

Securing Web Servers against Insider Attack

Shan Jiang

shan.jiang@alum.dartmouth.org

Sean Smith

sws@cs.dartmouth.edu

Kazuhiro Minami

minami@cs.dartmouth.edu

Department of Computer Science/Institute for Security Technology Studies *
Dartmouth College

Abstract

Too often, “security of Web transactions” reduces to “encryption of the channel”—and neglects to address what happens at the server on the other end. This oversight forces clients to trust the good intentions and competence of the server operator—but gives clients no basis for that trust. In this paper, we apply secure coprocessing and cryptography to solve this real problem in Web technology. We present a vision: using secure coprocessors to establish trusted co-servers at Web servers and moving sensitive computations inside these co-servers; we present a prototype implementation of this vision that scales to realistic workloads; and we validate this approach by building a simple E-voting application on top of our prototype.

By showing the real potential of COTS secure coprocessing technology to establish trusted islands of computation in hostile environments—such as at web servers with risk of insider attack—this work also helps demonstrate that “secure hardware” can be more than synonym for “cryptographic accelerator.”

1 Introduction

In the area of Web security, despite strong encryption on the browser-server channel, Web users still have no assurance about what happens at the other end. We present a security application that augments Web servers with *trusted co-servers* composed of high-assurance *secure coprocessors*, configured with a publicly known *guardian program*. Web users can then establish their authenticated, encrypted channels with a trusted co-server, which then can act as a trusted third party in the browser-server interaction.

*This work was supported in part by the U.S. Department of Justice, contract 2000-DT-CX-K001, and by Internet2/AT&T. However, the views and conclusions do not necessarily represent those of the sponsors. Preliminary versions of portions of this paper appeared as IBM Research Report RC-21851 and Dartmouth CS Technical Report TR2001-410.

General Scenario. Consider services where the user (or other parties) depends on the confidentiality or integrity of computation/data storage occurring at the server—but the server operator may benefit from violating these properties.

How can the client have any assurance that these properties still hold? The only feasible solutions require moving this critical computation to a trusted third party.

In our approach, we use the enabling technology of secure coprocessors to build this trusted third party at the server’s own site. The tamper resistance and authenticatable execution establish that the alleged computation is really occurring, but beyond the reach of even the server operator to subvert (except, of course, to destroy). In the dominant SSL framework of server-side authentication, users trust a CA to bind a server identity to a public key. In our framework, users would also need to trust a CA to bind a public key to a particular guardian program (either as an identity certificate or an attribute certificate)—and to have suitably evaluated that program. The guardian program, executing inside the coprocessor, would act as a trusted third party in the interaction. The underlying secure coprocessor technology we use permits a CA to make this authentication, and permits both CA and user to trust that this guardian can in fact as a trusted third party.

If deployed in the real world, this work can address a core trust problem in the Web. What’s more, this work would also provide a practical demonstration of the potential of secure coprocessing to resolve difficult trust problems—with real code, for real problems, with a feasible commercial and technological path to broader deployment.

We have been calling this project *WebALPS*: Web Applications with Lots of Privacy and Security.¹

Prior Work Prior work in secure coprocessing has established how to build tamper-responding secure hardware [17]; how to use this tamper-response technology (and

¹This name emerged at lunch one day, partly in jest: since it connotes “little Switzerland”: little neutral areas distributed on the Web.

many other techniques) to build general-purpose secure coprocessors, that can be trusted to carry out their computation correctly despite attempts to physically attack or modify this computation [16]; and how to use secure hardware for some individual specialized applications [19, 20].

Related work has also examined the use of secure hardware as SSL cryptographic accelerators for Web servers (although, with rare exception [8], most discussions of this application overlook the uncertain role of physical security in contexts where the server must be trusted anyway).

Our Project The WebALPS project unifies and moves beyond all of these, by using secure coprocessors to establish trusted third parties at Web servers. These trusted co-servers can then bring many additional security and privacy properties to a wide range of Web applications. Basing this work on having clients establish an SSL session *into* an application running inside the secure hardware at the Web server (instead of just using secure hardware to speed cryptography) provides a systematic way to enhance the security of a broad family of Web-based services, without requiring a substantial change to the currently deployed Web infrastructure.

We stress that, although some prior work discusses “secure web servers,” we believe that we are unique in examining web servers that are *secure against insider attack*.

This Paper Section 2 presents some background on secure coprocessing and on SSL. Section 3 presents the WebALPS vision, and how can address the Web manifestation of a basic trust problem. Section 4 describes our implementation and testing of a prototype WebALPS co-server. Section 5 describes our building a simple application on top of this prototype. Section 6 presents our conclusions and list some of the future research directions.

2 Background

2.1 Secure Coprocessing

The technology of high-performance, high-assurance programmable secure coprocessors enables the WebALPS project.

Motivation Distributed computation (and even centralized computation, with multiple parties) introduces a fundamental problem: *distribution dissociates dependency from control*. Consider a basic scenario: Alice and Bob participate in some computational activity. Alice’s interests \mathcal{I} depends on some correctness and/or privacy properties \mathcal{P} of some computation X at a computer that Bob controls. Consequently, Alice must depend on Bob to preserve and

protect her interests. However, Bob may have no motivation to do this; and, in fact, Bob’s interests may conflict with Alice’s, and motivate him to actively subvert Alice’s computation.

This scenario characterizes many real-world scenarios. In basic examples, Alice incurs a dependency on Bob that she would rather not have. However, it is enlightening to consider other dependency scenarios. For example, Alice may incur a dependency on Bob that *Bob* would rather not have. For example, consider a commerce application where Bob is a service provider. An ability to assert “customers can trust my service because they do not have to trust me” can be an effective marketing tool. Furthermore, an inability to fraudulently manipulate the service might preserve Bob from various litigation and legal hassles.

Secure Coprocessing as a Solution The technology of *secure coprocessors* has long been proposed as a foundation to address these problems. Quickly defined, a secure coprocessor is a general-purpose computer (possibly with cryptographic support) that is secure against all foreseeable physical and logical attacks (except, of course, denial of service stemming from actions such as complete destruction). Secure coprocessors can augment the security of an otherwise untrusted host computer, because the relevant parties have the potential to trust that the computation and data storage occurring in the coprocessor have been unmolested by adversaries with direct physical access to the machine.

Programmable, trusted secure coprocessors could enable systematic solutions to scenarios such as those sketched above. If X occurred in a secure coprocessor at Bob’s machine, and Alice was able to authenticate that X was occurring there, beyond Bob’s control, and Bob’s ability to manipulate his host and its network connections could not subvert \mathcal{P} , then Alice can trust that the important properties \mathcal{P} still hold of X , despite Bob’s potential attacks. Figure 1 sketches this revised scenario.

Some individual examples of these trust problems may be solvable—perhaps at a significant performance cost—via special cryptographic protocols (such as *secure multiparty computation* or *encrypted functions*). However, we still believe that only secure coprocessing can enable a systematic approach to practical, high-performance solutions.

The Enabling Platform The platform that enables practical realization of WebALPS is the IBM 4758 secure coprocessor² [16] The device provides a general-purpose computing environment for third-party code, with hardware support for cryptographic applications. Continuously active tamper-detection circuitry monitors tamper detectors and,

²Note that it important to distinguish the 4758 secure coprocessor platform from the IBM CCA application that many users install on it.

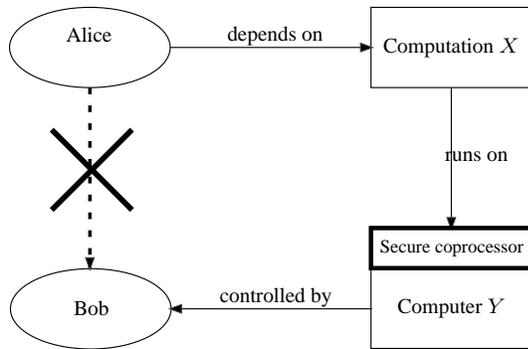


Figure 1. With secure coprocessing, Alice can trust the critical correctness properties of X , even if Bob may have motivation to subvert them. Here, by moving the computation on which Alice depends from Bob’s machine to a secure coprocessor added to Bob’s machine, Alice no longer incurs a dependency on Bob.

in case of physical attack, destroys sensitive secrets in secure memory before an adversary can access them. Code signing and a layered software architecture permits installation and maintenance of security control code, operating system, and application software. Hardware locks protect the keys and code of more critical layers from possibly malicious or faulty code, preserving the ability of each device to properly authenticate its configuration, and preventing a device with a rogue application from impersonating other devices and applications. The literature describes both this overall architecture [2, 16] as well as its independent validation at FIPS 140-1 Level 4 [15].

However, in order to support networked applications such as WebALPS, a secure coprocessor platform also must provide *outbound authentication*. Not only must Alice be able to publish an application and Bob install it in his coprocessor—Charlie, on the other side of the net, must be able to verify it’s really that application, in an untampered device. In the 4758, the initial version of security software generates a keypair which is certified by the factory. If the security software replaces itself, the old version generates and certifies a keypair for the new version, and destroys the old private key. When the security software installs or updates the OS or application layers, it generates and certifies a new keypair for the OS layer (and destroys the old private key). The OS layer then uses its keypair to generate and certify keypairs, and provide private key service, for the application layer. At each step in this process, the certificates identify (e.g., via software version and layer owner) the certifier and the certified. The private keys stay within memory protected both against physical tamper, as well against access from less-trusted software layers.

By building this internal PKI, this approach (further discussed in [13]) permits Charlie to draw conclusions about an entity that allegedly is a particular application in an untampered card, but enables these conclusions to be consistent with Charlie’s own beliefs about which software and layer owners can be trusted.

2.2 SSL

Readers not familiar with SSL might benefit from a quick introduction.

As commonly practiced, SSL permits the client to establish a shared symmetric key with a specific authenticated server through the *Handshake* protocol—one of the SSL components. The server has a private-public key pair, and a certificate from some CA attesting to something vague about the entity owning this public key. The client browser has some notion of which CA root keys it recognizes as valid. When a client opens an SSL connection, it verifies that the certificate from the server is correctly signed by a CA root that the client’s browser currently recognizes as legitimate. The client and server then carry out a key generation/exchange protocol that ensures that the client, and a party which knows the private key matching the server’s public key, share a symmetric key—that is (theoretically) shared by no one else, not even an adversary observing the messages between the client and server.

The remainder of the SSL session is then encrypted with this session key through the Record Layer protocol—another SSL component. Encryption (and MACs) with keys obtained this way provides several properties, including privacy and integrity of data along the channel, and the fact that each party can trust that, throughout the session, the entity claiming to be on the other end is the same entity that started the session (or, at worst, a colluding partner).

For protocols details, we refer the reader to the literature (e.g., [11]) or the specification itself (e.g., [3]).

3 Insider Attack at Web Servers

The previous section presented how secure coprocessing can address the question of trusting what happens at a remote site. In this section, we now present the WebALPS vision: how secure coprocessing can address the the basic trust problem in the context of insider attack at Web servers. (However, this work did not remain a vision; in Section 4 and Section 5, we discuss our completed prototypes.)

3.1 Fundamental Trust Problem

The World Wide Web is the grounds where, on a broad scale, our society’s business, government, and personal services are migrating and growing. As a basic model, a large

population of clients with browsers obtain services from a smaller population of service providers operating Web servers.

However, for each critical service that takes root in the Web (and arguably for many purely recreational services as well), the financial and personal interests of the clients forces them to trust the integrity and privacy of the computation and data storage at the service providers. But with the current state of deployed technology (i.e., SSL), all the client can be sure of is what the CA claims was the *identity* of the entity who originally possessed the public key in that server's certificate.³

At best, this identity establishes good intentions—if the alleged service provider has a pre-existing reputation that makes this hypothesis plausible. On the other hand: a service provider with an unknown reputation might be downright malicious; any service provider may have good intentions, but may be careless with general site security; the entity with which the client is currently interacting may not even be this original service provider, but rather an impostor who has learned the private key.

The threat that arises from this uncertainty is amplified by the Web's distribution of computation from server to client: via Java and Javascript, and also via more subtle executable content, such as Word documents infected with Macro viruses.

Furthermore, many interactions involve more parties than just the client and server—and these additional parties are *also* forced to trust the server integrity.

This situation leads to a *fundamental trust problem*:

Participants in distributed Web services are forced to trust server integrity, but have no basis for this trust.

This problem threatens a wide variety of Web applications. For one example, the current Web infrastructure provides no means for a server operator to plausibly deny that they (or adversaries who have compromised their machine) are not monitoring all client interactions for credit card numbers, passwords, personal profiles, and other sensitive data. If it exists in plaintext on the server at any point in time, then a rogue server operator has access. A companion paper [14] catalogs Web applications affected by this trust problem, including areas of non-repudiation and other issues.

3.2 The WebALPS Approach

To solve these problems, we need a systematic way for clients to trust the computation and data storage that occurs at a Web service provider of unknown credibility.

³Unfortunately, some recent Web spoofing work in our group [21] casts doubt even on this claim—in many scenarios, the client cannot even be sure of the fact an SSL session exists, let alone what the certificate really says.

What we propose is adding a secure coprocessor to the existing service provider infrastructure, as a *trusted co-server*, installing one of a well-defined, reasonably small set of guardian programs at the trusted co-server; training clients to use SSL to open an authenticated connection to the WebALPS guardian program at the co-server, not to the server directly; and changing the secure client-server interaction to center on client-co-server interaction, and then only bringing in the server as appropriate.

3.2.1 Properties

The WebALPS guardian program on the trusted co-server becomes a trusted “neutral” participant in the client-server interaction. (However, co-location at the server site allows us to have a “trusted third party” without actually having a third player, and should allow higher performance and easier integration than a remote third party would permit.)

By proper design of guardian program and server interaction, we can address the fundamental web security problem by raising the trust level of the computation and data storage at the server.

- The WebALPS guardian can witness the *authenticity* of certain data coming back to the client. This data can include assertions from the trusted guardian about the server content and configuration.
- The WebALPS guardian can provide *privacy* of data going back to the server, by keeping it encrypted between the client and the guardian, and then re-encrypting it before inserting it into the server.
- The user can trust the *integrity* of the computation occurring at the WebALPS guardian—even if the server operator might be motivated to subvert it.
- For computation relevant to third parties who may also have an interest in the client-server interaction, WebALPS provides a *haven* that all parties can trust to balance their interests.

Naively, one could also address such trust problems by putting the entire back-end service operation inside a coprocessor—but this would probably not constitute an effective solution in general, due to performance and maintenance reasons. The WebALPS approach avoids the problems of this naive straightforward approach.

Size The computational power, performance, and storage requirements for many Web servers probably exceed that provided by the 4758 platform. With WebALPS, rather than putting the entire server back-end into a coprocessor, we're only putting a small guardian.

Legacy Even if we could fit entire operations inside a coprocessor, such solutions would require substantial changes to existing e-merchants and service providers, who already have existing back-end systems. With WebALPS, rather than replacing the legacy server back-end, we're merely adding an additional component.

Authentication In order for the client to trust the server operations occurring in a secure coprocessor, the client has to be able to authenticate that the operation really is occurring in that type of system. Otherwise, we're back to "good intentions" again. With WebALPS, the proposed location of the guardian lets it easily use the existing server-side authentication features of the current infrastructure.

Usability Suppose we could fit the service software inside a coprocessor and could alter the client infrastructure so that clients could authenticate it really was that software. Each service provider potentially has their own software configuration. How could clients make any practical trust judgments about the server based solely on the identity of that software? With WebALPS, by limiting ourselves to a set of well-defined guardian programs, users do not need to make a new trust decision for each server configuration—but, rather, for each guardian program (as testified by the WebALPS CA).

Trusted Computing Base By minimizing what we put inside the coprocessor, we also minimize the computing base that stakeholders are forced to trust.

3.2.2 Architecture

When deployed in the real world, WebALPS will consist of a well-defined class \mathcal{A} of guardian types, each denoted by a human-understandable name; a trusted WebALPS Certificate Authority; and some mechanism in client browsers to clearly indicate: which portion of the browser window exclusively renders material from the other entity in this SSL session; when an SSL session has been established from a certificate signed the WebALPS CA; the identity of the member of \mathcal{A} to which this certificate belongs. (Unfortunately, the reality of Web spoofing raises some challenges to this assumption; we revisit this topic later.)

Certification Figure 2 shows an example process for a server operator to use a high-end secure coprocessor platform (such as the 4758) to install and certify a trusted co-server.

The operator first obtains a secure coprocessor platform and uses its software configuration architecture to install co-server application software $A \in \mathcal{A}$ from a co-server software vendor into this device. The co-server application then

generates a new key pair KP . The server operator uses the co-server application's ability to authenticate itself (via the coprocessor's outbound authentication API) to prove to the satisfaction of the WebALPS certificate authority that the new key pair KP belongs to an installation of A securely running on an untampered secure coprocessor platform at this server. Having evaluated the correctness of this co-server application software, the WebALPS Certificate Authority then issues an SSL-compatible certificate attesting to the public key of KP and the entity (WebALPS application A , running inside this secure coprocessor at this server) to which it belongs. The WebALPS application stores this certificate, and is then ready to participate as a trusted co-server to server operator's web application on his web server).

SSL A remote client using her Web browser initiates an SSL session with the co-server application within the secure coprocessor at the web site maintained by this server operator. Because the client's web browser indicates that the co-server application suitably demonstrates knowledge of the private key matching the public key in this application's SSL-certified key pair, the client can reasonably conclude that:

- server-client communications within this SSL session originated within the trusted co-server, and that
- client-server communications terminate inside the trusted co-server.

Since the private key in KP is safely confined inside the coprocessor, the server operator cannot know the session key and thus cannot eavesdrop or forge communications between the browser and the co-server.

The co-server can then work with the host Web server to provide enhances service to the remote client.

3.3 Solving the Fundamental Problem

Integrating WebALPS into the Web infrastructure solves the basic problem of how clients can establish trust in the computation and data storage at servers of otherwise unknown credibility.

We enumerate the (informal, for now) trust assertions.

Because of the security architecture of the tamper-responding secure coprocessors, the client Alice can trust that the only entities who can authenticate themselves as an instance of a particular type of WebALPS guardian are in fact *bona fide* WebALPS guardians of that type.

In particular, note that *the server operator cannot obtain the WebALPS private key*, since this is generated and protected inside the coprocessor.

Because she trusts the WebALPS Certificate Authority to do its job, the client Alice can trust that the only entity

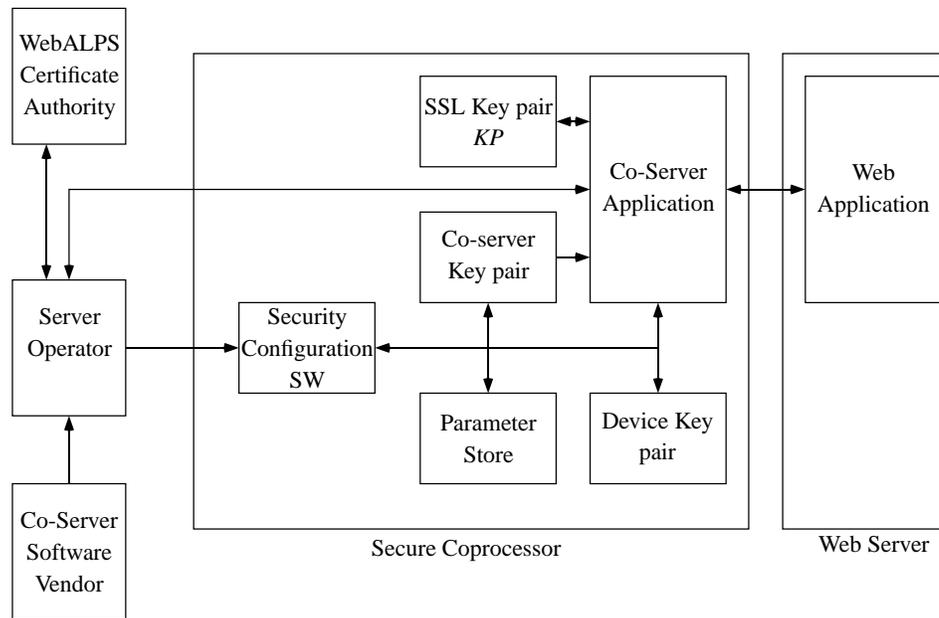


Figure 2. This figure sketches an example WebALPS installation and certification scenario. The server operator obtains a coprocessor from the coprocessor vendor, and the desired WebALPS guardian program (with all the necessary signatures) from the software vendor. The coprocessor’s security configuration software installs the guardian, which then generates an SSL key pair KP and uses the coprocessor’s outbound authentication to prove, to the CA, that this genuine guardian knows the private key in this pair. The CA then issues an SSL-compatible certificate for the public key in KP .

possessing a private key matching the public key in a WebALPS certificate for a member of \mathcal{A} is a *bona fide* WebALPS guardian of that type as well as the correctness of the services offered by this WebALPS guardian. (Essentially, to simplify the authentication process, Alice delegates the details of authenticating a tamper-responding coprocessor in a particular software configuration to the CA.)

Because of the nature of the SSL protocol, the client Alice can trust that, when she opens an SSL session to some entity, she has established an encrypted channel between her browser and an entity that possesses that private key.

Because her browser tells her unambiguously that this entity had an SSL certificate signed by the WebALPS CA, the client Alice can trust that this entity is a WebALPS guardian of that type.

Because a well-defined portion of her browser window exclusively renders material from that SSL entity, Alice can trust that the data she receives and sends via this window passes through this encrypted channel to a *bona fide* WebALPS guardian of that type.

Weaknesses Potential weaknesses of the system include compromise of the WebALPS CA that certifies the co-

server’s SSL key pair, and of the factory CA that certifies the coprocessor. Compromise of the code-signer key pairs are also a weakness, although the 4758 outbound authentication scheme would eventually allow rejection of compromised co-servers, should the code-signer compromise be discovered.

The existence of a trusted path from the co-server to the user is also an issue. As noted earlier, in a related project [21], we have demonstrated the ease with which a malicious server can spoof the existence of an SSL session, as well as the certificate data allegedly supporting such a session. We are exploring countermeasures [18]; in the meantime, prudent users should disable Javascript.

Of course, the guardian program itself needs to be sufficiently small and simple that its behavior can be reasonably verified by the WebALPS CA, and trusted by the user and other stakeholders.

Using WebALPS For each type of Web-based application that is at risk from insider attack, we can design a particular type of WebALPS guardian. For just one example, services which require clients to input sensitive data—passwords, credit card numbers, etc.—can have WebALPS strip out and

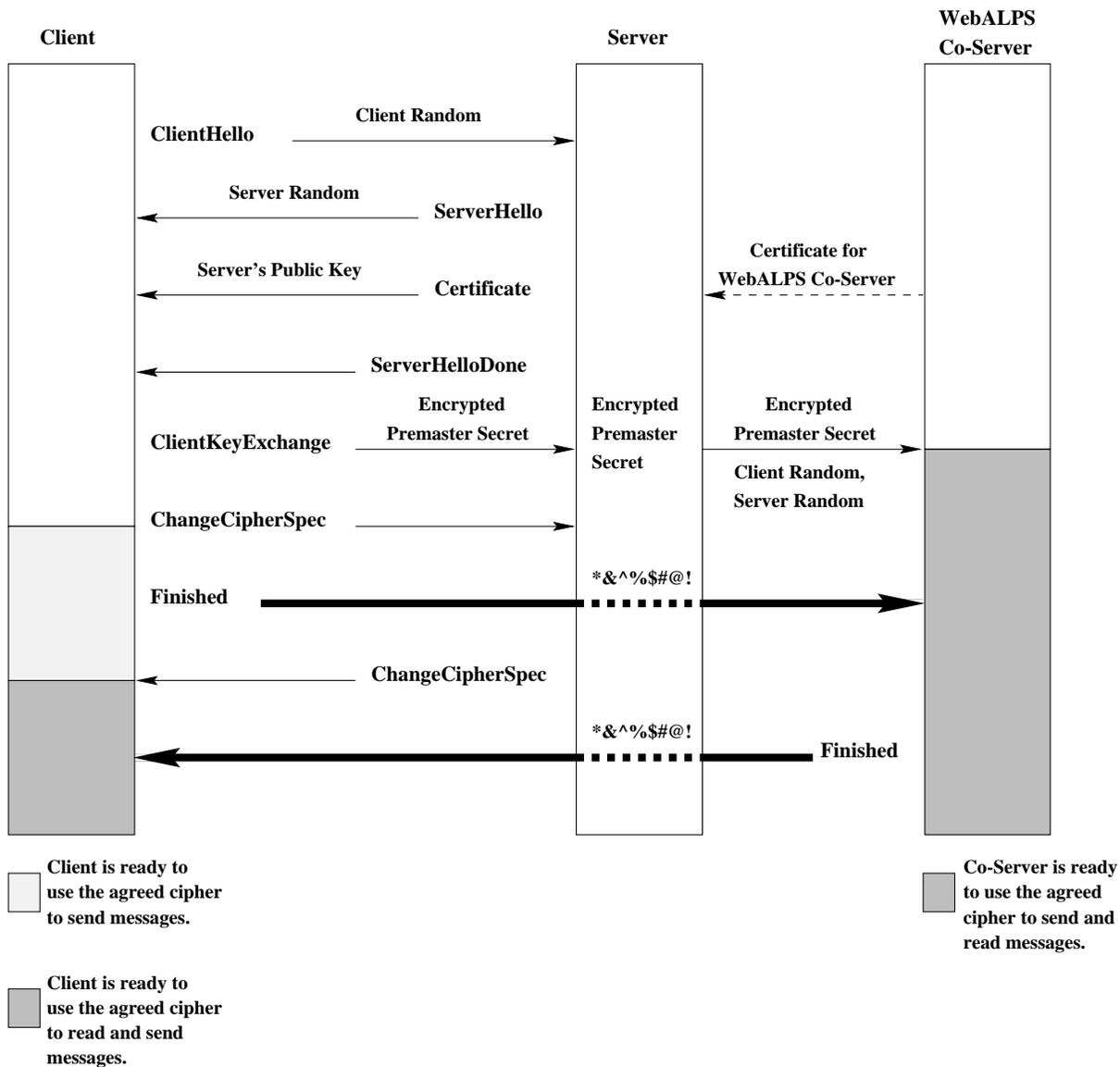


Figure 3. SSL Handshake process with WebALPS co-server. In this figure, strings that contain random characters such as “*~^%\$#@!” are used to denote illegible ciphertext and bold arrows are used to denote an established secure and authenticated communication channel. *client Random, Server Random, and Premaster Secret* are the three elements used in the shared-secret-generation process.

work with the sensitive data, before it goes to the server.

Through the above trust assertions, Web clients can validate the existence of the guardian programs and establish well-founded trust on the services provided by the servers with the help of WebALPS guardians.

4 Prototype

However, this paper does not just present a vision: we have implemented and tested this idea.

To provide secure and trustworthy Web services to Web clients, the WebALPS guardian programs have to communicate with clients through secure and authenticated channels. We designed and implemented a prototype WebALPS co-server that makes use of SSL protocol to estab-

lish such channels. We used a 4758 Model 2 platform, with the CP/Q++ system software, installed on a Linux host. The system we chose for the web server is Apache (v1.3.14)/mod_ssl (v2.7.1)/OpenSSL (v0.9.6). We used the Apache Web server because of its popularity and also because it is open-source. We tested the performance of our WebALPS-enabled Apache server and compared it with the performance of a normal SSL-enabled Apache server.

For the testing purpose of this prototype, there is no compelling reason to go through the trouble of using a realistic Web Certificate Authority. Instead, we modified the certificate-generation facility offered by the OpenSSL library to create a temporary certificate attesting to a temporary key pair generated inside the secure coprocessor for the WebALPS co-server.

Full details of the prototype can be found in [6].

4.1 SSL Sessions with Co-Servers

Handshake Process Figure 3 illustrates the handshake process with the participation of the WebALPS co-server.

Interactions between co-servers and servers are added to accomplish two goals: authenticating the co-server to the client, and establishing a shared secret between the co-server and the client. To achieve the first goal, the *certificate* message from the server to the client now contains the certificate for the WebALPS co-server generated by the secure coprocessor.

To achieve the second goal, server forwards the *Client Random* and *Server Random* messages 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. Consequently, the *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 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.

Record Layer 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. We modified the record layer in the Apache server to route every incoming SSL message from the client, as well as every outgoing message to the client, into the co-server for cryptographic operations including message encryption/decryption and MAC generation/verification.

(In this prototype, we encrypt *every* outgoing SSL message through WebALPS, so that the user need not manage two parallel SSL sessions, and so that we do not need to solve the problem of how two different streams of information—of different trust levels—can be securely multiplexed in the same SSL stream. Subsequent work [7] in our lab has begun exploring this issue.)

4.2 Performance Analysis

Performance of our system is important, because it will affect the willingness of the industrial world to accept the WebALPS approach. In particular, we are interested to see how much the communication overhead introduced by the co-server slows down the request-serving speed of the WebALPS-enabled server, and how well can the WebALPS-enabled server sustain heavy workload.

Section 4.2.1 and Section 4.2.2 address these questions, respectively. We did all tests on the same Apache server with three different virtual hosts configured to handle HTTP, HTTPS, and WebALPS-HTTPS requests respectively, using a null WebALPS application in order to measure the inherent costs of this approach. (When we prototype realistic WebALPS applications, these tests would need to be revisited.)

4.2.1 Speed

For the speed test, we used *http.load*,⁴ a free and easy-to-use tool from ACME software.

For the sake of comparison, we tested three systems including our prototype WebALPS-enabled HTTPS host, normal HTTPS host, and HTTP host. We tested each of these system by using one client which keeps sending requests for a randomly-generated file of size 2KB in a 2-second time span. We then compared the performance of these systems using three measurements including *speed* (the number of served requests per second), *connection time* (the TCP connection time for HTTP and HTTPS hosts plus the SSL handshake time for HTTPS hosts) and *request time* (the amount of time taken to process a request once the connection is established).

Table 1 shows the results from our tests. Not surprisingly, in all three measurements, the plain HTTP server offers the best performance, while the WebALPS server performs the worst. This result is expected given the overhead added by SSL and the even more overhead added by WebALPS.

However, what we really want to learn from these results is how the slowdown from HTTPS host to WebALPS-enabled host compares to the slowdown from plain HTTP

⁴http://www.acme.com/software/http_load/

	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 1. Speed test and comparisons of WebALPS host, normal HTTPS host, and HTTP host

	Speed (times)	Connection Time (times)	Request Time (times)
Slowdown caused by SSL (HTTP → HTTPS)	11.68	2.46	14.6
Slowdown caused by WebALPS (HTTPS → WebALPS-HTTPS)	5.85	0.14	6.1

Table 2. Comparisons of slowdowns caused by WebALPS with slowdowns caused by SSL

host to HTTPS host, and which phase (connection or request) is the major factor for the slowdown.

The data in Table 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 0.14 times 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.

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 as a method for securing e-commerce and other service-related applications. First, this usage of SSL offers a secure communication channel to an authenticated server—essential to many applications. Second, communications that go through SSL channel only occupy a very small percentage of total web 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 the WebALPS-enabled server. The kind of security and privacy that WebALPS co-server offers will solve the trust problems existing in a wide range of applications that are hard or even impossible to address otherwise. 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. (Convincing users who are only concerned about the final performance slowdown, and not about the relative

cost-benefit for what they gain from what they pay, is another issue.)

From Table 2, there is little difference (0.14 times) 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. (One might conjecture that using standard cryptographic acceleration with a non-WebALPS server would offer a similar speed-up.)

However, for request time, WebALPS-enabled Host is 6.1 times slower than normal HTTPS host. The time-limiting factor could be the bulk data transfer rate between the host and the coprocessor, and/or bulk data DES operation and MAC calculation rate in the coprocessor. The exact reason is still not clear. We are devising more elaborate tests to pinpoint the exact bottleneck here.

4.2.2 Scalability

For scalability testing, we chose to use WebBench⁵, a commercial-grade, web server benchmarking product from Ziff eTesting Labs Inc that enables creating large test suites using standardized (and allegedly realistic) test workloads.

We tested our prototype WebALPS system and the normal HTTPS Apache host using the standard e-commerce test suite provided by WebBench. This test suite uses a e-commerce workload with balanced HTTP (92%) requests and HTTPS requests (8%). During the test, we used a maximum number of 8 test clients with 3 threads per client. We then made two measurements—the request-serving speed (requests/second) and the throughput (bytes/second).

Figure 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

⁵<http://www.etestinglabs.com/benchmarks/webbench/home.asp>

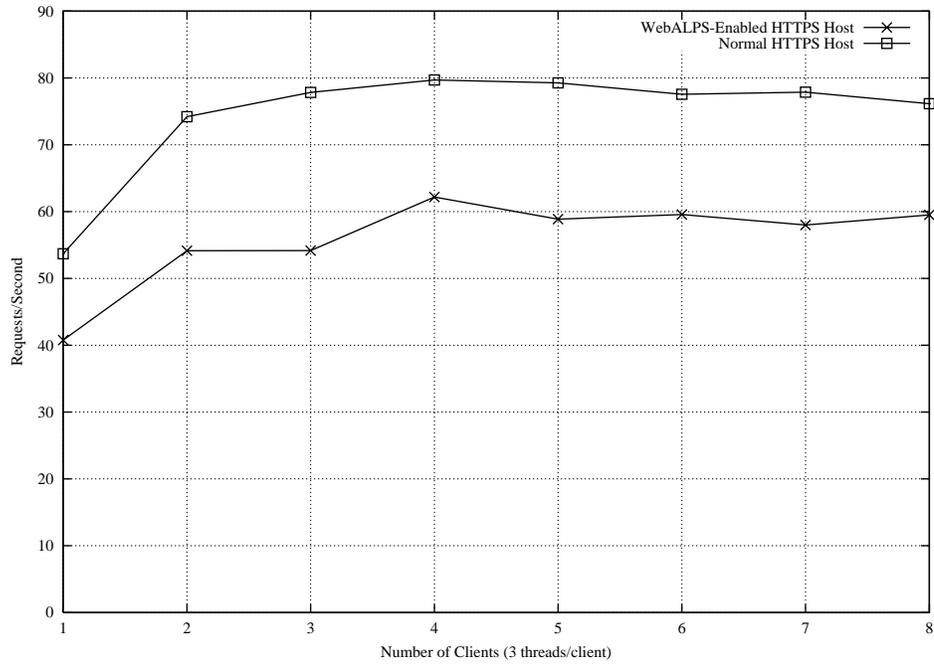


Figure 4. Scalability comparisons between a WebALPS-enabled HTTPS host and a normal HTTPS host (Requests/Second)

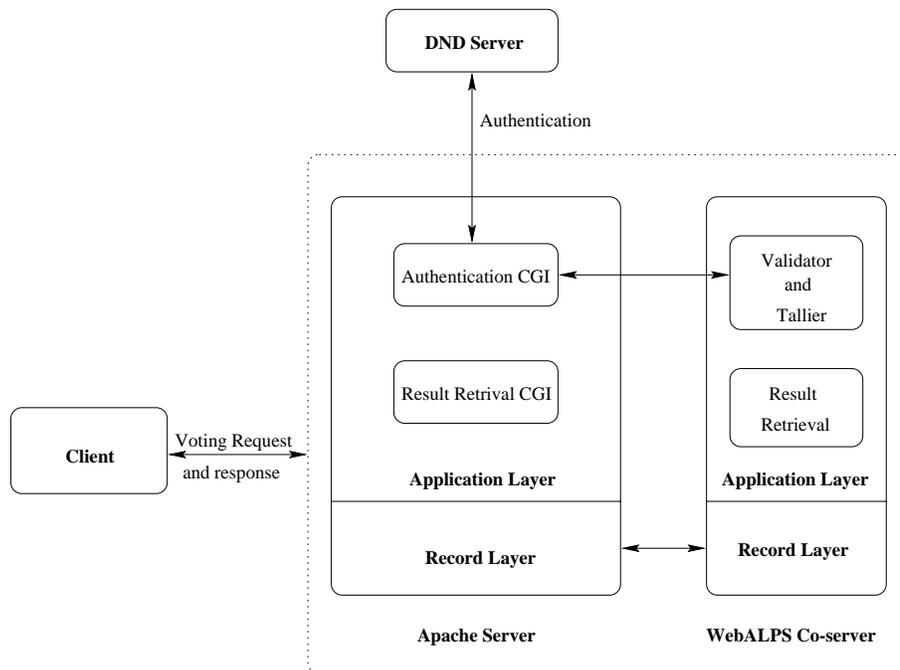


Figure 5. The structure of our demonstration voting system

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.

The result we obtained by measuring throughput is similar to the result when measuring speed (and the figure corresponding to Figure 4 can be found in [6]).

From these data (and assuming the WebBench suite provides a realistic workload), we can safely conclude that within our test range, WebALPS-enabled server are scalable under real-life e-commerce workloads.

Another thing we want to point out is that in this test, with the introduction of the non-secure components in the workload that represents real-life scenarios, the performance slowdown of WebALPS-enabled server has reduced from over 500% (as shown in Section 4.2.1) to only about 25%. This supports our previous argument that to users in real life, the performance slowdown of WebALPS approach would not be as dramatic as what Table 2 shows.

5 A Test Application

To demonstrate the feasibility of the WebALPS approach and the advantage it offers, we decided to build a simple application on top of the implemented prototype. We chose an E-voting system for this purpose because privacy/voter anonymity is one of the most important requirements for such applications [1, 9].

To protect the privacy of voters, it is often mandatory to assure that nobody, including the election authorities, can trace a vote back to the voter who cast it. Existing E-voting systems often employ complicated protocols to ensure this property [4, 10, 12]. In essence, the the problem here stems from the risk of insider attack at the server systems that are used for voting. With the introduction of trusted WebALPS co-servers, the task of ensuring the voter's privacy is greatly simplified.

Figure 5 illustrates the structure of our voting system. In the current implementation, we assume all the voters are users registered in our site's Name Directory (DND) and we use the voter's DND passwords for authentication. A voter casts his vote by sending it together with his name and password to the WebALPS co-server. The voting module running at the application layer inside the WebALPS co-server authenticates the user through the help of an authentication CGI program and the DND server, validates the vote, and counts the vote if the validation succeeds. During the whole process, the voter's sensitive information—the vote and the password—never leaves the secure coprocessor. Therefore, the voter can trust his privacy has been preserved—even if the server operator might be motivated to subvert it.

Discussion of additional applications can be found in [14].

6 Conclusion

In this paper, we presented an application of secure co-processing and cryptography that solves many manifestations of a real trust problem: the ability of insiders at Web servers to learn and change sensitive data and computation that clients depend on. We described our implementation and performance testing of this idea, and a simple E-voting application we built that further validates it.

Our performance analysis suggests 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. When tested with standard e-commerce workload benchmarks, the prototype demonstrates good performance and scalability that are comparable to what a normal SSL-enabled Apache server offers. We are confident that based on the secure features offered by secure coprocessors, the trusted WebALPS co-server will become a critical piece of enabling technology for security and privacy in Web-based services.

Many areas remain for future work.

First, we want to produce an end-to-end implementation of this idea. This requires extending our prototype to generate a key pair once, and setting up the WebALPS CA to authenticate and certify these key pairs. Whether the CA should use identity certificates—or whether we should develop a way to express these hardened server properties via attribute certificates—is an interesting question.

We also want to build WebALPS guardian applications for many of the more compelling examples in [14]. Ease of implementation will require some way to easily identify which pieces of the client-server and server-client traffic the guardian should re-write; extending HTML seems one natural approach. Once we have a real application in a live setting, the questions of performance measurement—and of “realistic workload”—will need to be revisited.

Our WebALPS prototype was built on the current production CP/Q++ operating system shipped with the 4758-2; the imminent release of Linux [5] for the 4758 could streamline development and experimentation (although out-bound authentication support would have to be added, for secure deployment).

Consideration of the full range of applications also raises some additional research questions. For example, most Web users probably are not even aware of what CAs their browser accepts, what this certification means, or what cryptography their browser has been configured to accept as suitable in SSL sessions. Of course, extending browser and web technology to meaningfully communicate certifi-

cate semantics—and server attributes and delegation—is arguably a necessary next step, even without WebALPS.

Acknowledgments

The authors are grateful for helpful discussion and suggestions from D. Chess, J. Dyer, N. Itoi, J. Kravitz, E. Palmer, A. Perrig, R. Perez, D. Song, and E. Ye, as well as advice from the referees.

References

- [1] L. Cranor. Electronic voting—computerized polls may save money, protect privacy. *Crossroads*, 2(4), April 1996.
- [2] J. Dyer, R. Perez, S. Smith, and M. Lindemann. Application support architecture for a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*. National Institute of Standards and Technology, October 1999.
- [3] A. O. Freier, P. Karlton, and P. C. Kocher. The ssl protocol version 3.0. <http://home.netscape.com/eng/ssl3/draft302.txt>, November 1996.
- [4] A. Fujioka, T. Okamoto, and K. Ohta. A practical secret voting scheme for large scale elections. In *Advances in Cryptology - AUSCRYPT '92*, pages 244–251, 1993.
- [5] Ibm research demonstrates linux running on secure cryptographic coprocessor. Press release, August 2001.
- [6] S. Jiang. Webalps implementation and performance analysis: Using trusted co-servers to enhance privacy and security of web interactions. Master's thesis, Department of Computer Science, Dartmouth College, June 2001. (Available as Technical Report TR2001-399). <http://www.cs.dartmouth.edu/~pkilab/shan-thesis.ps>.
- [7] E. Knop. Secure public-key services for web-based mail. Senior thesis, Department of Computer Science, Dartmouth College, August 2001.
- [8] Mudge. Security devices that might not be. *USENIX Login*, November 2000.
- [9] P. Neumann. Security criteria for electronic voting. In *16th National Computer Security Conference*, 1993.
- [10] H. Nurmi, A. Salomaa, and L. Santeau. Secret ballot elections in computer networks. *Computers and Security*, 36:553–560, 1991.
- [11] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison Wesley Longman, 2000.
- [12] A. Salomaa. Verifying and recasting secret ballots in computer networks. *New Results and New Trends in Computer Science*, pages 283–289, 1991.
- [13] S. Smith. Outbound authentication for programmable secure coprocessors. Technical Report TR2001-401, Department of Computer Science, Dartmouth College, March 2001. <http://www.cs.dartmouth.edu/~pkilab/oatr.pdf>.
- [14] S. Smith. Webalps: A survey of e-commerce privacy and security applications. *ACM SIGecom Exchanges*, 2(3), September 2001.
- [15] S. Smith, R. Perez, S. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*. National Institute of Standards and Technology, October 1999.
- [16] S. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, April 1999. Special Issue on Computer Network Security.
- [17] S. Weingart. Physical security for the μ abyss system. In *Security and Privacy*. IEEE, 1987.
- [18] E. Ye. Securing browsers. Master's thesis, Department of Computer Science, Dartmouth College. In progress.
- [19] B. Yee. *Using Secure Coprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1994. (Available as Technical Report CMU-CS-94-149).
- [20] B. Yee and J. Tygar. Secure coprocessors in electronic commerce applications. In *1st Electronic Commerce Workshop*. USENIX, 1996.
- [21] Y. Yuan, E. Ye, and S. Smith. Web spoofing 2001. Technical Report TR2001-409, Department of Computer Science, Dartmouth College, July 2001. <http://www.cs.dartmouth.edu/~pkilab/demos/spoofing/>.