Preventing Theft of Quality of Service on Open Platforms

Kwang-Hyun Baek, Sean W. Smith
Department of Computer Science
Dartmouth College

TR2005-539
Revision 1, of May 5, 2005

May 5, 2005

Abstract

As multiple types of traffic converge onto one network (frequently wireless), enterprises face a trade-off between effectiveness and security. Some types of traffic, such as *voice-over-IP (VoIP)*, require certain *quality of service (QoS)* guarantees to be effective. The end client platform is in the best position to know which packets deserve this special handling. In many environments (such as universities), end users relish having control over their own machines. However, if end users administer their own machines, nothing stops dishonest ones from marking undeserving traffic for high QoS. How can an enterprise ensure that only appropriate traffic receives high QoS, while also allowing end users to retain control over their own machines?

In this paper, we present the design and prototype of a solution, using SELinux, TCPA/TCG hardware, Diffserv, 802.1x, and EAP-TLS.

1 Introduction

Many enterprise IT infrastructures are experiencing "convergence." Increasingly many types of information services—such as telephony and multimedia—are moving onto users' general-purpose computers, and onto the network. Indeed, our university has already migrated to VoIP, and plans to run only one network in new dormitories, instead of three (intranet, telephone, and cable TV).

However, in order to provide acceptable performance, some of these applications require the network to ensure higher minimal QoS for their traffic. In a large campus environment, with users, occasionally uncooperative, who value individual and departmental autonomy over their machines, this situation creates a set of challenges.

1. The network and the user machines need to conspire together to ensure that traffic from appropriate applications receives the appropriate quality of service.
2. Rogue users, even with root access to their own machines, should not be able to steal high QoS for traffic from applications that do not merit it.
3. The users need to otherwise retain the control over their own machines to which they are accustomed.

Quality of Service Quality of Service (QoS) is a concept of how "good" offered networking services are. QoS can be characterized by a number of specific parameters, ranging from low-level parameters, such as packet loss and guaranteed bandwidth, to end-to-end notions, such as amount of delay and jitter. *Delay* is how long it takes to send information from one end node to another. *Jitter* describes the variance of delay when sending multiple packets. A packet can be lost when collision occurs or when router's routing queue is exhausted, resulting increased delay or jitter.

Specific applications may have specific QoS needs. For example, a multimedia network application might need to reconstruct audio and video signals from a stream of packets it receives. If the application experiences delay, it gives a half-duplex feel to the users. If it experiences jitter, the audio and video signals get interrupted causing distortion and unintelligible audio.

QoS architectures can provide QoS in terms of guaranteed services or differentiated services. Guaranteed services can guarantee certain QoS for a network application. Thus, a network application can request the minimal delay, jitter, packet loss for its traffic to the network. When using differentiated services, a network application can have the network routers treat its traffic with special priority. The routers may expedite the forwarding of the application's packets or assure that the application's packets do not get dropped. Differentiated services, however, do not make any guarantees on QoS.

Two networking architectures dominate current practice for QoS. *Differentiated Services (Diffserv)* [10, 31, 21, 23, 14, 12, 9, 19, 4, 3, 9] is an example of differentiated services. In contrast, *Integrated Services (Intserv)* [6, 5, 37, 38, 45, 44, 36, 15] offers guaranteed services. While Intserv gives the network more control over which application can be given specific QoS, Diffserv's approach is more simple and scalable.

Theft However, when Bob authorizes Alice to use his QoS-enabled network, he wants to make sure that Alice does not abuse the QoS architecture. For example, Bob may want to enforce that Alice's peer-to-peer file sharing traffic, which Alice modified to resemble VoIP traffic, does not hoard the network resources that other user's VoIP traffic needs. To guard against this kind of *theft of QoS*, Bob would set up a *QoS policy* that dictates what level of QoS that Alice's applications should receive.

Current QoS networking architectures enforce their QoS policies at the routers or gateways. These inspect the packets themselves, to determine which deserve higher QoS. Routers and gateways, however, cannot gather enough information from packet inspection to identify which application at the end nodes generated the packets—especially when the end nodes can shape their low priority traffic to appear as high priority traffic. Routers and gateways may attempt to use information in the packets—hardware address, IP address, port number, application protocol number, length of the data, identifiable patterns in the data. However, many existing techniques, such as MAC address spoofing and IP address spoofing, can easily change these values and bypass QoS policy enforcement at the router or gateway.

The Problem Standard QoS networking architectures thus require that the network believes the application-labeling information that end nodes put on packets—except the root users on some end nodes may wish to cheat. We need a way to ensure that the network can believe this information, despite such adversaries, while still providing users with open computing environments.

Our Solution In this project, we provide a solution to this problem using trusted computing hardware now becoming commercially ubiquitous. To gain access to the network, an end node must prove knowledge of

a private key. A kernel-level module at the end node can use this private key—and also is responsible for marking the QoS level on packets. When operating correctly, this module will follow the enterprise's QoS policy. Integrating measurements of the operating environment for this module into an on-board *trusted platform module (TPM)* ensures (for some level of adversary) that the module can use the private key only when it is operating correctly. Building this in Linux permits users to otherwise have freedom in configuring their machines; using the NSA's SELinux variant further protects against malicious users with root access.

This project builds on the previous Enforcer Linux and SELinux project [29, 28, 16], and uses the 1.1b TPM from the *Trusted Computing Group (TCG)* (formerly the *Trusted Computing Platform Alliance, TCPA*) [41, 40].

This Paper Section 2 reviews the building blocks of our design. Section 3 describes how we put these pieces together and presents the big picture of the proposed system. Section 4 discusses our prototype and presents some performance measurements. Section 5 provides a security analysis. Section 6 discusses related work. Section 7 concludes with some directions for future work.

2 Building Blocks

2.1 The Diffserv Architecture

We need a network architecture to provide quality of service.

Differentiated Services (Diffserv) [10] is a QoS architecture that categorizes network traffic into QoS classes so that traffic of a higher priority class receives better QoS than the traffic that belongs to a lower priority class. Diffserv appears to be the QoS architecture most widely used in practice.

Diffserv consists of following components: packet classification, packet marking and *per hop behavior (PHB)* enforcement. In Diffserv, network QoS policies divide network traffic into different QoS classes. The *Differentiated Services Code Point (DSCP)* value in the *Differentiated Services (DS)* field[1] in the IP header is used to classify a packet. The network specifies what type of packets belong to which class and maps the classes to different DSCPs (packet classfication). Ingress network nodes, such as border routers and gateways, inspect packets and mark the DSCP of each packet according to the QoS policy of the network (packet marking). Other routers of the network then handle the packet according to the QoS level associated with the DSCP value of the packet (PHB enforcement). PHB mechanisms, such as *assured forwarding(AF)* [21] and *expedited forwarding (EF)* [23], are used to provide better QoS to the traffic with higher priority.

As we discussed in Section 1, because the ingress network nodes do not know which application is issuing the packets, they must rely on the information within the header of each packet when they classify and mark the packets. This limitation hinders the network administrator from making an application-level QoS policy. Once the attacker figures out the packet-level QoS policy, the attacker can form low-priority packets to resemble high priority packets to bypass the QoS policy enforcement. See Figure 1.

Thus, it is desirable to move classification and marking to the end nodes that produce the packets, where we know which applications are issuing the packets. Then we can have more fine-grained, application-based

[1] The DS field supercedes the IPv4 Type of Service octet and the IPv6 Traffic Classifier octet.

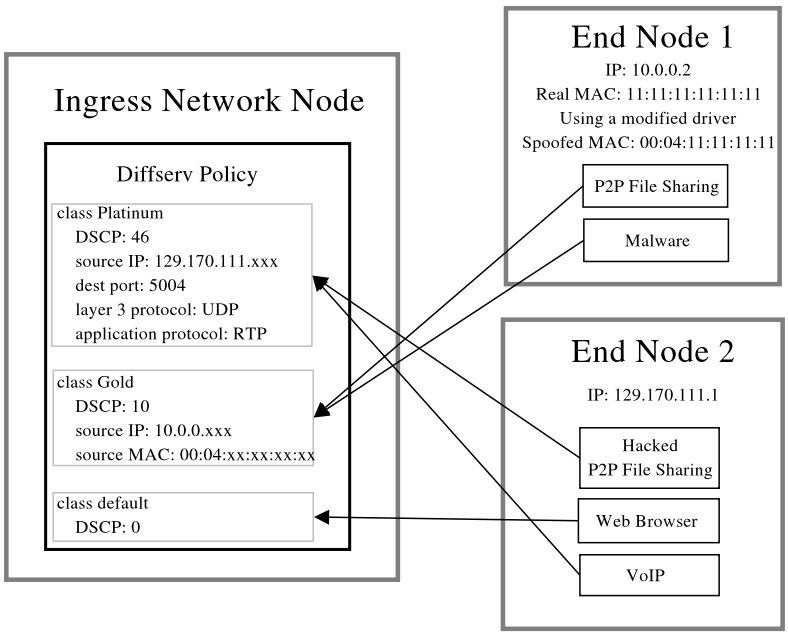


Figure 1: In Diffserv, because the ingress network node does not know what application issued the packets, its classification of traffic is based on packet inspection. However, packet inspection does not offer any control over what applications are allowed in the network. Thus, End Node 1 can get all its packets—including the ones issued by malwares in its machine—classified "Gold" by using a modified network hardware driver that can spoof the source MAC address. Furthermore, End Node 2 can get all of its peer-to-peer file sharing traffic marked "Premium" by hacking the program to modify the packets to resemble VoIP traffic, which often uses RTP protocol and destination port 5004.

QoS policy. However, this approach works only if we can guarantee that all the end nodes will obey the network QoS policy when marking the DSCP field of the packets. We can provide such guarantees using trusted computing.

2.2 TCG Hardware

We need a way for the network to be able to trust the QoS labeling carried out on end nodes.

Trusted computing tries to answer the the following question: how can Alice trust computation that occurs in Bob's computer? The TCG has come up with an answer for general-purpose computing platforms that targets a tradeoff between affordability and security. In the TCG design, a Trusted Platform Module [41, 40] measures and attests the configuration of Bob's computer to Alice. The TPM can provide further assistance in security protocols by providing *sealed storage* that binds data (such as private or secret keys) to a specific configuration of Bob's computer.

In the PC implementation, the TPM is mounted on the motherboard and (in careful interleaving with bootstrap) measures the host machine's hardware and software configuration to sixteen *platform configuration registers (PCRs)* in the form of SHA-1 digests, loaded in a one-way format. The first eight PCRs are filled

with hashes of BIOS, hardware configurations, ROM, the bootloader and its configuration. The usage of the other eight PCRs is up to OS and application designers, and may be used to record the configuration of the applications and drivers. As noted earlier, the TPM can also seal data items to the system configuration, as represented by a suite of specific PCR values. If a data item is an RSA private key, the TPM can also be configured to never unseal it—but rather only perform the RSA private key operation when the system configuration is appropriate.

The 1.1b TPM has been shipping on many IBM platforms for several years. A 1.2 version has been announced, but (at the time of our experiments) was not yet available.

Whether or not one agrees with the TCPA/TCG architecture or some of its potential commercial applications, two facts remain. This hardware is becoming commercially ubiquitous, so a feasible deployment base exists if one wants to build a trusted computing application. Also, the specifications for the TPMs are public, so one *can* build to it.

2.3 The Enforcer LSM

We need a way for the end node OS to actually work with the TPM.

The TCG has specified a *TCG Software Stack (TSS)*, and an open-source implementation of this recently became available [32]. Marchesini et al. also provided an open-source integration of the TPM with Linux, independent of the TSS [29]. Sailer et al. presented a design extending the TSS approach to the application layer in Linux [35].

For our experiments, we chose the Enforcer platform [16].

The TPM architecture binds data to host configuration. Standard PKI-based authentication and authorization systems use a keypair, whose private key belongs to that entity and whose public key has been certified by some authority. Using such a scheme in a TPM-equipped host naturally suggests using the TPM to bind this private key to the host configuration. The configuration of the host machine, however, may change frequently as a trivial file is changed. Does such a change preserve the original entity, or not? If so, do rebind the secret somehow, or must the host make another trip to the CA?

The Enforcer architecture solves the problem by separating the configuration of a host machine into three levels: long-lived kernel and hardware configuration, medium-lived software, and short-lived operational data.

The Enforcer software includes modified boot code and a *Linux Security Module (LSM)*. For PKI-based host authentication, the Enforcer uses the TPM to tie the use of the RSA private key to a known, trusted long-lived configuration of the host machine through the use of PCRs. Figure 2 illustrates the long-lived configuration of the host machine reported by the PCRs. During the boot-up of the host, the PCRs are populated with with measurements reflecting the hardware configuration, and key software such as BIOS, the kernel, and the Enforcer code. If the PCRs matches the known configuration, then the TPM makes secrets available to the Enforcer for that boot cycle. One of these secrets is a private key matching a public key that has been certified in an X.509 identity certificate isssued for this machine.

For medium-lived software and files, a remote *Security Admin* issues a signed policy which describes a file structure that it believes to be trustworthy. The Enforcer, at the beginning of each boot cycle, checks the signature of the Security Admin's policy and loads the policy into the memory. To prevent from using a

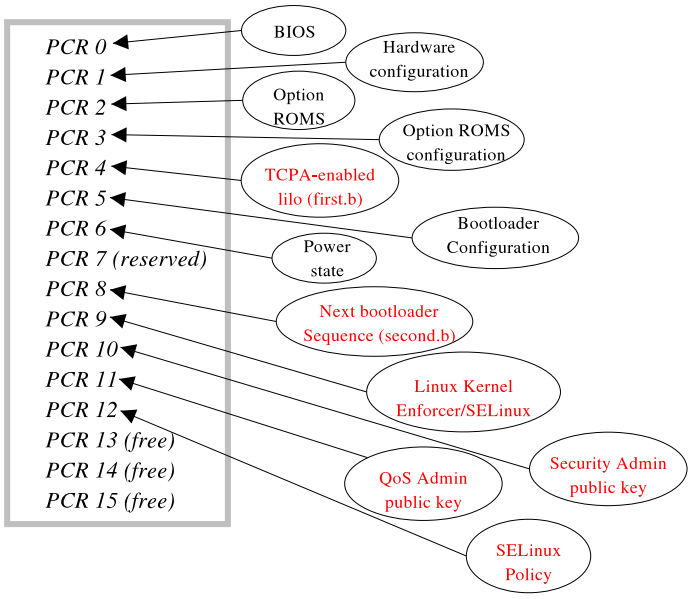


Figure 2: Enforcer's long-lived configuration. The TPM reports the configuration of hardwares, BIOS, ROMS, and the bootloader in the first 8 PCRs. The Enforcer uses PCRs 8–12 to report the configuration of the rest of the long-lived components.

forged policy, the Security Admin's public key is hashed in a PCR and is part of the long-lived configuration, so if an attacker tries to modify the public key file, the secret bound the long-lived configuration becomes not accessible. At run time, the Enforcer intercepts relevant syscalls touching files and checks them against this policy.

Finally, the short-lived operational data is protected through the use of an encrypted loopback filesystem. If tamper is detected in the long-lived or medium-lived configuration, the Enforcer can unmount the filesystem and deny the access to the secret necessary to decrypt the data.

Marchesini et al. [29] showed that the Enforcer platform can be used protect the integrity of SSL server to give higher level of assurance to its customer that the private key is well-protected. In their implementation, Apache Web server's private key is bound to the long-lived configuration of the Enforcer platform and is certified by the CA that vouches for the Enforcer kernel. The Enforcer enforces the medium-lived configuration of Apache, necessary scripts, and libraries through the use of the Security Admin's policy. Finally, operational data is kept in the encrypted loopback filesystem.

2.4 Adding SELinux

We need a way to protect against malicious root users.

Even with TCG hardware, the superuser in the traditional Unix/Linux access control model can nullify the binding between the configuration and the secret. For example, the superuser can use a debugger to read the memory location where the secret is loaded after it is released from the TPM. To solve this problem, Marchesini et al., in their later work [28] added *Security Enhanced Linux (SELinux)* to the Enforcer to

provide *software compartments* to limit the superuser. SELinux [27, 26] is a Flask-based operating system that offers *mandatory role-based access control*. In the mandatory role-based access control model, a policy assigns roles to subjects—processes and users—and types to objects—memory locations, devices, files and sockets—and then describes how the subjects in each role can access the objects in each type. Using the mandatory role-based access control policy, SELinux can confine each application to its own compartment of objects called *domain*. Separating objects in the system into separate domains can, thus, prevent the superuser from spying on arbitrary memory location of the domain where it does not have permission to read memory.

3 Putting the Building Blocks Together

TCG hardware with Enforcer and SELinux provides a practical way to bind a secret to trustworthy kernel-level code, which in turn can use a host-specific private key only when the rest of the machine abides by some policy schema. In the SSL example, the Enforcer will not export the SSL private key outside the use of Apache web server, and ensures that the server can use it only when it is configured according to the latest safety guidelines and the Web content is suitably guarded by the OS.

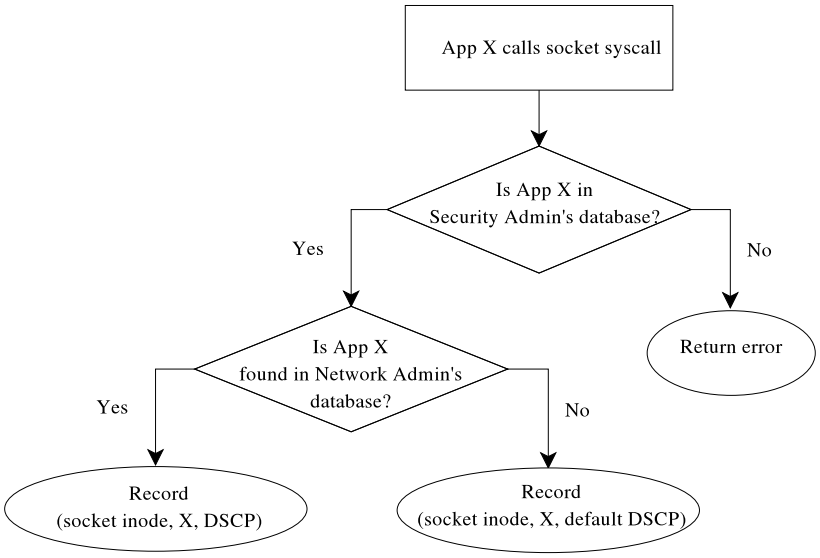
We can use a similar approach to solve the QoS theft problem. We choose a standard PKI scheme to require authorization for network access. We set up the trusted module at the end node to know the host's secret—and also to ensure that packets exiting the machine into the secure tunnel are marked correctly according to the network's QoS policy.

3.1 Distributed Packet Marking

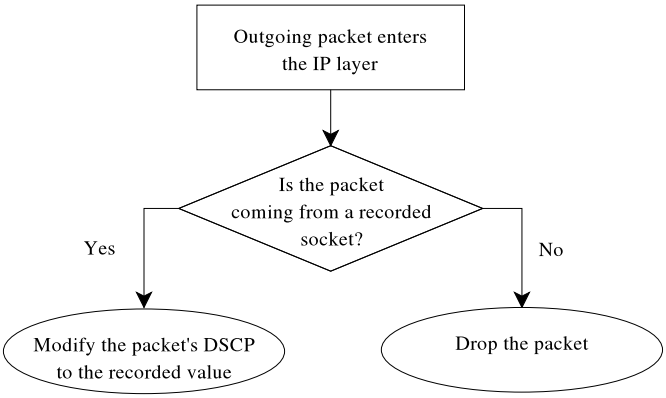
We introduce an additional remote party called the *QoS Admin*, who, similar to the Security Admin, issues a signed policy. This policy maps each network application to a DSCP. During the machine's bootup process, the QoS Admin's public key is also loaded into the PCR along with the Security Admin's public key, and the policy's signature is verified when the Enforcer loads the policy. The QoS Admin may need to consult the Security Admin policy when creating the QoS policy, since what an application does when executed can depend on other aspects of the system configuration besides the binary. Additionally, by granting an application the right to use a high QoS, the QoS Admin may also be granting that right to `fork`ed children.

Figure 3(a) describes how our kernel-level code can correctly identify the mapping between the packets and network applications. We added socket hooks to the Enforcer/SELinux LSM so that when an INET or INET6 type sockets are created, it records the socket's inode number, the program that created it, and the DSCP for the program in the QoS Admin's policy. The hook verifies that the program is listed in the Security Admin's policy and looks up the QoS Admin's policy for the hash of the program and the corresponding DSCP. If the program is not in the Security Admin's policy, the socket is not recorded and all the packets issued from the socket will be dropped. Moreover, if the DSCP for the program is not specified in the policy, the packets from the socket will be assigned the default DSCP and will receive best-of-effort service from the routers.

Figure 3(b) shows how we use the recorded socket inode and DSCP value to mark the outgoing packets. Packet marking happens at a POSTROUTING Netfilter hook [34] in the kernel IP stack to capture all outgoing packets at the IP layer just before it is sent to the network driver. (*Netfilter* is firewall utility integrated



(a) Socket hooks.



(b) Marking DSCP.

Figure 3: Socket and Netfilter Hooks. (a) We added hooks to socket syscalls to map socket inode numbers to programs. We then use this mapping to identify which programs are issuing which packets. If the socket is created by a program that is not listed in the Security Admin's policy, the hook does not record the socket inode; all the packets issued from such a socket to get dropped at the kernel IP stack. The hook also finds the DSCP for the program in the QoS Admin's policy and records this value along with the socket's inode number. If the QoS Admin's policy does not include the program, then the hook records the default DSCP. (b) The Netfilter POSTROUTING hook at the kernel IP stack uses the recorded socket inode and DSCP values to mark the DS field of the IP packet before the packet is sent to the device driver. If the hook does not recognize the socket inode, then the packet is dropped.

in the Linux kernels since version 2.4.) The Netfilter hook looks up the socket inode where the packet is coming from. If the socket is recorded, then it modifies the packet's DSCP value with the recorded DSCP. If the socket is not recorded, then the hook drops the packet.

3.2 Network Authorization

To restrict network access only to machines with trustworthy QoS markings, we use *Extensible Authentication Protocol-Transport Layer Security (EAP-TLS)* client authentication [1] for network authorization. EAP-TLS is a PKI-based authentication method for 802.1x port-based access control. In EAP-TLS authentication, the client holds a keypair and a certificate that is signed by a CA the network trusts. The client gains the access by proving to the network's authentication server the knowledge of his private key.

3.3 Big Picture

Figure 5 (appended after the paper) shows how the components of our system fit together. In our approach, the client's platform generates an RSA keypair and binds the private key to the long-lived configuration of the platform. An enterprise CA certifies the public key and issues a certificate vouching for the long-lived configuration of the client's platform.

In our current design, the private key file of the platform lives in the encrypted loopback filesystem, which is only accessible when the Enforcer is in the configuration that the private key is bound to. The Enforcer and SELinux limits the access to the key only to the 802.1x supplicant program that does not export the key or make visible to other processes. The Enforcer also detects integrity changes to the medium-lived software using the Security Admin's Policy.

During EAP-TLS authentication, the authentication server verifies that the certificate is signed by the enterprise CA and that the client has the knowledge of the private key. From these two facts, the authentication server can deduce that the client is in a "good" configuration, in which the Enforcer/SELinux will mark the packets according to the QoS Admin's policy. Figure 4 shows the resulting security functionality.

The power of our design lies in the non-restrictiveness of the approach. Our usage of the Enforcer/SELinux LSM allows Alice to have the freedom to administer her platform and add/remove applications as she wishes. The only restriction is that, should Alice want non-default network QoS, she needs to install an application that has been approved by the QoS Admin, in a configuration blessed by the Security Admin. SELinux compartmentalization will help here. Alice also cannot load arbitrary kernel modules (in the current architecture), since that could compromise the packet marking.

4 Prototype Implementation and Performance

We implemented our design on an IBM Thinkpad T40 with Pentium M 1.3GHz, 256MB RAM, TPM 1.1b, and Intel Wireless Pro 2100 device, running Linux kernel 2.6.11 with the `-mm` patch [30] and Debian testing distribution. This Thinkpad came pre-installed with a 1.1b TPM, and the kernel we used included the driver for the TPM. We also used `libtpm 2.0` from IBM [25] as the TPM library code. For SELinux libraries and utilities, we used Russell Coker's SELinux Debian packages [13]. We used the experimental Enforcer LSM patch that works with the SELinux LSM.

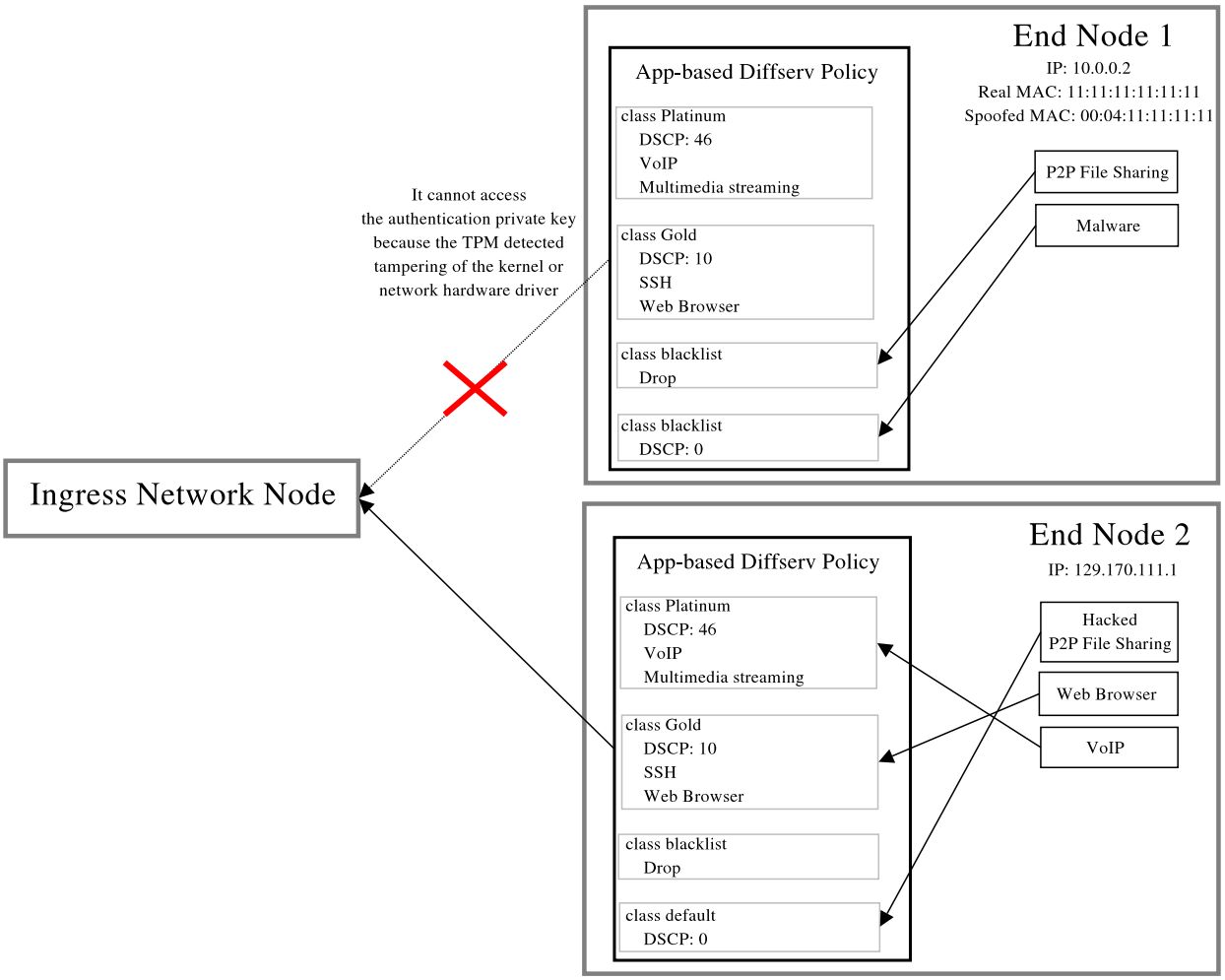


Figure 4: By moving the enforcement to each end node, we can enforce an application-based QoS policy. Coupled with the TPM's integrity protection of the platform (through binding the secret to the measured configuration), the Enforcer's intrusion detection mechanism can also identify and blacklist malware and maliciously modified programs. Moreover, note that TPM detects at boot time that the End Node 1 modified the hardware network driver and denies access to the authentication private key.

For 802.1x EAP-TLS client authentication, we used `xsupplicant` version 1.0.1 [46]. In addition, we also wrote a QoS Admin utility, which, using a configuration file, can build QoS Admin's policy and sign it using QoS Admin's private key.

To limit the power of the superuser account, we added new roles to the SELinux policy for the Security Admin and the QoS Admin. We restricted access to the Security Admin's policy and the QoS Admin's policy to the users who have these roles. We also added a domain for xsupplicant and all the kernel driver modules used for the wireless device, and restricted access to the domain to prevent the superuser from spying the shared key for network encryption. Finally, we created a domain for the TPM library functions and the TPM driver and restricted access to the domain to prevent the superuser from spying the private key for network authentication is loaded.

With the LSM framework, one can implement one's own socket hook functions, such as `socket_create()`, `socket_post_create()`, `socket_bind()`, and `socket_connect()`. These hook functions get called when user code makes socket syscalls such as `socket()`, `bind()`, and `connect()`. We modified `socket_post_create()` and `socket_shutdown()` to bookkeep which applications are using which sockets. Because a socket is completely shut down only when the `close()` syscall is called, we added an an LSM hook in the `close()` syscall to see which sockets get deleted. We also added a Netfilter hook function that gets called when a packet arrives in the predefined hooks in the kernel IP stack, called `trustednet_ip_postroute_last()`. This function captures all the packets right before they leave the kernel IP stack, and marks each packet according to the recorded DSCP value for the socket it is coming from. If no value is recorded, the function drops the packet. We also perform bookkeeping to detect packets a machine is sending to itself. Our bookkeeping occurs after the SELinux hook functions since SELinux policy may prevent the socket from being created and the packets from leaving the machine.

To evaluate the performance of our implementation, we measured how long the added hooks take to mark the packets. We measured the added latencies for the following five applications: `linphone`, a VoIP application, `ices`, audio streaming program, `firefox`, a Web browser, `exim4`, mail-transfer agent , and `ssh/sftp`, secure shell and file transfer protocol. We note that the time that the socket calls take is generally dependent on how much computing resources are available. If the platform runs out of physical memory and has to page and swap, the performance degrades dramatically. The maximum numbers reflect such events.

The socket hook operation is time-consuming, since we have to calculate the hash of the program to compare with the entry in the Security Admin's policy, and it also has to search the DSCP for the program in the QoS Admin's policy. Thus, the time that socket hook takes is proportional to the size of the program binary for which it has to calculate the hash. Table 1 shows the added latency for the socket hook for `socket_post_create()`. Even though the socket hook operation is costly, the program only needs to do this once to create the socket prior communication. Thus, the number of times socket hooks are called much less often than the Netfilter hooks.

The Netfilter hook operation is less costly, as shown in Table 2, since all it needs to do is look up the packet's socket inode number in the recorded values and mangle the DS field in the IP packet. The operation occurs much more frequently than the socket hook operations since a single socket may issue multiple packets.

Performance benchmarks for some specialized hardware show that Diffserv marking can be done without any added latency[20]. Thus, our approach may introduce some latency to the end-to-end communication that uses traditional. However, according to the International Telecommunications Union [18], the recom-

	linphone	ices	firefox	exim4	ssh/sftp
Min	2275	11550	5489	14379	1529
Max	47649	11931	7938933	325801	212549
Average	4859.47	11655.25	358324.71	24492	7065.02

Table 1: Added latency by the socket hook (µs).

	linphone	ices	firefox	exim4	ssh/sftp
Min	4	4	4	5	4
Max	3822	2083	5301	34	12926
Average	8.722	17.74	18.821	11.4835	40.5053

Table 2: Added lantency by the Netfilter hook (µs).

mended maximum round trip delay in a voice system is 0 to 150 milliseconds. Thus, the time that the local packet marking adds is easily absorbed in the total round trip time.

In summary, we showed that our implementation does not hinder the performance of the network applications. Furthermore, because we are distributing the computation that the edge routers or gateways traditionally perform for all packets they serve, our solution is more scalable than the centralized traditional Diffserv architecture.

Availability. Our code will be available for open-source download.

5 Security Analysis

In this section, we discuss several possible threat models to our distributed QoS enforcement and show how we can prevent them.

Since the DSCP marking happens at the kernel IP stack, it is possible for the attacker to modify the packet after it is marked by the Enforcer and before it arrives at the network interface driver. In Linux, a user can use netlink sockets and divert sockets to intercept, modify, reinject packets. We use SELinux to restrict the permission to use these sockets that can interfere with the QoS marking rules. Note that many useful programs, such as iptables firewall utility, use such sockets, however. Restricting netlink usage greatly decreases the functionality of these programs. Alternately, we can also use digital signature to detect tamper although this approach may greatly reduce the performance of the system.

The attacker may try to use another machine between the Enforcer platform and the network to try to modify the packet. Because EAP-TLS authentication produces Medium Access Control (MAC) layer encryption and integrity session keys, which encrypts the IP header of the packet, the attacker needs to migrate the session key to the untrusted man-in-the-middle machine and modify the DSCP in the IP header. Again, we use SELinux to ensure that the memory location of the session key is readable only by the network interface driver. Furthermore, we use the Security Admin's policy to include only the network interface driver which does not reveal the session key to other processes. Moreover, we use SELinux to guard against the superuser.

The attacker can also try to stand between routers to modify the packets after they have been decrypted by the access point or the switch. However, this attack requires physical access to the wires that connect the routers and access points. Thus, when physical security is present, this attack is very hard to do.

We also discuss the benefits our various components offer.

The TPM alone offers configuration measurement of the platform and binding of an RSA private key to the configuration of the platform so that the private key is only accessible only when the platform is in certain configuration. However, in order to provide high assurance, the TPM, if used alone, needs to report the comprehensive configuration of the platform including a snapshot of the filesystem. Thus, a small, trivial change in the platform's filesystem causes the secret key unaccessible.

The Enforcer LSM is added to break down the configuration of the platform into three levels—long-lived kernel and hardware, medium-lived software and libraries, short-lived operational data—and to let the TPM to bind the private key only to the long-lived configuration of the platform. The Enforcer LSM then works as an intrusion-detection system for the medium-lived componenets, enforcing the Security Admin's signed policy. The Enforcer LSM also provides encrypted storage for the short-lived configuration and can bind the secret key to the long-lived configuration, so that the encrypted data is only accessible when the correct version of the Enforcer is running. Thus, through the use of the Enforcer LSM, the small change that would have made the private key inaccessible would not affect the access to the private key in the Enforcer-assisted TPM.

However, the TPM and the Enforcer alone cannot prevent the superuser of the platform from spying the memory location where the secret is loaded from the TPM. Furthermore, the Enforcer alone can suffer from time-of-check/time-of-use attacks if the user can modify the memory location that stores the Security Admin's policy. SELinux is added to provide software compartments to protect the Enforcer platform from the superuser. It also gives more strict control over all the objects in the platform, including memory, to ensure that implicit flow of confidential information does not occur.

6 Related Work

Trusted Computing Our work is similar to *Trusted Network Connect (TNC)* [43], an ongoing effort by the Trusted Computing Group to enforce security for end-point host connections. The goal that TNC is trying to achieve is different from our work, however. TNC focuses on protecting the end nodes from malware and ties the security level of end nodes to the level of access to the network. Our work differs in several respects. We are trying to secure not just network authorization, but QoS markings as well; we are also distinguishing not just good nodes from bad nodes, but also between applications within nodes.

The specification of TNC has not been finalized yet. Current sources [42] indicate plans to use VLANs and 802.1x with EAP and *Remote Authentication Dial In User Service (RADIUS)*.

Secure QoS Intserv is a signal-based QoS architecture, where the node desiring better QoS makes a request to the network via Resource ReSerVation Protocol (RSVP) [11] and reserves network resources such as bandwidth and drop rate for the connection. There have been some work to secure RSVP messages from modification and eavesdropping [2, 39, 7]. However, they focus on outsider attacks and do not address insider attacks. Moreover, because Intserv requires the routers keep track of the flow that has reserved certain

resources, it is not as scalable and popular as Diffserv.

IEEE 802.11e The IEEE 802.11e working group is currently working on MAC-layer QoS for 802.11 wireless networks [22]. Zhu, et al. [47] give a good survey of the QoS schemes in the MAC-layer.

MAC-layer QoS for wireless networks focuses on a different problem from IP-layer QoS. Because an access point can service only one wireless node at a time, when more than one node transmits a packet at the same time, a collision occurs and the nodes must retransmit after a back-off period. In wireless QoS schemes, the nodes that are sending high priority traffic might back off for a shorter period of time than the nodes that are sending low priority traffic. However, misbehaving nodes may always back off for a short enough time to always win the transmission opportunity.

Some resarchers have proposed solutions to this greedy misbehavior [24, 33, 8]. These solutions use probabilistic algorithms for detecting MAC-layer misbehavior. Thus, it is possible for sophisticated attackers to bypass the detection, and the detection scheme can mistake a well-behaving node to be misbehaving.

In our design, we have the Security Admin approve only the network interface drivers that is tested to not misbehave. Thus, the users who wish to authenticate to the network cannot use network interface drivers that are not approved by the Security Admin.

7 Conclusions and Future Work

In this paper, we described a design and prototype of distributed enforcement of the network's QoS policy using trusted computing hardware, open source trusted computing tools, and Diffserv. In the Diffserv architecture, the network applications can receive the necessary QoS by classifying its traffic as a certain class by marking all of its outgoing packets. However, the enforcers of the network's QoS policy, such as the edge routers or gateways, are unaware of which network applications issued the packets they see, especially if malicious node can modify the low priority traffic to resemble the high priority traffic.

Thus, we argued that it would be ideal to move the enforcing to the end nodes, which are aware of the mapping between the outgoing packets and the programs as long as we can trust that each node will obey the network's QoS policy. We used trusted computing to bind the node's network authentication secret to the node's configuration that obeys the network's QoS policy. Using our design, we allow all authenticated nodes to mark the packets, and at the same time we are assured that node's marking obeys the QoS policy of the network. We implemented our design by adding socket and Netfilter hooks to the Enforcer LSM and by binding the node's private key for EAP-TLS authentication to the modified Enforcer with marking abilities. We showed that our implementation does not suffer performance and increases scalability due to distribution of marking duties.

Several areas remain for future work.

Policy Update Verification Currently, there is no way for the authentication server to know whether the policies of the Security Admin and the QoS Admin in the end node's Enforcer platform is up-to-date; the

server knows only that the policy the Enforcer is enforcing has a valid signatures[2]. Once an update is released, it is the responsibility of the Security Admin and the QoS Admin to patch the system in timely fashion. Although it is possible to attest the policies to an unused PCR in the TPM and map the new configuration to the certificate, this idea goes against the Enforcer's design of separating the medium-lived software configuration from the long-lived core. Moreover, adding attestation to EAP-TLS authentication may require changes to the current standards.

We are thinking of ways to allow the Enforcer, rather than the TPM, attest the medium-lived core's policy. This idea allows the attestation hierarchy to match the hierarchy of the configuration. We propose the use of *X509 attribute certificates* [17] in conjunction with the enterprise certificate for EAP-TLS authentication. In our proposed scheme, in the beginning of each boot cycle, after the Enforcer/SELinux LSM checks the signature of the Security Admin's policy and the QoS Admin's policy, the Enforcer/SELinux LSM generates attribute certificates for these two policies and signs them with the authentication private key—thus vouching for the timestamp, version number, and issuers of the policies. When the Enforcer platform authenticates to the network, it presents these attribute certificates along with the network certificate to the authentication server so that the authentication server can find out what version of the policy the platform is enforcing. Since EAP-TLS authentication supports validation of a chain of certificates, we are hoping that using the attribute certificates may save us from making any changes to the standard EAP-TLS authentication protocol.

Automated Policy Update Policy update verification is also useful when the Enforcer platform is used in multiple domains that have different QoS policies. Thus, it is desirable for the Enforcer platform to keep multiple QoS policies signed by different QoS Admins, and use the appropriate one when crossing the domain. For users who may not have multiple policies installed, it would be ideal to support dynamic update of the QoS policy. In this model, the network detects a Enforcer platform which is not properly configured with the policy that the network approves. By creating an "quarantined" VLAN, we can limit the user's network access to where the user can download the signed QoS policy of the network. Once the policy's signature is checked, the Enforcer platform can restart using the updated QoS policy and be able to gain access to the network.

Usability of Policy Writing In general, it can be hard to come up with a policy that matches the conceptual model and allows the system to operate correctly. If not carefully written, the policy may not deliver the security that we need. Too restrictive policy may cause the system to lose important functionality. Dynamically linked executables requires the Security Admin to validate the linkers and dynamically loadable libraries as well as the program binaries. In our current implementation, we included all the libraries and modules and linkers in the Security Admin's policy to increase assurance against attacks that use dynamically linked executables.

In our current implementation, the policies from the Security Admin and the QoS Admin are locally generated. The timestamps on executables cause the hashes of the binary compiled in different platform to be unique, making it hard for the Security Admin and the QoS Admin to issue general, universal policies for all the Enforcer/SELinux platforms. In order to support automatic policy update, we plan to investigate on ways to make the policies more portable across different platforms.

[2]The Enforcer does offer a way to record the serial number of the Security Admin's policy it uses and to reject its use if the policy's serial number is less than the recorded one. This feature, however, only addresses rollback, not freshness.

Usability of Open Platforms Another area of future work will be to validate, with user studies, that this approach (a trustworthy network QoS module within SELinux) still results in a sufficiently flexible and open platform to satisfy users who like to retain administrative control of their machines. Verification of the SELinux policy (once we finish adapting it from the prior Enforcer work) will also be necessary.

Acknowledgement

This work was supported in part by the Mellon Foundation, by Cisco, by Intel, and by the Office for Domestic Preparedness, U.S. Dept of Homeland Security (2000-DT-CX-K001). The views and conclusions do not necessarily represent those of the sponsors.

We also gratefully acknowledge John Marchesini, Chris Masone, Jason Jeffords, and Josh Stabiner for helpful suggestions and advice.

References

[1] B. Aboba and D. Simon. PPP EAP TLS Authentication Protocol. IETF RFC 2716, October 1999.

[2] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, Steve Muir, and Jonathan M. Smith. Secure Quality of Service Handling SQoSH. *IEEE Communications Magazine*, April 2000.

[3] G. Armitage, B. Carpenter, A. Casati, J. Crowcroft, J. Halpern, B. Kumar, and J. Schnizlein. A Delay Bound alternative revision of RFC 2598. IETF RFC 3248, March 2002.

[4] F. Baker, K. Chan, and A. Smith. Management Information Base for the Differentiated Services Architecture. IETF RFC 3289, May 2002.

[5] F. Baker, J. Krawczyk, and A. Sastry. Integrated Services Management Information Base Guaranteed Service Extensions using SMIv2. IETF RFC 2214, September 1997.

[6] F. Baker, J. Krawczyk, and A. Sastry. Integrated Services Management Information Base using SMIv2. IETF RFC 2213, September 1997.

[7] F. Baker, B. Lindell, and M. Talwar. RSVP Cryptographic Authentication. RFC 2747, January 2000.

[8] J. Bellardo and S. Savage. 802.11 Denial-of-Service Attacks: Real Vulnerabilities and Practical Solution. In *Proceedings of USENIX Security Symposium*, August 2003.

[9] Y. Bernet, S. Blake, D. Grossman, and A. Smith. An Informal Management Model for Diffserv Routers. IETF RFC 3290, May 2002.

[10] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. IETF RFC 2475, November 1998.

[11] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP)—Version 1 Functional Specification. RFC 2205, September 1997.

[12] A. Charny, J.C.R. Bennett, K. Benson, J.Y. Le Boudec, A. Chiu, W. Courtney, S. Davari, V. Firoiu, C. Kalmanet, and K.K. Ramakrishnan. Supplemental Information for the New Definition of the EF PHB. IETF RFC 3247, March 2002.

[13] Russell Coker. Security Enhanced Linux Information. `http://www.coker.com.au/selinux`.

[14] B. Davie, A. Charny, J.C.R. Bennett, K. Benson, J.Y. Le Boudec, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis. An Expedited Forwarding PHB (Per-Hop Behavior). IETF RFC 3246, March 2002.

[15] B. Davie, D. Oran, S. Casner, and J. Wroclawski. Integrated Services in the Presence of Compressible Flows. IETF RFC 3006, November 2000.

[16] Enforcer homepage. `http://enforcer.sourceforge.net/`.

[17] Stephen Farrell and Russell Housley. An Internet Attribute Certificate Profile for Authorization. IETF RFC 3281, April 2002.

[18] ITU-T Recommandation G.113. Transmission Impairments Due to Speech Processing, February 2001.

[19] D. Grossman. New Terminology and Clarifications for Diffserv. IETF RFC 3260, April 2002.

[20] The Tolly Group. LinleyBench 2002 Test Results–Ezchip NP-1c 10 Gigabit 7-Layer Network Processor. `http://www.linleygroup.com`, 2001.

[21] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured Fowarding PHB Group. IETF RFC 2597, June 1999.

[22] IEEE 802.11e WG. Draft Supplement to IEEE Standard for Telecommunications and Information Exchange between Systems—LANMAN Specific Requirements,. Part 11:Wireless LAN Medium Access Control (MAC) and Physical Layer. IEEE Std 802.11e-D3.3, October 2002.

[23] V. Jacobson, K. Nichols, and K. Poduri. An Expedited Fowarding PHB. IETF RFC 2598, June 1999.

[24] Pradeep Kyasanur and Nitin H. Vaidya. Detection and Handling of MAC Layer Misbehavior in Wireless Networks. In *Dependable Systems and Networks (DSN'03)*, pages 173–182, San Franciso, California, June 2003.

[25] IBM Watson Research—Global Security Analysis Lab: TCPA Resources. `http://www.research.ibm.com/gsal/tcpa/`.

[26] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, February 2001.

[27] Peter Loscocco and Stephen D. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[28] John Marchesini, Sean Smith, Omen Wild, Josh Stabiner, and Alex Barsamian. Open-Source Applications of TCPA Hardware. In *20th Annual Computer Security Applications Conference*, December 2004.

[29] John Marchesini, Sean W. Smith, Omen Wild, and Rich Macdonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report TR2003-476, Department of Computer Science, Dartmouth College, December 2003.

[30] The -mm patches to the Linux kernel. `http://www.kernel.org/patchtypes/mm.html`.

[31] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. IETF RFC 2474, December 1998.

[32] TrouSerS—The open-source TCG Software Stack. `http://trousers.sourceforge.net/`.

[33] Maxim Raya, Jean-Pierre Hubaux, and Imad Aad. DOMINO: A System to Detect Greedy Behavior in IEEE 802.11 Hotspots. In *ACM MobiSYS 2004*, pages 84–97, June 2004.

[34] Rusty Russell and Harald Welte. Linux netfilter Hacking HOWTO. `www.netfilter.org/`, July 2002.

[35] R. Sailer, Zhang X., T. Jaeger, and L. Van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th Usenix Security Symposium*, San Diego, CA, August 2004. USENIX.

[36] S. Shenker, C. Partridge, and R. Guerin. Specification of Guarateed Quality of Service. IETF RFC 2212, September 1997.

[37] S. Shenker and J. Wroclawski. General Characterization Parameters for Integrated Service Network Elements. IETF RFC 2215, September 1997.

[38] S. Shenker and J. Wroclawski. Network Element Service Specification Template. IETF RFC 2216, September 1997.

[39] Vanish Talwar and Klara Nahrstedt. Securing RSVP for Multimedia Applications. In *Proceedings of the 2000 ACM workshops on Multimedia*, pages 153–156, Los Angeles, 2000.

[40] TCG PC Specific Implementation Specification. `http://www.trustedcomputinggroup.org`, August 2003.

[41] TCG PC Specific Implementation Specification Version 1.1. `http://www.trustedcomputinggroup.org`, August 2003.

[42] Trusted Network Connect Frequently Asked Questions. `http://www.trustedcomputinggroup.org`, May 2004.

[43] Trusted Network Connect to Ensure Endpoint Integrity. `http://www.trustedcomputinggroup.org`, May 2004.

[44] J. Wroclawski. Specification of the Controlled-Load Network Element Service. IETF RFC 2211, September 1997.

[45] J. Wroclawski. The Use of RSVP with IETF Integrated Services. IETF RFC 2210, September 1997.

[46] Open Source Implementation of IEEE 802.1x. `http://www.open1x.org`.

[47] Hua Zhu, Ming Li, Imrich Chlamtac, and B. Prabhakaran. A Survey of Quality of Service in IEEE 802.11 Networks. *IEEE Wireless Communications*, 11(4):6–14, August 2004.

Appendix

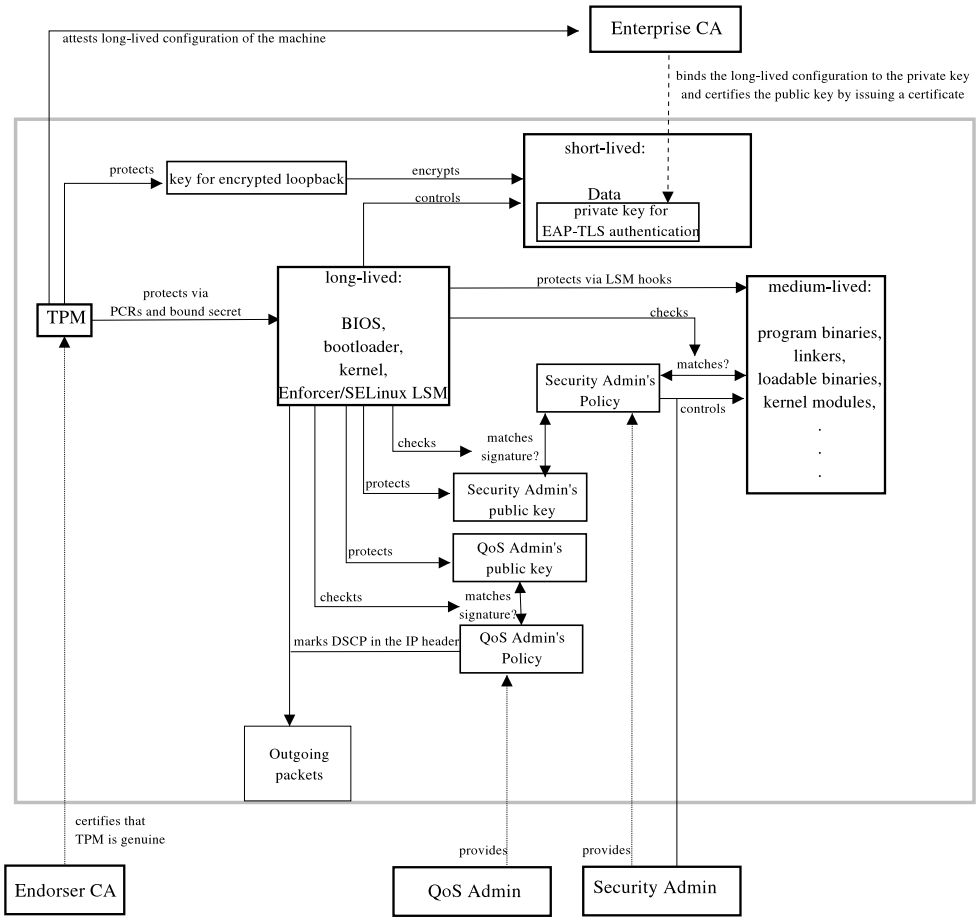


Figure 5: Our modified Enforcer/SELinux platform. (This figure is based on Figure 2 in [28].) The Endorser CA certifies that the TPM is genuine and its certificate comes with the hardware. The TPM vouches for the long-lived configuration of the platform through the use of the value of the PCRs and binds the short-lived encryption key to it. The Enterprise CA certifies the long-lived configuration of the platform and the TPM by issuing a certificate for the platform's network authentication keypair. The private key for network authentication should live in the encrypted loopback filesystem (short-lived component), which is only accessible when the long-lived configuration matches the value that the Enterprise CA certified. The long-lived components then protect the medium-lived components by enforcing the Security Admin's signed policy. When the long-lived components detect discrepancies between the Security Admin's policy and the medium-lived components, the Enforcer LSM will unmount the encrypted filesystem and panic the kernel. We introduce another remote party called the QoS Admin who issues a signed application-based QoS policy. The long-lived configuration also enforces the QoS Admin's policy by marking all its outgoing packets according to the policy. The authentication server can be assured that the platform will obey the network's QoS policy simply by the fact that the platform can prove the knowledge of the authentication private key, which is certified by the Enterprise CA because the CA vouches for the fact that platform's kernel will obey the QoS Admin's policy.