

High Performance Audited Traffic Capture

Eric Rescorla, Kevin Dick

{ekr,kevin}@claymoresystems.com

Claymore Systems Technical Report TR-2003-02-01

Abstract

In [1], Rescorla and Dick proposed using passive session recording and timestamping to provide third-party auditing capabilities for SSL. This approach enables dispute resolution for electronic transactions without modifying existing applications or installing client-side PKI. The practicality of such a system approach requires both cost-effective deployment and secure capture at network speeds. This report describes the design and implementation of the recording portion of the system. Using \$2000 worth of commodity hardware combined with an off-the-shelf hardware security module, we developed a device that could achieve successful captures at peak speeds up to 380 Mb/s and sustained speeds of 178 Mb/s. We also demonstrate the feasibility of clustering multiple such devices, which would enable cost-effective capture at 1 Gb/s.

1 Introduction

SSL [2] and its follow-on protocol TLS [3] (collectively referred to as SSL) are by far the most widely used Internet security protocols, to a great degree because they are easy to implement and deploy. However, SSL is a channel security protocol so its security properties are ephemeral—third parties cannot verify them at a later date. If two parties engage in a transaction over SSL and subsequently dispute its contents, neither party can demonstrate which of their asserted versions is correct. This difficulty limits the usefulness of SSL for electronic transactions to providing confidentiality and in certain cases authentication.

The standard response to this issue is to recommend that the parties cryptographically sign their messages using a standard format such as [4] or [5]. While theoretically sound, this approach has seen only limited deployment. For example, one might imagine that the banking industry would be at the forefront of cryptographic signature adoption. However, according to a banking industry publication [6], while VeriSign generated 25,000 server certificates for financial institutions from 1998 to 2001, it generated only 345 client certificates. The obvious explanation for this disparity is that financial institutions require server certificates to support the use of SSL, but don't require client certificates because these institutions do not widely employ cryptographic signatures.

In [1], Rescorla and Dick observed that it is possible to use authenticated recordings of SSL traffic to provide an evidentiary trail. The asserted benefit of this approach over cryptographic signatures is that it has very low impact on the existing transaction system. To get fairly strong security guarantees, it requires simply installing a device on the network and deploying a small piece of software on all server machines. To get very strong security guarantees, it requires replacing the servers' SSL implementations with hardened SSL accelerator devices. Most organizations can implement either of these options purely from an operational perspective, without the full software upgrade project usually required to deploy cryptographic signatures. Moreover, this type of auditing has the additional advantage of providing a substantial amount of forensic evidence to investigate fraud or attacks executed using an SSL-protected connection.

In September 2002, we began a project to determine the practical feasibility of using an SSL auditing system in a typical commercial or governmental setting. We had already built a software-only prototype that demonstrated the capture of SSL traffic and subsequent decoding of SSL sessions without compromising the server's private key. Therefore, technical feasibility was not a concern. Rather, we hoped to demonstrate that we could build a fully secure device for reasonable cost that would achieve acceptable performance.

This work was sponsored by DARPA contract SPO700-98-D-4000.

The authors would like to acknowledge the support of Broadcom, for providing us with our gigabit ethernet switch.

2 Overview of SSL Auditing

Understanding the course of our project requires a basic understanding of SSL auditing. Unlike cryptographic signatures, SSL auditing is a system-level rather than an application-level solution. Therefore, it is straightforward to build an understanding of the solution by starting with topology of the target system.

2.1 Target Topology

Figure 1 shows a typical topology for an SSL-based transactional system. An SSL client and an SSL server communicate on behalf of a client application and a server application, respectively. The SSL client initiates a single TCP connection to the SSL server via the Internet and runs the application protocol over SSL. The application protocol typically has either no security capabilities or authentication capabilities only. Thus, once the transaction is finished, neither side can prove the contents of the transaction. As will become clear below, we also have to take into account the stores where the SSL implementations keep their private keys.



Figure 1 Typical SSL Transaction System

2.2 Capture Components

Capturing the transactions executed using this topology requires several discrete elements. First, we must create a record of SSL sessions by sniffing the network connection, identifying SSL packets, and saving them. We call this component the recording application or Recorder and it saves packets in a session store that may reside on the local disk or attached storage. Second, we must secure these records against tampering by hashing them, combining them with a timestamp, and signing them with a securely stored private key. We call this component the master application or Master and it returns the signatures to the Recorder so that it can save them along with the captured data. Figure 2 shows the addition of these components to the SSL transaction system.

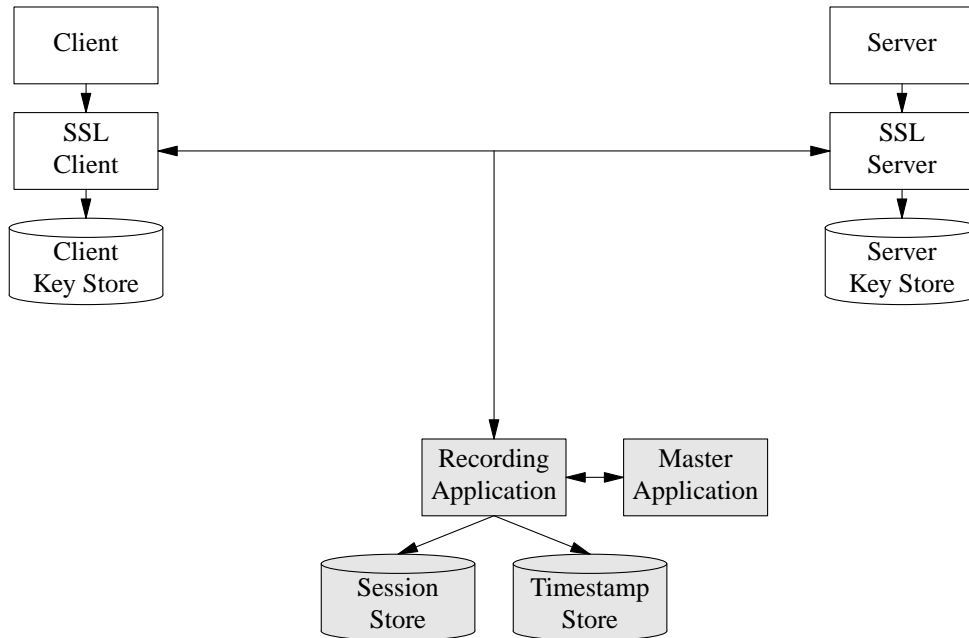


Figure 2 Adding Capture Components]

2.3 Replay Process

The Recorder and Master operate while the client and server applications execute transactions using SSL. When we want to review a transaction at a later time, we require more components. The replay process requires a number of steps to ensure the overall security of the system. A human operator initiates a replay request. We call the component that handles this request the replay application or Player. To fulfill the request, the Player must retrieve the recorded session and corresponding timestamp from the session store and signature store, respectively. Once it validates the timestamps, all the Player needs is the SSL server's private key to replay the session. However, we don't want the SSL server to completely expose its private key because that will compromise every session.

Therefore, the Player submits the handshake portion of the SSL session to a component on the SSL server machine that can access its key store within the machine's trust boundary. We call this component the revelation application or Revealer. This component applies the server's private key to the handshake and extracts only the traffic keys for the session of interest. The Revealer encrypts these traffic keys under the Master's public key. Therefore, the Player must take the additional step of sending these keys back to the Master for decryption. This extra step ensures that the Master, which is highly secure, can perform access control and logging. Once it has the traffic keys from the Master, the Player can replay the session of interest. Figure 3 shows the complete set of components necessary for capture and replay.

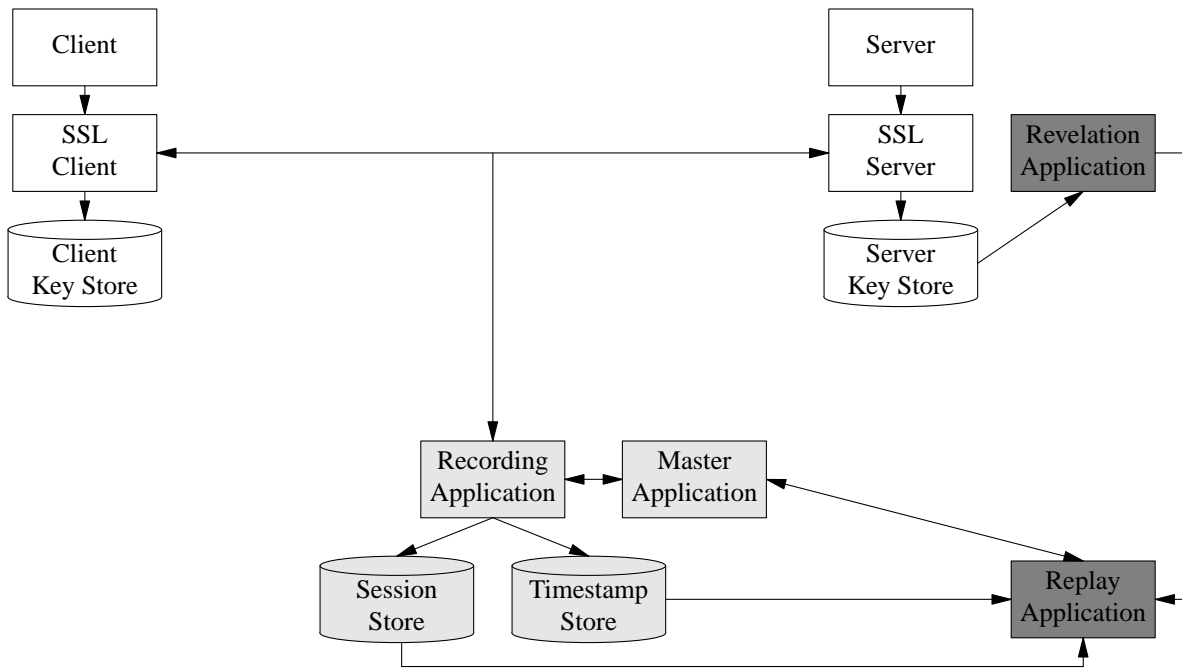


Figure 3 Adding Replay Components

2.4 Potential Configurations

Figures 1 through 3 essentially reflect the logical configuration of the system; there are a number of potential different physical configurations. Figures 4 through 6 show three of the most important ones. Figure 4 is the most obvious configuration: all the client components run on a client machine, all the server components plus the Revealer run on the server machine, all the capture components run within a single device, and there is a dedicated machine for the Player.

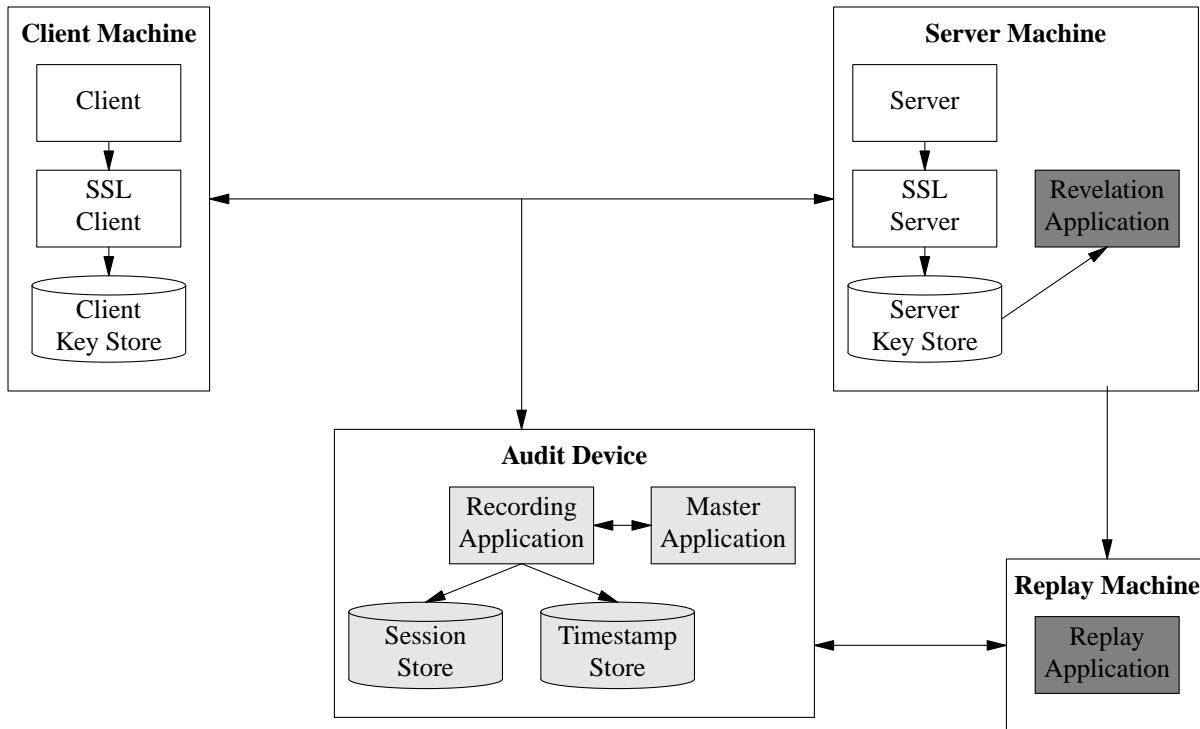


Figure 4 Basic Physical Configuration

However, at the start of the project, we did not believe that a single capture device could possibly capture SSL traffic at 1 Gb/s. Because capture at this speed appeared to be a potential commercial and government requirement, we knew we would need to cluster multiple devices. We thought it might be more cost effective to have multiple Recorders in their own devices with timestamping performed by a single Master device. Therefore, we would need the capability to split the capture into separate physical components as shown in Figure 5. From a development standpoint, enabling this architecture required exercising the necessary discipline to make sure the Recorder and Master software were truly independent.

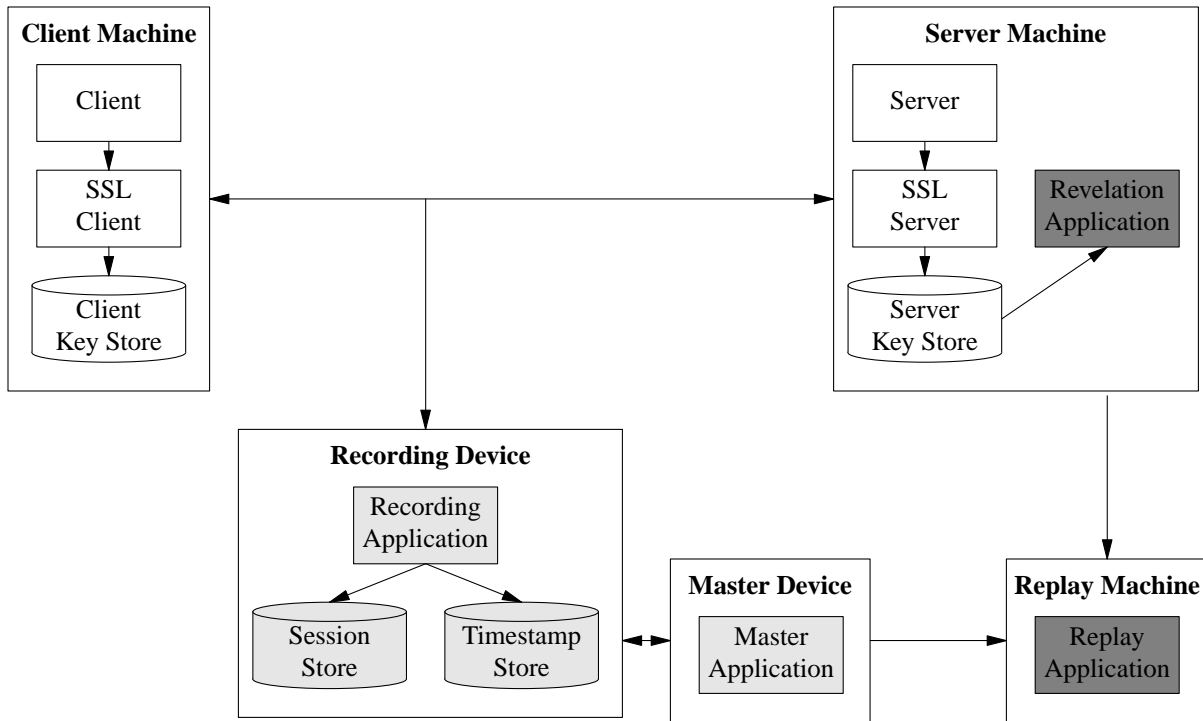


Figure 5 Split Capture Configuration

Furthermore, there is a trend towards offloading SSL processing from server application machines to gain efficiency through the use of dedicated cryptographic processing and decouple the scalability of SSL connection processing from the scalability of application transaction processing. Companies such as F5, Intel, and SonicWall all make such devices. Moreover, as discussed below, using dedicated SSL hardware offers potentially greater security guarantees because they can guarantee that the server's private key never leaves its key store. Figure 6 shows the SSL transaction system with this physical configuration.

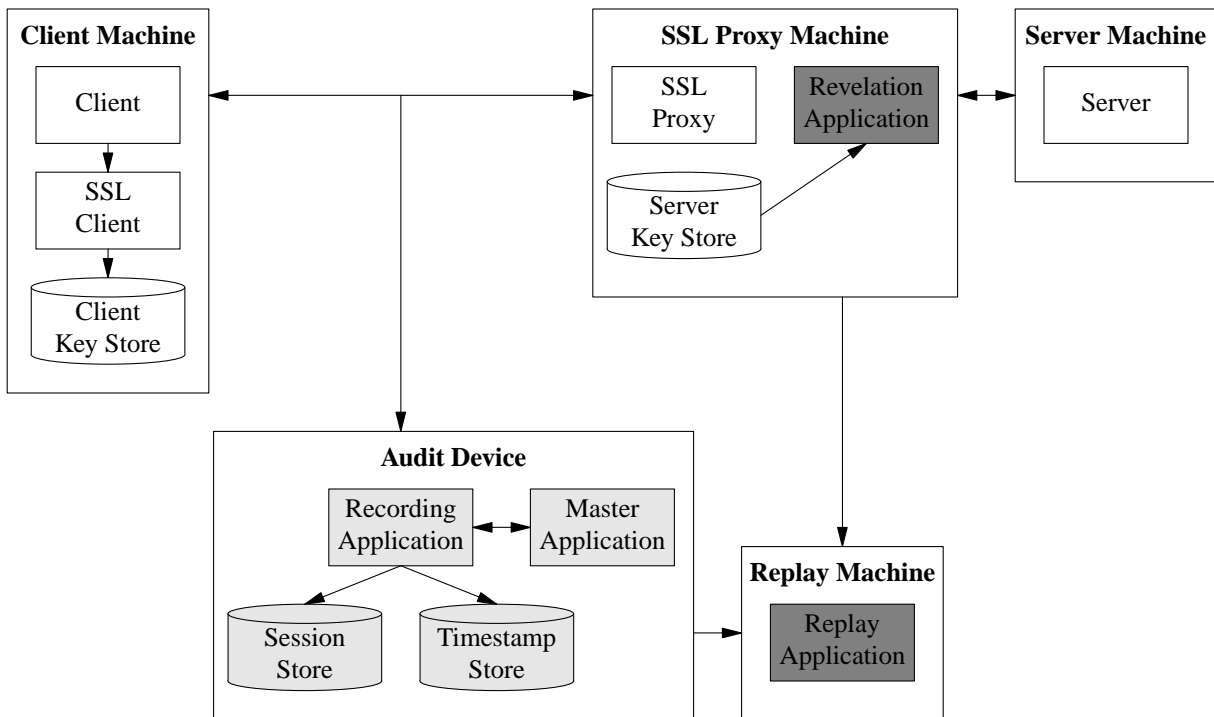


Figure 6 SSL Device Configuration

2.5 Security Guarantees

We separate security guarantees into two classes based on the point in time where one party might want to generate a false record of a transaction. Guarantees against reactive fraud prevent the parties from altering records after the device has captured a transaction. We consider this case the most common—one party realizes that it has made an error and does not want to bear the cost of remediation. Guarantees against proactive fraud prevent a party from planning to generate alternative records before the device has captured a transaction. We consider this case less likely—the cost of installing the device-based system and then circumventing it is probably greater than for other means of fraud.

There are three primary types of reactive fraud. First, a party may attempt to modify the record of a transaction. However, each record has a cryptographic timestamp, so such a modification would have to include a forged timestamp, which requires compromising the private key used for timestamping. We prevent this attack by storing this key in a hardware security module (HSM) and performing all timestamping operations within the HSM's trust boundary. These security modules are available off-the-shelf and conform to the Federal Information Processing Standards for cryptographic devices (FIPS 140-1) [7]. A level four device is certified against any electronic or physical attack.

Second, a party may attempt to insert a new record into the system and then claim it is the correct record of a previous transaction. Again, timestamping using the HSM prevents this attack. Each timestamp has both a transaction time and a sequence number, making it easy to unambiguously determine which of two potential records of a transaction came first. An attacker might think that he could simply remove the original record, bringing us to the third form of reactive attack, an attempt to delete records. A party that owns the device containing the record in question could perform such a deletion. However, this attack would leave a gap in the record sequence numbers. By presuming that any gaps are the fault of the owner, we prevent the attacker from gaining any benefit from such deletions.

Proactive attacks are more problematic. Consider the typical transaction case. A human user logs onto a server system with a password and then executes a transaction. In most cases, the party that owns the server either controls this password or can gain access to it. Once he has it, he can arrange for a confederate to execute fraudulent

transactions that look exactly like legitimate transactions from that user. Even if the user employs SSL client authentication, it is possible for an attacker with access to the server's private key to instrument the server so that he can hijack session. Note that this hijacking is not possible if the server uses a hardened SSL device because nobody can access the device's private key. In the password case, the only real protection is taking the responsibility for recording transactions out of the server party's hands. The user or his organization could run its own Recorder and then compare the server party's version of the transaction to its own. Alternatively, both parties could agree that a third party, such as the server party's ISP, would securely administer the Recorder.

3 Project Goals and Approach

Given the operational ease-of-use and substantial security guarantees, we felt that the practicality of SSL auditing for commercial or governmental use deserved investigation. We proceeded by setting the system goals, creating an initial design, and allocating time to overcome the likely obstacles. As discussed below, we were lucky enough to have our results closely match these expectations.

3.1 Goals

Our goals revolved primarily around cost constraints and desired throughput. We were almost certain that we could build a useful system with very expensive custom hardware, but commercial and governmental deployment would rely on keeping costs in line with widely available network devices such as VPNs, firewalls, and intrusion detection systems. To meet this budget, we knew we would have to use two primary platforms: a server based on commodity x86 hardware and an off-the-shelf HSM from a reputable manufacturer. We deliberately set the cost constraints for the commodity hardware on the low end, figuring even modest success at that cost could scale effectively to large installations. Moreover, we knew that most HSMs were fairly expensive and we would need to reserve a substantial budget for this component.

For throughput goals, we considered the typical infrastructure at commercial and governmental installations. Medium-sized installations typically run their transaction servers on a 100Mbps LAN while large-sized installation typically use a 1Gb/s LAN. We wanted the ability to serve medium-sized installations with a single device, so our prototype would have to provide at least 100Mbps of throughput. We wanted to serve a large-sized installation with a cluster of no more than eight devices, so our prototype would have to provide at least 125Mbps of throughput with linear scaling of additional devices. Because the devices would operate in a completely passive sniffing mode, we would have to achieve this throughput with zero packet loss.

3.2 HSM Timestamping

We were very concerned about the throughput of timestamping on the HSM. Cryptographic signing operations typically consume a lot of processing power. Moreover, due to the additional security engineering and government certification, HSM processing power typically lags substantially behind that of commodity hardware. Therefore, before the project started, our initial design split timestamping into two pieces. The first piece comprised queuing captured packets, hashing them, and marshalling them for signing. We planned to execute these functions on commodity hardware, and then pass the marshaled hashes to the HSM. Using this approach, the HSM would expend processing power only for signing operations.

In choosing an HSM, we relied on our previous investigation into the capabilities and prices of the alternatives. The IBM 4758 offered the best choice. It has FIPS 140 level 4 certification and can accommodate custom applications running on its processor. Its list price (\$3000) is among the cheapest and its wide availability means that discounts are sometimes possible. We paid \$600 for our units. Finally, IBM does not charge for the custom application development kit. While they don't support it either, the community of 4758 developers is large enough to provide some level of peer assistance. The 4758 has drivers for Windows and Linux. To keep costs down, we therefore chose Linux as our platform operating system.

3.3 High Speed Capture

In choosing our hardware platform, we considered the potential bottlenecks in the system. Capturing on a 1Gb/s network would require a lot of I/O bandwidth so we knew the commodity hardware required a 64-bit 66MHz PCI bus. To accommodate the 4758, it would also require a 32-bit 33MHz PCI bus. We also knew that the network data processing would be processor intensive. AMD Athlon MP's appeared to have the best price/performance ratio, so we settled on a dual processor Athlon MP platform. We had a great deal of difficulty deciding on the disk technology. While we knew that recording captured transactions would be disk intensive, the highest throughput technology is SCSI RAID. This technology is quite expensive because manufacturers charge a premium for both the host bus adapters and the disks themselves. Eventually, we decided to give price more weight and settled on 7200 RPM IDE disks.

3.4 Clustering Feasibility

As noted above, we wanted the capability to capture transactions on a 1Gb/s LAN with a cluster of devices. Therefore, we planned to investigate the implementation of such a capability during this project. Unfortunately, fully implementing and testing clustering would require more time and money than we had in our budget, so we settled for examining feasibility only. Our plan was to implement a simple algorithm for partitioning SSL sessions among clustered machines using a hash that included some combination of machine addresses and SSL handshake information. We would then use this algorithm to run a cluster of two machines to verify that we could successfully capture transactions and roughly determine the scaling properties of a cluster.

3.5 Potential Obstacles

As with any research project, we had some a priori expectations about the potential obstacles to implementation. We divided these obstacles into two categories, HSM and sniffing, reflecting the physical partitioning of our system. For the HSM, our biggest concern was actually getting it working in a device. We faced two primary obstacles. First, we had to integrate the HSM into the combined hardware environment. The IBM 4758 Linux driver was unsupported and known to work only for an old version of the operating system. Second, we had to port our prototype signing application to the 4758. Executing such applications is also not supported and the 4758 uses its own arcane operating system called CP/Q. We foresaw a substantial amount of effort overcoming these obstacles. From reviewing IBM's specifications, we felt that the 4758's performance was likely to be adequate for our purposes and we could do little to affect this performance in any case.

For sniffing, we had more of a general concern about performance. Most of these concerns had to do with dropping packets. We really couldn't afford to drop any, so we would have to service the Ethernet interface very efficiently. We foresaw that context switching between capturing data and processing would be a potential bottleneck. Because of our decision to use IDE disks, we were also concerned with the amount of time it would take to write captured packets to disk. Even through the absolute throughput of the disks would have been enough to sustain a high capture rate, we couldn't afford diverting too many processing resources away from the network data to perform disk writes.

4 Implementation

4.1 Platform

As mentioned in Section 4, all components are currently in the same physical unit. We use a commodity rack mount PC, as shown in Figure 7.

Processor	Dual Athlon MP-2000+
Motherboard	Tyan S466-4M
Memory	1024 MB
Network	Intel PC/Pro 1000 (for sniffing) onboard 3c59x (management)
Disk	2 Maxtor 80GB 7200 RPM IDE
Operating System	Red Hat 7.3 (Linux Kernel 2.4)
HSM	IBM 4758 model -002

Figure 7 Recording system specifications

Linux was chosen as our operating system specifically because Linux drivers were available for the IBM 4758. However, we quickly discovered that the 4758 SDK was only compatible with Linux kernel 2.2, which is extremely downrev. For stability and performance reasons we deemed it better to use a more modern version. We therefore had to port the SDK to kernel 2.4 (Red Hat 7.3). This port is described in Appendix A.

4.2 Recorder

The recorder has four primary tasks to execute:

1. Read sniffed packets from the network.
2. Write `ts_reqs` to the Master for signing.
3. Read timestamps from the Master.
4. Write timestamps and signed packets to disk.

The Recorder is implemented as a single Linux process. Rather than use threads, all events are handled via a centralized callback mechanism. This allows maximum portability between operating systems (we might someday want to move to FreeBSD) as well as allowing tight control of scheduling and avoiding the need to lock data structures when passing them between threads. However, it does create the necessity for extreme care in the amount of time expended by any individual callback to avoid starving other tasks. Figure 8 shows the relationship between the various stages.

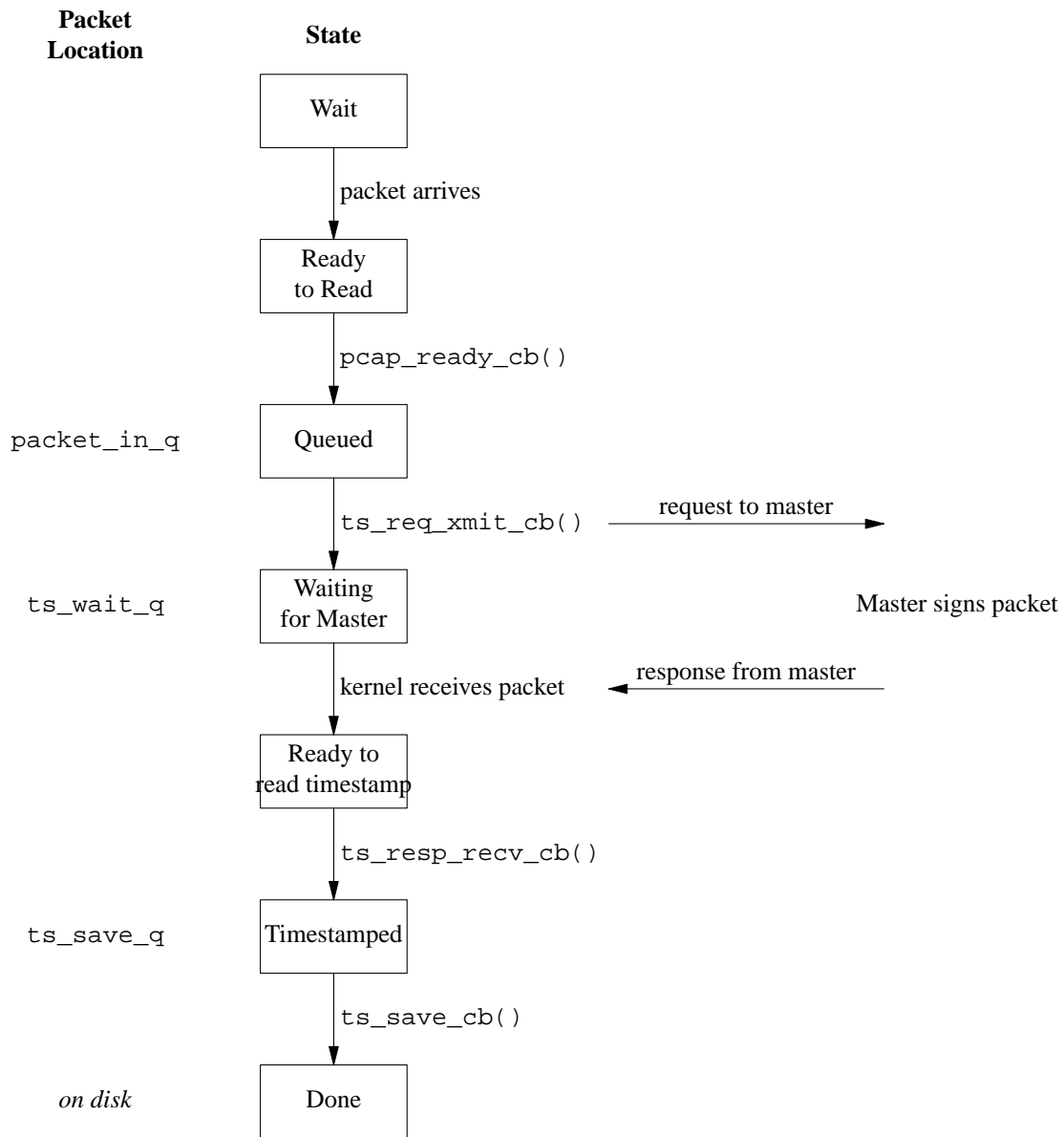


Figure 8 Recorder state machine

4.2.1 Packet Capture

Packet capture is accomplished by way of the PCAP packet capture library (Phil Wood's prototype libpcap-0.8.1104 <<http://public.lanl.gov/cpw>>) An event is registered on the file descriptor associated with the `pcap_t` context structure and when the file descriptor is ready to read the readable callback fires. The reading callback invokes PCAP to read packets up to a maximum of 10,000 packets (compiled into the Recorder code). This read is done in non-blocking mode so if fewer packets are available the call returns anyway.

As each packet is captured, a `ts_req` structure is created containing the packet data and some metadata such as the connection number, packet index and arrival time. This `ts_req` structure is then threaded onto the `packet_in_q`. If necessary, an event is scheduled for the write to the Master.

Preventing Packet Loss

Servicing the packet input from PCAP is the most critical task performed by the Recorder and controls its ability to handle peak loads. If any other part of the Recorder has a performance glitch, the size of the various `ts_req` queues increases and an excessive amount of memory is consumed. Once the load is reduced these queues eventually empty. Under sustained periods of high load, of course, the queues will grow to consume all available memory, but the amount of such memory is very large and bounded only by the amount of swap available.

By contrast, the amount of space in the kernel read buffers that we are servicing is fixed and generally quite small. By default it is 64k but in our Recorder we increase it to 50MB. Nevertheless, this represents less than 3 seconds of traffic on a 100 megabit Ethernet. When this buffer fills up the kernel starts dropping packets. Packet dropping isn't ordinarily a problem with TCP because the sender will just retransmit. However, because we're a passive listener, as long as the packet is received and ACKed by the receiver no retransmit will occur. Thus, capturing every packet is critical.

In order to prevent packet loss, we need to do three things: First, we need to ensure that all other callback actions are relatively short so that the packet reading callback is serviced regularly. Second, the packet reading callback must be greedy. Our callback reads up to 10,000 packets at a time. If it actually succeeds in reading a full buffer it assumes that it's falling behind and goes into *emergency mode* by deactivating other tasks in order to save CPU time for packet capture. Finally, we need a relatively large input buffer to smooth out temporary spikes and starvation. As previously mentioned we increase the size of the buffer to 50MB¹.

4.2.2 Sending requests to the Master

Sending requests to the Master is relatively straightforward. The next group of packets is chosen out of the `packet_in_q`. The packets are hashed and the hashes plus metadata are marshalled into a single data structure (see Figure 9) which is then transmitted to the Master over an SSL connection. (The specification language is from TLS.)

```
struct packet_info {
    uint16 length;
    uint64 conn_num;
    uint64 packet_seq;
    uint8 direction;
    uint64 recorder_sequence_number;
    uint64 arrival_time;
    uint32 tcp_sequence_number;
    opaque data_digest<255>;
    opaque running_digest<255>; // unused
}
```

Figure 9 Request data structure

`length` is the overall request length

`conn_num` is the connection number

`packet_seq` is the sequence number of the packet in the connection

`direction` is the direction the packet was flowing (initiator to responder)

`recorder_sequence_number` is the overall sequence number of this packet as received by the recorder

`arrival_time` is the time that this packet arrived at the recorder

`tcp_sequence_number` is the TCP sequence number in the packet

1. In performance tests we discovered that the MMAP packet mode of PCAP produces very substantial packet loss rates at high network speeds. We attribute this to the small size of the MMAP packet ring (32,768 packets).

`data_digest` is a SHA-1 digest of the data

In order to keep the cost of this callback small the number of requests that may be transmitted is limited to what will fit in a single SSL record (197). In order to prevent starvation, writes are performed in non-blocking mode. If a write returns with a would-block error then a flush event is scheduled. In any case, once a request has been successfully written to the Master, the relevant packets are threaded onto the `ts_wait_q` while the Recorder waits for the master to return a timestamp.

When to schedule a sending event

The decision about when to schedule a sending event requires balancing promptness and overhead. If we schedule sending events only when a large number of packets are ready for timestamping then timestamps will always be out of date. If the data rate is high then this isn't a problem but if the data rate is low then the timestamps may be quite incorrect. On the other hand, if we schedule a sending event whenever a packet is ready we may end up with each `ts_req` in its own SSL record. This would result in a large amount of per-packet SSL overhead.

In order to balance these two goals we adopt a strategy adapted from TCP's Nagle algorithm [8]. Sending events are only scheduled when either there are no outstanding unsigned timestamp requests or when a full SSL packet's worth of `ts_reqs` is ready to be sent. This approach is self-clocking. Under conditions of high load `ts_reqs` are sent in large groups for low overhead. In conditions of low load they are sent singly to maximize prompt signing.

4.2.3 Receiving responses from the Master

The Recorder registers a callback to fire when the socket from the Master becomes readable. The callback reads the timestamp from the socket and writes it to disk. It then moves the signed `ts_reqs` to the `ts_save_q` and schedules a saving callback if one is not already registered.

4.2.4 Saving packets to disk

In principle, saving the timestamped packets to disk is straightforward: simply loop through all the packets, create `stored_packet` structures, and write them to disk. However, we encountered a few difficulties with our initial design that required us to modify it for acceptable performance.

The first problem we encountered was starvation from multiple writes. Any given timestamp may cover a large number of packets, especially at high data rates. If all of these packets are written to disk at once enough time may be consumed to starve the capture task. Instead, we write a single packet from the `ts_save_q` to disk per cycle through the event loop. This strategy recognizes that writing packets to disk is far less critical than reading them from the network. Reading from the network is a bottleneck on peak capture whereas writing to disk is only a problem for sustained high capture rates.

The second problem we encountered was a result of wanting to save each connection in an individual file, as discussed in Section 4.2.4. Unfortunately, having a large number of files open means that the kernel has to do a lot of seeking in an attempt to write each file contiguously. This results in substantial slowdown and once again causes starvation and packet loss. The fix is to write all packet data to a single file and demultiplex later. In order to avoid exceeding filesystem limits we automatically close the current file and open a new one at 1 GB.

Storing Packets

Packets are stored as a sequence of the structures shown in Figure 10.

```
struct stored_packet {
    uint16 length;
    uint64 connection_number;
    uint64 packet_seq;
    uint32 flags;
    uint64 timestamp_id;
    uint16 timestamp_index; // Unused
    uint32 tcp_sequence_num;
    opaque packet_data<0..2^16-1>;
}
```

Figure 10 Stored packet format

length describes the total length of the structure

connection_number is an index that increases by one for each connection the recorder has seen.

packet_seq is the sequence number of the packet within the connection

flags is a flags word used to indicate special packets such as the start and end of the capture

timestamp_id is the sequence number of the timestamp that covers this packet

timestamp_index is intended to point to the timestamp entry within the timestamp for this packet. It is currently unused

tcp_sequence_num is the TCP sequence number of the packet

packet_data is the actual captured packet

Because each stored_packet structure contains a connection number and sequence numbers they are self-contained. This allows a number of arrangements. For maximal convenience when replaying a connection, each connection would be stored in its own disk file. However, for the performance reasons discussed above, it is necessary to store the data initially to a single file. It would be easy to write a background process to demultiplex the single packet stream into per-connection files, leaving us with the data structure shown in Figure 11. In this figure, we see two separate connection files pointing to entries in a single audit log. Each audit entry covers one or more packets.

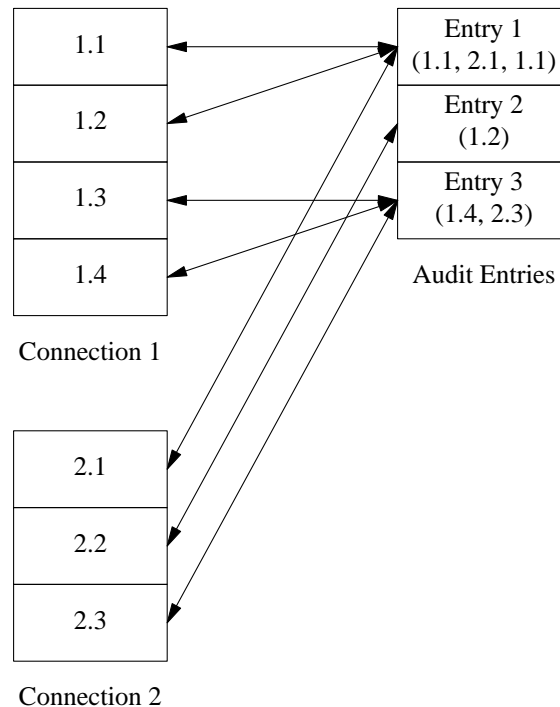


Figure 11 Recorder data structure

The stored connection files live on a pair of IDE drives arranged in a RAID 0 configuration. They are formatted with the ext2fs [9] filesystem. We configure the filesystem without journalling because in our tests with we found that the journalling daemon would occasionally consume substantial amounts of CPU and result in packet loss when operating at very high loads. Without journalling ext2fs seems to have more smooth behavior.

Storing Timestamps

Although timestamps are inherently sequential, even two consecutive packets might have nonconsecutive timestamps. Accordingly, it is very convenient to be able to have random access to timestamps based on their sequence

numbers. Initially we used Berkeley DB for timestamp storage, however the quantity of timestamps, both in terms of number and size quickly led to errors.

However, the fact that timestamps are written sequentially—even though they may be randomly accessed—lends itself to a simple file based data structure. We use two files: a data file to store the actual timestamps and an index file to store their file offsets. Each index entry is fixed at 8 bytes long. Thus, to find timestamp x we can simply seek to position $8x$ in the index file, read the relevant file offset and then seek to the appropriate position in the data file. This data structure can be written very quickly and read comparitively quickly and is thus ideal for our purposes.

4.3 Master

In our original design, the Master was made up of two pieces, the Master application and the Signer. The Master application lives on a general purpose computer and proxies data to and from the Signer. The Signer was to live on the 4758 and is responsible for actually signing the timestamp requests. The Master application has three jobs:

1. Read requests from the Recorder.
2. Assemble them into batches and pass them to the Signer
3. Return signed timestamps to the Recorder.

Unfortunately, we soon discovered that this design did not allow for asynchronous signing. The Linux API provided by the IBM with the 4758 only has a synchronous mode of operation. Using that API directly would require that the Master completely stall while waiting for the Signer to complete a signing operation. This leads to poor performance.

In order to allow asynchronous operation, we split the Signer into two pieces, as shown in Figure 12. The proxy application (`csm4758`) lives on the host and is a subprocess of the Master. It communicates with the Master via `stdin` and `stdout`. In turn it communicates with the application on the 4758 (`app4758`) via the 4758 host API. This approach allows the Master to treat the Signer as just another file descriptor to read and write from and thus asynchronously transmit requests and receive responses.

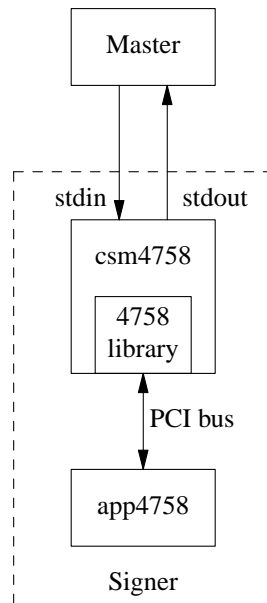


Figure 12 Master interaction with Signer

app4758 is pretty simple. It sits in an infinite loop reading commands from csm4758, executing them, and returning the results. For our purposes, the only relevant command is the request for a timestamp. app4758 uses its local key to sign the timestamp and returns the sequence number, signing time, and the signature.

4.4 Performance

In order to characterize the performance of our system we built a simple testbed (shown in Figure 13) consisting of four load machines and a capture machine on a switched gigabit Ethernet network. We used a prototype Broadcom gigabit switch configured to mirror all traffic to the capture machine.

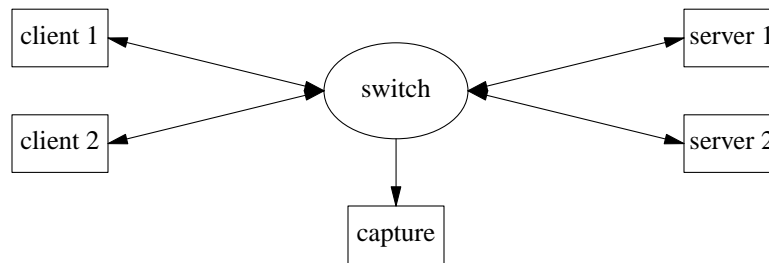


Figure 13 Testbed configuration

Our load machines are commodity rack mount PCs, as shown in Figure 14.

Processor	Pentium III 1GHz
Motherboard	ACorp 6PLEF ATX (VIA 8633)
Memory	512MB
Network	Intel PC/Pro 1000 (for sniffing) Realtek 8139 10/100
Disk	Maxtor 40GB 7200 RPM IDE
Operating System	FreeBSD 4.7

Figure 14 Testbed system specifications

To generate load, we use the nttcp [10] client server pair. nttcp simply moves TCP data at maximum possible speed from one machine to another. With our equipment, a single client/server pair of machines can do roughly 380 megabits of data traffic/second. Note that although we are generating TCP traffic rather than SSL traffic, this is irrelevant from the perspective of capture because we do not examine the traffic but merely store it.

4.4.1 Profiling Results

Because the Master operates under low load and self-adjusts to high load conditions, the Recorder is the performance bottleneck. Profiling indicates that the three major time contributors to the Recorder are:

1. Storing packet data to disk.
2. Digesting captured packets.
3. Reading captured packets.

There are relatively obvious avenues for improving the performance of each of these tasks. The simplest is the packet data store. We use IDE disks and software RAID, both of which consume a fair amount of CPU. Upgrading to faster SCSI hardware RAID would likely significantly reduce the amount of CPU time consumed by storage.

We currently use SHA-1 to digest packets as they arrive. Converting to MD5 should produce a substantial performance improvement with an acceptable reduction in security. It might also be possible to further optimize the

SHA-1 code or employ hardware acceleration. Finally, the MMAP packet capture code is likely substantially faster than the ordinary PCAP code. However, as previously noted, we believe that its limited capacity was the reason for high packet loss rates. This may be fixable.

Because we were able to achieve our performance goals without employing these optimizations, we did not explore them any further. However, they represent a potential avenue of future research for situations that cannot be handled by a current recorder but where clustering is undesirable.

4.4.2 Sustained Load

We characterize the performance of the system via two primary metrics: sustainable capture rate and peak capture rate. Due to the inherent buffering of the system, these two numbers are rather different.

We have successfully used the recorder to capture sustained TCP traffic at 178 Mb/s data rate (193 Mb/s of actual traffic). At this load, the CPU consumption is on the order of 90%. Thus, we expect that although the actual peak sustainable load is 200 Mb/s or so, running at these speeds one would be extremely susceptible to packet loss from minor operating system glitches such as page faults or periodic cron jobs.

4.4.3 Peak Load

As previously mentioned, our scheduler is biased to favor packet capture over packet timestamping and storage. Thus, under peak load conditions, the vast majority of the CPU is spent reading packets from the network and allocating space to store them. We have successfully captured packets at peak loads as high as 380 Mb/s (400 Mb/s of actual traffic). With some tuning to the allocation and buffering algorithms we believe that it would be possible to see peak loads in the range of 500 Mb/s. Obviously, performing at this kind of load for any length of time requires quite a bit of memory, because the data is being buffered rather than stored to disk.

4.4.4 Simulated Web Traffic

Our experiments show that nttcp allows us to maximally load the network. However, we also performed tests with simulated Web traffic of various numbers of connections and file sizes using the Apache Web Server [11] and the ab traffic generation tool. After some initial performance issues involving connection lookup we observed similar performance under these conditions. This is unsurprising because almost all of the Recording logic is independent of the number of active connections.

4.5 Multiple Recorders

We expect that the ability to capture at 200 Mbp/s will be sufficient for nearly all applications. However, in certain special cases it may be necessary to capture at higher speeds than a single unit can sustain. In such cases, the obvious approach is to use more than one Recorder machine. There are essentially two ways to deploy such machines, clustered and independently.

4.6 Clustering

In a clustered system all the traffic is mirrored to multiple ports instead of one, with one mirror port allocated for each Recorder. Each Recorder sees all of the traffic and they cooperate to divide up the space of traffic so that each Recorder processes only a subset of packets.

The simplest method of dividing the space of traffic is to use a hash function as described in[12]. Essentially, we have some function.

$$f(\text{source_addr}, \text{source_port}, \text{dest_addr}, \text{dest_port})$$

Each Recorder gets a portion of the hash values. If the hash lies for a given packet lies in that hash space the packet is accepted. Otherwise it is rejected. In order to ensure that each direction of traffic on a given TCP connection goes to the same Recorder, f must be symmetrical with respect to the direction of packet flow. One simple such function is:

$$f = \text{source_addr} + \text{source_port} + \text{dest_addr} + \text{dest_port}$$

The clustering approach has two obvious problems, both related to aggregate packet load. Because all of the traffic to be sniffed must be available to each Recorder then the maximum amount of aggregate traffic that can be sniffed is the capacity of a single link (in most cases 1 gigabit). In most cases this will not be an issue, but might be a problem if traffic rates are extremely high.

The second problem is load on the individual Recorder machines. Even though the Recorder only records the slice of traffic that is assigned to it, it must still examine each packet in order to determine whether it must handle it or not. This can consume a significant amount of CPU time.

Our initial attempt at this sort of packet segmentation was to simply add a rejection function to the Recorder's packet handler. Unfortunately, profiling quickly revealed that just reading the packets from PCAP was consuming a very substantial amount of CPU time. As a consequence, the Recorder's ability to handle traffic was substantially degraded.

A better approach is to reject the packets before they ever reach application space. Linux and FreeBSD both allow applications to install BPF [13] filters on capture ports. BPF filters are actually small programs executed by the kernel. Therefore it is quite easy to write a small hash function as a filter. With such a filter in place, only the relevant packets reach the Recorder application.

Because our switch was unable to mirror to multiple ports we were unable to directly measure clustering performance with two hosts. However, we were able to test with a single host and an aggregate traffic load of 760 Mb/s. With a BPF filter designed to partition the traffic in half our Recorder was able to handle its share of the load. Because the clustered Recorders would operate independently except for the occasional management message, traffic coverage should scale appropriately with additional devices. We also tested multiple Recorders on the same physical unit against a single Master and verified that that configuration operated correctly.

4.6.1 Segmented Topologies

The second way to use multiple Recorders is to arrange the topology of the network in such a way that it splits the traffic between multiple Recorders, as shown in Figure 15. This has the advantage that the aggregate traffic limit no longer exists. Because each Recorder only sees the traffic destined for it, the aggregate Recording capacity scales linearly with the number of Recorders.

The primary difficulty with this approach is that the topology is relatively fixed. As a consequence, the administrator needs to determine in advance what percentage of traffic is likely to be on each link in order to design the topology. If the traffic characteristics of the network change, the load allocation will be unresponsive and packets may be lost.

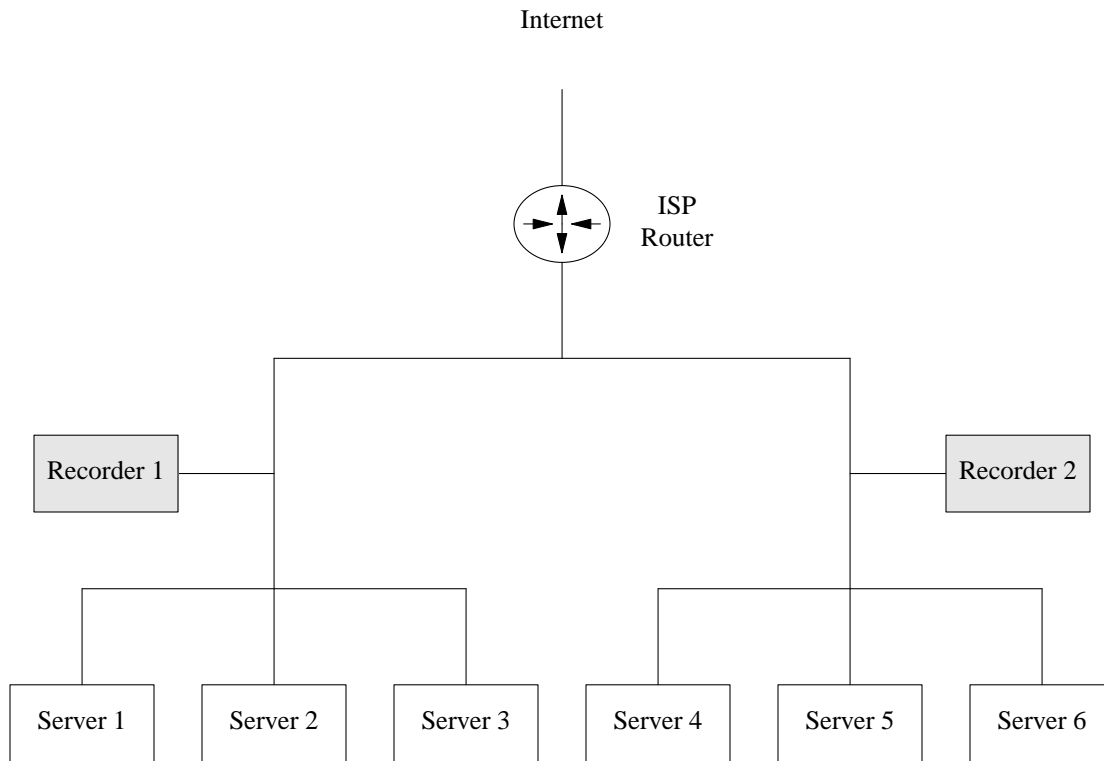


Figure 15 A segmented network

4.6.2 Shared Master

In the previous sections we have assumed that each Recorder would have its own Master. However, as noted in Section 4.5.1, the load on the Master is comparatively low. It would therefore be possible for a number of Recorders to share a single Master. As discussed in Section 5, the Master is explicitly designed with this configuration in mind.

5 Conclusions and Future Work

Our goal for this project was to develop a practical version of our SSL Recording system. This meant that it had to meet two constraints: cost and performance. We were largely successful on both counts. The system described in this paper has a current hardware cost of approximately \$2600 (\$2000 for the general purpose machine) and can perform sustainable captures at 180 Mb/s and peak captures at 380 Mb/s.

This project demonstrates the feasibility of building an economical high speed capture appliance. However, the weak point now is the absence of tools to handle the large amounts of captured data. There are two problems here. The first is that the amount of disk storage consumed will quickly exceed the total amount of disk in our machines. The fix for this is to use hierarchical storage such as network attached storage and/or offline storage. The Advanced Packet Vault [14] has already shown this to be possible when the data is spooled to tape, but we really want a more complicated hierarchy to provide ready access to more recent data. The second problem is being able to find the appropriate connections for a given transaction. This requires some way to bind transactions to connections.

The problem of replay is also largely unsolved. Although the original paper [1] described how to do simple replay, this is only useful for single connection manual replay. In many cases it will be desirable to replay a large number of connections and export them into some common format. We have not explored this avenue to any great extent.

The problem of performance still deserves some attention. Currently, it is not clear that two recorders can

actually handle a gigabit of aggregate traffic. As previously noted, we believe that with sufficient tuning, peak loads of up to 500 Mb/s can be handled with a single recorder. This tuning has yet to be done.

A.1 Developing for the 4758

5.2 Drivers

IBM supplies 4758 drivers for Linux and Windows. The Linux driver is an Open Source driver distributed by IBM Alphaworks. Unfortunately, this driver is only compatible with Linux kernel version 2.2. This version of Linux is extremely old and we wanted to use a more modern version. Because the kernel had changed quite a bit between versions 2.2 and 2.4 (mostly in the direction of adding more abstraction) we needed to port the driver to kernel 2.4. The issues are briefly summarized below.

Wait queues have changed. In the 2.2 kernel there was a single queue element. In the 2.4 kernel, a BSD-style queue was used with separate `wait_queue_head_t` and `wait_queue_t` elements. We ported the wait queue handling to the new style.

Signals are now handled indirectly. The old way to access the signal stack was to simply examine data structures directly. The new way is to use the kernel-provided API calls such as `signal_pending()`⁷. We converted the relevant driver code.

Driver IO addresses are acquired indirectly. The old way to find IO addresses for devices was to examine various structures directly. Kernel 2.4 has a `pci_resource_start()` call to do the same job. We use that.

Assorted structure fiddling. In a number of cases structures added or deleted fields that were unused by the driver in any case. We added or deleted zero initializers as appropriate.

We have contributed the driver modifications back to IBM for redistribution by Alphaworks.

5.3 Building Binaries

IBM's SDK uses a somewhat unusual procedure to build binaries for the 4758:

1. Compile the objects using the ordinary platform compiler.
2. Link the objects using the ordinary platform linker and IBM-provided libraries (not ordinary `libc!`). The "binary" is linked as a shared library, not a final form object.
3. Translate the object into a translated binary using IBM's `cpqxl` tool.
4. Sign the object and put it in a package for download using IBM's `sccrodsk` tool.

This procedure presents two problems when moving from Red Hat 6.2 to Red Hat 7.3:

The SDK tools are linked to the wrong library versions. The IBM tools are provided in binary form only and are linked to various libraries from Red Hat 6.2. Red Hat 7.3 uses newer versions of these libraries. In order to get the tools to run we had to copy the relevant libraries from a 6.2 system to our 7.3 development system.

Translation doesn't work with Red Hat 7.3's toolchain. Red Hat 6.2 uses `egcs` version 1.1.2-30 and `GNU ld` version 2.9.5.0.22. Red Hat 7.3 uses `gcc` version 2.96 and `GNU ld` version 2.11.93.0.2. Unfortunately, `cpqxl` is very sensitive to output format and cannot translate binaries compiled with the newer toolchain. In order to get successful compilation we installed the older versions of `gcc` and `GNU ld` on our Red Hat 7.3 system.

5.4 A 4758 Emulator

In general, developing programs on embedded platforms is relatively unpleasant and the 4758 is no exception. In order to make development easier we developed a primitive 4758 emulator so that we could work on an ordinary PC. The emulated program runs as a separate UNIX process on the same machine as the host-side application.

Communication between the host-side and "card-side" programs is by a pair of pipes.

Implementing this is comparatively simple. We designed a pair of libraries that replace those provided by IBM. The host-side library implements the calls to communicate with the card as calls to send and receive data from the subprocess. The card-side library implements the calls to communicate with the host as calls to communicate with the parent process. In addition the card-side library implements the cryptography and state management functions that would ordinarily be provided by the IBM-provided card side library.

This approach has a number of advantages. First, it allows us to develop even while working on systems that do not have a 4758 card—such as laptops. Second, it allows us to fix most bugs before we ever have to use the card. Because debugging on the card is a relatively difficult and slow process, this speeds up development. Third, it allowed us to do performance tuning without worrying about the performance of the 4758. This was invaluable in the early stages of profiling when we wanted to deal with as few variables as possible.

Currently, the emulator is in a rather rough state. However, we believe that with some modest polishing it would make a useful tool for other 4758 developers. We are still exploring how best to make it available to the community.

References

- [1] Rescorla, E., and Dick, K., "*Secure Auditing for SSL*" (submitted to USENIX Security).
- [2] Freier, A.O., Karlton, P., and Kocher, P.C., *The SSL Protocol Version 3.0* (November 1996).
<http://home.netscape.com/eng/ss13/draft302.txt>
- [3] Dierks, T., and Allen, C., "The TLS Protocol Version 1.0," RFC 2246 (January 1999).
- [4] Dusse, S., Hoffman, P., Ramsdell, B., Lundblade, L., and Repka, L., "S/MIME Version 2 Message Specification," RFC 2311 (March 1998).
- [5] Atkins, D., Stallings, W., and Zimmermann, P., "PGP Message Exchange Formats," RFC 1991 (August 1996).
- [6] "Digital Certificates: A Solution in Search of a Problem," *Future Banker* (2001).
- [7] National Institute of Standards and Technology, "Security Requirements for Cryptographic Modules," FIPS PUB 140-1, U.S. Department of Commerce (January 1994).
- [8] Nagle, J., "Congestion Control in IP/TCP Internetworks," RFC 0896 (Jan 1984).
- [9] Card, R., Ts'o, T., and Tweedie, S., "Design and Implementation of the Second Extended Filesystem," *Proceedings of the First Dutch International Symposium on Linux*.
- [10] Bartel, E., *New TTCP Program*.
<http://www.leo.org/~elmar/nttcp/>
- [11] *Apache*.
<http://www.apache.org/>
- [12] Nokia, *The IP Clustering Power of Nokia VPN* (April, 2001).
http://www.nokia.com/vpn/pdf/ip_clustering.pdf
- [13] McCanne, S., and Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture," (*USENIX Winter*, pp. 259-270 (1993).
- [14] Antonelli, C., Coffman, K., Fields, J., and Honeyman, P., *Cryptographic wiretapping at 100 Megabits*.