Abstract—There have been increasing demands for proper monitoring and control in end-host systems, mainly for security and QoS purposes. Nevertheless, existing technologies are insufficient as primitives for end-host security. For example, Berkeley Packet Filter (BPF), the most popular monitoring infrastructure for many Unix systems, is intended for packet capturing at physical interfaces, and thus, not appropriate for monitoring of applications, which is sometimes critical for system security.

This paper presents a simple solution to the problem, utilizing hierarchical virtual network interface (VIF) mechanism. VIF is a new OS abstraction that can be hierarchically structured and attached to OS entities to control their network I/O. We extend VIFs to allow filtering and monitoring of their traffic, and show that it has desirable properties for end-host monitoring and control of traffic. We present our prototype implementation on FreeBSD, and evaluate it qualitatively and quantitatively.

Demonstrated advantages include: i) ability to monitor terminating entities at arbitrary granularity, ii) a single consistent framework for both network security and network Quality of Service, iii) OS independence, iv) efficiency as a control primitive, v) compatibility with BPF interface and its applications, and vi) flexibility for future functional expansion.

I. INTRODUCTION

The explosive popularization of internetworking technology caused massive growth of traffic, and at the same time, it has bred a variety of subversive applications, such as Internet worms, computer viruses, Denial of Service attacks, etc. This trend necessitated technologies that can restore public order in the global packet switching network. Firewalls are among such a technology that brings about peaceful and secure territories by establishing solid barriers between network boundaries [1].

This notion of conventional firewall is, however, rapidly becoming obsolete due to potential threats for their territorial integrity [2]. For example, the secure territory is easily threatened by a viral-infected node within the trusted network; cryptography technology have increased difficulty in packet inspection at the point of border defense; port-forwarding might easily penetrate even the hardest firewall ever built, if the firewall considers just types of application protocols.

All of these defiant situations compromise the access control scheme at network border. Accordingly, there emerged greater needs for detailed inspection and control of traffic at endpoint of communication. The Distributed firewall is the most notable approach in this regard, providing network-wide top-down framework for management and enforcement of security policy at the distributed checkpoints on each end-node system [1], [3]. In the same context of network security, but from a bottom-up perspective, are a variety of software firewalls for major Operating Systems (OSs): the Packet Filter (PF) of OpenBSD [4], ipfw mechanism on FreeBSD [5], IPchains [6] and IPfilter [7] for Linux, and various personal firewall products for the Windows OS, such as [8]–[10].

It seems, however, that these mechanisms for end-host network security are posing several architectural difficulties, which we call the structural mismatches, as stated below.

[M1] Monitoring Granularity: The first mismatch is that, although the end-node mechanism must be capable of monitoring activity of applications on the host, this basic functionality is not always provided. For example, we may allow fetching of web documents by a legitimate web browser, but, may not want a Trojan horse program to communicate with intruders waiting outside for a chance to get inside, using the same protocol. The end-node firewall needs to deal with threat of this sort appropriately, and, actually, every personal firewall package on Windows operating system supports this feature. Nevertheless, this is not the case with most firewall implementation on Unix systems. Packet monitoring mechanisms, such as the Berkeley Packet Filter (BPF) [11], does not allow monitoring on a per-application basis. What is needed here is flexibility in the control granularity so that we can properly discriminate malicious applications and conforming ones.

[M2] Monitoring Facility and Control Primitive: A second mismatch is of traffic monitoring and control. Packet monitoring infrastructures have been designed mainly for monitoring of traffic, and thus, they cannot control the traffic based on the monitored information; conversely, most control mechanisms, such as packet filters and advanced queuing mechanisms, are designed mainly for controlling of traffic, and rarely provide monitoring facility. The mismatch in monitoring and control implies that we cannot confirm the
effect of our control, and that it limits the way we secure the system. For example, if the system needs more data to evaluate a suspicious flow, they might want to slow down the flow temporarily, rather than to drop everything. Although there are exceptions, such as [7], the existing implementations do not possess the flexibility in their granularity. We believe that a complete mechanism needs to possess monitoring features as well as control features, to somehow bridge the gap.

[M3] Network Perspective and End-host Perspective: The third and last mismatch we identify is that network-wide scheme cannot control applications, and mechanisms for controlling applications cannot be deployed network-wide. For example, frameworks for network-wide security, such as [1], [3], [12], have been focusing on network traffic, and thus, they lack perspective of monitoring end-host applications. Meanwhile, for monitoring of applications, researchers have been mostly focusing on system call (syscall) activities [13], [14], and thus, their security policies have had tight association with the underlying operating system, which is not desirable for network-wide deployment. Since these approaches need to scrutinize each syscall, and each OS has a different set of syscalls, we say that the end-host approach is OS dependent. Consequently, here again, we have a structural mismatch. An example is when a vulnerability in a multi-platform application, such as sshd, is discovered. In fact, updating of sshd at every security advisory is annoying periodical ceremony for most system administrators nowadays. It would greatly reduce management cost, if we could simply activate one additional policy that patches the security hole throughout the network. In other words, the primitive must be OS independent.

In summary, as a building block for security solution in the network age, we need an efficient mechanism that realizes i) monitoring of network applications, ii) integration of monitoring and control, and iii) OS independence, at end-host. In this paper, we propose such a mechanism, the Netnice Packet Filter, or NPF. This is built on the concept of hierarchical virtual network interface (VIF), which is an abstraction, hierarchically structured and attached to OS entities to control their network I/O. We extend VIFs to allow filtering and monitoring of their traffic, and show that the proposed scheme bridges the presented mismatches. For proof of concept, we present a prototype implementation we built on FreeBSD, and evaluate it qualitatively and quantitatively, showing that it has desirable properties in this problem domain and it is comparable with other implementations in terms of performance.

The composition of the paper is as follows. First, we give short notes on existing studies in this field (Section II). Then, we present the design and implementation of our primitive for end-node control (Section III), and evaluate the proposing mechanism (Section IV). We address application issues in Section V and conclude the paper (Section VI).

Note that focus of the paper is on the end-node mechanism, and policy issues are out of the scope of this paper, including network-wide rule management and its secure deployment. We believe that they are orthogonal components, and independence of the layers between policy and mechanism would contribute to lower design complexity of future systems. Indeed, this mixture of security policy and enforcement mechanism has been major limitation of existing mechanisms. We discuss issues related to actual security applications built on the primitive in Section V.

II. RELATED WORK

In this section, we review previous efforts that have been made in this domain, and clarify that they cannot bridge the architectural mismatches mentioned above. In this regard, distributed firewalls, packet filters, and packet monitors, are of distinct importance. Note that, there have been implementations that may meet some of the requirements, but, the point we would like to make here is that there has not been any approach that meets all of them, which is particularly important for system security.

A. Distributed Firewall

Conventional firewalls [1] are built on several assumptions. First, the trusted network is trustworthy. Second, all the traffic travels through a checkpoint. Third, the checkpoint is capable of inspecting the packets they transfer. These assumptions have been, however, threatened by several trends in internetworking, and accordingly, the concept of distributed firewall was introduced, as presented in the introduction. In this category, we have two approaches; top-down and bottom-up.

Reference [2] is considered to be the first article that conceptually proposed the distributed solution in network security. Based on the proposal, they presented a prototype implementation [3], also. In this top-down framework, security policy is centrally defined by a policy language, which is securely delivered to each host by secure connections, and then, enforcement of localized security policies takes place on each endpoint. The proposed architecture possesses topology-free property, and higher level of network security is achieved by the control at end-hosts.

However, as their emphasis is on the distributed framework, its underlying primitive has several shortcomings. First of all, monitoring granularity is inflexible, since their monitoring primitive is connection oriented (M1). Second, similarly to most other primitives, it offers just a simple accept/reject capability (M2).

The bottom-up approach is built upon functional similarities among implementations on various platforms, probably resulted from their field experience. For example, all of the
firewalls [4]–[7], [15] accept standard packet filtering rules (based on IP address, protocol, and port number), and user-ID based rule. They use these mechanisms as building blocks, and introduce another layer above the underline mechanisms, to reduce their cost for a network-wide policy management [12]. However, this way, their functions are limited by their underlying mechanisms which are far from flawless, as we review next.

B. Packet Filter

Almost every operating system nowadays is equipped with software firewall (packet filter) to protect the system from network-borne threats. In their implementation, Unix gurus have been focusing mostly low-end routers and servers with routing capability, whereas the Windows developers have been targeting the dominating client use, in the design of such filters. Accordingly, they are categorized roughly into two groups: server oriented [4]–[7], [15], and client oriented [8]–[10]. Although it works in a different layer, we may add library based solutions, like [16], in this category.

For this class of implementations, OS independence (M3) is relatively kept, and their limitations come mostly from M1 and M2. First of all, most of the implementations do not allow monitor of applications, since association of their target traffic and its owner applications is somewhat lost in the network layer they reside (M1). Secondly, they provide fairly limited controllability over traffic, accept or drop, which is limiting the possibility of the primitives as we suggested in the slowdown example above (M2).

We may provide the filtering and monitoring functionalities by hooking function calls for communication in the userspace. The approach is, however, susceptible to fraud by malicious applications. In this regard, personal firewalls on Windows [8]–[10] circumvent the problem by hooking communications at device driver level.

C. Packet Monitor

A packet monitor, a.k.a. packet capturer, realizes monitoring of traffic, a basic functionality essential for network security. They mostly come with packet filtering (classification) mechanism to reduce overhead of data movement between kernel and user space, by selecting specific subset of packets. Since the beginning of this field [17], developers have worked toward better capturing performance, and a well-known implementation, the Berkeley Packet Filter [11], has gained popularity, ported onto most of the popular operating systems. Distinctive feature of the mechanism is its innovative micro-machine architecture for packet classification. In this scheme, the high-level description of packet matching rules is compiled in the userspace, and efficient packet matching code, expressed in a virtual Instruction Set Architecture (virtual ISA), is passed into the kernel. The advantage of the scheme is two fold. First, we can eliminate the overhead caused by the interpretation of rules in high-level description, for each packet. Second, in the compilation process, we can optimize the rule matching algorithm, to generate more efficient code.

Aside from the de-facto standard, many approaches have been proposed, which achieve even further performance by dynamic code generation [18], by migration of monitoring entity into kernel space [19], [20], by compiler optimization technique [21], by merging of filters [22], and with hardware support [23], respectively. There exist other lines of products in this category, which targeted other aspects of traffic monitoring, such as remote monitoring and standardization of its API [24]–[26].

For this class of implementations, OS independence (M3) is relatively kept, again. Their limitations mainly lies in M1 and M2. First of all, the monitors are concerned just with flow of packets, and thus, selective monitoring of specific application is hard to achieve (M1). Besides, control over the monitored traffic (M2) is not addressed by most implementations.

III. DESIGN AND IMPLEMENTATION

As reviewed thus far, an end-node primitive needs to realize monitoring of terminating applications with flexible granularity, and controlling of traffic they monitor, while keeping OS independence. To accomplish these goals, we propose to extend the hierarchical virtual network interfaces [27]. In this section, we first review the VIF concept briefly, and provide high-level description of the extension we propose, followed by implementation details.

A. Virtual Network Interface model

The hierarchical virtual network interface is a general purpose primitive for network control on end-host operating systems, which provides traffic control capability through hierarchical virtualization of network I/O. Figure 1 illustrates the concept. A physical machine (rectangular box) has a network interface (a cylinder at bottom, numbered as 6). Circles denote processes (p1, p2, and p3) and the arrows are data flow, driven by system calls. The approach allows hierarchical creation of VIFs, which connect the physical interface to terminating entities, such as processes, threads, and sockets.
On this framework, we can separately choose a traffic control scheme at each of the VIFs. For example, we can set bandwidth limit, using Leaky Bucket algorithm or, for better link utilization, we may employ a Weighted Fair Queuing (WFQ) or Priority Queuing (PQ) scheduler. As Application Programming Interface (API), the VIF structure and its control parameters are represented as directories and files under process file system (procfs), and protection of resources is naturally realized with access control semantics of the file system: a user cannot control resources of other users, unless they have appropriate permission. Because of this property, system users, as well as system administrators, can control their own network I/O, while protecting their resource from inappropriate access.

There are a variety of advantages for network security at end-host system in this model. First of all, the hierarchical structuring allows users to attach VIFs to OS-supported entities at desired granularity, such as a process group, a process, or sockets. This property lays desirable foundation for network security at end-host, as we claimed in the M1 discussion. Second, the framework favors flexible extension of VIF functions. As we have provided various scheduling disciplines, such as WFQ and PQ, we may add new functionality without violating the control framework. Third, resource protection is inherently realized by the partitioning property of VIF structure.

**B. Extension of the VIF model**

We propose to make just two minor changes on the basic VIF model, to efficiently support the features needed for end-host security. However, before making the statement, we would like to briefly cover an issue which governed our design process: performance.

In monolithic kernels like Unix, the cost for buffer copy is enormous, and there has been no efficient way to do the security control at userland, for example by migrating processing of network protocols there. Meanwhile, in operating system kernel, modules can directly access the packet data structure, without costly buffer copies needed to deliver data into user space. For this reason, many works have been promoting control inside kernel [11], [18], [20], [21]. Packet filter (software firewall) packages, such as [4]–[7], [15], are implemented on exactly the same ground, although their goals and implementations differ greatly.

The hidden cost of the approach for high performance is its compromised flexibility and restriction in programing constructs available for use. For example, kernel modules tend to lack easy access to file system and peer communication. It also requires code safety checking feature, not to crash the system by a user-written misbehaving piece of code. This limitation leads to the second category of network security mechanisms for end-host operating systems; userland mechanism.

Since userland provides programming freedom, solutions can compensate the hidden costs listed above. A typical application is Intrusion Detection System (IDS), such as [28], which has logging (using file system access) and alerting (using mail system access) capabilities. For this type of control, systems need to provide a mechanism for data transfer between the kernel and the user space, such as [11]. Similar approach is diverting of packets into user space, for detailed inspection of packets and sometimes even for modification of payload (e.g. to remove mail attached viruses) [8]–[10].

Clearly, there is a trade-off between performance and flexibility, and each direction has its own supporting ground and useful application. For this reason, in our end-host network primitive, we propose to support both types of the operations to provide users with appropriate options.

First, we add in-kernel packet filtering and monitoring capability, filter, and extend the VIF model so that we can place a "filter" onto each VIF. This is intended mainly for fast filtering of packets, but can be used for in-kernel monitoring (statistics and accounting, like [19], [20]) as well.

Next, we add a monitoring facility, port, with which packets traveling through a particular VIF of interest can be captured and sent to user space for application-level monitoring. The mechanism serves as a peephole of VIF, and we allow system users to run a variety of applications on the captured packets, such as packet monitors and IDS. To facilitate this process, we keep it compatible with the dominant mechanism for packet monitoring, BPF, so that widespread applications for network monitoring, such as tcpdump [29] and ethtool [30], can be used also for monitoring of applications.

Next section addresses how the functionalities are implemented.

**C. Implementation**

For the implementation, we extended the existing virtual network interface system to accommodate the new functionalities. In this section, we briefly review how the base VIF system is implemented, and then, implementation detail for the filter mechanism and for the port mechanism are given. For further detail of the base implementation, readers are referred to [27].
1) **Base system:** The implementation overview is shown in Fig 2. The four thin arrows in the figure denote packet entry and exit points of the virtual network interfaces. Packets are hooked by stealing of function calls, between the physical interface layer and the network layer. Once forwarded to the VIF tree, they are scheduled by local scheduling mechanism of each VIF, and passed into next VIF until delivered to its destination (a socket, for incoming packets, or a network interface, for outgoing packets).

Since packet classification and routing at each VIF would be too costly, we use *flow labeling* and *source routing*. For outgoing packets, each packet is marked with its target VIF when they leave their originating process. After the protocol processing, in transport layer and network layer, they are hooked into the VIF system, and the system adds, onto each packet, a list of VIFs on the path from the current VIF down to the root interface. This way, each VIF does not need to classify, nor route, the packet, since all the information needed for the control is available on the packet data structure.

Input packets are handled in a slightly different manner. When they are hooked into the VIF system (in this case, they are always inserted into the root VIF), the system first tries to find the destination socket of the packet. Starting from the VIF directly connected to the socket, we travel across the VIFs down to the root VIF, and, reversing the order, we get the routing information needed to guide the packet up to the target VIF.

2) **The Filter mechanism:** The objective of the filtering mechanism is to realize in-kernel packet filtering and monitoring. The scheme is implemented as follows.

First, we modified the process file system (*procfs*) interface, so that we can put the filter code onto each VIF. From users’ perspective, the filtering mechanism of a VIF is activated simply by writing filter code (written in BPF binary form) to a *filter* file in a VIF directory under `/proc/network` that corresponds to a VIF of interest.

Second, we needed to address the performance issue, because existing mechanisms for packet filtering seems quite inefficient. For example, [5] takes filter rules as a *rule chain* written in a high-level description format. The kernel modules keep the rules in the presented form, and for each packet, it iteratively calls functions for rule matching, which causes unnecessary overhead. Accordingly, the filter micro-machine approach of BPF can be a desirable solution, which can remove the control overhead, by optimization of rules and their compilation. However, although the optimization could significantly reduce the cycles needed for processing of each packet, it introduces another overhead for emulation of the virtual machine architecture. Hence, to avoid the emulation overhead, we decided to augment the micro-machine architecture approach with native binary translation (or, Just-In-Time assembler, pursued in approaches like [18], [21], [31]), aiming at further performance gains.

The binary translator we wrote converts a filter program in BPF micro-code into IA32 native binary (Figure 3). The translator function, `npf_gen()`, first converts the original code into native instructions on allocated memory space, and then, `npf_patch()` is called to back-patch unresolved branch targets in the filter code. The translation happens when the filter code is passed to the kernel, using the `procfs` user interface described above. Memory allocation strategy is simple; allocate the memory with roughly estimated size first, and, if the translation process fails due to underestimation, simply discard everything and start all over with another memory space with doubled size. Unused space in the allocated memory is used as persistent scratch memory for the binary code, and thus, we can realize stateful inspection of packets. We also extended the process file system so that interested programs can access the scratch memory of the VIF by a simple read operation (useful for statistics).

Third, we modified the packet scheduling routine of VIF, to call the filter binary if the VIF has an activated filter code (dashed arrow in Figure 3). The filter code is called just like another function, which takes a packet data structure as an argument, and returns positive value if accepted, or zero, otherwise. It is easy to observe that packet filtering is substantially simplified in this scheme, one function call to a native filter binary, compared with iterative rule-chain matching in ordinary packet filters.

This mechanism serves as a platform for various solutions. We may simply use it as a high-performance in-kernel packet filtering, or we may use it for detailed packet inspection. Note that, if the filtering is applied to the root VIF directly connected to physical interfaces (VIF 6 of Figure 1), it would serve as a conventional firewall. But now, in addition to the physical interface, we can filter the packets at any location, e.g. “after the firewall”, and “just before an application”. We may also use it as an in-kernel statistics mechanism, which works at any control granularity. These features are not currently available for any systems, to the best of our knowledge.

3) **The Port mechanism:** For the network monitoring and control mechanism at userland, we implemented the *port* mechanism. This is a feature that BPF provides on a per-NIC basis; NPF does per-VIF, realizing far greater flexibility. The challenges we faced in the implementation were mostly related to how to keep the BPF compatibility, which is
addressed as follows.

First, we represented the tapping interface as a regular file, port, in its corresponding VIF directory under /procfs/network. Additionally, to make the port interfaces under procfs visible at the standard mount location of BPF (/dev/bpf*), we devised a technique, which we call device stealing. It uses a soft link from the BPF location to the VIF port, to temporarily "steal" application's access to BPF device, as shown below. (The procedure is shown as a sequence of user commands just to deliver the idea.)

```
# ls /proc/network/em0
bandwidth recv type port
drops send weight
# cd /dev
# rm bpf0
# ln -s /proc/network/em0/port ./bpf0
```

Second, we extended packet format of the original BPF, to pass richer information to the userland, while providing compatibility to legacy programs. Figure 4 illustrates the change: bh_pld and bh_direction are added at the end of the BPF header. Note that the change does not require legacy applications to recompile, since we could adjust the size of the structure by changing bh_hdrlen (good design practice by BPF people!).

Third, for BPF compatibility of the port abstraction, we changed semantics of some ioctl() commands, upon which the BPF control is based. For example, each VIF is already attached to physical interface, and there is no need for explicit attachment, as is done in the BPF case. Hence, a command to attach a BPF device to a physical interface (BIOCSETIF) is simply ignored.

Lastly, we changed semantics of write operation on the interface. Packet diverting is an operation to divert packets onto userland for security inspection and proxy services, used in many security solutions. To support such an operation, we slightly modified the port interface, so that after proper operation the packets are re-inserted into the original flow. In the original mechanism, writing onto an interface causes injection of a raw packet into the physical layer. However, in our scheme, after the inspection of each packet, users can return it back to its original flow, just by writing the buffer to the same device file. Note that just this simple modification realizes diverting operation, by dropping the packets in the original flow through the filtering mechanism described above.

Since packet diverting violates BPF spirit of data writing, our mechanism is not BPF-compatible in a strict sense. Nevertheless, the dominant infrastructure is mostly for packet capturing, and there is only one major program (rarpd) that uses BPF writing on most Unix systems, to the best of our knowledge. Further, the most popular library, libpcap, is used for packet monitoring purpose. Hence, we believe that this change does not have a great impact on the compatibility.

With the changes described here, we can inherit the great assets of BPF and libpcap for monitoring of application behavior, while providing far greater functionality. Further, we can use the diverting mechanism for modification of packet data, for example, to remove viral-infected email attachments. It is also possible to employ proxy modules, utilizing the same mechanism. To improve the performance of packet filtering, we reused the binary translator system we developed for the filter.

IV. Evaluation

A. Qualitative Evaluation

In this section, we qualitatively evaluate whether our proposed primitive meets our claims.

[M1] Monitoring Granularity: Our scheme enables network monitoring of terminating entities at flexible granularity. For the example in Figure 1, VIF 1 and VIF 2 can monitor network I/O of p1 and p2, respectively. VIF 3 and VIF 4 are used to monitor sockets used by threads in p3. VIF 5 can be used to monitor the entire activity of p3, and VIF 6 of all the processes on the host. This flexibility is not possible with other mechanisms that work only at the network layer. For example, BPF cannot be used that way, since it is attached to the physical hardware at the bottom of network subsystem. Our mechanism can also be used for monitoring of packets that pass through a firewall (VIF 6 in Figure 1), to confirm functionality of the firewall rules, by monitoring a VIF connected to the root VIF.

[M2] Monitoring Facility and Control Primitive: Our approach realizes an integrated framework for network security and QoS, since each VIF has great control capability. We may grant all network resources to an application (for QoS), or revoke network access (for security). Indeed, they are opposite sides of the same coin, namely, resource control. Although we have had just the two extremes in the control, we can exploit various possibilities in between them, with this primitive. For example, we may just slow down suspicious traffic, not dropping all the packets.

The integration is beneficial to reduce system complexity, since network security and QoS both work toward resource management of the system. Another benefit is performance advantage: the integration of security and QoS would reduce overhead by eliminating redundant operations in the network subsystems, such as packet classification.

[M3] Network and End-host Perspectives: Recent trends in system security have been moving toward end-host and application oriented security control [13], [14]. However, controls in this direction tend to be OS-dependent, and thus, their security policy cannot be shared network-wide. Network oriented control, on the other hand, provides OS independent means for system security, but, in turn, it is unable to properly monitor process activities.

Our approach bridges the gap, by realizing detailed monitoring of network application, while keeping OS independence. It would also enable network-wide deployment
of a consistent security policy, once supported by every operating system in the target network.

All of the advantages above are realized by unique properties of the VIF model. However, they could be easily offset by poor performance. Hence, we examine efficiency of the implementation, next.

B. Quantitative Evaluation

In this section, we evaluate performance of the implementation, namely the Filter and the Port. Basic performance of the VIF kernel, such as overhead for VIF management and for traffic control, is better described in [27].

The hardware setting used in the experiments are shown in Figure 5. The hardware specification used in the experiments is as follows; Processor: Intel Celeron processor (2.0GHz), Chipset: Intel E7205, Main Memory 512MB, Hard Disk Drive: Maxtor 6E040L0 (40GB IDE), NIC: Intel 82562 (on-board, em driver), and Intel Pro/100 Ethernet (on PCI bus, fxp driver), Motherboard: GIGABYTE 8INXP. We used a D-Link DGS-1008D switch to carry out experiments (data plane), and a 100Mbps D-Link DI-604 switch for management of the experiments (control plane). We used FreeBSD 4.9 as a base system, and used the GENERIC kernel in the traffic source and a kernel with NETNICE patch in the target host.

1) The Filter mechanism: In this section, we evaluate performance of the filter mechanism. First, we measure overhead for filter instrumentation, a time it takes to translate the instructions in BPF machine language into native binary.

Second, we measure microscopic filtering performance, by counting CPU cycles needed for each packet processing. Lastly, for macroscopic measurement of the performance, we show impact of the controls on end-to-end throughput. In the last two experiments, we compare the results with a standard software firewall implementation available on the platform, the IP Firewall (ipfw) [5].

To measure the instrumentation overhead, we measured the latency for a sequence of system calls to install a filter code; open(), write(), and close(), to a filter file in a VIF directory. By the sequence of system calls, the filter code is translated into native binary form, and installed onto target VIF. We prepared several filter rules with different sizes, and plotted the latency of code generation against the number of instructions in the code (we converted the measured CPU cycles into milliseconds in this graph). We used the clock counter of Pentium processor by rdtsc(), and took 100 samples each. The minimum value in the run was used for further analysis, because other samples could be affected by interrupts.

The result is shown in Figure 6. As shown, there is a linear growth in the latency, depending on the number of instructions in the filter code. However, the slope is quite flat. The figure indicates that the minimum latency is approximately 25μsec, which can be justified for most network applications. Each additional host entry in the rule set added just two instructions, and thus, the code generator would scale well for most practical rule sets.
Next, we measured microscopic performance of the filter mechanism, comparing it with \texttt{ipfw}. For \texttt{ipfw}, we measured the latency of its filtering routine in \texttt{ip\_input()}. For NPF, we measured corresponding code segment, which calls the filter binary. We counted CPU cycles because the cycles consumed in the filtering processing is considered to be most accurate metric, which well describes nature of the implementations, not easily influenced by system factors, such as CPU clock rate, application protocol, system load, etc. To clarify the difference of the two implementations, we conducted several runs, changing the number of filter rules for host address matching. We used the minimum values, again, as their representatives.

Result is shown in Figure 7. It is easy to observe the linear growth in the cost for \texttt{ipfw}, as the number of the rules increases. On the other hand, the performance of NPF remains almost constant, regardless of the number of the rules applied. This owes in part to the efficiency of our scheme, and also to a rule optimizer of the code generator.

Although we may filter unnecessary traffic in userland [16], it is beneficial to drop unnecessary packets, as early as possible [32], [33]. This is particularly true with public servers, which might be targeted by DoS attacks. So, lastly, we measured application layer throughput under attack, to evaluate its overall behavior. For this purpose, we run traffic generator on the controller host, and generated traffic destined for an application on the NETNICE host, changing its intensity. On the NETNICE host, we run \texttt{nttcp} [34], a traffic benchmark program, and measured application layer throughput, between NETNICE and GENERIC.

Result is shown in Figure 8. It is easy to observe that both implementations exhibited similar performance. In this setting, NPF did not outperformed the \texttt{ipfw}, mainly due to the overhead of VIF processing needed for the handling of packets inside the VIF tree. A note on this issue is made in the discussion section.

2) The Port mechanism: To profile the port mechanism, next, we conducted a microscopic study and macroscopic study, again. The former is intended to measure the microscopic behavior of the implementation for each packet processing. The latter is to measure its overall behavior, including overhead of the VIF processing. In these experiments, we compared the implementation with the standard packet capturer, BPF.

For the microscopic measurement, we again used the clock counter and took 10,000 samples each, measuring CPU cycles needed to process each packet. We prepared three scenarios: (a) Accept, (b) Host match, and (c) Host miss. In the first scenario, all packets generated are accepted, utilizing a filter rule which matches every packet. In the Host match case, all generated traffic was to a single destination (say port 80), and therefore all traffic matched the target rule (something similar to “accept 80”), and therefore all traffic was accepted. In the last case, Host miss, packets are generated to an unused port on the host, and none of the traffic matched a rule, dropping all the traffic. For BPF, we measured cycles to execute a tapping routine. For NPF, we measured cycles to execute a corresponding routine for packet capturing. We used the minimum numbers for each run, as noted above. A listener process was run with a real time priority, to avoid packet dropping by the listener.
Figure 9 shows the result. We observed the advantage of NPF in all the cases. The better NPF performance is due to the binary translation and the VIF hierarchy. The dramatic difference between the accept and reject packets cases (i.e., first two vs. last scenarios) is because the packets are copied to userland when it is accepted.

Next, we assessed the impact of the implementations on throughput of mainstream traffic, as a macroscopic measurement. We used nttcp while running a monitoring process on the traffic.

Figure 10 shows the result. In the Accept All case and Host Match case, BPF exhibited better performance. We did not observe difference in the Host Miss case. As shown, the advantage we observed in the microscopic metric was offset by the VIF processing overhead, resulting in the overall performance.

3) Packet Diverting: Lastly, we measured performance of the diverting operation. We, again, used nttcp on a diverting VIF, and measured the throughput for both directions, input traffic and output traffic. Additionally, in this experiment, we tested both Fast Ethernet (FE) speed and Gigabit Ethernet (GbE) speed, to see how it scales. Again, to avoid packet dropping, we run the listening (diverting) process with a real time priority.

Figure 11 shows the result. As shown in the four bars on the left, we did not observe significant difference between the modes, in the Fast Ethernet setting. This is because the Celeron CPU was fast enough to divert all the packets at the link speed.

Right half of the figure is for the Gigabit Ethernet setting. In this case, we observed significant slowdown, in the diverting mode. The peak performance of the baseline cases were 526Mbps and 524Mbps, for Output traffic and for Input traffic, respectively, while the divert performance hit 236Mbps (45% of Baseline), and 258Mbps (49%). Since the diverting doubles the traffic between user space and the kernel, it is reasonable that performance of the diverting mode is bound by one half of the peak rate. Clearly, this mode should only be used when it is worth the benefits of diverting packets with such performance degradation.

V. DISCUSSION

We have presented the Netnice Packet Filter and profiled its properties. In this section, we first discuss issues related to the VIF mechanism as a building block for such security solutions. Then, we make some notes on the performance, and comments on applications of the mechanism, to clarify the contribution of the primitive for system security.

A. Toward a standard primitive for Security Control

It is hard to satisfy efficiency and flexibility with a single mechanism. Therefore, a desirable way is to allow users to choose where and how actual control takes place, considering objectives and performance requirement, with a reasonable framework. The filter mechanism takes charge of the performance part, and the port mechanism the flexibility part, in the framework. They realize a kernel service for security control with unprecedented characteristics, as discussed below.

First, because the mechanism can be used to inspect network I/O of applications, it allows for detailed stateful analysis of communication semantics, based on protocol knowledge. The monitoring program need to think just about communication protocol, leaving the selection of packets to inspect, to the tree-structure of the VIF system. This feature greatly simplifies monitoring programs, and realizes a variety of controls that existing mechanisms cannot easily realize.

Second, the filter mechanism realizes easy instrumentation of filter programs inside the kernel, for high-performance and flexible packet processing. The mechanism can also be used for statistics purpose. It is also easy to extend the functionality of the filter, just by supporting various filter architectures. For example, we may extend the interface to support context-free parsing of application protocols, or in-kernel IDS which may take signature file of Snort [28]. Since packet selection feature is inherently realized with the VIF structure itself, such a module can concentrate just on the parsing task or signature matching, significantly reducing design complexity of such extensions.
Third, the port mechanism provides monitoring and control of traffic with great flexibility, while providing compatibility with the dominant monitoring infrastructure, BPF. This allows users to benefit from all applications already built for the traditional interface, in addition to the great programming freedom it offers for future systems.

The downside of the scheme is that, first and foremost, it requires modification of the OS kernel, which is not always an acceptable option. Another limitation includes performance of the userland monitoring and packet diverting, due to the cost of buffer copy. We believe that this limitation can be overcome by clever buffer management schemes, like [35]. We further discuss the performance issue, in the following section.

B. Performance Consideration

In the measurements of throughput, we observed that the performance of NPF is equivalent to, or, slightly less than, the performance of existing applications. This is mostly due to the overhead incurred by VIF system, not by performance of the extensions we added, as we reviewed in the microscopic evaluations. Note that the overhead of the VIF system includes cost for queuing control and bandwidth management, which are not included in the control cases (ipfw and BPF in our study). Accordingly, we underestimate its performance unless the controls are coupled with queuing and bandwidth control modules. We omitted comparison with such cases, to avoid controversy about fairness in the experiments.

Another source of performance gain comes from structured nature of the VIF scheme; we can selectively apply filter and port mechanisms to VIFs, thereby avoiding packet classification, needed for every other packet controlling and monitoring scheme. For example, to monitor traffic for HTTP, we simply wrap a web browser and tap every packet on the VIF, although we need to apply packet classifiers to selectively monitor the packets destined for port 80 of TCP in other approaches. In the NPF case, we simply do not see unnecessary packets, since they are not delivered to the VIF that we monitor.

C. Possible Security Applications

We believe that fair evaluation is made only under realistic conditions with real applications which make most of the control primitive. Actually, because it is a control primitive for various solutions, it must be coupled with security policy and its management mechanism, to be an actual security solution. For example, by coupling with policies for a particular user, the solution would be personal firewall. Likewise, if an administrator provides policies consistent across a specific network, it would become distributed firewall. Hence, we make some comments on the application issue of the control primitive, in this last section.

a) Personal Firewall: The Netnice Packet Filter (NPF) facilitates a variety of controls that neither intermediate node firewalls, nor most of the end-host firewall implementations, can realize. NPF can allow communication just to authorized application programs while prohibiting communication by unauthorized programs. One way to realize such control is to wrap each process by a VIF, and apply filters for each of the applications. This procedure can be automated by a security manager with policy database, which keeps filters for each application.

Note that some of the functionality has been already realized on personal firewalls on Windows operating system [8]–[10]. They are, however, built upon hooking at device driver layer, and do not possess flexibility of control, or monitoring granularity we provide.

Further, our proposed primitive realizes integration of security management and QoS management, that could inspire programmers to write various new applications, some of which are not possible with existing approaches; For example, feedback-based QoS control, based on the statistics capability, or automated reservation of path bandwidth, utilizing the monitoring capability.

b) Intrusion Detection Systems: Thanks to the BPF compatibility, the Netnice Packet Filter can run standard IDS, such as Snort [28], without any modification. However, we expect much more in this direction. Although most IDS systems are passive entities in network security, the Netnice Packet Filter can make it an active center of security control, by the integration of monitoring and control.

For example, we may detect DoS attack and automatically set filter rules onto the VIF it is monitoring to drop the offensive packets. This approach would be beneficial particularly for high-performance servers, not to degrade its service throughput under attack.

c) Intrusion Prevention Systems: Latest approaches for advanced security avoid using predefined signature files, but rather, try to define appropriate behavior of each application, denying any violating attempts at endpoint [13], [14]. These proactive security approaches would realize far secure systems, which protect the systems even from attack of unknown types. However, they have been built on monitoring of system calls, and thus, OS independence is compromised in the approaches, restricting deployment of the approach.

Our approach easily expands the limit, simply by applying appropriate filters onto each VIF that describes approved behaviors of each application. We believe that the application monitoring capability and its OS independent property would bring about a breakthrough in the pioneering security solutions of this sort.

d) Distributed Firewall: Coupled with a network-wide policy management mechanism, like [3], NPF would serve as a building block for distributed security solution. Current distributed firewall systems are built on rules described in network oriented manner, allowing defiant applications to communicate using permitted protocols. In this regard, our monitoring scheme furnishes desirable infrastructure to realize more robust systems, although a proposal of complete solution is left for future work.

VI. CONCLUDING REMARKS

We addressed the problem of structural mismatches in end-host operating systems for network monitoring and control,
utilizing hierarchical virtual network interfaces. We proposed
to extend the basic VIF model to support filter mechanism and port mechanism, for integration of in-kernel and userland security solutions, with a unified control model.

Utilizing a prototype implementation we developed on FreeBSD, we qualitatively and quantitatively evaluated the scheme, and showed that the VIF extensions can bridge the three gaps for end-host oriented network monitoring and control: i) control at flexible granularity, ii) integrated monitoring facility and control primitive, and iii) network perspective and end-host perspective. We believe that identification of the gaps is one of the major contribution of the paper.

As a primitive for network security and QoS control, the model exhibited advantages over existing conventional software firewalls and packet monitors, as follows: i) ability to monitor terminating entities at arbitrary granularity, ii) a single consistent framework for both network security and network Quality of Service, iii) OS independence, iv) efficiency as a control primitive, v) compatibility with BPF interface and its applications, and vi) flexibility for future functional expansion. In addition, this mechanism serves as a quick solution for monitoring of application network I/O, and checking of incoming traffic that pass through a firewall, with various libpcap tools, such as tcpdump.

We have ported the mechanism onto several major operating systems, and made the implementations publicly available at http://www.netnice.org. We are currently implementing a module for a GUI rule-builder, Firewall Builder [12], and a module for a famous PC IDS, Snort [28], for integration with the VIF system. Our future work includes design of a multi-platform distributed security solution, and expansion of the in-kernel packet processor for more generic control.

ACKNOWLEDGMENT
This work was inspired and owes greatly to the original authors of the Berkeley Packet Filter. Authors thank Michael Guaitieri and Nicolas Baldi for their assistance in the development of the software. The work was supported in part by NSF under grants 0087609, 0121658, and 0325535, as well as by the Information-technology Promotion Agency, Japan.

REFERENCES