Dartmouth College Computer Science Technical Report TR 2001-399

WebALPS Implementation and Performance Analysis: Using Trusted Co-servers to Enhance Privacy and Security of Web Interactions

Shan Jiang
Advisor: Professor Sean Smith
Dartmouth College

June 3, 2001

Abstract

The client-server model of the Web poses a fundamental trust issue: clients are forced to trust in secrecy and correctness of computation occurring at a remote server of unknown credibility. The current solution for this problem is to use a PKI (Public Key Infrastructure) system and SSL (Secure Sockets Layer) digital certificates to prove the claimed identity of a server and establish an authenticated, encrypted channel between the client and this server. However, this approach does not address the security risks posed by potential malicious server operators or any third parties who may penetrate the server sites.

The WebALPS (Web Applications with Lots of Privacy and Security) approach [24] is proposed to address these weaknesses by moving sensitive computations at server side into trusted co-servers running inside high-assurance secure coprocessors.

In this thesis, we examine the foundations of the credibility of WebALPS co-servers. Then we will describe our work of designing and building a prototype WebALPS co-server, which is integrated into the widely-deployed, commercial-grade Apache server. We will also present the performance test results of our system which support the argument that WebALPS approach provides a systematic and practical way to address the remote trust issue.

Acknowledgments

I am deeply grateful to Professor Sean Smith for his advisement, support, and encouragement throughout this project. He has been extremely generous with his ideas and time. I have learned a great deal about research as well as writing from him. I cannot imagine having a better advisor both in terms of intellect and personality.

I want to express my gratitude to other members of my thesis committee – Professor Chris Hawblitzel and Edward Feustel for carefully reading the draft of this thesis and providing invaluable feedbacks.

I want to thank Joan Dyer for her advice and tool support. I also want to thank Guanling Chen for providing one of the machines that was used for testing and Qun Li for helping me with Linux drawing tools.

This work was supported in part by U.S. Department of Justice, contract 2000-DT-CX-K001, by Internet2/ATT, and by an equipment loan from IBM Watson. Therefore, my thanks go to them for making this project possible.

Finally, I am very thankful to Mary for her unconditional support and understanding, and for always being there for me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Challenge

Ever since its origination in the early '90s, the World Wide Web has grown explosively. From May 1999 to September 1999, according to Alexa Research's Internet archiving project [3], the number of Web hosts rose from 2.5 million to 3.4 million. That is a 31% growth rate in just 4 months, or 125% per year. As to the number of Web pages, Cyveillance, an internet consulting company, estimated that the Web contained 2.1 billion unique, public accessible pages by July 2000, and is growing at a rate of 7.1 million pages per day [9]. Besides the growth in size, the Web also grows in terms of the number of services it offers. Among many other things, people go to the Web for shopping, banking, trading, training, and entertainment on a daily basis. The Web has evolved from a static information provider to an interactive, dynamic environment, which has become an inseparable part of many people's lives. According to ActivMedia, global on-line population had reached 300 million in 2000 [2]. The impact that the Web imposes on the business world is tremendous too. The retail e-commerce sales in the fourth quarter of 2000 were $8.7 billion, an increase of 35.9% from the

previous quarter [28].

As the Web and e-commerce grow, so do cybercrimes. The most recent survey conducted by the Computer Security Institute at Carnegie Mellon University on 538 U.S corporations and government agencies [7] revealed that system penetration by outsiders grew by 60% in 2001. Unauthorized access by insiders rose too, by 15%. The financial loss caused by these security breaches was reported to be over $370 million. In addition to significant financial loss of big corporations, cybercrimes poses a serious threat to internet users' privacy. Cyberthefts of users' private information have been reported regularly. In March 2000, it was reported that a computer intruder stole information on more than 485,000 credit cards from an e-commerce site [5]. More recently, anti-globalization computer hackers broke into the computer systems used by the World Economic Forum and stole personal information of most of the participants in the Forum's latest annual meeting [17]. The information stolen includes 27,000 names paired with e-mail addresses, phone numbers, travel schedule, Web-site passwords, and credit card numbers.

From the above statistics and cases, it is not hard to see that as more mission and corporate-critical information is provided via the Web, and as the sophistication and number of cyber-related crimes continues to expand, to enhance privacy and security of Web interactions has become essential. How to enforce Web security is a very broad problem. In this thesis, the writer will address one particular trust issue that is intrinsic to the Web due to its distributive nature: *Why should a user trust the secrecy and correctness of computation occurring at some remote server, where an adversary might be motivated to subvert it?*

1.2 Current Solutions

For a client to have trust on sending sensitive information to some remote server for computation, the following properties must be preserved and proved to the client:

- the authenticity of the server: the server must be able to prove its claimed identity to the client before the client sends out any sensitive information;
- the secrecy of the communication channel: any sensitive information must be immune to eavesdroppers during transmission;
- the confidentiality of the client's information at the server: the server must employ strong enough security measures to keep the user's information secret, despite of any hacking attempts from inside or outside;
- the correctness of the computation that provides the service: the service offered by the server must do exactly what it claims to do, nothing less and nothing more.

A large amount of research, development, and standardization have been done to enforce these properties. One field of research focuses on the design of secure Web communication protocols. Netscape Communications proposed the Secure Socket Layer (SSL) protocol [12], which enables the client and the server to agree on a set of ciphers before transmitting any higher-level protocol (HTTP in most cases) messages. Integrated with a Public Key Infrastructure (PKI) system, SSL also makes it possible for the server to authenticate itself with an SSL digital certificate. Through these two measures, the client and server can establish an authenticated, encrypted channel for communication, which

- protects the client's information during transmission and

- makes certain that the client's information reaches the desired destination.

Other standards that aim at the same goals include Secure HTTP (SHTTP) [21] proposed by CommerceNet and IP Security Protocol (IPSec) [16] by The Internet Engineering Task Force (IETF). However, these protocols do not address security risks posed by server-side insider attack. Nor do they provide any assurance to the clients about the correctness of the service that the server provides or what kind of security measures the server employs to guard the client's sensitive information.

Another field of research aims at enhancing server-side security against attacks from inside and outside. Firewalls are developed to thwart outside assaults. Many commercial server systems employ techniques such as strong public key encryption, two-factor authentication, and policy-based access control to ward off security threats posed by insiders as well as outside hackers. Even though these measures make hacking and cybersabotage a lot harder, weaknesses remain:

- exploitable security holes still exist in the systems that use them;
- there is no valid means for the server to prove to the clients that it has employed these measures.

Moreover, even if it were possible to build a 100% secure server that could defeat any security attacks, clients still may not trust this server because the site operator could be motivated to misuse clients' sensitive information or cheat the clients by offering fraudulent services. Imagine a distributed online blackjack gambling system; with the amount of money involved, why should a high-roller client trust that the server is dealing the cards honestly?

Clearly, even with secure communication protocols and a server equipped with state-of-the-art security measures, the fundamental trust problem remains unsolved: *Participants in distributed Web services are forced to trust server integrity, but have no basis for this trust* [24]. The WebALPS (Web Applications with Lots of Privacy and Security) approach is proposed to provide a systematic

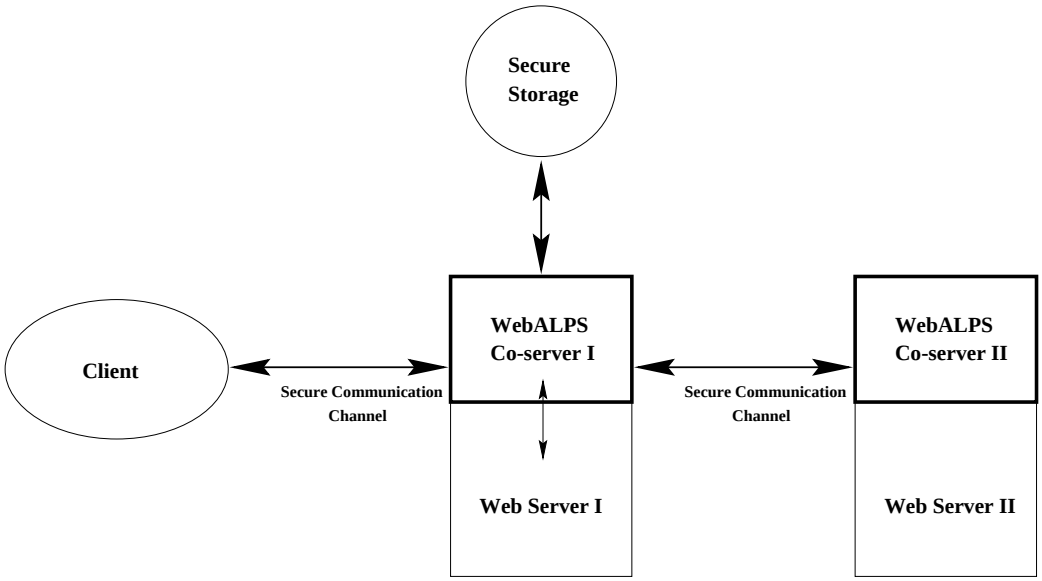

Figure 1.1: Web Communication with WebALPS Co-servers

and practical way to establish the foundations of this trust.

1.3 The WebALPS Approach

WebALPS approach is based on the secure coprocessing architecture [22] built at IBM Watson. The core idea of this approach is to augment Web servers with trusted co-servers running inside high-assurance secure coprocessors that handle the sensitive computations that previously clients have to blindly trust the servers with. The secure coprocessor offers a secure place to carry out those sensitive computations beyond the observation or manipulation of insiders as well as adversaries with direct physical access to the server system.

Figure 1.1 illustrates the model of Web communication with the participation of WebALPS co-servers. When *Client* accesses services offered by *Web Server I*:

- *WebALPS Co-server I* opens an authenticated, encrypted communication channel with *Client* via a secure communication protocol such as SSL;
- *Client* sends his request and sensitive information to *WebALPS Co-server I* through the established channel;
- depending on the type of service the server offers, *WebALPS Co-server I* can take one or more actions from below:
  - process *Client*'s request;
  - selectively forward *Client*'s information to *Web Server I* to process *Client*'s request;
  - selectively store *Client*'s information in an encrypted format in some external storage media;
  - establish a new authenticated, encrypted channel with another trusted entity (*WebALPS Co-server II*) and selectively forward *Client*'s information to that entity to continue processing *Client*'s request.
- after request has been processed, *WebALPS Co-server I* writes the generated response back to *Client* through the established secure channel.

In this scheme, it is still the secure communication protocol that assures the client of the authenticity of the server and the privacy of his information during transmission. However, the introduction of the trusted co-server brings two properties that cannot be guaranteed or proved before:

- the privacy of the clients' information at the server: the client's sensitive information is sent over to the WebALPS co-server, which runs inside a highly tamper-proof secure coprocessor. A third party, including the site operator with root/administrator privilege, will not be able to steal this information even with physical access to the coprocessor;

- the correctness of the computation that provides the service: applications running inside the coprocessor are normally publicly reviewed and have their correctness certified. Furthermore, mechanisms are provided for these applications to authenticate their identities and status during run-time to remote clients.

These properties, plus the ones provided by the secure communication protocol, build a solid foundation for the clients to cast their trust on WebALPS-enabled Web services.

1.4 The Thesis

Although theoretically sound, the WebALPS approach will not be adopted into the current Web model until it is prototyped and integrated into industrial-strength Web servers. Therefore, the work presented in this thesis is aimed at making the WebALPS idea real by providing an efficient prototype and analyzing its performance. Below is a brief sketch of what is in the remaining chapters:

- **Chapter 2** outlines the project requirements and goals;
- **Chapter 3** and **Chapter 4** describes in detail the two important pieces of technology that form the foundation of WebALPS approach: Secure coprocessors and SSL;
- **Chapter 5** explains how Apache and OpenSSL implements the SSL protocol;
- **Chapter 6** contains the details of the WebALPS implementation;
- **Chapter 7** lists and analyzes the results of performance tests;
- **Chapter 8** concludes the thesis and presents the directions for future work.

Chapter 2

Project Outline

This chapter lists the requirements for the implementation of WebALPS co-servers and outlines the road map of the project.

2.1 The Requirements

The requirements on the prospective system directly determine its design. The following subsections describe the major requirements.

2.1.1 Security

To enhance web security is the major motivation behind WebALPS approach. Naturally, security becomes the top concern during the implementation of WebALPS co-servers. These co-servers will be responsible for managing the clients' sensitive data. To reduce the likelihood that this data could be compromised, the co-servers must assume that the web servers may have been written or modified

by an adversary in an attempt to mount an attack on them. Defensive coding should be enforced throughout the implementation. For example, the co-server's code must thoroughly validate every argument passed in from the web server.

2.1.2 Performance

As Figure 1.1 shows, WebALPS approach complicates the current Web communication model by introducing extra entities (co-servers) and extra steps (interaction between co-servers and Web servers) into the picture. Therefore, one can safely predict that this approach will degrade Web servers' performance. On one hand, just as SSL-enabled host is generally slower than non-SSL host, it should be expected that WebALPS-enabled servers gain security at the cost of performance. On the other hand, for the WebALPS approach to become widely accepted by the industrial world, the performance it offers must be reasonable. Therefore, keeping the speed of the WebALPS-enabled server from dropping too much is another requirement of the project.

2.1.3 Scalability

In addition to performance, scalability is another important standard for evaluating commercial Web servers. To be ready for deployment in the real world, the performance of WebALPS-enabled server should scale well under realistic server workloads.

2.1.4 Configurability

Because of the expected performance slowdown, it is not always desirable to use WebALPS co-servers. For Web services that weigh performance far more than security and privacy, for example, a host that acts as a non-interactive, static page provider, employing WebALPS approach may not

provide much advantage. This suggests that WebALPS should be an option that can be turned on and off by site administrators according to their needs.

2.1.5 Ease of Deployment

The less difference there is between our implementation and the currently existing infrastructure, the easier it is to make the world accept WebALPS. Partially for this reason we chose Apache server, the most popular Web server in the world, as the server platform for the project. The ease-of-deployment requirement also motivates the guideline that we should modify the Apache server as little as possible. Presumably, asking the site administrators to install a small patch to their existing servers would be much more acceptable than forcing them to install a whole new server.

2.2 Road Map

The following is the list of major milestones during the development of the project:

- background preparation: Thoroughly understand the SSL protocol, the Apache server SSL implementation, and the IBM 4758 secure coprocessor software interface;
- system design: come up with a detailed diagram illustrating how client, Web server, and WebALPS co-server interact with each other to establish an authenticated, secure communication channel between the client and the WebALPS co-server;
- system implementation:
  - generate a public keypair and the corresponding X.509 [13] certificate for the WebALPS co-server, and integrate this certificate into Apache server. For this prototype implementation, the keypair and the certificate are both temporary. Live deployment of WebALPS

co-servers will require a formal certificate that can be used to verify that the keypair really belongs to the co-server. [24] has a more detailed discussion about this ;

  - port the server-side SSL session key generation code into the IBM 4758 secure coprocessor installed on the server site;
  - implement co-server session management and integrate it with the Apache server session management;
  - choose and design a prototype application.

- performance test and analysis: measure the performance of WebALPS-enabled server under load, and compare with the performance of the same server without WebALPS in order to validate that the implemented system actually works under real server loads.

Chapter 3

Enabling Technology I

— Secure Coprocessor

Secure coprocessing is an essential piece of enabling technology for the WebALPS approach. The secure features that secure coprocessors offer lead to the credibility of WebALPS co-servers. This chapter introduces the general secure coprocessing technology and describes the IBM 4758 secure coprocessor, the foundation of our implementation, in detail. It also discusses the design decisions made based on the features of the IBM 4758 coprocessor.

3.1 What is Secure Coprocessing

In distributed environments such as the Web, it is often very difficult and sometimes even impossible to provide a secure physical environment for sensitive computing occurring at a remote site. Secure coprocessing technology deals with this problem by offering secure, general-purpose computing de-

vices that can withstand all foreseeable physical and logical attacks, and thus can be trusted with sensitive processings in a hostile environment. Such devices are called secure coprocessors. A secure coprocessor must:

- correctly execute the program as it is supposed to;
- be able to prove the identities of itself and the applications running inside it upon the request of a remote user.

In addition, secure coprocessors normally offer cryptographic support for secure communications between coprocessor-resided applications and other distributed entities.

3.2 IBM 4758 Secure Coprocessor

The IBM 4758 PCI Cryptographic Coprocessor is the state of the art in the programmable secure coprocessor industry. Model 023, one of the newest members from the Model 2 family of IBM 4758 products, was used for our co-server implementation. In the rest of this thesis, we will refer to this coprocessor as the "IBM 4758" or the "card" and refer to the machine on which the IBM 4758 is installed as the "host". In the following subsections, first we will take a look at the security properties of the IBM 4758. Then we will describe its hardware and software architecture to see how these properties are achieved.

3.2.1 Certified security properties

As we stated in Chapter 2, the top requirement for WebALPS project is security. Since WebALPS co-servers rely on the IBM 4758 to provide a safe execution environment, for a Web client to trust

the WebALPS co-server's security, he has to know what security properties the IBM 4758 offers and have confidence in these properties.

The IBM 4758 provides [22]:

- safe execution: Even placed in a hostile environment, the IBM 4758 provides a safe haven for the execution of code from genuine, trusted sources. In more detail, if Authority M owns a particular software layer L, the IBM 4758 guarantees two properties for M:
  - control of software: only M, or a superior designated by N, can load code into layer L;
  - access to secrets: only code trusted by M running in the appropriate context can access secrets that belong to Layer L.
- authenticated execution: A remote client is able to authenticate:
  - an untampered IBM 4758 device;
  - the software configuration of this untampered IBM 4758 device.

The assurance of the above security promises can only be established through standard, independent validations. Federal Information Processing Standards (FIPS) 140-1 is a set of testings established by U.S. government for such purpose. FIPS 140-1 offers multiple levels of validation, among which Level 4 is the highest possible standard. For a cryptographic module to pass level 4 validation, its physical security must resist any attack the evaluation lab attempts, and its software documentation must extend to a full formal mathematical model and formal proof of security with that model. IBM 4758 (model 01 or 02) was the first ever and is to date the only general purpose computing device that has earned this level of certification [23]. Therefore, it provides solid ground for the clients' trust that WebALPS is aiming to obtain.

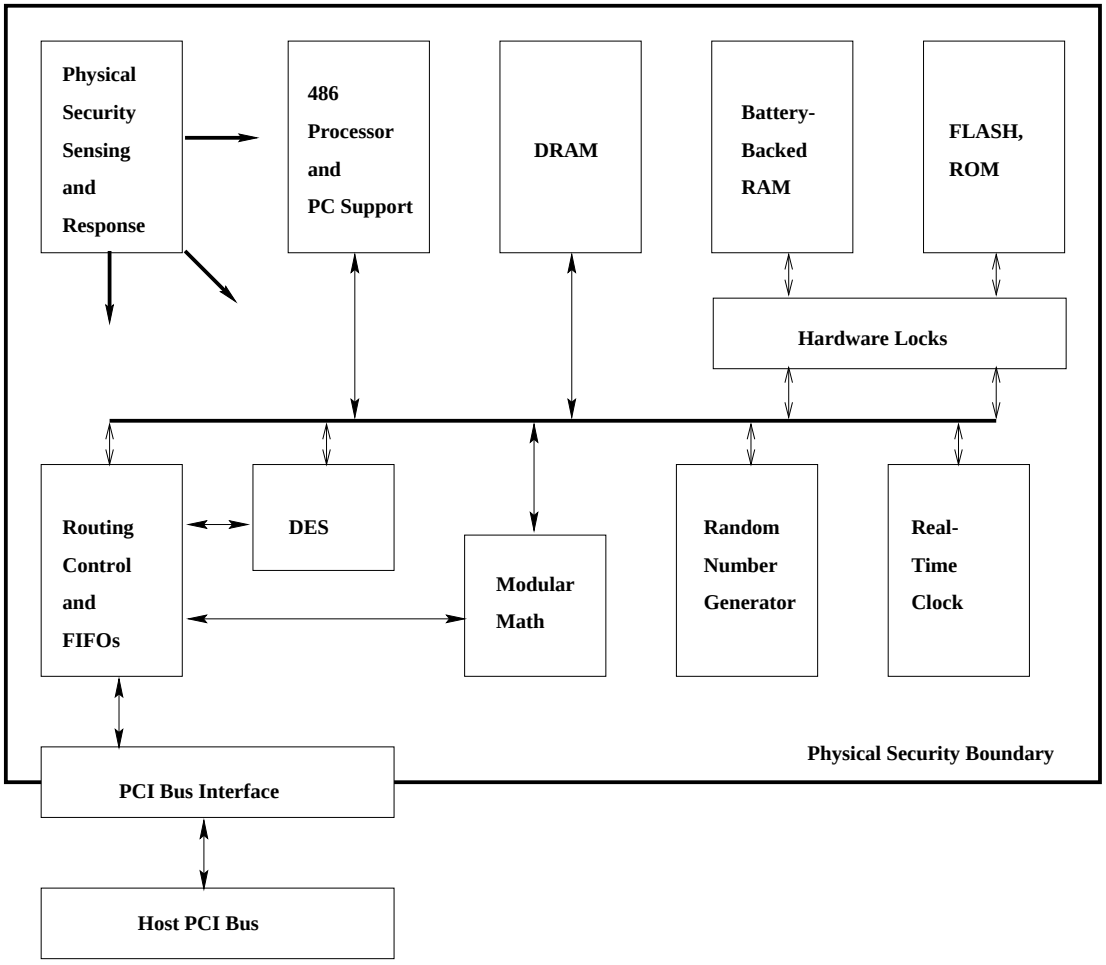

Figure 3.1: Hardware architecture of the IBM 4758 Secure Coprocessor (originally appeared in [10])

3.2.2 Hardware

Physical security of the IBM 4758 is achieved through hardware. Figure 3.1 sketches the hardware structure of the IBM 4758. These hardware components can be functionally divided into three groups [15]:

General-purpose computing Components

- CPU: Intel 486 99MHZ;

- memory:
  - DRAM: 4MB;
  - Flash Memory: persistent storage for bootstrap software, operating system, and applications;
  - BBRAM: 16KB, battery-backed SRAM for storing sensitive data. The content of this memory is set to zero upon tamper detection.
- external interface: PCI bus and serial interface for communicating with the host;
- pipelining hardware: mainly FIFO buffers that are connected to the internal and external DMA channels as well as to the DES engine. These buffers enable fast bulk data movements between
  - the host and the card;
  - two points internal to the card (e.g., from RAM through DES and back);
  - two points external to the card (e.g., bulk DES from host RAM through the card).
- internal Bus

Security components

- security enclosure: two sealed steel enclosures (separated by resin) with electrical circuits embedded to detect physical penetration;
- physical security sensing and response: include tamper detectors and always-active circuit that reposes to physical penetration, high and low temperatures, radiations, and abnormal power sequencing. When any of these abnormal conditions is detected, the following actions will take place:
  - Zeroize BBRAM;

  - Refresh DRAM;
  - Shut down CPU.

- hardware locks: independent circuitry that controls the access to the Flash and to BBRAM by the code executing on the main CPU. This measure protects crucial code and secrets from possibly malicious or faulty application code.

Cryptography support components

- DES engine: provides DES encryption/decryption at high sustained rates;
- modular math engine: supports the computations that are the basis of public cryptographic algorithms such as RSA, Diffie-Hellman, and DSA;
- hardware random number generator: consists of an electronic noise source and a random bit-value accumulator.

3.2.3 Software

The IBM 4758 features a software system [10] that ensures the integrity of its software configuration, and at the same time offers the flexibility of software installation and updates in a hostile environment. This section introduces IBM 4758's multi-layer software architecture, code-loading scheme, and its authentication feature.

Architecture

Figure 3.2 shows the multi-layer software architecture of the IBM 4758:

- miniboot layer: the bootstrap layer that manages security and configuration. It includes:

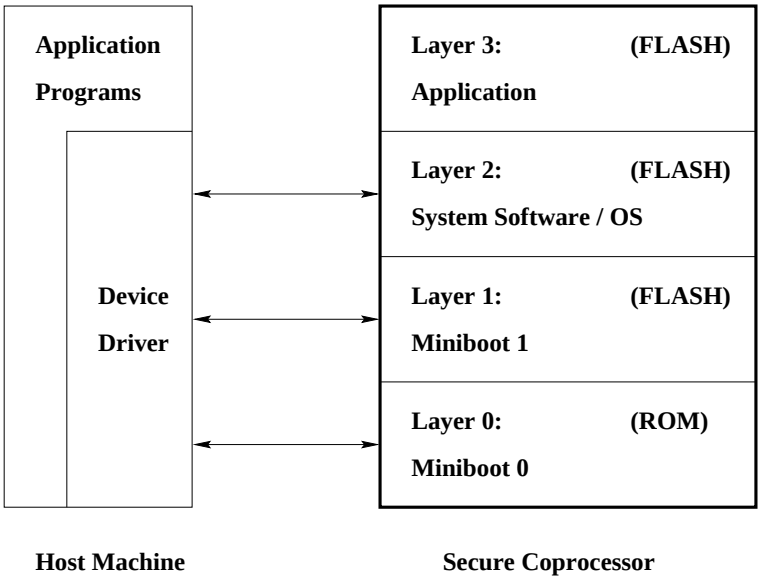

Figure 3.2: Software architecture of the IBM 4758 Secure Coprocessor (originally appeared in [10])

  - Miniboot 0: Permanent portion that supports DES operation and secret-key authentication;
  - Miniboot 1: Rewritable portion that supports public-key cryptography and hashing, and carries out code installation and update tasks.

- OS/control system layer: manages computational, storage, and cryptographic resources;
- application layer: the layer where WebALPS co-servers will reside and provide services to WebALPS clients with the IBM 4758's resources.

Code-loading scheme

A secure code-loading scheme is essential for WebALPS to earn clients' trust. Without such a scheme, a malicious web site operator could buy a secure coprocessor, install his own code to impersonate as WebALPS co-servers in order to steal sensitive information from the client.

The IBM 4758 employs a complicated set of security measures to ensure the integrity of code installation and update. Suppose Alice is the developer of WebALPS application and Bob is the interested customer who operates a Web site, the WebALPS loading works like this (paraphrase from [24]):

- Alice obtains the following things from IBM:
  - a unique identifier;
  - a signed command telling coprocessors that Alice can be the owner of their application layers if there is no current owner;
  - a signed command telling coprocessors that Alice has a specified public key.
- Alice signs the WebALPS code with her private key;
- Bob installs a coprocessor on his server machine and obtains the WebALPS code together with IBM-provided commands from Alice;
- Bob presents the code and commands to the coprocessor for installation. First, the security configuration software running inside the coprocessor validates the commands against the built-in coprocessor's public key and other parameters in a parameter store. If the validation is successful, the coprocessor continues the installation:
  - the parameter store is updated to record Alice's unique ID and her public key, as well as the fact that now Alice owns the application layer of this coprocessor
  - WebALPS is installed to the application layer;
  - a key pair is generated for WebALPS installation on this coprocessor;
  - a certificate is generated using the coprocessor's built-in key pair to assert that the newly generated key pair belongs to WebALPS written by Alice running inside this coprocessor;

– the key pair and certificate are stored in the BBRAM and FLASH segments that is accessible by WebALPS during run time.

With such a scheme, if Bob is not a valid coprocessor application developer, even if he has obtained the commands from Alice, he cannot use his own code to impersonate as WebALPS because he cannot provide Alice's signature. Even if Bob has obtained his own set of commands from IBM so that he is able to install code into the coprocessor, his code still cannot fool clients as WebALPS written by Alice because the certificate for his code includes his identity which can be verified by WebALPS clients during run time.

Outbound authentication

As mentioned above, a WebALPS co-server can authenticate its identity during run time to remote clients – a feature called *Outbound authentication* [25]. Upon request, a WebALPS co-server provides a certificate chain to the remote client. As Figure 3.3 shows, this certificate chain starts from WebALPS's own certificate generated by the OS-layer software (IBM CP/Q++) inside the coprocessor, followed by the IBM CP/Q++'s certificate generated by this coprocessor, this coprocessor's certificate, and ends with the IBM class root certificate [14]. The co-processor's certificate is generated and signed by the private half of IBM's class root key pair before shipment. Therefore, as long as the client has the public key of the certificate authority who signs the IBM class root certificate, he will be able to verify the identity of WebALPS co-server as well as the software configuration on this card through this certificate chain.

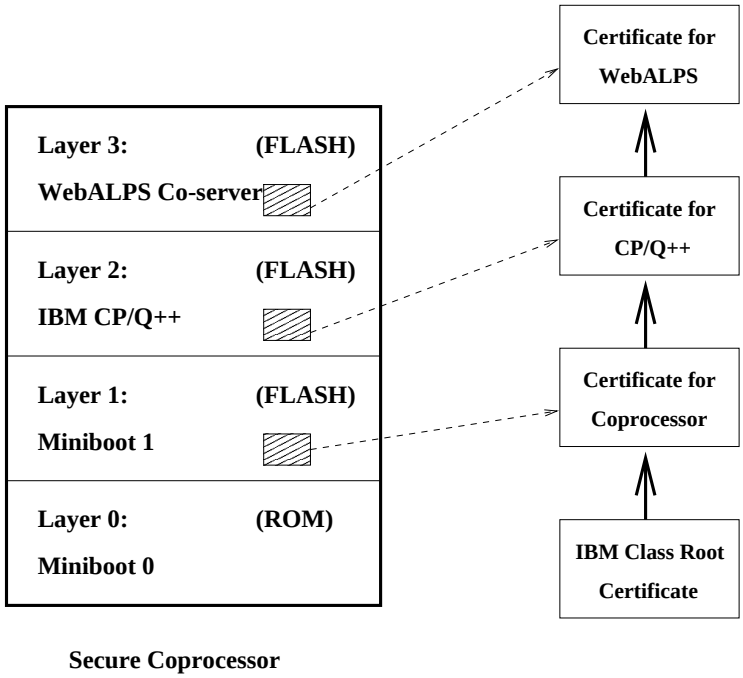

Figure 3.3: Certificate chain used during outbound authentication

3.3 Design considerations

Reviewing the hardware and software architecture of the IBM 4758 not only helps to understand why this device lays the foundation of trust that WebALPS needs, it also facilitates the decision-making in designing the WebALPS system:

- the functional boundary between the host and the coprocessor: the IBM 4758 offers general computing ability. But it does not offer network support. Nor does it provide a standard operating system. These limitations make it impossible to port an entire web server inside the card. Furthermore, the card's limited CPU speed and memory size could become a bottleneck for performance. For this reason and also for the reason that we want to keep the trusted code base in our system as small as possible, we decided to keep the computation that the WebALPS co-server carries in the coprocessor as simple as possible. In particular, the Web

server running on a powerful host will be responsible for TCP connections, book-keeping and other non-sensitive functions of a Web server, while the co-server handles the establishment of a secure communication channel with the client and performs computations that involve the sensitive data from the clients;

- choice of cipher: To establish a secure, authenticated channel, the client and the co-server first have to agree on a choice of the ciphers that will be used to encrypt the application data. Because the IBM 4758 offers a fast DES engine, to boost the performance of WebALPS system, we decided to use DES for this choice.

Chapter 4

Enabling Technology II

— SSL

The goal of WebALPS approach is to provide secure, trustworthy Web services to Web clients. To achieve this goal, WebALPS co-servers have to communicate with clients through secure and authenticated channels. The establishment of such channels require secure network protocols. We chose SSL for this purpose because it is the most widely accepted and deployed secure communication protocol among all the available ones. SSL is supported by nearly all the major commercial browsers, including Netscape Navigator and Microsoft Internet Explorer, and Web servers, including the ones from Netscape, Microsoft, and IBM.

Section 4.1 gives a brief introduction to SSL. Section 4.2 sketches our design of WebALPS-enabled SSL protocol.

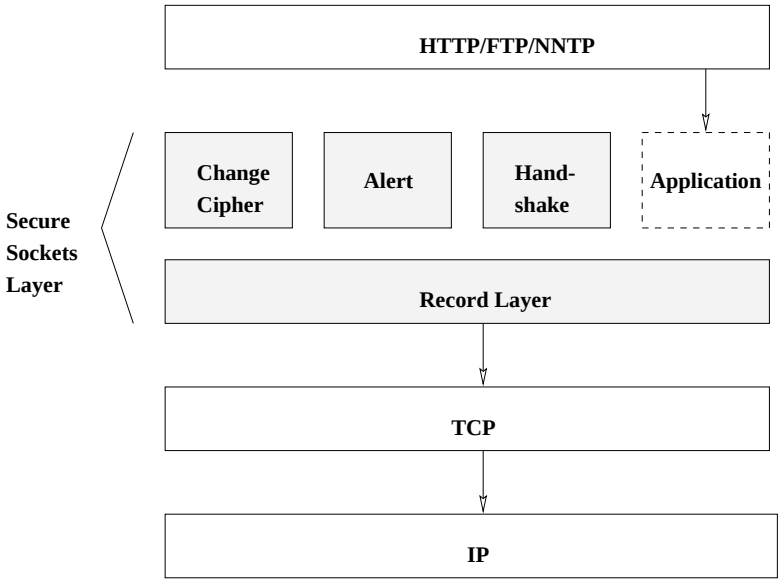

Figure 4.1: SSL protocol components and Internet protocol stack (based on [27])

4.1 How SSL works

Section 4.1.1 gives an overview of the design and components of SSL protocol. The next two sections describe the record layer protocol and the handshake protocol, the two protocols that are directly related with WebALPS implementation, in more detail.

4.1.1 Structure Overview

The designers of SSL decided to add a new protocol layer in the internet protocol stack for security services. This approach has several advantages [27]:

- it requires very few changes to the existing protocols;
- it gives SSL the ability to support multiple applications.

Figure 4.1 illustrates the internet protocol stack after the introduction of the SSL protocol. There are four SSL component protocols on two different layers:

- the upper layer consists of three protocols:
  - Handshake protocol: negotiate the set of ciphers that will be used for secure communication between the server and the client;
  - ChangeCipherSpec protocol: instruct the other party to use the ciphers negotiated through the Handshake protocol;
  - Alert protocol: signal the other party an error or caution condition.
- the lower layer has a single protocol:
  - Record Layer protocol: format and frame the messages from the upper-layer protocols, and passes them to TCP layer for transmission.

In addition to the three SSL component protocols, upper layer also includes application protocols, such as Hyper Text Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Net News Transfer Protocol (NNTP), that use the security service provided SSL.

4.1.2 Record Layer Protocol

All SSL messages are encapsulated through the Record Layer protocol. In addition to message framing, typing and fragmentation, it also provides encryption, Message Authentication Code (MAC) generation, as well as message verification. As Figure 4.2 shows, This protocol defines a universal format to frame message from Handshake, ChangeCipherSpec, Alert, and application protocols. A record layer protocol message contains two parts:

- header: including protocol, version, length;

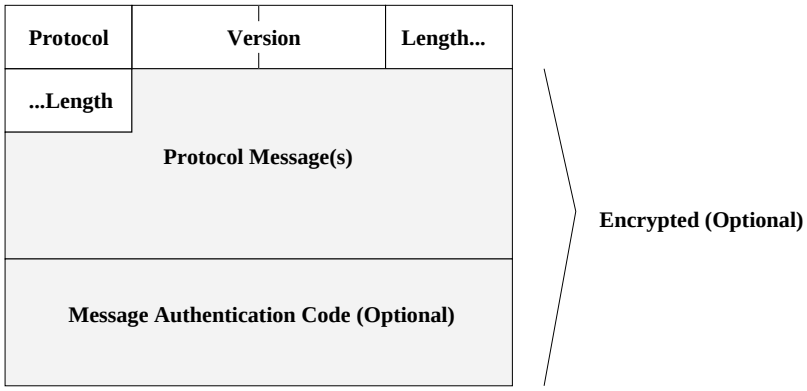

Figure 4.2: Record Layer protocol message format (originally appeared in [27])

- message body: including messages from the upper layer protocol. If a security service has been negotiated, the message body will also include the Message Authentication Code (MAC) of those messages, and the whole message body will be encrypted using the agreed cipher.

4.1.3 Handshake Protocol

The Handshake protocol is of particular importance to WebALPS project because this protocol is responsible for establishing a set of shared secrets between the client and the server, and for authenticating the server to the client. Both of these processes will be changed with the introduction of trusted co-servers. In this section we will describe the Handshake protocol in detail.

Figure 4.3 illustrates the whole handshake process:

- the client sends *ClientHello* message which contains *Client Random*, a 32-byte random number to seed cryptographic calculations later for key materials, and the cipher suites that the client prefers to use (each cipher suite contains a key-exchange algorithm, an encryption algorithm, and a hash algorithm for MAC generation);

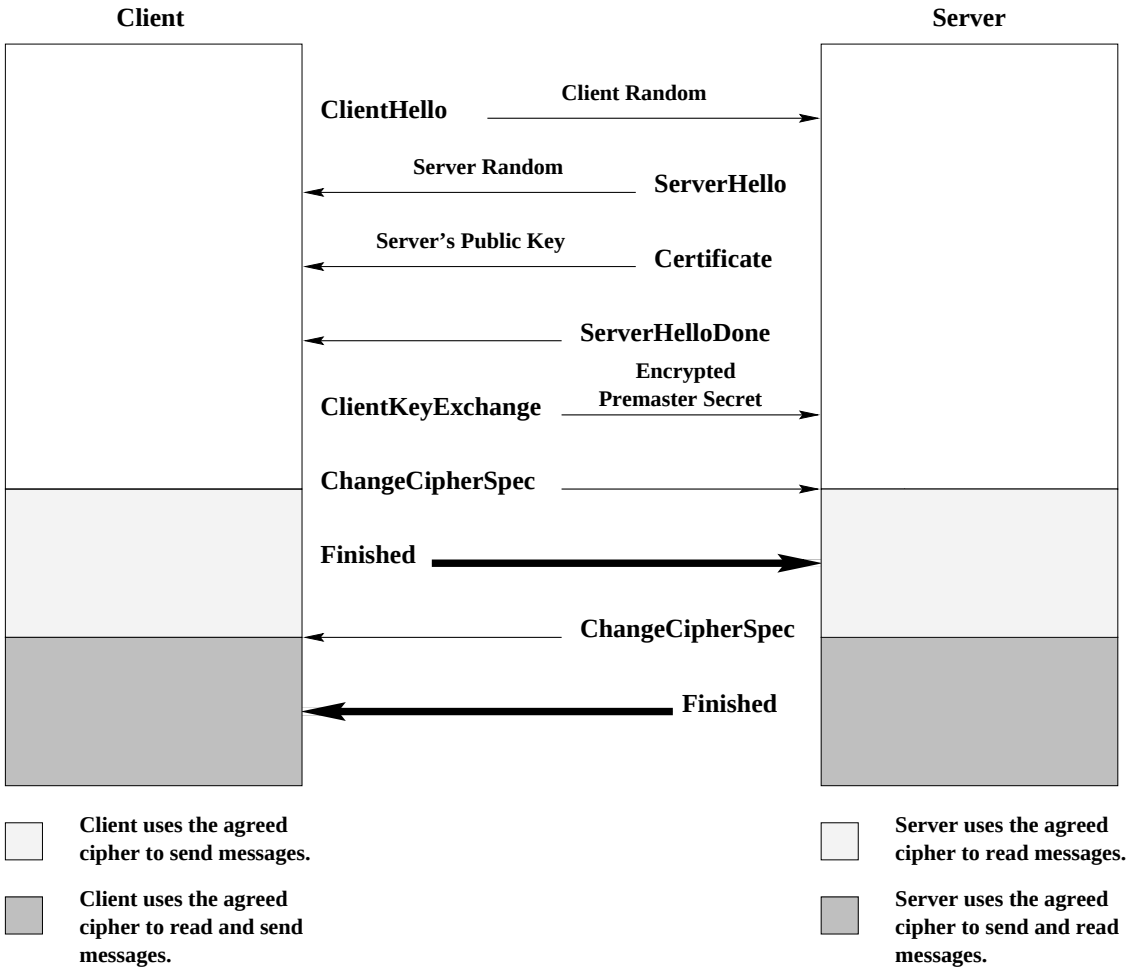

Figure 4.3: SSL Handshake protocol

- the server responds with *ServerHello* message which contains *ServerRandom*, a 32-byte random number that functions the same as *ClientRandom*, and the cipher suite that server chooses to use from the client's propositions;
- the server sends its public key certificate within the *Certificate* message;
- the server concludes this part of handshake with *ServerHelloDone* message;
- after the client receives and verifies the server's certificate, he will use the server's pubic key to encrypt another random number, known as *Premaster Secret*, and send it within the *ClientKeyExchange* message;
- the client calculates the shared secrets for sending data based on *ClientRandom*, *ServerRandom*, and *Premaster Secret* and sends a *ChangeCipherSpec* message to activate the negotiated ciphers for all future messages it will send;
- the client sends a *Finished* message to let the server check the newly activated options. This message contains MAC for all the previous handshake messages and is encrypted;
- after the server receives the *ChangeCipherSpec* message, it calculates the shared secrets for reading data (the same as the secrets for the client to send data), and verifies the *Finished* message from the client;
- the server now calculates the set of secrets for sending data and send a *ChangeCipherSpec* message to activate the negotiated ciphers for all future messages it will send;
- the server sends a *Finished* message which is the first message sent from the server that uses the negotiated ciphers.

At this point, the client and the server have negotiated a cipher suite and exchanged the shared keys for data encryption and MAC generation. Because the key calculation is based on the premaster secret, which is encrypted using the server's public key during transmission, it is assured that no

third party can steal the shared secrets as long as the server keeps its private key secure. In addition, the client has authenticated the server's identity with the server's certificate. Therefore, an authenticated, secure communication channel is established between the client and the server.

4.1.4 Reuse a Previous Session

An SSL *session* refers to the state in which the client and the server have negotiated a cipher suite and exchanged shared secret keys for communication. Each session is identified by a *Session ID*, which is a random 32-byte number generated by the server. As demonstrated in the previous section, using the Handshake protocol to establish a new SSL session is a complicated process that involves a significant number of protocol messages as well as time-consuming cryptographic computations. To reduce this overhead, SSL protocol offers the option of reusing a previous established SSL session between the client and the server.

Figure 4.4 shows this process:

- the client proposes to reuse a previous session by including the *Session ID* of that previous session in the *ClientHello* message that is sent to the server;
- the server decides to grant the client's session-reuse request. So it includes the same *SessionID* in its *ServerHello* message and send the message back to the client;
- then the server sends the *ChangeCipherSpec* message to activate the agreed cipher suite in the previous session for all future messages it will send;
- the server sends the *finished* message to conclude its part;
- the client verifies the server's *Finished* message and sends his *ChangeCipherSpec* message to activate the agreed cipher suite in the previous session for all future messages he will send;

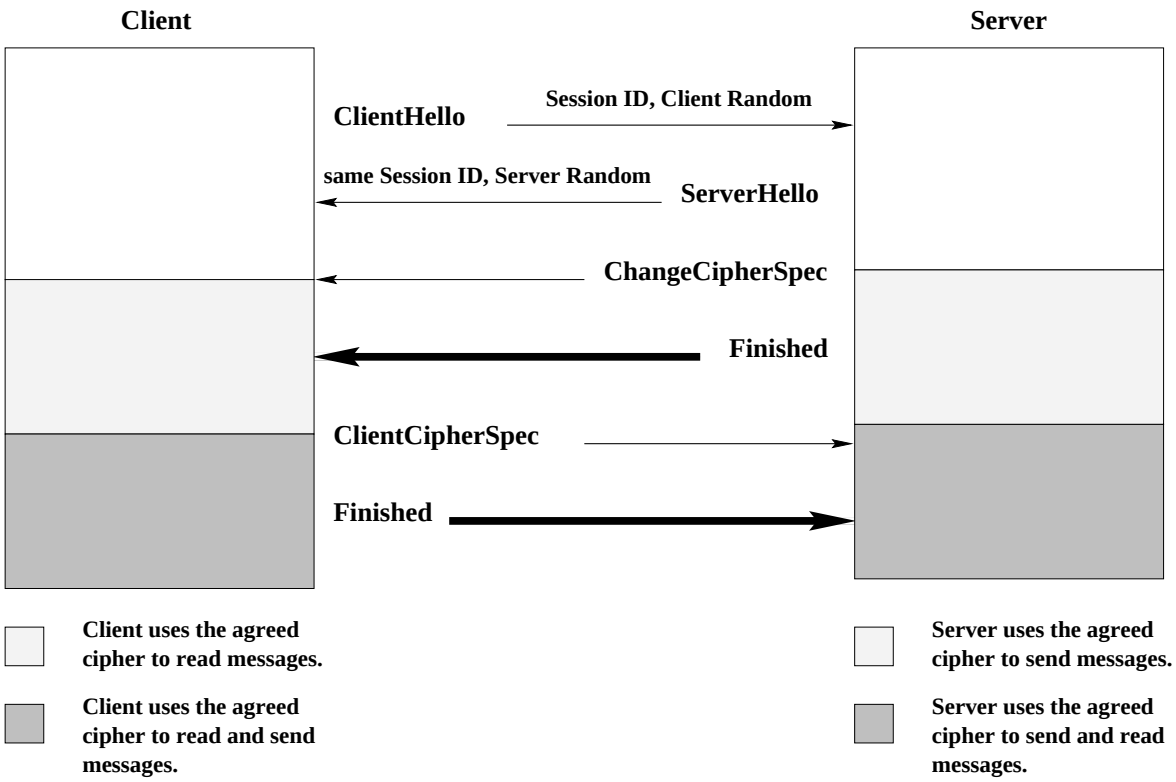

Figure 4.4: SSL protocol: reuse a previous session

- the client concludes his part by sending the *Finished* message.

This process involves only six protocol messages and reduces the cryptographic calculation for secret generation.

One thing worth noting is that the server can choose to decline the client's session-reuse request by putting a new session ID inside the *ServerHello* message. If this happens, the SSL session establishment falls back to the normal Handshake protocol.

4.2 How WebALPS-enabled SSL works

With WebALPS approach, the previous two-party SSL protocol now has to accommodate three parties. Besides, instead of servers, WebALPS co-servers become the trusted entities that will share secrets with the clients. Therefore, components of SSL protocol has to be modified accordingly. In particular, the Handshake protocol and the Record layer protocol will be changed. Section 4.2.2 and Section 4.2.4 describe these changes. But before we talk about our revised protocol, we will first take a look at some design choices we have to make regarding the variety of options SSL offers.

4.2.1 Options with SSL

In addition to the protocol described in the previous section, SSL offers a couple of variations. One of such variation is that there are three different modes for authentication: no authentication, server authentication, and mutual authentication. For WebALPS, no authentication is not an acceptable choice. Client authentication is not common in real-world applications because there isn't yet a widely-deployed PKI system for authenticating clients. Normally, a client will use other means to authenticate himself (through credit card number and billing address, for example). Therefore, for

the sake of simplicity, we chose to only implement the common case which requires server authentication.

Another option is which version of SSL to support in our project. The options are SSLv2.0, SSLv3.0, and Transport Layer Security(TLS)v1.0. TLS is the new name for SSL protocol after its design was taken over by IETF. Again, we decide to support the most common case — SSLv3.0.

4.2.2 Handshake with WebALPS co-server

Intuitively, the handshake process with the participation of the WebALPS co-server occurs as illustrated in Figure 4.5. Interactions between co-servers and servers are added to accomplish two goals:

- authentication of co-server to the client: The *certificate* message from the server to the client now contains the certificate for WebALPS co-server generated by the secure coprocessor;
- shared-secret establishment between the co-server and the client:
  - the server forwards *Client Random* and Server Random to the co-server since these numbers will be used in the secret generation process;
  - the client uses the co-server's public key to encrypt the *Premaster Secret* in the *ClientKeyExchange* message. Consequentially, *Premaster Secret* is known to the co-server but not to the server;
  - the key calculation process takes place inside the co-server;
  - the encrypted *Finished* message from the client becomes incomprehensible to the server. The server has to send this message to the co-server for decryption and MAC verification;
  - before the server sends the *Finished* message, it has to forward it to the co-server for

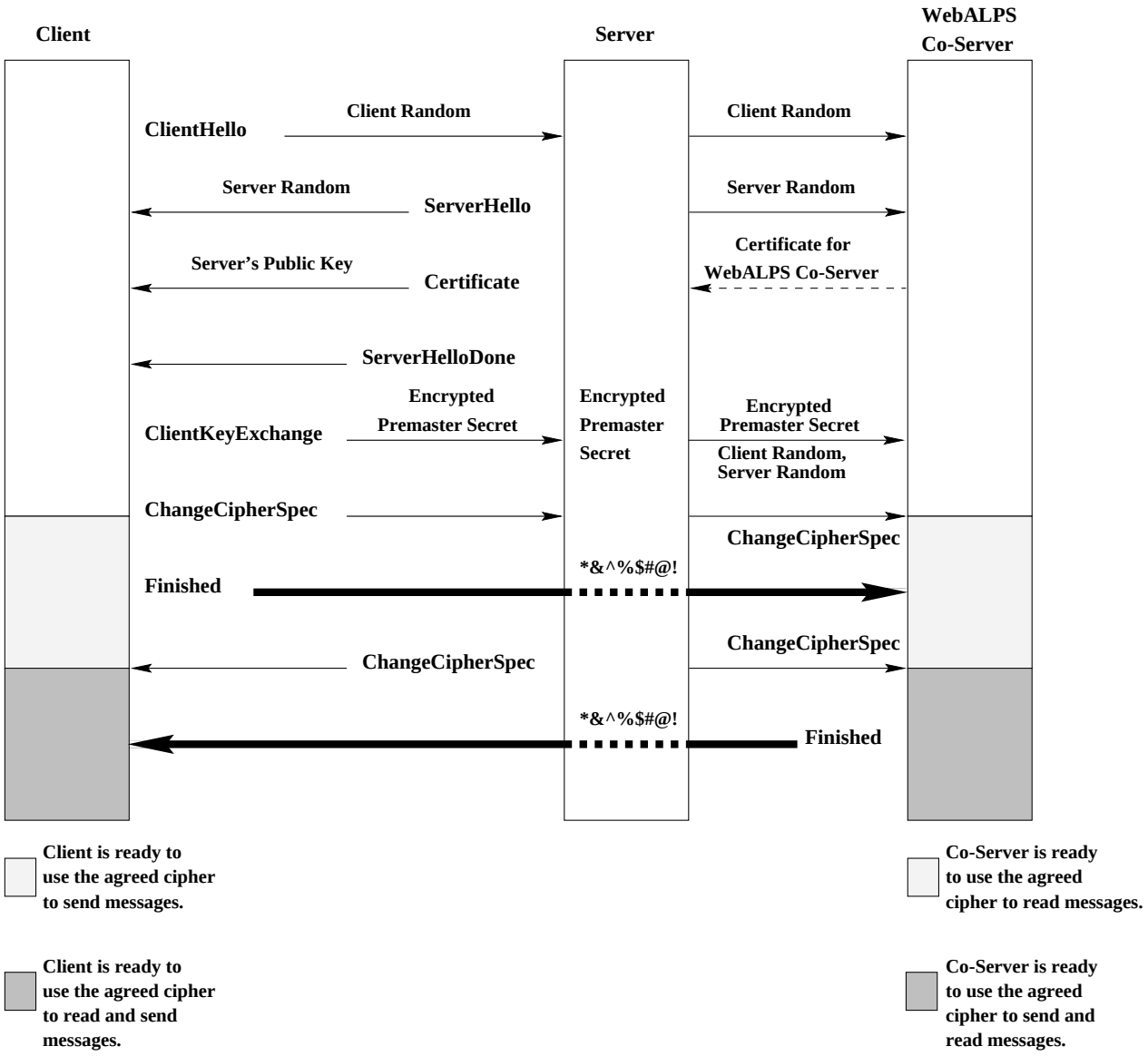

Figure 4.5: SSL Handshake process with WebALPS co-server (intuition). In this figure and some of the following figures, strings that contain random characters such as "*&^%$#@!" are used to denote illegible ciphertext and bold arrows are used to denote an established secure and authenticated communication channel.

MAC generation and encryption. Otherwise, the client will not be able to verify this message, and thus will not send any sensitive information to the server-side.

This process adds 8 messages to the previous SSL protocol. When there are vast numbers of handshake requests, the message passing between the server and co-server could become a performance bottleneck. Unlike the messages exchanged between the server and the remote client, the messages between the server and the co-server do not need to carry any information about cipher negotiation. Therefore, we can reduce the message-passing overhead by concatenating several messages together. Figure 4.6 shows the revised diagram of the handshake process that we used in our implementation. The following changes are made:

- elimination of *Certificate* message from the co-server to the server: the certificate for the WebALPS co-server is stored in the server file system and is loaded into the server during initialization. Therefore, the co-server does not need to send it again and again during handshake;
- elimination of *Client Random* and *Server Random* messages from the server to the co-server: *Client Random* and *Server Random* are piggybacked into the message that contains *Encrypted Premaster Secret*;
- elimination of *ChangeCipherSpec* message from the server to the co-server: once the co-server receives *Client Random*, *Server Random* and *Encrypted Premaster Secret*, it will calculate the shared secret for both reading and writing immediately.

With these improvements, the communication overhead between the server and the co-server are reduced to 4 messages per handshake.

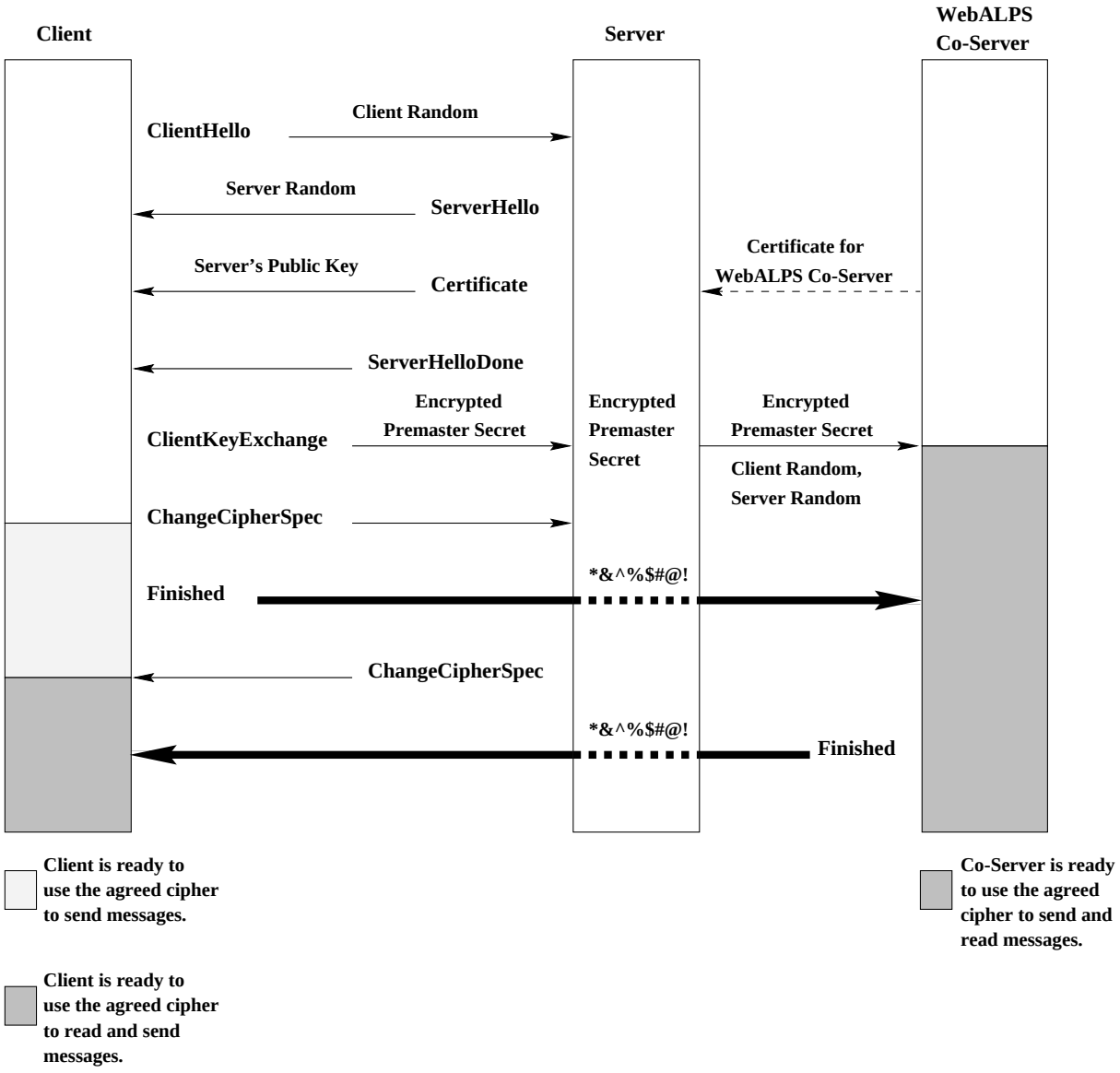

Figure 4.6: SSL Handshake process with WebALPS co-server (real)

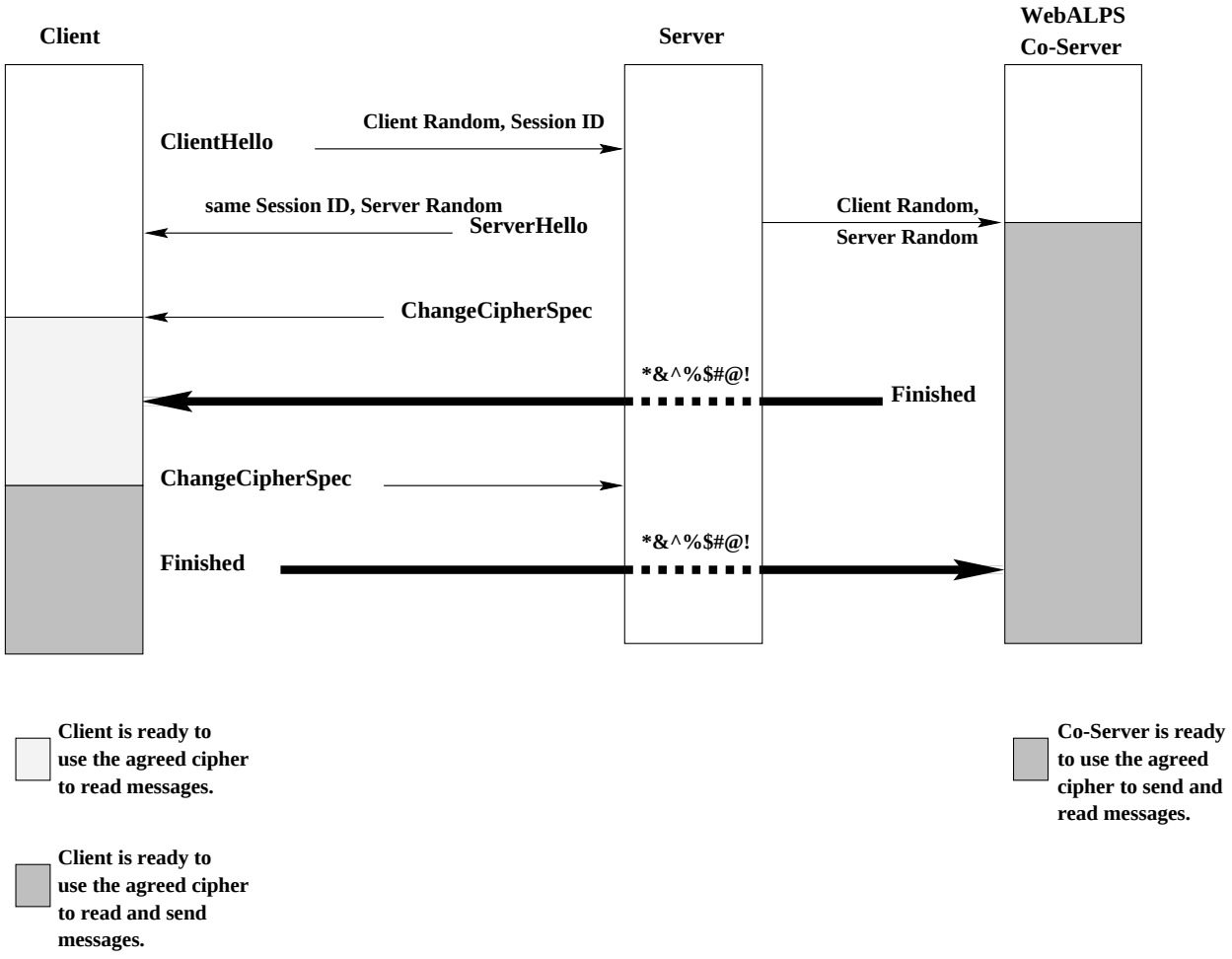

Figure 4.7: Session reuse with WebALPS co-server

4.2.3 Session reuse with WebALPS co-server

Figure 4.7 shows how reusing a previous session is done with the participation of WebALPS co-server:

- after the server receives the client's *ClientHello* message and agrees to reuse a previous session, it sends the new *Client Random* and Server Random to the co-server;
- the co-server will calculate the new set of secret keys using the new *Client Random*, the new *Server Random*, and the old *master secret* (an intermediate result during key generation) from the previous session;
- the server sends the *ChangeCipherSpec* message to the client;
- the server asks the co-server to generate the MAC for and encrypt the *Finished* message;
- the client sends the *ChangeCipherSpec* message to the server;
- the client sends the *Finished* message to the server and the server asks the co-server to decrypt and verify this message.

4.2.4 Record Layer protocol with WebALPS co-server

Now that the shared keys for encryption and MAC are in the co-server, the record layer protocol has to call for the co-server to carry on security services.

In our revised protocol, the client-side remains unchanged. But on the server-side, the record layer uses the server only for framing, typing and fragmenting the application layer messages, and uses the co-server for data encryption/decryption and MAC generation/verification. Figure 4.8 illustrates this process. It shows how the server's record layer uses the security services provided in the WebALPS co-server in the process of retrieving a HTTP request from the client and sending the HTTP response back in fragments:

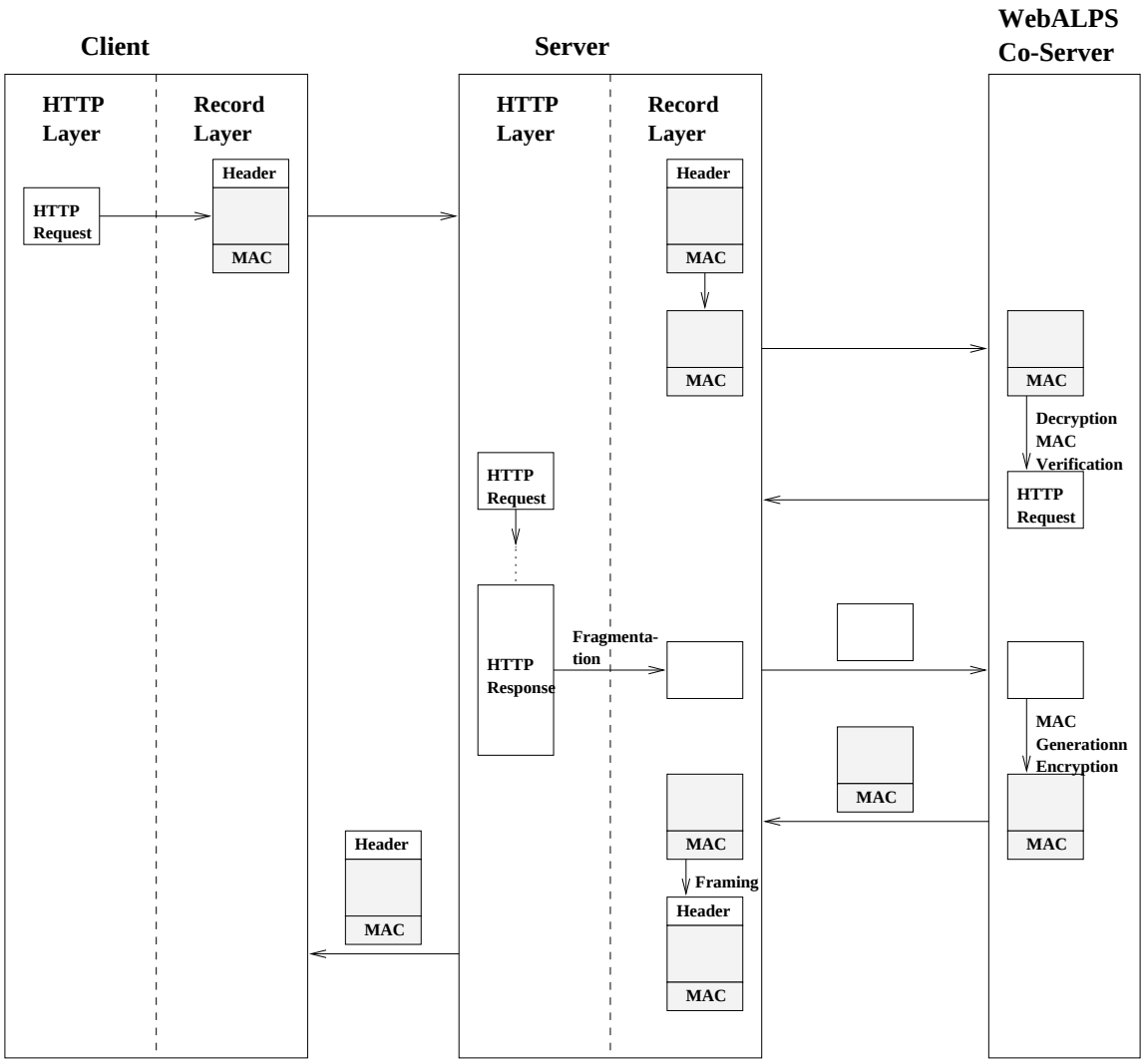

Figure 4.8: Record Layer protocol with WebALPS co-server

- after the server's record layer receives the encapsulated HTTP request message from the client, it deframes the message by taking off the header and then sends the rest of the message to the WebALPS co-server;
- the co-server's record layer decrypts the message, verifies the MAC, and sends the plaintext HTTP request back to the server. The co-server's application layer (not drawn in this diagram) can process this request before sending it back to the server;
- the server's HTTP layer processes the request and generates an HTTP response;
- the server's record layer cuts the large HTTP response into several fragments and sends them to the WebALPS co-server;
- the co-server's record layer generates MAC for each segment, encrypts it, and sends the encapsulated fragment back. If there is an application layer in the co-server, it can process the HTTP response message before the cryptographic processings take place;
- the server's record layer adds a header for each fragment and sends it to the client.

Chapter 5

Apache, Mod_SSL, and OpenSSL

As described in Chapter 3, because of the secure coprocessor's system restrictions, WebALPS co-servers are designed to only provide necessary functionalities to ensure the security of sensitive services. Therefore, to be fully functional, a WebALPS co-server has to be supported by a Web server, which deals with TCP connection, logging, forwarding requests from clients into the co-server, and other non-sensitive tasks. When we started implementation, the very first decision to make was on which server platform we should build WebALPS co-server. We decided to choose Apache HTTP server [4] for this purpose because:

- according to Netcraft's most recent survey, it is the most popular web server in the world with 62.55% market share [18]. Since the goal of the project is to integrate WebALPS approach with an industrial-strength server, Apache seems the best fit;
- it is open source, which facilitates our implementation.

This chapter starts with an introduction to Apache. Then it describes mod_ssl [19], the Apache module that supports SSL, and OpenSSL [8], a crypto toolkit that mod_ssl relies on for its SSL implementation. This chapters finishes with the description of the major data structures used by Apache, mod_ssl, and OpenSSL and how these data structures interact with each other to enable the configurable SSL support in Apache server.

5.1 Apache

As an HTTP server, the primary task of Apache server is to listen to TCP connections for incoming HTTP requests and serve these requests. In addition, Apache server supports functionalities such as Common Gateway Interface (CGI), HTTP authentication, access check, through an extensible, modular structure. Apache also allows users to host multiple web sites in a single server by setting up multiple virtual hosts. Section 5.1.1 describes the basic model that Apache employs to serve HTTP requests. Section 5.1.2 gives an outline of Apache's modular structure. Section 5.1.3 talks about the support for virtual hosts in Apache.

5.1.1 Life Cycle of Apache Server

On Unix systems, Apache server employs a pre-forking model to serve multiple requests from different connections at the same time. The handling of each request is divided into a set of phases. Figure 5.1 illustrates this model and the life cycle of Apache server:

- server startup and configuration: after the server starts, it parses the command-line arguments, initializes resources such as memory pool, processes the directives in the configuration files, and open all the configured log files;

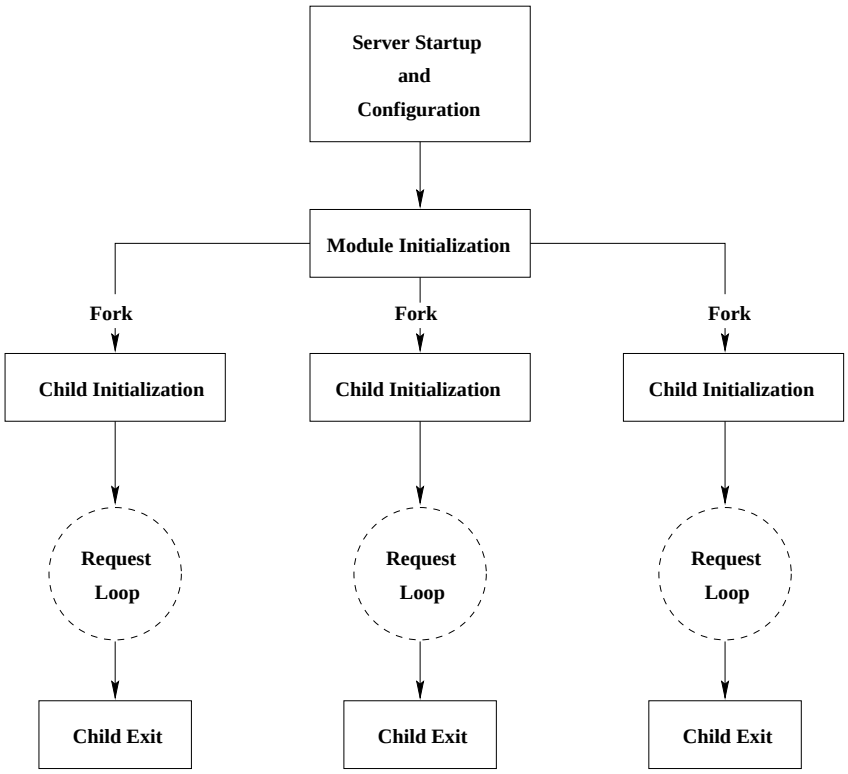

Figure 5.1: Apache server life cycle (originally appeared in [26])

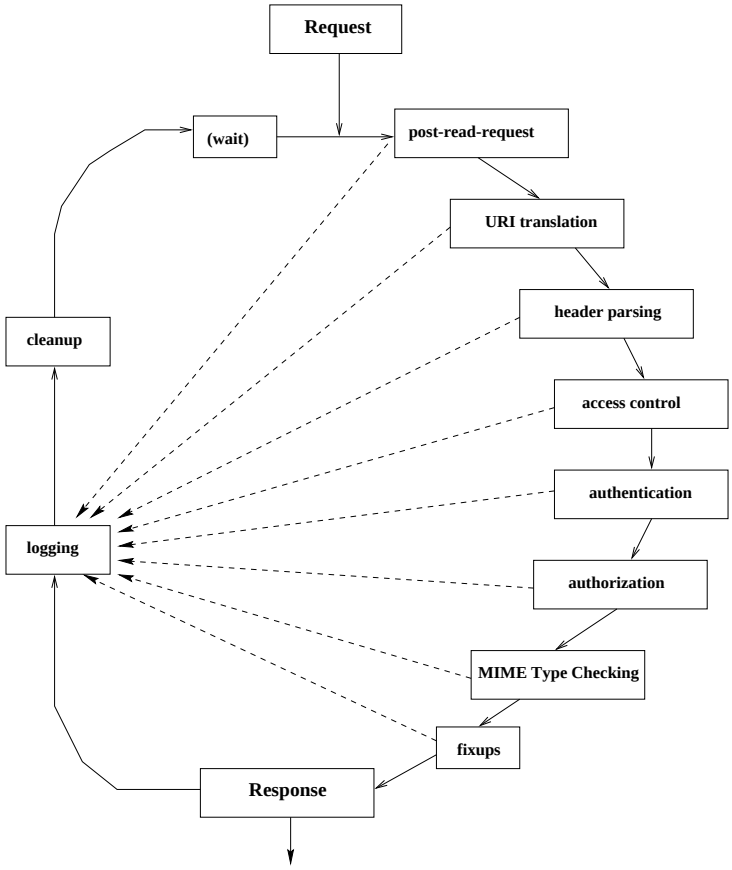

Figure 5.2: The Apache request loop (originally appeared in [26])

- Module initialization: initialize all the modules that are loaded into the server (modules are described in the next section);
- pre-forking and child initialization: the server spawns off several copies of itself to run as unprivileged processes. The parent process will stick around to monitor the status of its children and will fork off more children if the incoming requests become overwhelming;
- request loop: each child will do the actual work of accepting and processing incoming requests. Each request goes through a set of fixed phases that form a loop (Figure 5.2). Through each phase, a decision is made about the current request. For example, the *URL translation phase* decides which document/CGI script is being requested and the *Response* phase decides how to generate the response;
- child exit: When a child has served a maximal number of requests (set in the configuration file) or the entire server receives a shutdown or restart signal, the child exits and thus completes its life cycle.

5.1.2 Modular Structure

Apache offers great flexibility and extensibility through its modular structure:

- users can add/delete functionalities of their Apache servers to meet the requirements for their individual web sites by adding/deleting modules;
- users can add a previously non-existing functionality to his Apache server by writing his own module.

Figure 5.3 sketches the modular structure of Apache server. Conceptually, an Apache server consists of a *Core* which offers the minimum set of functionalities for a Web server, and a set of standard

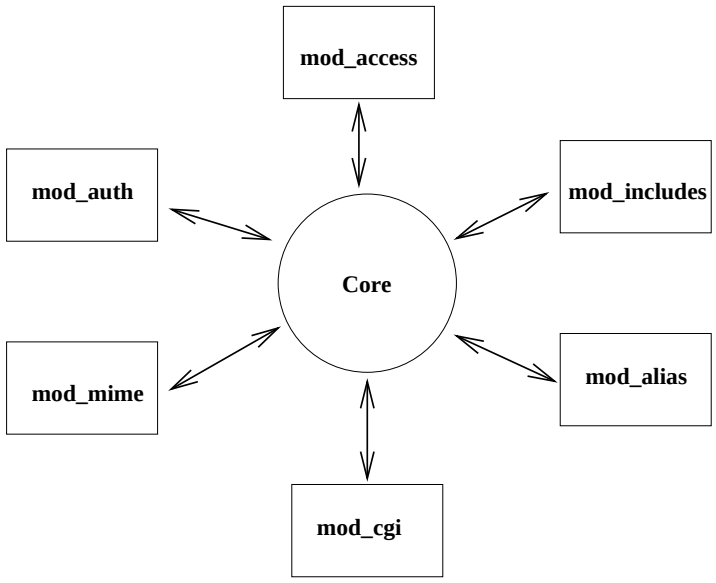

Figure 5.3: Apache server modular structure

modules that interact with the *core* and are responsible for other important functions including access control (mod_access), authentication (mod_auth), MIME type checking (mod_mime), CGI script support (mod_cgi), Server-Side Includes (SSI) support (mod_includes), and file name looking-up (mod_alias).

As described in the previous section, Apache breaks down the request loop into several phases. Modules participates in a particular phase during the request handling by registering a *handler*, a small function with standard signature, for this phase. Each phase can have multiple handlers from multiple modules registered. When this phase is reached during request handling, the Apache core will pass the control to these handlers by calling them one by one, in a sequence determined by module loading sequence at server startup time. If there are no module handlers that are registered to handle this phase, a default handler provided by Apache Core will be called. The results of request processing by each handler are collected through a data structure called *request_rec*, which contains the information of the current request, current connection and current server.

5.1.3 Support for Multiple Virtual Hosts

Very often people want to host more than one web site. To use a separate server for each site would be highly inefficient. Apache supports the mechanisms of using virtual hosts to address this problem. Each virtual host acts like a separate server – it can have its own set of configurations including document root location, server name, error log. One common exercise for Apache servers providing E-commerce services is to configure a virtual host to listen to port 443, from which comes the SSL communication, and to serve HTTPS requests.

As stated in Chapter 2, we want to make WebALPS configurable – it is an option that can be turned on or off for a server. To be able to show this and also to facilitate the performance testing, we decide to set up our Apache server to include:

- a normal HTTPS virtual host that employs the original SSL protocol;
- a WebALPS-enabled virtual host that uses our revised SSL protocol as described in Chapter 4.

5.2 Mod_SSL

Among all the modules, the one that concerns us the most is mod_ssl, the module that enables Apache server to provide SSL support.

As all the other modules, mod_ssl accomplishes its functionalities by registering handlers for particular phases. Table 5.1 sketches those phases and the actions taken by mod_ssl handlers. Two phases, *new connection* and *close connection*, are not from the request loop (Figure 5.2). They are related to connections, which have longer life time than requests – multiple requests can come through one connection. mod_ssl has to register handlers for these phases to deal with the establishment and

Phases	Actions
post-read-request	Link the data structure about this request to the data structure that contains information about the current SSL connection
URI translation	Log information about incoming HTTPS requests
access control	Check if the settings of current SSL connection satisfy the requirement set by the directory being accessed, and either declines the access or forces an SSL renegotiation if not;
authentication	Use the name in the client's certificate to authenticate the client;
authorization	Deny the access if the directory is marked as *StrictRequire* and SSL is not enabled for the current directory
response	Return an error message if a plain HTTP request is received on an SSL-enabled server port
new connection	Establish an SSL session through handshake protocol
close connection	Terminate an SSL connection by flushing the communication buffer, sending alert to the other party, and deallocating data structures for this SSL connection

Table 5.1: mod_ssl registered handler functions

termination of SSL communication channels.

In addition to providing handler functions to carry on SSL-related tasks during the life time of a connection, mod_ssl is also responsible for processing SSL-related configuration directives that a user defines in his server configuration file. In a way, mod_ssl works like an interface between Apache and OpenSSL. It understands what the Apache server is configured to do, and relies on the crypto library and SSL protocol implementations provided by OpenSSL to do the real work.

5.3 OpenSSL

OpenSSL is an open-source cryptography toolkit that implements the SSL and TLS protocols, as well as the related cryptography standards required by these protocols. Functionally, this toolkit can be divided into three parts:

- SSL library: implements SSLv2.0, SSLv3.0, and TLSv1.0 protocols with the following major APIs:
  - *SSL_library_init*: initialize the library;
  - *SSL_CTX_new*: create a data structure **SSL_CTX** as a framework to establish TLS/SSL connections. This data structure accommodates options regarding certificates, crypto algorithm choices, etc.;
  - *SSL_new*: after a TCP connection is established, a data structure **SSL** is created based on the previous **SSL_CTX** structure and is bound to that connection;
  - *SSL_accept*/*SSL_connect*: performs the handshake process for server/client;
  - *SSL_read*/*SSL_write*: after the handshake, use the agreed cipher suite to read/write to the SSL/TLS connection;
  - *SSL_shutdown*: terminates the secure communication, send alert messages about the shutdown of connection, and clean up data structures related to this connection.
- Crypto library: supports following cryptographic functions:
  - symmetric ciphers: DES, IDEA, RC2, RC4, Blowfish;
  - public key cryptography: RSA, DSA, Diffie-Hellman (DH);
  - MAC and hash functions: MD2, MD4, MD5, HMAC, SHA;
  - certificates: X.509.
- OpenSSL: a command-line tool that supports the following functions:
  - encryption and Decryption with symmetric ciphers;
  - calculation of message digests;
  - creation of RSA, DH and DSA key parameters;
  - creation of X.509 certificates, certificate requests, and Certificate Revocation Lists (CRLs).

Because OpenSSL implements the SSL Handshake protocol and Record Layer protocol, the Web-ALPS implementation interacts closely with OpenSSL code.

5.4 How Apache, mod_ssl, OpenSSL Interact Through Data Structures

Three different layers of code, Apache, mod_ssl, and OpenSSl, work together to provide the support for configurable SSL communications in the Apache server. It is important for us to understand the implementation details about how these three parties work, because our own implementation will be based on this knowledge.

5.4.1 Important Data Structures and Their Roles

For Apache [26]

- **ap_global_ctx**: the global context that stores the global configurations of the server that come from each module;
- **server_rec**: per-server data structure to store configuration of each virtual host including the server name, port, log file name, etc.;
- **conn_rec**: per-connection data structure to store the connection-related information including the local and remote sockets for the communicating parties, the IP of the remote client, the user name of the remote client, if the connection should be kept alive, etc.;
- **request_rec**: per-request data structure to store request-related information including the request protocol, the request method, the requested file name, among many other things.

For mod_ssl

- **SSLModConfigRec**: stores the global configuration defined by mod_ssl for each httpd process;
- **SSLSrvConfigRec**: stores the per-server configuration defined by mod_ssl for each virtual host.

For OpenSSL

- **SSL_CTX**: stores the information that will be used to create new SSL/TLS connections including the list of supported cipher suites, the server's certificate (when used with servers), the SSL session cache, among many other things. When used with Apache, the contents of this data structure will be filled in according to the configuration of each HTTPS virtual host;
- **SSL**: stores information about each SSL connection including the negotiated cipher suited and established secret keys. During initialization, the contents of this data structure is filled in from the corresponding **SSL_CTX**.

5.4.2 How they interact

Figure 5.4 shows the lifetime of the aforementioned data structures:

- **ap_global_ctx** and **SSLModConfigRec** exists almost throughout the lifetime of the server process;
- **server_rec** is created during the configuration time based on information from **SSLModConfigRec** and other module configurations;
- **SSL_CTX** is created based on **SSLSrvConfigRec**;

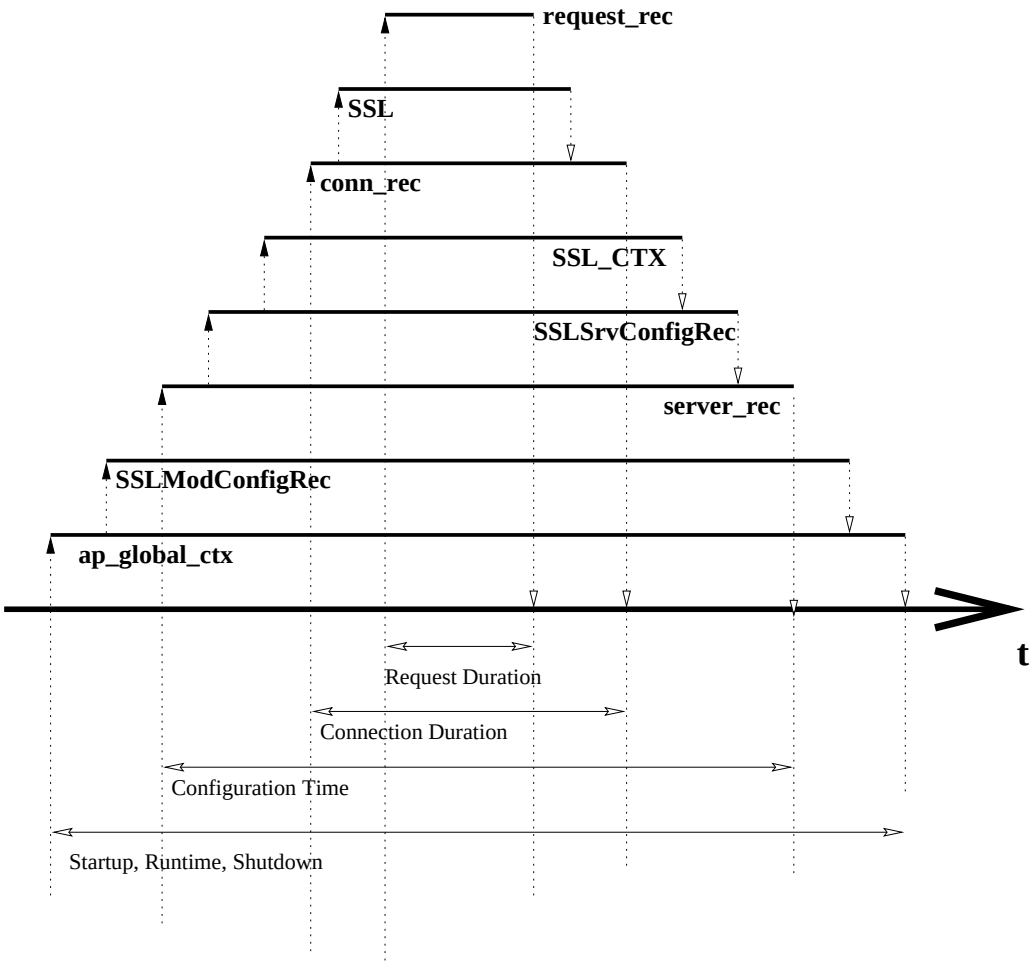

Figure 5.4: The lifetime of Apache, mod_ssl, OpenSSL data structures (based on [20])

- **conn_rec** and **SSL** exists during the lifespan of a connection;
- **request_rec** has the shortest life time – it exists during the processing of a particular request.

With this data structure hierarchy, the configuration information of the server is passed from **SSLModConfigRec**, through **server_rec**, **SSL_CTX**, and eventually reaches **SSL** so that the correct subset of the functionalities provided by OpenSSL will be used to serve requests as the server operator desired.

Chapter 6

WebALPS Implementation

The previous chapters introduced the two important pieces of enabling technology for WebALPS, described the structure of the web server that will host the WebALPS co-server, and discussed important design considerations about the project. Now is the time to talk about the implementation. This chapter starts with a summary of all the aforementioned design issues and the specification of the system we used, and then describes how we actually implemented the project.

6.1 Design and the System

Based on the previous discussions, here are the design decisions that we made and the system we chose to use:

- about the secure coprocessor:
  - we chose to use IBM 4758 PCI cryptographic coprocessor model 023;
  - we decided to only port functionalities that are necessary for providing security services

into the coprocessor;

  - we decided to use DES as the choice of symmetric cipher for WebALPS-enabled SSL connection.

- about the SSL protocol:

  - we chose to support SSLv3.0 without client authentication;

  - we revised the Handshake process and session reuse process to accommodate the introduction of trusted WebALPS co-servers;

  - we revised the Record Layer protocol to accommodate the introduction of trusted WebALPS co-servers.

- about the web server:

  - we chose Apache server (v1.3.14)/mod_ssl (v2.7.1)/OpenSSL (v0.9.6) to host the WebALPS co-servers (these versions had been the most recent stable release by the time we started the project);

  - we will configure our server to include a normal HTTPS virtual host and a WebALPS-enabled HTTPS virtual host.

6.2 Configurability

The following changes were made to achieve the goal of configurability:

- we added a new configuration directive, *SSLWebalps*, to mod_ssl, which takes either *On* or *Off* as argument;

- we added a state variable *bWebalps* to the mod_ssl per-server data structure **SSLSrvConfigRec**;

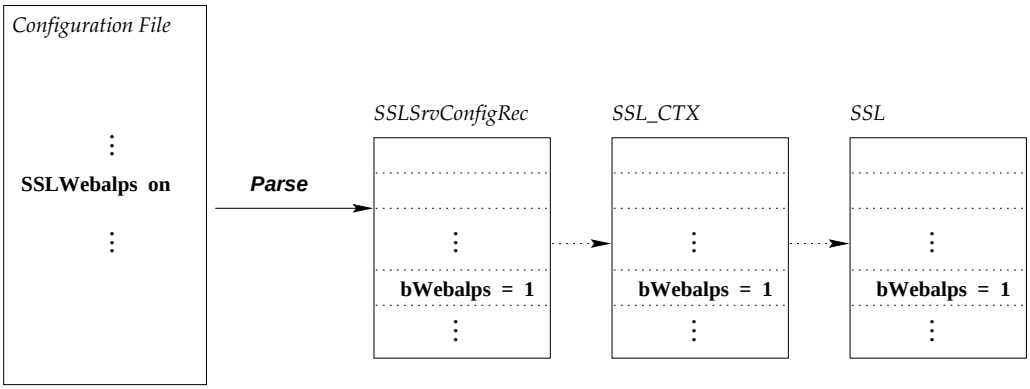

Figure 6.1: Implementation of Configurability

- we added the configuration handler functions *ssl_cmd_SSLWebalps* to parse the *SSLWebalps* configuration directive and set *SSLSrvConfigRec->bWebalps* accordingly;
- we added a state variable *bWebalps* to the OpenSSL per-server data structure **SSL_CTX**;
- we added a state variable *bWebalps* to the OpenSSL per-connection data structure **SSL**.

With these changes, when a user include *SSLWebalps on* in his server configuration file, this information will be passed through **SSLSrvConfigRec**, **SSL_CTX**, and eventually reaches **SSL**, the data structure that keeps all the information about an SSL connection (Figure 6.1). Then all the functions that deal with handshake protocol and record layer protocol can check **SSL** to see if they should use co-servers or not. This effectively accomplishes the goal of a configurable WebALPS system.

6.3 Porting WebALPS co-server's Certificate

As illustrated in Figure 4.6, the Apache server should cache the certificate provided by WebALPS co-server that can be used to authenticate co-servers to a remote client. Unfortunately, the IBM

4758 does not provide built-in mechanisms for parsing and generating X.509 certificate (the format that SSL uses) yet. To avoid being distracted by the complexity of implementing this mechanism by ourselves, we decided to reuse the *OpenSSL* certificate generation utility in our prototype implementation. The whole process involves three steps:

- key generation: we rewrote part of the *OpenSSL* code so that the generated key pair contains:
  - the WebALPS co-server's public key as the public part;
  - some garbage value as the private part;
- certificate request (CSR) generation: we rewrote the *OpenSSL* code so that the last step of CSR generation sends the CSR into the secure coprocessor to be signed by the co-server's private key;
- certificate generation: just as the CSR generation process, last step of certificate generation is rewritten to send the certificate into the secure coprocessor to be signed using co-server's private key.

The certificate generated is a self-signed temporary certificate. When a client tries to access this server using a commercial browser, a warning for unrecognized Certificate Authority (CA) will pop up. However, this temporary solution suffices for our prototype-testing purpose. For real deployment, we will need a formal certificate signed by a CA, so that it can be used to verify that a keypair really belongs to the WebALPS co-server. [24] has more discussions about this.

6.4 Implement WebALPS-enabled SSL Protocol

This section describes the central piece of implementation work in the project.

6.4.1 Storage of SSL Session Information in the Co-server

In the WebALPS co-server, we use a hash table (Figure 6.2) to store the minimal set of information of all the alive SSL sessions. Each entry in the hash table stores a data structure that contains the following data about one particular SSL session:

- **session_id**: the session ID of the session. This is used as the key for this entry in the hash table since each session has a unique ID;
- **client_ms**, **client_key**, **client_iv**: the MAC secret, DES key, and initialization vector for verifying and reading messages from the client;
- **server_ms**, **server_key**, **server_iv**: the MAC secret, DES key, and initialization vector for MAC generation and encryption of messages to be sent from the co-server;
- **master_secret**: an intermediate result during the key generation process and will be reused if the server decides to reuse this session;
- **read_sequence**, **write_sequence**: the number of messages that the co-server has received from the client and number of messages that the co-server has sent to the client. These state information is needed during the MAC generation process;
- **app_data**: the application-specific data.

6.4.2 Implement WebALPS-enabled Handshake protocol and Record Layer protocol

With the clear design of the Webalps-enabled Handshake protocol and Record Protocol shown in Chapter 4, the implementation follows naturally. Table 6.1 outlines the major functions that are involved in this process.

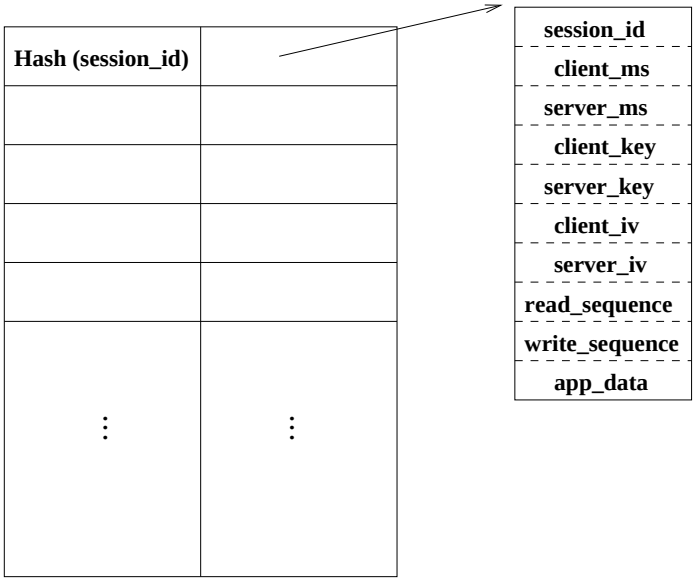

Figure 6.2: The data structure that stores session information in the WebALPS co-server

State	Host SSL functions	Card WebALPS functions
read *Client Hello*	ssl3_get_client_hello	
write *Server Hello*	ssl3_send_server_hello	
		webalps_resume_session_handler
write *Certificate*	ssl3_send_server_certificate	
write *ServerHelloDone*	ssl3_send_server_done	
send TCP frame	BIO_flush	
read *ClientKeyExchange*	ssl3_get_client_key_exchange	webalps_new_session_handler
read *ChangeCipherSpec*	do_change_cipher_spec	
	ssl3_final_finish_mac	webalps_ssl3_finish_mac
read *Finished*	ssl3_get_finished	
	ssl3_get_record	webalps_retrieve_message
		webalps_des_cipher
		webalps_ssl3_mac
write *ClientKeyExchange*	ssl3_send_change_cipher_spec	
write *Finished*	ssl3_send_finished	
	do_ssl3_write	webalps_assemble_message

Table 6.1: Implementation of WebALPS-enabled Handshake protocol and Record Layer protocol

Among these functions,

- *webalps_retrieve_message* and *webalps_assemble_message* are the co-server's record layer functions. They use *webalps_des_cipher* to provide message encryption/decryption and use *webalps_ssl3_mac* to calculate the MAC of a message. These two functions also each accommodate a hook function, which will be called to carry on application-specific tasks;
- *webalps_new_session* establishes a new SSL session by calculating the shared secret and storing them inside the hash table described previously;
- *webalps_resume_session* deals with session reuse by retrieving the stored **master_secret** of the previous session and recalculating the shared secret.

6.5 The Design of a Simple Application

With the implementation of WebALPS-enabled SSL protocol done, our prototype WebALPS co-server has emerged. Unlike the previous situation in which a client has to blindly trust a remote server, now the client can put well-founded trust on the remote party, the WebALPS co-server, that he is communication with. How to make use of this nice feature and turn it into a meaningful, appealing real-world application is the very first thing on our future task list. To illustrate in general how WebALPS can provide meaningful services, in this section we describe the paper design of a simple application.

Imagine the common grade-retrieval system where a student provides a password to a server and the server returns his grade. With the help of WebALPS-enable co-server, we can build a system in which the password is totally invisible to the server. Figure 6.3 sketches such a system:

- the client, a student named Shan, sends the request of retrieving his grade to the server together

with his name and password (for the sake of illustration, we assume this request uses GET method);

- the server can not get the client's password from the request or even understand the request since it does not know the secret key the client uses to encrypt the request. The server forwards the request to the co-server;
- the co-server gets the encrypted request, decrypts it, and gets the submitted password;
- the co-server calls a hook function from inside *webalps_retrieve_message* to erase the password part of the request and gives the modified plain-text request back to the server;
- the server now knows the name of the client and looks up Shan's record in its database. The record, which is encrypted using a key only known to the co-server, contains three pieces of information:
  - name of the student;
  - password of the student;
  - the grade of the student.

  In a real application, this record should also contain mechanisms to ensure its integrity (such as an internal MAC) as well as mechanisms to defeat replay attacks.
- the server sends the encrypted record back to the co-server;
- the co-server decrypts the record and calls a hook function from within *webalps_assemble_message* to check:
  - if the name in the record matches the name in the request;
  - if the password in the record matches the password in the request.

  and generate an HTML page dynamically according to the result of the check: if the check succeeded, Shan's grade will be included in the responsec.

- the co-server sends the generated page back to the client using the secure channel established during the handshake.

In a real world application, the response sent back to the client often contains a significant amount of non-sensitive information (images, for example) for presentation, advertisement, and many other purposes. To accommodate for this, during the response phase, we can ask the server to generate a HTML template that contains the non-sensitive part of the response, as well as a tag marking where in the page the co-server should insert the sensitive information. The co-server then finds this tag and put the necessary information in there. This can be achieved through an HTML extension or maybe a new standard defined through XML.

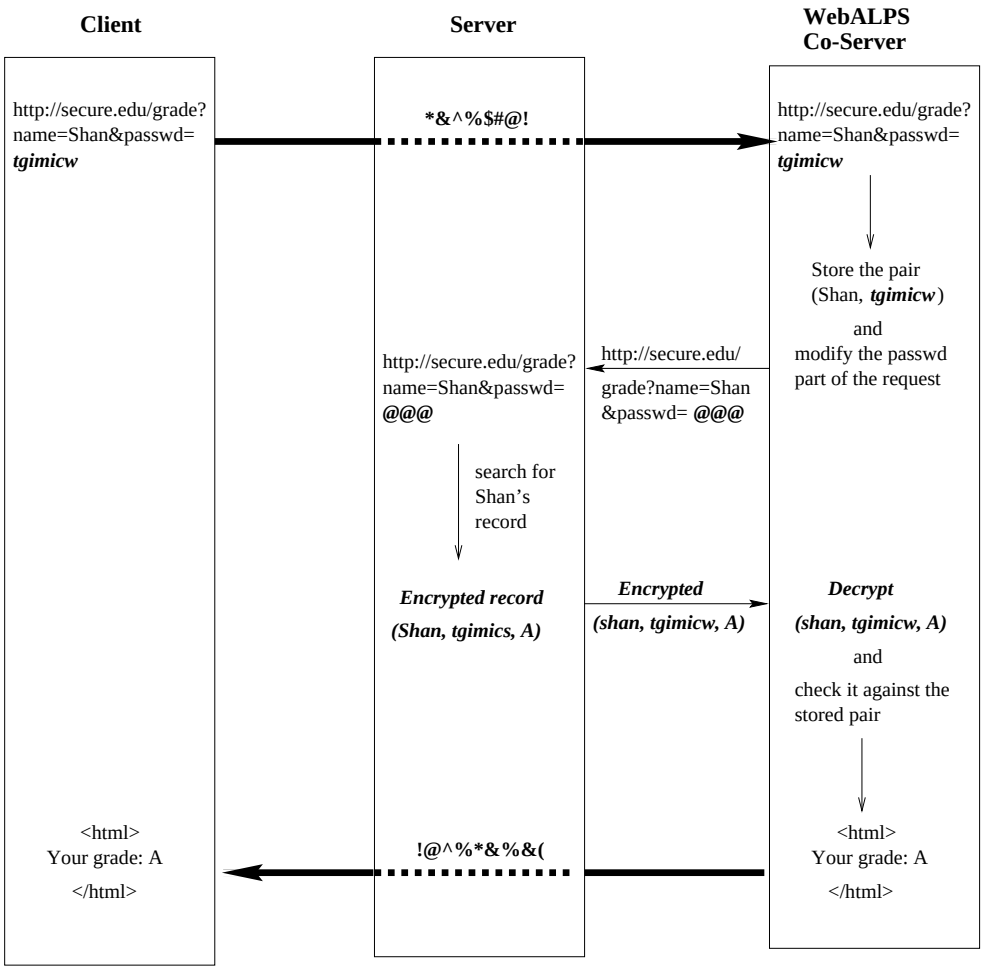

Figure 6.3: WebALPS application: secure password-based grade retrieval system

Chapter 7

Performance Analysis

As stated in Chapter 2, performance and scalability are two important requirements for the WebALPS-enabled server. This chapter presents the results of out performance testing on the prototype WebALPS-enabled server (without any application processings) using popular web server benchmarking tools, and analyzes these results.

7.1 Test Goal

We are interested to see:

- how much does the communication overhead introduced by the co-server slow down the performance of the WebALPS-enabled server;
- how well can WebALPS-enabled server sustain heavy workload.

In addition to these two goals, we also want to establish the correctness of our implementation through intensive testings.

In the next two sections, we will describe the performance test and the scalability test respectively. These tests are done on a WebALPS co-server with no application level processing.

7.2 Speed Test

Speed here refers to how fast a server serves requests from the clients. It is normally measured in number of requests served per second.

7.2.1 Test Tool

For speed test, we used *http_load* [1], a free, easy-to-use, and effective tool from ACME software. The test with *http_load* can be configured by specifying a value for the following command line options:

- *parallel*: how many test clients;
- *rate*: how many requests each client sends out per second;
- *seconds*: in terms of time, how long the test lasts;
- *fetches*: in terms of served request numbers, how long the test lasts.

Normally, only one from *parallel* and *emph*, and one from *seconds* and *fetches* needs to be specified.

7.2.2 Testing Setup

Workload

We chose to use a randomly generated file of size 2KB as our test load. All the requests during the test are for this file.

http_load Parameter Setup

What we are interested to know through this speed test is that for one single connection, how fast the WebALPS-enabled server serves a single request. This information can tell us how much overhead is produced by the interaction between the server running in the host CPU and the co-server running inside a PCI card. Therefore we set *parallel* to 1 and set *seconds* to 2. Section 7.3 will talk about the server's performance under multiple connections for prolonged tests.

Data Collection

For the sake of comparisons, we will run the same tests on three types of hosts:

- WebALPS-enabled HTTPS host;
- normal HTTPS host;
- HTTP host.

On each type of host, we perform the same test ten times and each time we collect the following three pieces of information:

- speed: the number of served requests per second. This tells us the speed of the host from the

	Speed (requests/sec)	Connection Time (msec)	Request Time (msec)
WebALPS host	9.8922	0.7235	100.368
normal HTTPS host	67.7246	0.6335	14.1461
HTTP host	858.7989	0.1832	0.9060

Table 7.1: Speed test and comparisons of WebALPS host, normal HTTPS host, and HTTP host

experience of the client – how long it takes from when client starts trying to talk to the host until the client gets back the response. This time is the sum of the following two measurements;

- connection time: the amount of time taken to establish the connection. For HTTPS hosts, this time includes both the TCP connection time and the SSL handshake time;
- request time: the amount of time taken to process a request once the connection is established.

The results from all ten tests will be averaged to get the final result that we present in the next section (more elaborate statistical analysis will be done in the future).

7.2.3 Testing Result

Table 7.1 shows the results from our tests. Not surprisingly, in all three measurements, plain HTTP server offers the best performance, while WebALPS server performs the worst. We knew this result even before the test. What we really want to learn from these results is:

- how does the slowdown from normal HTTPS host to WebALPS-enabled host compare to the slowdown from plain HTTP host to HTTPS host?
- which phase is the major factor for the slowdown of the WebALPS-enabled server, connection phase or request phase?

	Speed	Connection Time	Request Time
Slowdown caused by SSL (HTTP -> HTTPS)	11.68	2.46	14.6
Slowdown caused by WebALPS (HTTPS -> WebALPS-HTTPS)	5.85	14.2%	6.1

Table 7.2: Comparisons of slowdowns caused by WebALPS with slowdowns caused by SSL

Comparing the slowdown Caused by WebALPS to the slowdown Caused by SSL

The data in Table 7.2 shows how much SSL slows down HTTP communication and how much WebALPS slows down SSL communication:

- for the overall speed, normal HTTPS host is 11.68 times slower than HTTP host while WebALPS host is 5.85 times slower than normal HTTPS host;
- for the connection time, normal HTTPS host is 2.36 times slower than HTTP host while WebALPS host is only 14.2% slower than normal HTTPS host;
- for the request time, normal HTTP host is 14.6 times slower than HTTP host while WebALPS host is 6.1 times slower than normal HTTPS host.

Figure 7.1, Figure 7.2, and Figure 7.3 provide a more direct view of these comparisons. In every category, SSL technology has a far greater negative performance impact on HTTP than what WebALPS technology has on SSL.

Despite of the performance slowdown, SSL technology still prevails in today's e-commerce world. We think that there are at least two reasons behind SSL's success:

- SSL offers security communication channel which is essential to many applications;
- communications that go through SSL channel only occupy a very small percentage of total web

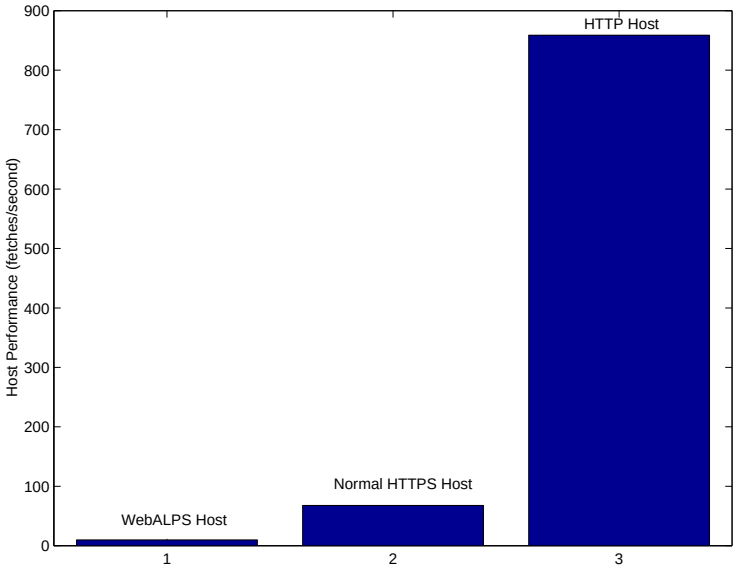

Figure 7.1: Comparisons of server speed among WebALPS host, normal HTTPS host, and HTTP host

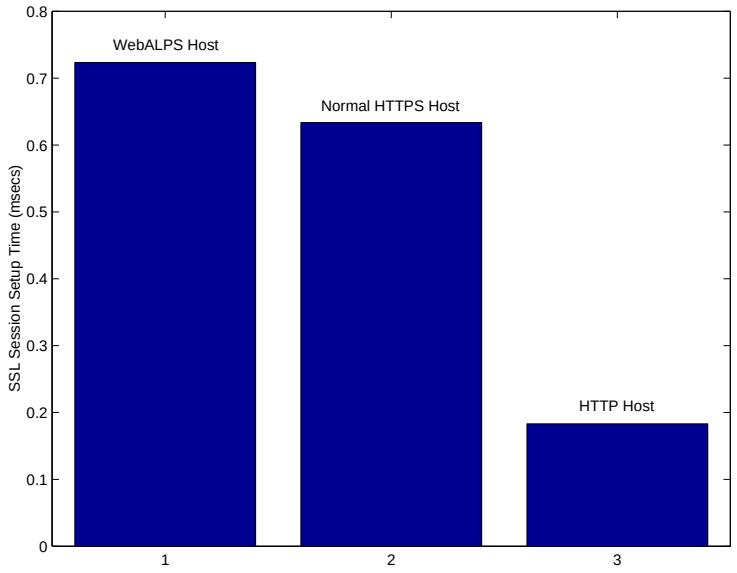

Figure 7.2: Comparisons of connection time among WebALPS host, normal HTTPS host, and HTTP host

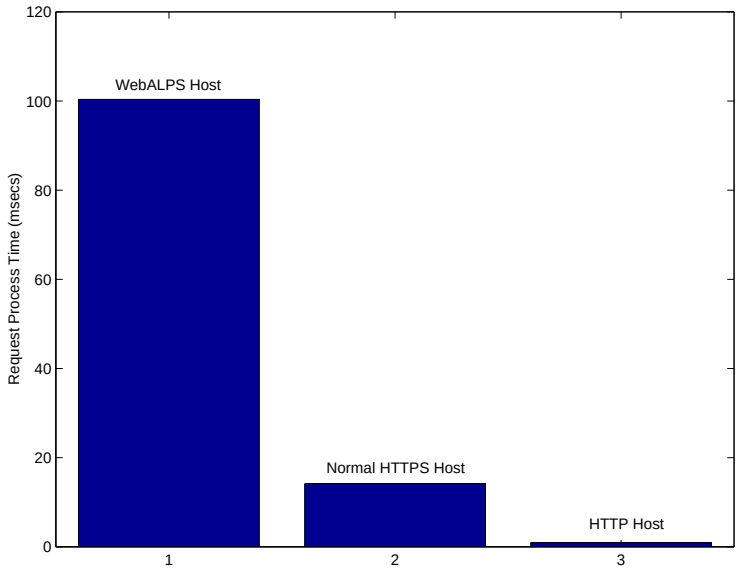

Figure 7.3: Comparisons of request process time among WebALPS host, normal HTTPS host, and HTTP host

traffic. A user's experience about a server's speed is built up based on the overall performance of the server, which is largely determined by non-SSL communications.

These two arguments remain true for WebALPS-enabled server. The kind of security and privacy that WebALPS co-server offers will enable a variety of applications that are hard or even impossible to realize before [24]. Based on this similarity with SSL, we argue that although WebALPS approach slows down the secure communication, it could still win as a technology because of the important security features it offers.

Where is the Bottleneck

From table 7.2, there is little difference (14.2%) between the connection (TCP plus SSL handshake) phase for WebALPS-enabled host and normal HTTPS host. This could be attributed to the fact that WebALPS co-server uses the IBM 4758's fast modular math engine for RSA operations that

are required during the SSL handshake process.

However, for request time, WebALPS-enabled Host is 6.1 times slower than normal HTTPS host. The time-limiting factor could be one or more from the following:

- bulk data transfer rate between the host and the coprocessor;
- bulk data DES operation and MAC calculation rate in the coprocessor.

The exact reason is still not clear. More tests are being devised to pinpoint the performance bottleneck here.

7.3 Scalability Test

7.3.1 Testing Tool

For scalability testing, we chose to use WebBench [11], a commercial-grade, web server benchmarking product from Ziff eTesting Labs Inc. We made the choice because:

- it offers a very easy way to create large test suites – different combinations of testings that can run for hours, a feature that is essential to scalability test;
- it offers several standard test workloads, which are widely acknowledge and used to test commercial web servers.

WebBench's standard test suites produce two overall scores for the server being tested:

- speed as measured in requests per second

- throughput as measured in bytes per second.

WebBench offers both static standard test suites and dynamic standard test suites. The test requires one or more client machines running Windows 95/98/NT/2000, as well as a controller machine that runs Window NT/2000.

7.3.2 Test Setup

Workload Document Tree

We used the standard workload tree provided by WebBench. It contains of 6,160 HTML or GIF files with file sizes ranging from 223 bytes to 529KB. The file type and size distribution of this workload tree is determined by the data collected from popular commercial web sites including the Internet Movie Database, Microsoft, USA Today, and ZDNet.

The Test Suite

Among all the standard test suites offered by WebBench, we chose to use the e-commerce standard test suite. As stated in Section 7.2.3, a user's satisfaction about a web server's performance is built on the server's overall performance when executing the set of workloads that this user enforces on the server. The e-commerce test suite is a good representation of the kind of workloads our WebALPS-enabled server might face in the real life.

Below is the characteristics of the e-commerce test suite:

- 80% static requests and 20% dynamic requests (invoke a CGI program on the server side);
- 92% HTTP requests and 8% HTTPS requests;

- For each client, the minimum number of session reuse is 5 and the maximum number is 15.

Number of Clients

Due to the limited number of Windows machines available to us, we ran 8 clients on two machines with 4 clients per machine and 3 threads per client. During the whole test process, we monitored the system usage on the client machines and found that it never exceeded 50%. This fact ensures us that the test results are not restricted by the ability of the clients to send out requests.

7.3.3 Test Results

In this section, we compare the test result we obtained from using the same WebBench e-commerce test suite on the WebALPS-enabled HTTPS virtual host and the normal HTTPS virtual host.

Figure 7.4 shows the scalability comparisons between WebALPS-enabled host and normal HTTPS host in terms of the server's ability to sustain its serving speed (measured in number of requests served per second) under high workload. From the figure, it is clear that both hosts scale well under the workload provided in the e-commerce suite. As for the performance, on average, WebALPS-enabled host is about 25% slower than a normal HTTPS host.

Figure 7.5 shows the scalability comparisons in terms of throughput. The result is similar to what we got from the previous figure. Both hosts show good scalability and on average, the throughput provided by WebALPS-enabled host is about 28% less than normal HTTPS host.

From these data, we can safely conclude that within our test range, WebALPS-enabled server are scalable under real-life e-commerce workloads.

Another thing worth to mention is that in this test, with the introduction of the non-secure com-

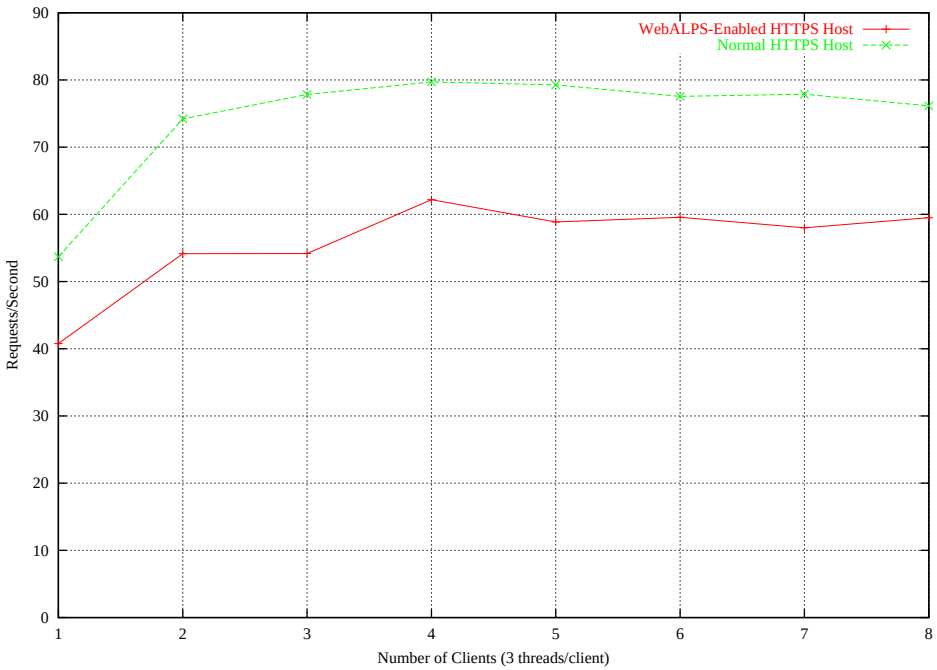

Figure 7.4: Scalability comparisons between a WebALPS-enabled HTTPS host and a normal HTTPS host (Requests/Second)

ponents in the workload that represents real life scenario, the performance slowdown of WebALPS-enabled server has reduced from over 500% (as shown in Section 7.2.3) to only about 25%. This proves our previous argument that to users in real life, the performance slowdown of WebALPS approach would not be as dramatic as what Table 7.2 shows.

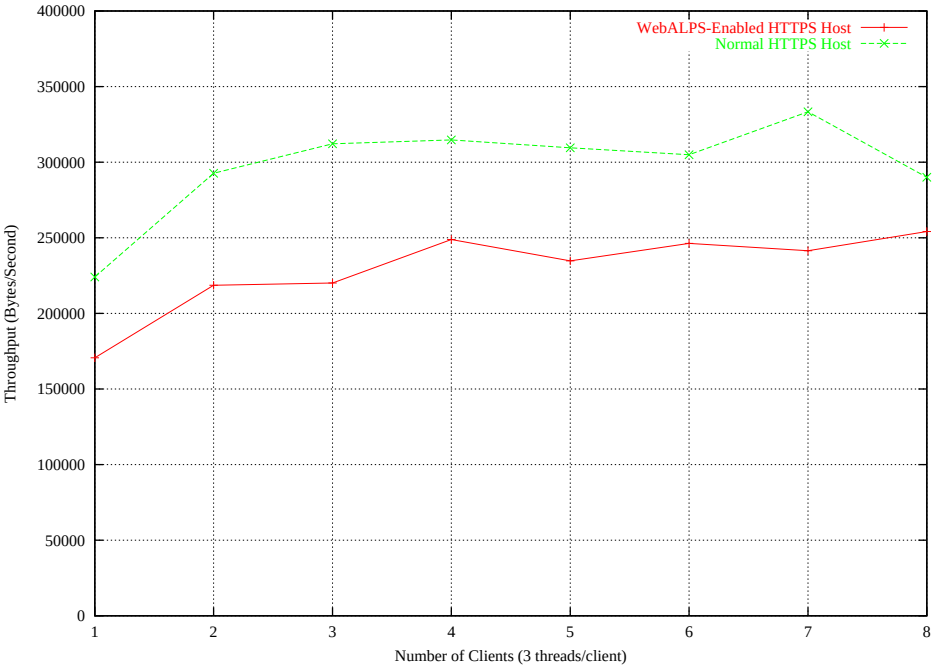

Figure 7.5: Scalability comparisons between WebALPS-enabled HTTPS host and normal HTTPS host (Throughput)

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis work, we successfully designed and implemented a prototype WebALPS co-server which is able to establish an authenticated and secure communication channel to a remote client through our revised SSL process. Based on the certified security features offered by IBM secure coprocessors, this prototype co-server is able to act as a trusted entity in the Web interactions and has the potential of offering sensitive web-based services to remote clients.

The performance test of this prototype shows that the WebALPS co-server can offer security and privacy to web clients at a reasonable performance cost compared with how much SSL technology sacrificed for similar goals. Moreover, when tested with a real-life e-commerce workload, our prototype demonstrates good performance and scalability that are comparable to what a normal SSL-enabled Apache server offers, proving WebALPS to be a practical technology. The fact that the prototype has survived the intensive tests also establishes our confidence about the correctness

of our implementation. To further test this, we plan to make our code and demos publicly available.

We also illustrate the scheme of implementing WebALPS-based applications through our simple secure grade-retrieval system. Now we are working on building more complicated systems based on our WebALPS co-servers.

In summary, our work in this thesis realized the WebALPS approach and proved its feasibility. We are confident that the WebALPS trusted co-server will become a critical piece of enabling technology for security and privacy in web-based services.

8.2 Future Work

First, several things can be done to improve the prototype co-server:

- the coprocessor has a limited memory. Under heavy workload, it could become a bottleneck piling up with session information. Moreover, the single connection between the co-server and multiple server processes could also become a performance concern. One way to address this problem is to use multiple coprocessors to implement a one-server-multiple-co-server structure. This approach has the potential of improving the WebALPS-enabled server's performance with the introduced parallelism;

- we plan to add support for more SSL options including more cipher suites and client authentication;

- as illustrated in Figure 1.1, many services cannot be done at one web site (for example, the Dartmouth basement web mail system needs to talk to the DND server for user authentication). We need to give the co-server the functionality of acting as a client to open a new SSL connection to another server and accomplish the services it needs to provide to the clients.

Second, the WebALPS approach not only requires the server-side work, it also needs support from the client side. For example, for this approach to work, users have to be able to distinguish the traffic from trusted WebALPS co-servers from the traffic from untrusted servers. However, current commercial browsers don't even offer a clear and secure way to indicate which CA signed the certificate provided by a server.

Moreover, we are also considering other ways of implementing the WebALPS approach. For example, if the IBM 4758 is improved to provide a larger memory, networking support, as well as a standard OS, to build a whole web server inside the coprocessor would become a valid option [6].

Finally, the successful prototype implementation invites application development. We need to identify the application scenarios that can benefit from using WebALPS co-servers, design the scheme for the interactions between the server and the co-server for those scenarios, and eventually implement these applications.

Bibliography

[1] ACME Laboratories. *http://www.acme.com/software/http_load*

[2] ActivMedia Report. "Real Numbers Behind Net Profits 2000." June 2000. *http://www.activmediaresearch.com/real_numbers_2000.htm*

[3] Alexa Research. "Grow Web, Grow! Internet Trends Report: 1999 Review." February 1, 2000. *http://www.alexaresearch.com/top/report_4q99.cfm*

[4] Apache Software Foundation. "Apache HTTP Server Project." *http://httpd.apache.org*

[5] APBNEWS news. "Massive Cybertheft Case Revealed." March 17, 2000. *http://apbnews.com/newscenter/internetcrime/2000/03/17/credit0317_01.html*

[6] Paula Austel, Ron Perez. Personal communication. 2001.

[7] Computer Security Institute Press Release. "Financial losses due to internet intrusions, trade secret theft and other cyber crimes soar." Mar 12, 2001. *http://www.gocsi.com/prelea_000321.htm*

[8] Mark J. Cox, Ralf S. Engelschall, Stephen Henson, Ben Laurie, et. al. "OpenSSL project." *http://www.openssl.org*

[9] Cyveillance press release. "Internet Exceeds 2 Billion Pages." July 10, 2000. *http://www.cyveillance.com/web/us/newsroom/releases/2000/2000-07-10.htm*

[10] Joan Dyer, Ron Perez, Sean Smith, Mark Lindemann. "Application Support Architecture for a High-Performance, Programmable Secure Coprocessor." *Proceedings, 22nd National Information Systems Security Conference.* October 1999.

[11] eTesting Labs, Ziff Davis Media Inc. *http://www.zdnet.com/etestinglabs/stories/benchmarks/0,8829,2326243,00.html*

[12] Alan O. Freier, Philip Karlton, Paul C. Kocher. "The SSL Protocol Version 3.0." November 18, 1996. *http://home.netscape.com/eng/ssl3/draft302.txt*

[13] R. Housley, W. Ford, W. Polk, D. Solo. "Internet X.509 Public Key Infrastructure Certificate and CRL Profile." January 1999. *http://www.ietf.org/rfc/rfc2459.txt*

[14] *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference Version 2: 4758-002 and 4758-023.* March 2001.

[15] *IBM 4758 PCI Cryptographic Coprocessor Models 001, 013, 002, and 023 General Information Manual.* January 2000.

[16] S. Kent, R. Atkinson. "Security Architecture for the Internet Protocol." November 1998. *http://www.ietf.org/rfc/rfc2401.txt*

[17] MSNBC news. "Hackers breached Davos security." February 5, 2001. *http://www.msnbc.com/news/526270.asp?cp1=1*

[18] Netcraft. "The Netcraft Web Server Servey." April 2001. *http://www.netcraft.com/survey*

[19] Ralf S. Engelschall, Ben Laurie. "mod_ssl project." *http://www.modssl.org*

[20] Ralf S. Engelschall. *Figure Apache+mod_ssl+OpenSSL Data Structure Overview.* Included in the mod_ssl source distribution.

[21] E. Rescorla, A. Schiffman. "The Secure HyperText Transfer Protocol." May 1996. *http://www.terisa.com/shttp/1.2.1.txt*

[22] Sean W. Smith, Steve Weingart. "Building a High-Performance, Programmable Secure Coprocessor." *Computer Networks.* (Special Issue on Computer Network Security). 31:831-860. April 1999.

[23] Sean W. Smith, Ron Perez, Steve Weingart, Vernon Austel. "Validating a High-Performance, Programmable Secure Coprocessor." *Proceedings, 22nd National Information Systems Security Conference.* October 1999.

[24] Sean W. Smith. "WebALPS: Using Trusted Co-Servers to Enhance Privacy and Security of Web Interactions." *Research Report RC-21851, IBM T.J. Watson Research Center.* October 2000.

[25] Sean W. Smith. "Outbound Authentication for Programmable Secure Coprocessors." Draft, March 2001.

[26] Lincol Stein , Doug MacEachern. *Writing Apache Modules with Perl and C* O'Reilly, 1999

[27] Stephen Thomas. *SSL and TLS Essentials: securing the Web* John Wiley & Sons, 2000

[28] United States Department of Commerce News. "Estimated Quarterly U.S. Retail E-commerce Sales: 4th Quarter 1999 - 1st Quarter 2001." May 2001.
http://www.census.gov/mrts/www/current.html