback. In: Proceedings of Network and Distributed System Security Symposium, February.

Descaper, D., Dittia, Z., Parulka, G., Plattner, B., 1998. Router plugins: a software architecture for next generation routers. In: ACM SIGCOMM, September.

Dong, Y., Yau, D.K.Y., Lui, J.C.S., 2004. Composition of Java-based router elements and its application to generalized video multicast. IEEE Network (November).

Goddard, S., Tang, J., 2000. EEVDF proportional share resource allocation revisited. In: Work-in-Progress Sessions of the 21st IEEE Real-Time Systems Symposium (RTSSWIP00), November.

Kohler, E., Morris, R., Chen, B., Jannotti, J., Frans Kaashoek, M., 2000. The Click modular router. ACM Transactions on Computer Systems 18 (3).

Kohler, E., Morris, R., Chen, B., 2002. Programming language optimizations for modular router configurations. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), October.

Law, K.T., Lui, J.C.S., Yau, D.K.Y., 2002. You can run, but you can't hide: an effective methodology to traceback DDoS attackers. In: The Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOT), October.

Law, K.T., Lui, J.C.S., Yau, D.K.Y., in press. You Can run, but you can't hide: an effective statistical methodology to trace back DDoS attackers. IEEE Transactions on Parallel and Distributed Systems.

Li, P., Ravindran, B., 2002. Proactive QoS negotiation in asynchronous real-time distributed systems. Journal of Systems and Software (December).

Merugu, S., Bhattacharjee, S., Zegura, E., Clavert, K., 2000. Bowman: a node OS for active networks. IEEE Infocom (March).

Nieh, J., Vaill, C., Zhong, H., 2001. Virtual-time round-robin: an O(1) proportional share scheduler. In: Proceedings of the 2001 USENIX Annual Technical Conference, Boston, MA, June.

O'Malley, S.W., Peterson, L.L., 1992. A dynamic network architecture. ACM Transactions on Computer Systems (May).

Park, K., Lee, H., 2001. On the effectiveness of probabilistic packet marking for IP traceback under denial-of-service attack. IEEE INFOCOM (April).

Rescorla, E., 2000. SSL and TLS: Designing and Building Secure Systems. Addison Wesley Professional, Reading, MA.

Ross, S., 1996. Stochastic Processes. John Wiley, New York.

Savage, S., Wetherall, D., Karlin, A., Anderson, T., 2000. Practical network support for IP traceback. In: Proceedings of ACM SIGCOMM, August.

Song, D.X., Perrig, A., 2001. Advanced and authenticated marking schemes for IP traceback. In: Proceedings of IEEE INFOCOM, April.

Spalink, T., Karlin, S., Peterson, L., Gottlieb, Y., 2001. Building a robust software-based router using network processors. ACM SOSP (October).

Stoica, I., Abdel-Wahab, H., 1995. Earliest eligible virtual deadline first: a flexible and accurate mechanism for proportional share resource allocation. Available from: <http://www.cs.berkeley.edu/~istoica/eevdf.ps.gz>.

Striegel, A., Manimaran, G., 2003. Dynamic class-based queue management for scalable media servers. Journal of Systems and Software 66 (2).

Sun, H., Lui, J.C.S., Yau, D.K.Y., 2004. Defending against low-rate TCP attack: dynamic detection and protection. In: IEEE International Conference on Network Protocols (ICNP), Berlin, Germany, October.

Tsai, C.-F., Tsai, C.-W., Chen, C.-P., 2004. A novel algorithm for multimedia multicast routing in a large scale network. Journal of Systems and Software 72 (3).

Vaughn, R., Ronda, H., Kevin, F., 2002. An empirical study of industrial security engineering practices. Journal of Systems and Software 61 (3).

Viega, J., Messier, M., Chandra, P., 2002. Network Security with OpenSSL. O'Reilly & Associates.

Yau, D.K.Y., Chen, X., 2001. Resource management in software programmable router operating systems. IEEE Journal on Selected Areas. In communications 19 (3).

Ben Chan graduated from the Chinese University of Hong Kong with the degree in computer engineering. He is currently a graduate student in Purdue University, USA. His research interests include network communication, design and implementation of operating systems, as well as system security issues.

John C.F. Lau graduated from the Chinese University of Hong Kong with the degree in computer engineering. He is currently a member of technical staff in the Cluster Technology Inc. His research interests include parallel and distributed computing systems, communication networks and reconfigurable embedded systems.

John C.S. Lui was born in Hong Kong. He received his Ph.D. in Computer Science from UCLA. When he was a Ph.D. student at UCLA, he spent a summer working in the IBM T.J. Watson Research Laboratory. After his graduation, he joined the IBM Almaden Research Laboratory/San Jose Laboratory and participated in various research and development projects on file systems and parallel I/O architectures. He later joined the Department of Computer Science and Engineering at the Chinese University of Hong Kong. For the past several summers, he has been a visiting professor in computer science departments at UCLA, Columbia University, University of Maryland at College Park, Purdue University, University of Massachusetts at Amherst and Universit degli Studi di Torino in Italy. Currently, he is leading a group of research students in the Advanced Networking and System Research Group. His research interests span both in system and in theory/mathematics. His current research interests are in theoretic/applied topics in data networks, distributed multimedia systems, network security, OS design issues and mathematical optimization and performance evaluation theory. He received the CUHK Vice-Chancellor's Exemplary Teaching Award in 2001. He is an associate editor in the Performance Evaluation Journal, member of ACM, a senior member of IEEE and an elected member in the IFIP WG 7.3. He is the TPC co-chair of ACM Sigmetrics 2005. His personal interests include films and general reading.