An operating system independent API for firewall control: design and implementation for Linux

Sebastian Kiesel
NEC Europe Ltd., NEC Laboratories Europe
kiesel@nw.neclab.eu

Jochen Kögel
Institute of Communication Networks
and Computer Engineering (IKR)
Universität Stuttgart, Germany
jochen.koegel@ikr.uni-stuttgart.de

***Abstract*—Firewalls are a crucial building block for securing interconnections between networks of different security domains. Newer IP based applications such as IP telephony require that packet filters are configured dynamically by session-aware entities. Several architectures and protocols were proposed for firewall remote control (e. g., IETF MIDCOM and NSIS). However, on the nodes that could host the control entity no common local interface for managing packet filter rules exists. This forces developers of firewall control entities to use firewall specific interfaces and limits portability across operating systems and firewall implementations. This paper describes the design of an operating system independent interface, which allows the firewall control entity to modify firewall rules from user space. We implemented this interface for Netfilter chains of the Linux operating system kernel taking functional and security requirements into account. Based on measurements, we analyze the performance characteristics and conclude that the approach is feasible for small to medium network setups.**

I. Introduction

Firewalls are a proven and widely deployed means for securing network interconnections. However, the usage of out-of-band signaling protocols such as SIP for IP telephony and multimedia applications imposes new challenges on them. While in the past firewall configuration usually was static and self-contained these new applications require dynamic changes to the rulesets, depending on the session signaling. Various protocols for firewall control have been proposed and implemented as prototypes. However, few of the prototypes can actually change the firewall ruleset with reasonable efficiency. This is because the interface for modifying firewall rulesets in the operating system is not standardized and usually rather complex. This paper proposes an operating system independent API, which intentionally covers only the most important functions, in order to keep it as easy to use as possible. A prototype implementation shows the feasibility; measurements are used to assess the performance of this approach.

The reminder of this paper is organized as follows: Section II shows why dynamic firewall control is needed and gives a classification of the various control architectures that have been proposed. The first operating system for which we have implemented our API is Linux. Its firewall subsystem Netfilter will be introduced in section III. Section IV covers the design of the API. Section V details the relevant parameters for characterizing firewall performance in a VoIP scenario. Measurement results from our API running on Linux are presented in section VI. The paper concludes with section VII.

II. Firewall Control Frameworks and Protocols

A. The SIP/RTP firewall traversal problem

A *firewall* is one or a group of network elements that enforce an access control policy on the traffic at the border between network domains with different security levels and requirements. That is, a firewall is basically a gateway that relays traffic from one domain to the other, but only if the traffic is compliant to a specific security policy. Often, it is implemented as a *packet filter*, i. e., as a router that forwards only compliant packets while discarding the others.

The firewall policy is usually expressed by a list of *policy rules*, each being “the binding of a set of actions to a set of conditions” [1]. A policy rule that allows specific packets to pass a packet filter is also called a *pinhole*. Traditionally, in many realistic deployment scenarios a *static* configuration of the pinhole list has been sufficient, often complemented by *connection tracking* inside the packet filter (i. e., keeping state information and allowing inbound replies only after an outbound request). This is possible due to the in-band signaling and the “well known port numbers”-concept most traditional Internet services such as WWW or e-mail follow.

However, some applications differ from these concepts. In the considered IP telephony scenarios, the Session Initiation

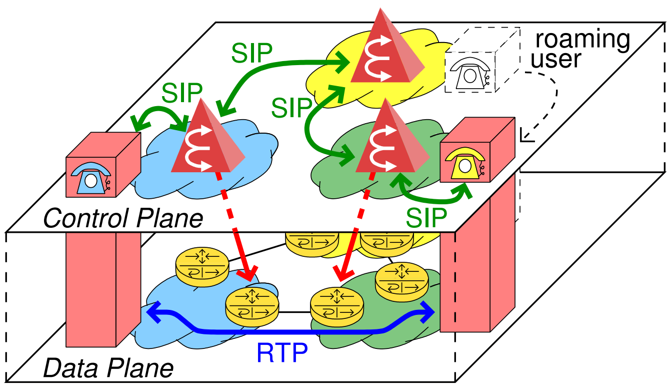

Fig. 1. SIP signaling messages and RTP media streams may travel on different paths through the network

Protocol (SIP) is used for call control signaling. The actual user data (speech) is transported in separate flows, using the Real-time Transport Protocol (RTP). RTP stream parameters are negotiated *dynamically* using Session Description Protocol (SDP) messages embedded in the SIP signaling. These parameters include the codec, bit rate, etc., but also information which is relevant for firewall configuration, such as IP addresses and UDP port numbers. This dependency between protocols is complicated by the fact that signaling messages and media streams may travel on different paths through the network (e. g., to roaming users, see Fig. 1).

Due to this dynamic nature of SIP/SDP and RTP, a static configuration of packet filters on the media path is no real option, as it would either block all RTP streams or would have to allow all UDP traffic, rendering the firewall's protection almost useless in many network scenarios. Instead, firewalls have to interact with the session signaling, in order to allow only those media streams that have been announced by means of signaling first. Several architectures and protocols have been proposed for this purpose; an overview will be given next.

B. *Classification of SIP/RTP firewall architectures*

Fig. 2 shows a classification of SIP/RTP firewall architectures. In the following, the logical entity that handles the signaling messages is referred to as *signaling component*, the entity for the media streams will be called *media component*. An important criterion is whether the signaling component and the media component are located in one common or two separate physical network elements. In both cases, if several parallel network interconnection points are available between the security domains, one can further distinguish whether the architecture forces a media stream to traverse one specific media component, or whether the decision which one to use is left up to a dynamic routing protocol.

The main advantage of placing signaling and media component into one network element is that the control interface between these components will be an internal one, i. e., no signaling protocol is needed. However, this solution prohibits scenarios like the one depicted in Fig. 1, where the signaling messages for a roaming user are sent via his home network, whereas the media streams flow on a direct path to the communication partner, resulting in lower latencies.

1) Protocol helpers: If an enforcement of the media path is not required, a full-fledged protocol entity for the session signaling protocol is not always needed. A passive analysis of the signaling messages may be sufficient to determine the parameters needed to open the pinholes for the media streams. The *protocol helpers* within the *Netfilter* framework (see below) implement this concept successfully for many protocols such as FTP. However, there are several drawbacks of integrating a SIP/SDP parser directly into the packet filter:

- SIP is a text based protocol allowing various formats for the same message. It may use different transport protocols that may fragment a message over several IP packets. Therefore, a rather complex parser is required to extract the SDP information. This is error-prone and may increase the likelyhood of implementation weaknesses.
- SIP messages may be encrypted in order to provide confidentiality in insecure networks. In such cases, the protocol helpers are not able to extract the required information from the SDP.
- Passive analysis of SIP does not allow policy enforcement on a session level, e. g., rejecting calls from specific users. Therefore, an additional SIP proxy or B2BUA will be needed in many scenarios.
- It is difficult to determine when the pinholes have to be closed again, as even SIP messages may travel on different paths through the network (e. g., the INVITE might be sent via proxies while the BYE is sent directly).
- As already mentioned this scheme fails if media streams flow through a different firewall than the corresponding signaling messages. Therefore, this solution most useful for smaller sites that have only one Internet access link.

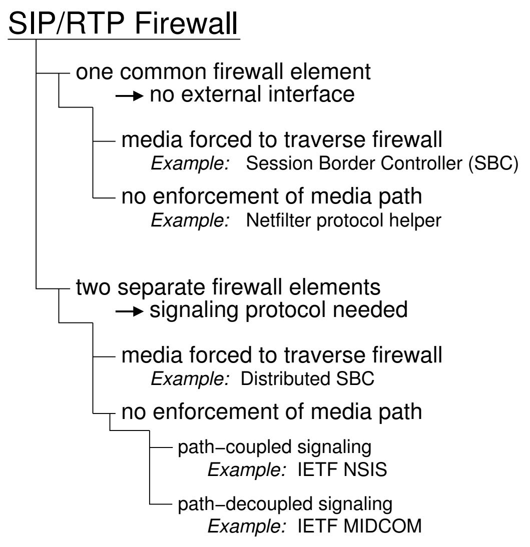
SIP/RTP Firewall
one common firewall element
→ no external interface
media forced to traverse firewall
Example: Session Border Controller (SBC)
no enforcement of media path
Example: Netfilter protocol helper
two separate firewall elements
→ signaling protocol needed
media forced to traverse firewall
Example: Distributed SBC
no enforcement of media path
path−coupled signaling
Example: IETF NSIS
path−decoupled signaling
Example: IETF MIDCOM

Fig. 2. Classification of SIP/RTP firewall architectures

2) Session Border Controllers: An SBC combines a SIP B2BUA as signaling component and a media component, such as an RTP proxy or a packet filter, in one box. SDP information in the signaling messages will be modified by the B2BUA, causing the multimedia endpoints to send the media streams not to the other party but to the SBC, thereby forcing the media path to go through the SBC. This solution may overcome the above-mentioned problems at the cost of having to process SIP messages and keeping call state at every network border, which increases the total effort. Furthermore, SBCs introduce single points of failure that cannot be circumvented by rerouting on the IP layer.

3) Distributed Session Border Controllers: Having the signaling and media components placed on different network elements allows to process signaling messages at a central place in the network, in conjunction with the authorization on a session basis. Then, only the media streams will be filtered at the edge. This also allows media streams taking an optimized path with potentially smaller end-to-end delays, as discussed above. However, a signaling protocol is needed between the two components. When using a distributed SBC, its signaling component will modify the SDP information in order to force the media traverse one specific media component.

In contrast, the following two solutions do not influence the path of the media streams between the two multimedia endpoints. It is determined by the IP layer routing protocol, which may be able reroute in case of a node or link failure.

4) IETF NSIS: The NSIS architecture [2], implemented by the GIST and NAT/FW NSLP protocol layers, follows the principle of *path-coupled firewall signaling*. Based on the normal routing tables, signaling messages are sent along the (future) media path, in order to announce a new flow and to create and maintain the necessary state information.

5) IETF MIDCOM / SIMCO: When using *path-decoupled signaling*, firewalls on the media path may be controlled directly from SIP entities in the core of the network (Fig. 3). As fewer parties are involved, authentication is less complex and the protocol can work more efficiently. However, the central entities need to know the network topology and routing state, in order to configure those firewalls which are actually on the media path. Several protocols implement this idea, e. g., the MIDCOM MIB [3], H.248 [4], or SIMCO [5].

C. The need for a packet filter control API

The previous section has outlined that there is a wide solution space for the coordination between the signaling and media components of a SIP/RTP firewall. As each of these schemes has its individual advantages and disadvantages it is likely that different protocols will be deployed in different networks. On the other side, there are also various options for the media component, e. g., RTP proxies or packet filters.

A popular choice for prototype development and deployment in small to medium sized networks is to use the packet filter included in many personal computer operating systems, such as Linux or OpenBSD. However, the application programming interface (API) for adding packet filter rules into the operating system kernel is not standardized and in many cases rather complex. To facilitate development of new firewall solutions, this paper proposes such an API which can be used by firewall control frameworks. An implementation for Linux shows the feasibility and the performance of this approach.

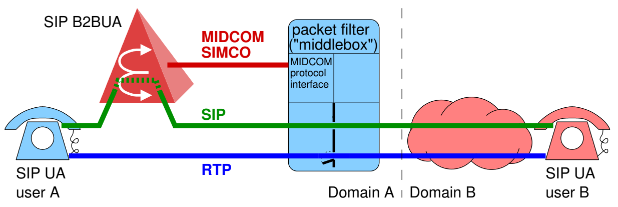

Fig. 3. The IETF MIDCOM architecture (acc. to [6])

III. Linux Netfilter

Netfilter is the Linux packet filtering framework that provides functions for packet filtering and mangling in the IP stack of the Linux kernel. On initialization, the functions register with hooks in the kernel and are called when packets pass the hook. Command line tools and libraries allow to configure Netfilter from user space.

A. Filter and Conntrack

The Filter function performs stateless packet filtering and allows to specify sophisticated rules. In contrast, Conntrack is the stateful connection tracking function. For other Netfilter functions (mangle, nat) and further details on the Linux networking architecture refer to [7].

Filter registers at hooks for communication to and from the local host (input and output hooks) as well as at the forwarding hook that is called for packets routed through the host. A Filter rule consists of conditions (called matches) and an action (called target) that applies if all conditions hold. Basic matches are, e. g., protocol, IP address and port numbers while targets are, e. g., accept or drop. Filter keeps several filter rules in linear structures, so-called chains. A global filter table keeps references to all chains and distinguishes between built-in and user-defined chains: the former are named according to the hooks and are always present. They serve as starting point for filtering. The latter can serve as targets and allow better structuring and administration of the firewall configuration. Once a packet passes a hook, the kernel processes the rules of the corresponding chain linearly until a rule matches. If the target is a user chain, the kernel continues on that chain until a target determines the final action (e. g., accept, drop). If no rule matches, the per-hook default policy applies. There are also hooks for packet mangling and NAT, that operate on separate tables and chains. They are not considered here.

In contrast to the Filter function, the connection tracking system (Conntrack) enables stateful packet filtering. Conntrack registers with highest priority at the very first hook that is called when a received packet enters the stack. It classifies packets whether they belong to a new, established or expected flow. It may also verify sequence numbers in transport layer protocols. The classification is based on a hash table lookup, where the hash is calculated over the five tuple or parts of it. Conntrack stores the classification result in packet metadata that can be accessed from Filter rules, e. g., in order to accept all packets belonging to established connections. When connections terminate or after an inactivity timeout Conntrack removes the flows from its hash table.

B. Focus on Filter

Obviously, the hash-based conntrack module performs faster for a high number of rules than linear search in the Filter chains. However, a hash-based algorithm works only for exact (five tuple) matches and not for address ranges. Yet, many firewall control protocols, such as MIDCOM or NSIS, which could make use of the framework developed here, require support for address and port ranges and wildcarding (which can be mapped to ranges). Therefore, our first approach, which is presented here, is based on Filter rules only. Using combinations of chains, conntrack or other extensions like ipset (hash, bitmap and tree based filtering) is for further study.

Our solution works on a user-defined chain that is modified by the control framework, while all other chains and configurations remain untouched. This enables the administrator to set up other firewall rules and to define how the controlled chain is embedded in this configuration. However, one problem with Conntrack and firewall control remains: typical configurations start with rules that accept all packets belonging to established connections, i.e., that are present in Conntrack's hash table. In such cases, sending a packet that matches a pinhole in the controlled chain results in a new Conntrack entry, allowing all further packets to be accepted until the connection terminates. Removing pinholes, e. g., in case of attacks, would not neccessarily cut off all unwanted traffic. This is not acceptable.

There are three possible solutions to this problem: The first and simplest solution is to completely disable Conntrack by not loading the modules and not referring to Contrack from Filter chains. In a more sophisticated second solution, the firewall administrator enables Conntrack only for permanent rules that are not going to be managed by the control framework. Thus, Netfilter chains use the Conntrack classification result, e. g., for TCP connections and DNS only, but not for other UDP traffic (possibly VoIP). Removing flows from Conntrack once the pinhole is closed would be the third solution. This requires much more additional functionality in the backend and knowledge about which Conntrack entries originate from a certain rule (n:1 mapping in case of pinholes with ranges).

C. Managing filter rules

The library *libiptc* provides functions for managing filter rules that operate on chains of a table (see Fig. 4). Libiptc performs changes on a user space representation of the chains, which differs from the kernel space representation: In the kernel, Netfilter rules are stored consecutively in memory as a *blob* (binary large object block), optimized for fast parameter matching. Contrary, the user space representation consists of linked lists of dynamically allocated objects. That is, the user space representation is optimized for efficient modifications.

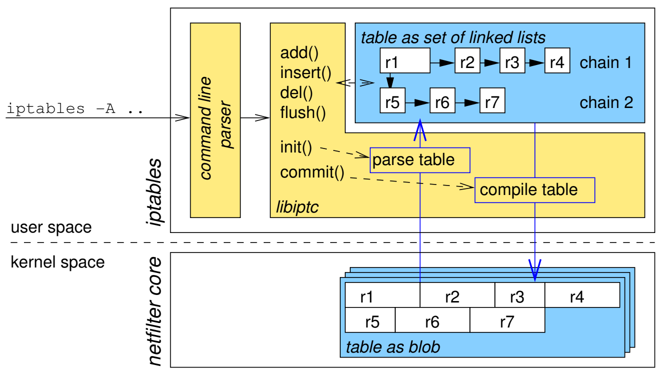

Fig. 4. Management of Netfilter rules with libiptc

The usual way for Netfilter configuration is using the command line tool *iptables*, which is basically a command line parser that uses libiptc for rule management. Before iptables can perform any changes to filter rules, it has to call the *init* function of libiptc that parses the blob of the requested table and creates the user space representation. Afterwards, iptables calls libiptc functions, which change the user space representation (e. g., adding or deleting rules). When finished, it calls the *commit* function, which triggers the compilation of the user space representation into the blob and eventually the replacement of the old ruleset in the kernel.

For large rulesets, parsing and compiling the ruleset requires a high effort compared to adding or deleting a few rules. That is, from a performance point of view, rarely changing many rules at once is better than doing frequent minor changes. It becomes obvious, that developers did not design libiptc for dynamic rule management in the first place. In most Netfilter-based firewalls, this is not a problem, since static rules and protocol helpers together with Conntrack perform well. However, the requirements for this work are different and thus this approach of ruleset management can become a bottleneck. Thus, the overhead imposed must be considered when designing and deploying the framework.

IV. Design and implementation of the pinhole API

Given the multitude of firewall control frameworks and the different UNIX-like operating systems the goal of this project was to specify a simple, flexible, and secure API for adding packet filter rules into the operating system kernel. The API implementation was first used to connect our SIMCO server [8] with the Linux Netfilter.

A. Requirements and architecture

The API was designed to be as easy to use as possible: the *add()* call allows to open a new pinhole in the firewall, with specified parameters such as IP adresses and port numbers. It returns an ID, which can be used to remove the pinhole with the *delete()* call. As changing the firewall ruleset may be resource-consuming, these requests are not processed immediately. Instead, *commit()* has to be called in order to make all pending requests effective. The decision when to call *commit()* is left to the application. Our SIMCO server measures the time spent for each *commit()* call. The next call may be scheduled only after a pause, which is at least C times as long. This scheme yields short response times as long as the system is only slightly loaded. If, however, the ruleset grows larger, no more than $\frac{1}{1+C}$ of the CPU time will be spent for updates.

The code was split into two modules, a frontend module to be integrated into the firewall control protocol entity and a backend which runs as a standalone auxiliary process. Fig. 5 shows these modules together with our SIMCO server. This design has been chosen in order to fulfill various requirements:

- Simple integration of the frontend into various firewall control frameworks: no special libraries such as libiptc have to be linked against the main process.

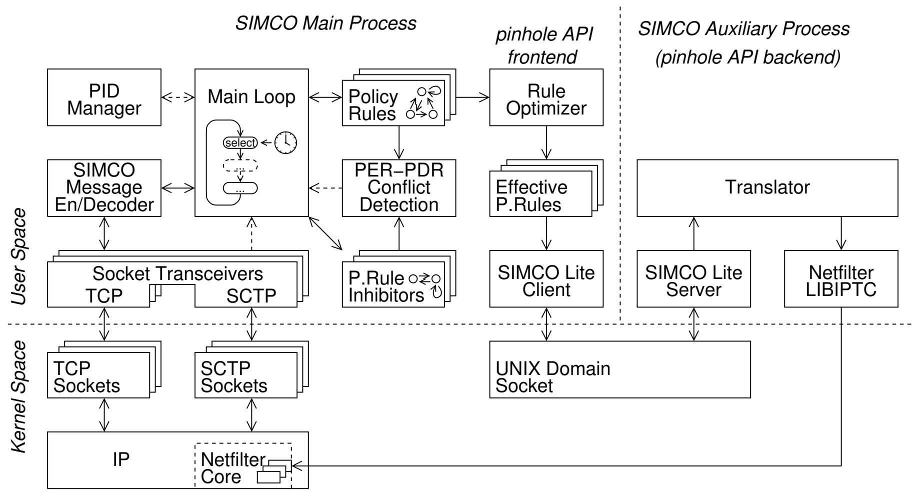

Fig. 5. SIMCO server architecture block diagram with Netfilter backend

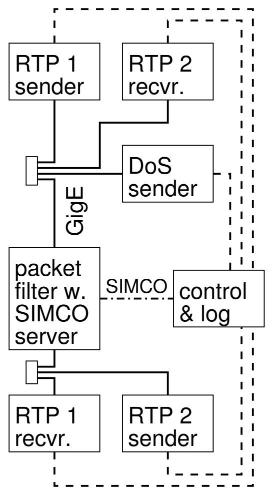

Fig. 6. Measurement testbed

- Operating System independence: In order to support different operating systems only the backend module has to be modified. Then, all firewall control frameworks that use the frontend may be ported easily.
- Robustness: in case of malfunction of the main process the auxiliary process can remove all the firewall rules which were installed by the firewall control framework. In case of errors while installing a new ruleset it can perform a rollback to the last known good state.
- Security: changing the firewall ruleset in the kernel requires elevated (e. g., “root”) privileges. However, processes that accept connections from the network often are vulnerable against attacks due to implementation weaknesses, such as buffer overflows. Therefore, it is desirable to execute the main process with lower privileges, as this may limit the damage caused by potential exploitations. The interface between frontend and backend is used for this privilege separation.
- Optimization: the ruleset may be optimized for better filtering performance. Some general optimizations, such as merging adjacent or overlapping pinholes, may be performed in the main process. Other optimizations specific to the packet filter itself are better performed in the operating system specific backend. Currently, no optimization algorithms are implemented.

B. Internal communication

Communication between the main and auxiliary process uses a Unix domain socket and a simple protocol, which reuses the message format of the SIMCO PER message. Therefore, this internal protocol is labeled as “SIMCO lite” in Fig. 5. Compared to SIMCO, some simplifying assumptions are made: The backend process can handle only one incoming connection from the frontend. No cryptographic message protection is necessary as communication is host-internal. The backend cannot handle rule lifetimes or conflicts between contradicting rules. Therefore, these checks have to be done in the frontend. The protocol allows for incremental changes as well as for a complete exchange of the ruleset. As explained above, libiptc performance is largely independent of the number of rules that are updated at once. Therefore, our SIMCO server simply transmits the complete ruleset to the backend whenever changes are required.

C. Measurement Testbed

For functional tests and performance measurements (see next section) a testbed was installed (see Fig. 6). It consists of seven personal computers, each with a 2.8 GHz Pentium 4 CPU, Intel Gigabit Ethernet cards and the Linux kernel version 2.6.22. The firewall computer with our SIMCO server and the firewall API is located between the insecure network (top) and the secure “inside” network (bottom), each consisting of a switched Gigabit Ethernet segment. A third Ethernet segment (dashed lines) is used for control and management.

Using the SIMCO protocol the controller can add or remove pinholes from the firewall. Packet senders on both sides of the firewall generate corresponding UDP packet flows, with packet sizes (160 bytes payload) and interarrival times (20 ms) that are typical for many VoIP applications. The packet receivers report delays and lost packets to the control server. Using four computers for sending and receiving the emulated RTP flows ensures that the performance bottleneck is actually in the firewall computer. One further computer allows to send “DoS attack” traffic at a configurable rate, which does not match any pinhole in the firewall.

V. Packet filter performance characteristics

In the following we discuss the performance impact of a Linux based firewall using our API and SIMCO. We assume a scenario where the firewall has to inspect the traffic from a large number of simultaneous calls.

A. Relevant characteristics

From a subscriber’s viewpoint, three parameters are relevant to characterize the performance of a VoIP packet filter:

- The mean processing delay δ_F of legitimate packets while traversing the packet filter contributes to the end-to-end delay of IP packets and thus to the ear-to-mouth delay, which should be as small as possible [9].
- Packet losses decrease the voice quality. Even though most speech codecs can tolerate some packet losses, the probability p_F that a legitimate packet is lost in the packet filter should be as close to zero as possible.
- The mean delay δ_u for adding a new pinhole to the packet filter contributes to the post-selection delay or the answer signal delay [10]. Other contributors to these call setup delays, which are unpleasant to subscribers, are the firewall control signaling and the transport of the corresponding signaling messages. The latter was studied in detail in [11].

From the network operator's viewpoint the most important performance characteristic is the number of simultaneous sessions m that can be handled by the packet filter without exceeding limits for the quality parameters given above. Attention has to be paid to the fact that a media stream may have to traverse several firewalls, each contributing to these end-to-end metrics. Furthermore, the packet filter has to be dimensioned such that it can process and reject illegal packets at the highest assumed rate (esp. Denial-of-Service attacks by "flooding") without impeding its ability to forward the legitimate traffic.

B. *Influencing factors*

A personal computer usually has only one or few central processing units (CPU). Several tasks can be processed quasi-simultaneously by allocating CPU time slices to them by turns. In the considered firewall scenario the main tasks are:

- The SIMCO main process, which communicates with the SIMCO agents, and manages the policy rules and their lifetimes, etc.
- The SIMCO auxiliary process, which installs the effective rule set into the operating system kernel.
- IP packet handling in the kernel: reception, access control, routing, transmission, etc.
- Other tasks in the operating system kernel or in other processes, e. g. recording of event log files.

Some of these tasks are implemented as processes in the user space, others take place in the operating system kernel or in interrupt handlers. Context switches between these different tasks cause additional latencies. IP packets may be buffered at various places while travelling through the IP stack. Several packets may be processed in one time slice, especially at high packet rates. Due to this complex interdependencies it is hardly possible to develop a simple analytic model for the packet delays or the maximum packet rate that can be handled. However, it is possible to identify basic parameters on which the CPU utilization for updating the ruleset and for filtering the packets depends on. According to our measurements these two tasks consume the biggest share of CPU ressources. The limited availability of these ressources eventually leads to delays under high loads and causes the system to become overloaded if these parameters exceed certain limits.

Assuming a mean number of simultaneous multimedia sessions m and a mean session duration h, the mean session arrival rate follows as $\lambda_\mathrm{C} = \frac{m}{h}$. For each multimedia session two pinholes (one per direction) have to be opened at the beginning and closed at the end. Therefore, the mean rate of ruleset changes is

$$\lambda_\mathrm{uc} = 2 \cdot 2 \cdot \lambda_\mathrm{C} = 4\frac{m}{h} \tag{1}$$

and the number of pinholes for the media streams is $2m$ Furthermore, a (usually constant) number c of policy rules is needed for SIP and other auxiliary protocols, e. g., DNS. Thus, the mean total number of policy rules in the packet filter is

$$U = 2m + c\ . \tag{2}$$

The CPU utilization caused by ruleset updates depends both on the number of rules U in the ruleset and on the ruleset change rate λ_uc. The dependence on U is due to the fact that for each change, the complete ruleset is read from kernel space to user space, modified there, and written back (see Sec. III-C). If λ_uc increases this process will be triggered more often, causing higher CPU utilization. However, to ensure consistency, ruleset updates must not be performed in parallel. That is, an arriving change request may have to wait until the current update process is completed. With higher λ_uc the probability increases that several requests arrive during that waiting time; they can be processed at once in the next update cycle.

A packet filter with white list configuration compares the characteristic parameters of each received IP packet with the ruleset. If a matching policy rule can be found, the packet will be forwarded, otherwise it will be discarded. Netfilter usually searches linearly in the list of policy rules. Assuming that the legitimate traffic spreads evenly over the pinholes, $\frac{U}{2}$ comparisons are needed per RTP packet in the mean. In contrast, disallowed packets have to be compared with all U rules until it is clear that they do not conform with the security policy and have to be discarded. Regarding the number of rule comparisons the illegal traffic therefore causes a higher CPU utilization than the legitimate traffic. Other (e.g., hash-based) approaches for storing and searching within the ruleset, may reduce this dependence from the ruleset size (see above). However, problems occur if the pinhole specifications include ranges for some parameters. In the following we will analyze a configuration based on Netfilter's Filter function only.

With two media streams per session and assuming that all used codecs geneate the same mean packet rate λ_M the packet rate at the packet filter is

$$\lambda_\mathrm{i} = 2m \cdot \lambda_\mathrm{M} + \lambda_\mathrm{X}\ , \tag{3}$$

where λ_X denotes the rate of illegal packets sent by attackers. The signaling traffic is neglected here. This requires a rate of

$$\lambda_\mathrm{F} = 2m \cdot \lambda_\mathrm{M} \cdot \frac{1}{2}U + \lambda_\mathrm{X} \cdot U = U(m \cdot \lambda_\mathrm{M} + \lambda_\mathrm{X}) \tag{4}$$

rule comparisons. They compete with the other tasks of the firewall (see above) for the same CPU ressources.

VI. Measurement results

A. Ruleset update transaction delay

The first measurement examines the transaction delays for adding or deleting two pinholes corresponding to a multimedia session, as a function of the number U of pinholes that are already in the ruleset (Fig. 7). In order to isolate this effect, no traffic was sent through the packet filter during this measurement. One hundred measurement cycles were performed for each measurement point. In every cycle two policy rules were added and removed again. The figures show the mean transaction delays.

As already mentioned, for each change the complete ruleset is read from kernel space to user space, modified there, and written back to the kernel space. These operations are performed in libiptc and the contribution of the expensive *commit* operation is indicated in Fig. 7. The delays grow almost perfectly linear with the ruleset size.

The figure also shows measurement results for changing the ruleset with the standard *iptables-restore* command. This mechanism is faster than the *iptables* command, as it is also able to commit several changes at once. Thus, this is the next best choice for a firewall control daemon that does not directly use libiptc or our API and therefore, we give those values for comparison. As expected, the commit operation takes the same time for our API implementation and *iptables-restore*. The larger overall delay of *iptables-restore* might result from the effort for processing text based rule descriptions.

B. Packet loss and packet delay with static ruleset

The second series of measurement studies the packet processing delays δ_{F} and the packet loss probability p_{F} of legitimate packets in the packet filter, depending on the number of simultaneous multimedia sessions m.

In this measurement, no attack traffic was sent (i. e., $\lambda_{\mathrm{X}} = 0$). Furthermore, no sessions were established or terminated during one measurement, i. e., the SIMCO server was not active ($h \to \infty$).

Fig. 8 shows δ_{F} and p_{F} over the number of pinholes $U = 2m$, which were added before each measurement. Load generators were used to send a packet flow through each pinhole. Emulating RTP streams with G.711 voice codec, these flows consisted of 50 UDP packets per second, each carrying 160 bytes payload. In the depicted parameter range, the relative utilization of the Gigabit Ethernet segments was only few percents. The measurements show that up to about $\hat{m} \approx 500$ simultaneous sessions, which correspond to an overall packet rate $\hat{\lambda}_{\mathrm{i}} \approx 50000\frac{1}{\mathrm{s}}$ and a ruleset size of $\hat{U} \approx 1000$, no packet losses occur. Packet delays are very small and in the same order of magnitude as the measuring accuracy; they cannot be distinguished from the measurements with Netfilter disabled (i. e., the computer working as router). Above this stability boundary $\hat{m}$ both packet loss and delays increase dramatically and prohibit any reasonable communication.

C. Defense against malicious packet floods

The main task of a firewall is to reject traffic that does not conform to the security policy, including packet floods from Denial-of-Service attacks. At the same time it must be able to forward the legitimate traffic. The maximum rate of malicious traffic that can be rejected without impairing the legitimate flows has been studied in the next series of measurement.

The basic parameter for the graphs shown in Fig. 9 is again the number of simultaneous sessions m. Using the SIMCO server and a SIMCO load generator pinholes with a constant lifetime of $h = 180\mathrm{s}$ were added. The interarrival times of new pinholes were negative-exponentially distributed and the mean value was chosen such that $U = 2m$ pinholes were open at the same time in the mean (lower X-axis). Packet generators were dynamically configured to send flows with a packet rate of $\lambda_{\mathrm{M}} = 50\frac{1}{\mathrm{s}}$ through each pinhole, i. e., the mean overall rate of legitimate packets was $m \cdot 100\frac{1}{\mathrm{s}}$ (lower part of the bars, upper X-axis). In this state, still without sending malicious traffic, the mean delay for a ruleset update was measured (lower line, right Y-axis). Again, this delay turned out to be small, with a strong increase above ca. 450 to 500 simultaneous sessions.

Next, an additional packet generator was used to send UDP packets (160 bytes payload) not matching any pinhole, which therefore must be discarded. Starting at zero the packet rate of this attack traffic was increased until the packet filter's ability to forward the legitimate traffic was impaired so much that the packet loss probability exceeded 0.1% (upper and middle part of the bars, left Y-axis). This maximum rate of attack traffic

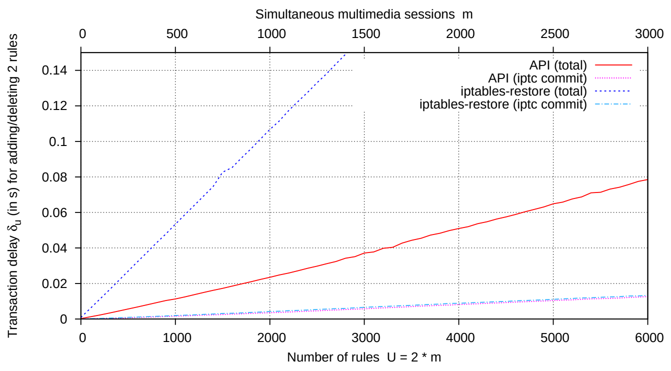

Fig. 7. Ruleset update delay (no traffic)

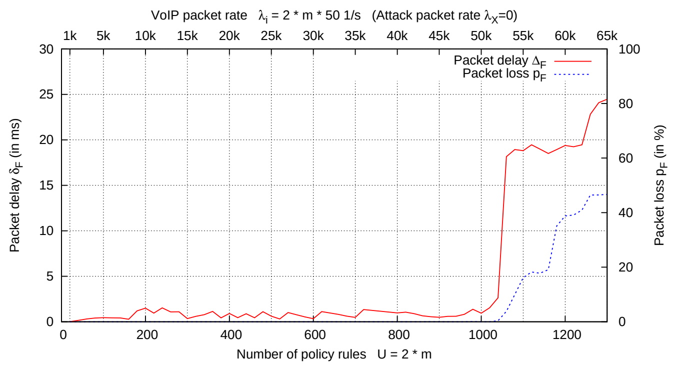

Fig. 8. Packet delay and loss over load (no ruleset updates)

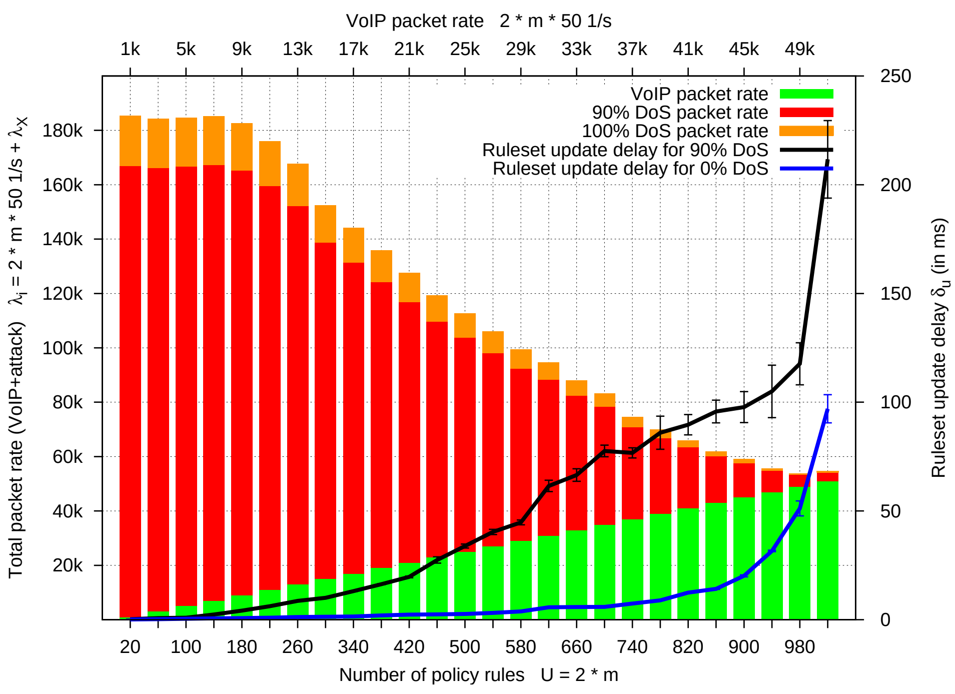

Fig. 9. Maximum packet rate (RTP and DoS traffic) and ruleset update delay

starts at 180,000 packets per second for only 10 simultaneous sessions and reaches zero at $\hat{m} \approx 500$. This effect has to be considered for capacity planning: Under normal conditions the packet filter should have to handle significantly fewer than the $\hat{m}$ simultaneous sessions that are possible in principle, in order to have enough free capacity to handle the malicious traffic.

At this maximum packet rate the CPU is completely used for packet handling; the ruleset update process, which has lower priority, suffers from starvation. Therefore, in a third step the attack packet rate was decreased to 90% of the maximum value determined before. Then, ruleset update delays were measured again (upper line). They were significantly higher than when measured without attack traffic. Nevertheless, for $m < \hat{m}$ they are still acceptable, even if several firewalls have to be configured sequentially during call setup.

VII. Conclusion

Traditionally, firewall rulesets were configured rather statically. However, newer applications such as SIP-based VoIP require dynamic reconfiguration based on the session signaling. Unfortunately, the API for adding firewall rules to the operating system kernel has not been standardized as, for example, the socket interface for TCP connections has. Therefore, this paper proposes such an API, which intentionally covers only the most important functions, in order to keep it as easy to use as possible. It can be used by protocol entities of various firewall control protocols that have been proposed recently.

Our code that provides this API for Linux is open source licensed under the GPL. It is available for download from http://sourceforge.net/projects/simco-firewall.

Our measurements show that a standard PC with Linux can filter the traffic of several hundred simultaneous IP telephony sessions. However, packet filters need resources not only for forwarding legitimate traffic, but also for discarding illegal traffic such as DoS floods. Our measurements also consider this effect and can be used for firewall capacity planning.

Acknowledgment

The authors would like to thank Christian Blankenhorn and Sebastian Meier for working on the implementation of the SIMCO server and the testbed.

References

[1] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, “Terminology for Policy-Based Management,” IETF, RFC 3198, Nov. 2001.
[2] R. Hancock, G. Karagiannis, J. Loughney, and S. V. den Bosch, “Next Steps in Signaling (NSIS): Framework,” IETF, RFC 4080, Jun. 2005.
[3] J. Quittek, M. Stiemerling, and P. Srisuresh, “Definitions of Managed Objects for Middlebox Communication,” IETF, RFC 5190, Mar. 2008.
[4] ITU-T, “Gateway control protocol: IP NAPT traversal package,” ITU-T, Rec. H.248.37, Sep. 2005.
[5] M. Stiemerling, J. Quittek, and C. Cadar, “NEC’s Simple Middlebox Configuration (SIMCO) Protocol Version 3.0,” IETF, RFC 4540, May 2006.
[6] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan, “Middlebox communication architecture and framework,” IETF, RFC 3303, Aug. 2002.
[7] K. Wehrle, F. Pählke, H. Ritter, D. Müller, and M. Bechler, *Linux Network Architecture*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.
[8] C. Blankenhorn, “Evaluation of SCTP as transport protocol for transaction-based applications at the example of a protocol for firewall control,” Student project (in German), University of Stuttgart, IKR, 2005.
[9] ITU-T, “One-way transmission time,” ITU-T, Rec. G.114, May 2003.
[10] ITU-T, “Network grade of service parameters and target values for circuit-switched services in the evolving ISDN,” ITU-T, Rec. E.721, May 1999.
[11] S. Kiesel and M. Scharf, “Modeling and performance evaluation of transport protocols for firewall control,” *Computer Networks*, vol. 51, no. 11, pp. 3232–3251, Aug. 2007.