Surf'N'Sign: Client signatures on Web documents

by A. Herzberg
D. Naor

The emergence of World Wide Web-based systems and Web transactions has led to the need to find a mechanism that provides electronic signature capabilities as a replacement for written signatures. Such a mechanism should guarantee authentication and nonrepudiation. Many Web applications could benefit greatly from such a mechanism, e.g., banking systems, tax filing, reservation systems, and corporate procedures. This paper discusses the various approaches that could be taken to provide such a mechanism and suggests a solution that provides client commitment on Web documents by means of digital signatures. The architecture and implementation of the solution, called Surf'N'Sign, is outlined in detail. Our design of the solution gives special consideration to the semantics of such a signature and to its proper and secure use on the Web. Its prototype was implemented at the IBM Haifa Research Laboratory as a plug-in to the Netscape Navigator™ browser and is integrated naturally into the browsing process. It provides a signing mechanism at the client, as well as the capability to archive and preview the signed documents. Surf'N'Sign lends itself to easy integration with existing applications on the Web.

In paper-based transactions, handwritten signatures are used to authenticate the document, to serve as the signer's agreement to the information it contains, and also as evidence that can be shown to a third party in case of repudiation. With the emergence of World Wide Web-based systems generating Web transactions, the need for a function and a mechanism analogous to a written signature becomes imminent. Examples of such systems include, among others, banking systems, tax filing, reservation systems (car, travel, etc.), and corporate procedures involving expenses, evaluations, etc. Such systems are *fully* Web-enabled only if some type of signature capability is integrated into them. The signature capabilities are needed either at the client or at the server.

Digital signatures (a.k.a. "electronic signatures") are the digital equivalent of handwritten signatures. Digital signatures provide data that can serve as evidence that the signer agreed on some digital message. Such data should be "easy" to produce and verify, but "difficult" to forge by anyone other than the signer, and can be used as a proof to a third party. The concept of a digital signature was first introduced in the classical paper of Diffie and Hellman,[1] and the first—and most well-known and widely used—implementation is RSA.[2] The legal state of digital signatures is currently not so clear, and adequate legislation concerning digital signatures has not yet been introduced. This situation is likely to change as digital signatures become more widely used (see, for example, Utah's Digital Signature Development Program[3]). However, digital signatures clearly offer the closest—or only—known alternative to handwritten signatures when moving paper processes to the Web.

Digital signatures have been successfully incorporated into a large number of electronic applications,

in particular, electronic mail,[4–6] groupware and workflow (e.g., Lotus Notes**), and electronic commerce (e.g., References 7 and 8). Incorporating the digital signature mechanism into Web technology is now called for. Such extension requires proper design of the integration between the algorithm that computes digital signatures and the Web interface and implementation. This paper considers a particular scenario where the client needs to commit to a Web document; the document may be a purchase order, a contract, or any other committing statement. The paper discusses the issues and difficulties involved in extending digital signatures to such scenarios and presents the architecture and design of a solution, called Surf'N'Sign, that provides client commitments on Web documents by means of digital signatures.

Digital signatures and the Web. The first question that needs to be addressed when designing a signature mechanism for the Web is the exact definition of the "message," or the document, that is being signed. The appearance of Web documents, unlike physical documents, is a function of various parameters besides its HTML (HyperText Markup Language) representation. However, the main goal of Surf'N'Sign is to provide the client with a mechanism that allows him or her to commit to the document appearance at the time of the signature, and as a result it adopts "what you see is what you sign" semantics. It is natural to sign the "document source" of a Web document, but then it is not clear to what extent the document representation (HTML source in our case) reflects the Web document that is seen by the signer at the time of signing. Web documents, written in the HTML language, are typically complex hypertexts composed of several types of data such as plain text, images, links, and executables (JavaScript** procedures, Java** applets, and ActiveX** controls).

Links pose only a semantic problem: does the mechanism sign the link name only, or does it sign the content of the link at the time of signing? If the latter approach is taken, the content of the uniform resource locators (URLs) pointed to by all links should be signed as well, and this must be done recursively. Surf'N'Sign adopts the first approach: documents containing links are signed, but the link content is not included in the signed message.

Some components, such as *images* or *frames*, may change with time so that the signature on the document source no longer reflects the altered document. Moreover, some images, especially *background*, may be abused by the creator of the document: it may hide some essential parts of the document and thereby completely change the meaning of the document. In order to protect the client from such situations, Surf'N'Sign filters out HTML documents that include "changeable" components such as embedded images, background, and frames and refuses to sign them. An alternative approach could be to sign the "changeable" components along with the document source. This solution protects the client at a time of dispute but introduces additional complexity and burden on the server administrator, who must keep track of the original embedded images to prove the authenticity of the signature in the future.

Signing *active content* of the document, such as JavaScript procedures and Java applets, is dangerous if one wants to commit to the *view* of a document generated at the time of signing. This view may depend on such things as time, user input, and others (for example, a JavaScript procedure may display different amounts of money depending on the time of day). To avoid such situations, Surf'N'Sign filters out HTML documents with active content since it does not intend to commit to the *behavior* of such components. We should note that code signing may be very meaningful in other scenarios, for example, when certain assertions about the code are signed with it. This approach is taken by the Digital Signature Initiative,[9] for example.

A long document may not be contained in the screen viewed by the signer. One may permit long documents (and rely on the signer to scroll the display to read all of the contents). We take the more conservative approach—which we found appropriate to use on the Web, since there has not been sufficient public experience with digital signatures—and restrict the length of the signed documents.

Finally, in addition to its complex nature, the document is viewed by a specific browser with a specific interface that may affect the outlook of the document. In turn, Surf'N'Sign considers the identity and version of the browser as part of the signed message.

Complying with the nature of the Web, the new mechanism should be well-integrated with the common browsing process, with its user interface, simplicity of use, and speed, thus minimizing delay caused by the signature computation. In addition, the open interconnection nature of the Internet im-

poses real threats to any application that requires both authentication and "unforgeability." Some of these issues are addressed by general protocols that provide communications privacy over the Internet, such as Netscape's SSL (Secure Socket Layer)[10] protocol. Other issues, such as ensuring that the software indeed produces the real signature, must be addressed separately.

Signing and verifying with public-key cryptography. A digital signature is the process that transforms a message M into a signed message S. If S_X is the secret signing transformation of a user X, then for any message M in the message-universe, $S = (M, S_X(M))$ is the signature of user X on M. The process of verifying a signature is a *public* transformation V_X, which, when applied to S, returns True or False: $V_X(S, M)$ = True if and only if the signature originated from the message M. Also, it is computationally infeasible for any user other than the one using S_X to compute M and S such that $V_X(S, M)$ = True. The private and public transformations S_X and V_X are performed with a key K_X of user X that consists of a private and public component. The association between the signer's ID (identification) and her or his public key is done via a *certificate*, which is issued to the signer by some *certificate authority* (CA)[11] or by a certification mechanism internal to the application. A certificate is basically a signature on the pair (ID, K_X) and on some assertions where ID is the identification of the person who holds the key K_X. A common implementation uses the RSA[2] signature algorithm for the signature scheme and MD5 as the One-Way Hash Function.

Review of existing solutions. Several general security protocols exist for the Internet. The SSL protocol[10] of Netscape provides secure communication between the server and the client, but this by itself does not provide the nonrepudiation features for the signature. S/MIME (Secure/Multipurpose Internet Mail Extension), PEM (Privacy-Enhanced Mail), and PGP (Pretty Good Privacy)[4–6] are designed mainly to sign e-mail messages, and can be adapted, if desired, to sign Web documents. However, this adaptation has not yet been done. S-HTTP (Secure Hypertext Transfer Protocol)[12] does provide a signature mechanism. However, it requires that the browser use S-HTTP. Currently, this protocol is not widely supported in the market (i.e., by neither Netscape Communications Corporation nor Microsoft Corporation). Shen, an alternative security scheme for the World Wide Web,[13] does not specify nonrepudiation among its immediate goals. There are several server-oriented signature mechanisms. The Digital Signature Initiative,[9] carried out by the World Wide Web Consortium, suggests a mechanism that is designed to " . . . provide a comprehensive solution to the basic problem of helping users decide what to trust on the Web." The Signature Labels of the Digital Signature Initiative sign *assertions* regarding a URL together with its content, and its main application is code signing. Another protocol, the S[3] Server-Supported Signatures protocol, is suggested in the paper by Ashokan et al.[14] It presents a nonrepudiation scheme in which signature generations are done at some designated *signature server*. This technique has not yet been implemented.

A Web signature mechanism may be designed to be performed by either the server or the client.

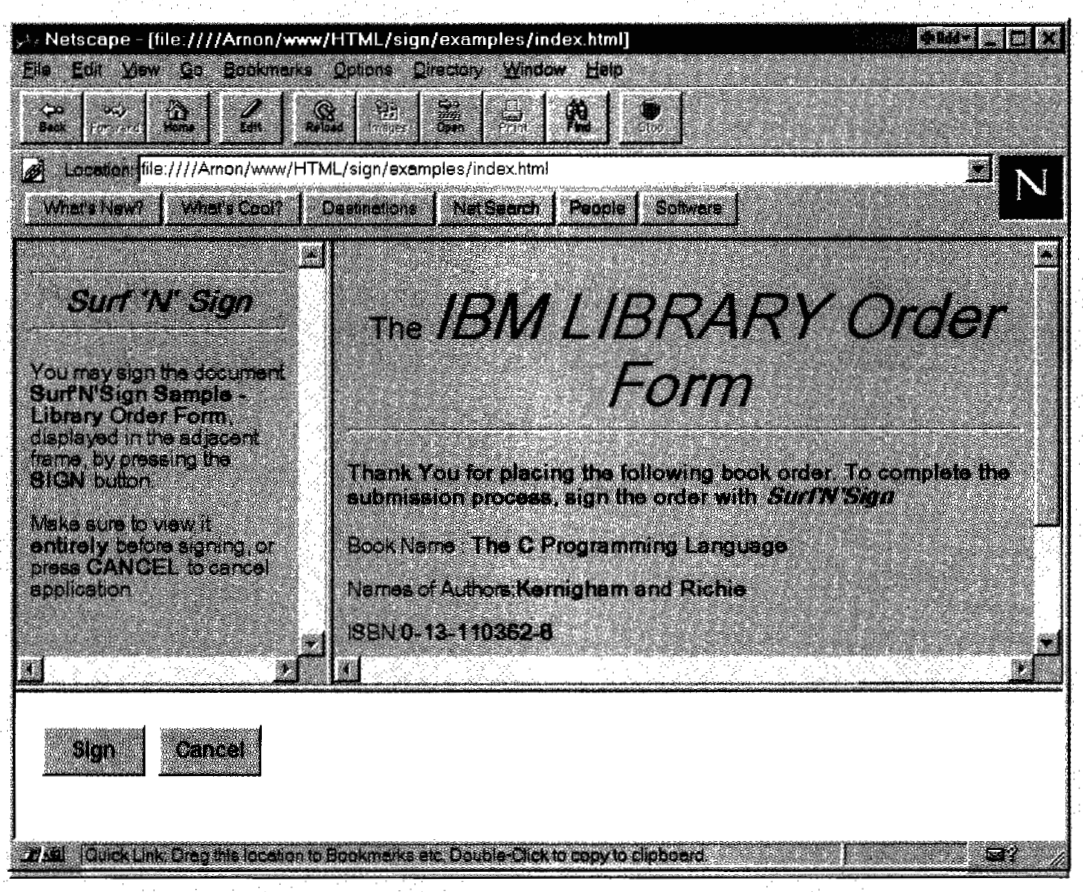
Surf'N'Sign

Surf'N'Sign is designed to provide a mechanism to support *client commitment on Web documents* by the means of digital signatures. As such, it emphasizes clarity and simplicity, thus avoiding possible misunderstanding at the client's end, while providing adequate authentication of the signer, nonrepudiation, and the ability to prevent the forging of signatures. It is also designed to be an easy-to-use mechanism that becomes an integral part of the surfing process and, as much as possible, adopts the interface of the browser. The signature semantics adopted by Surf' N'Sign have been described and justified in detail in the previous section.

Characteristics of Surf'N'Sign. Some of the main features of Surf'N'Sign are now described.

Security and trust model. In general, a Web signature mechanism may be designed to be performed either by the server or by the client. However, for client-based signature applications, the signature mechanism that has access to the signer's sensitive data (i.e., private key) and that results in a commitment of the client, should be fully trusted by it. Hence, it is log-

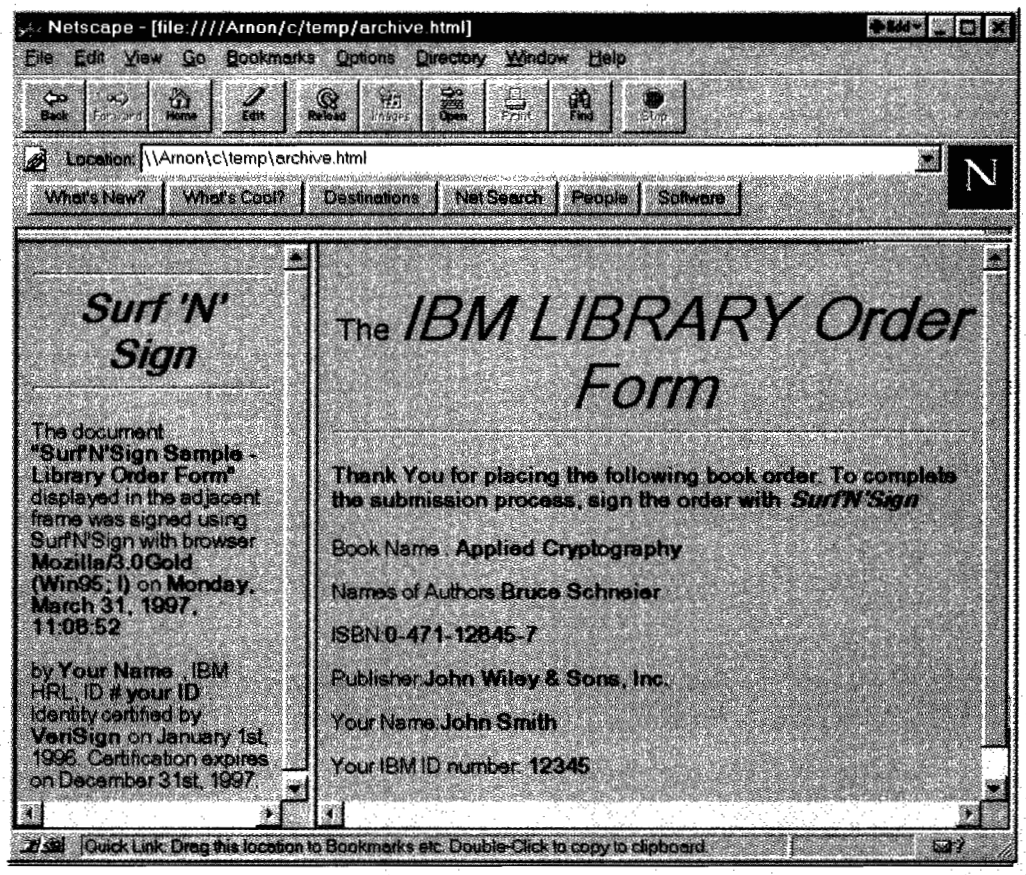
Figure 1 Main screen of Surf'N'Sign, displaying document to be signed with sign and cancel buttons



ical to require that the software performing the signing operation must reside *at the client* and be a trusted piece of code *independent* of the server that contains the document to be signed. For that reason, performing the signature at the server, by a Common Gateway Interface (CGI) for instance, is unacceptable. Another natural alternative is loading a Java applet from the server, while exploiting the trust mechanism of a signed applet supplied by the Java language. The problem with this solution is having access to the private key data of the signer—it will require the private data to be sent repeatedly to the server. Moreover, the Java applet is not independent of the document that is being signed, since it is the same server that supplies both the signing code (applet) and the document. As a result, we have chosen to implement Surf'N'Sign as a plug-in to Netscape Navigator**—the exact architecture of the system is explained in detail later. By this method we achieve the goal of having the signature computation done locally at the client by code independent of the server.

Implementation. Surf'N'Sign is implemented as a plug-in to the Netscape browser, which interacts with the server by a CGI script. As such, it uses a common browsing mechanism and interface. It consists of the *signature protocol*, which lets the client sign a document and send the signature to the server. This method requires one public-key operation that causes a negligible delay. Another component is the

Figure 2 Browser window displaying an archived signature

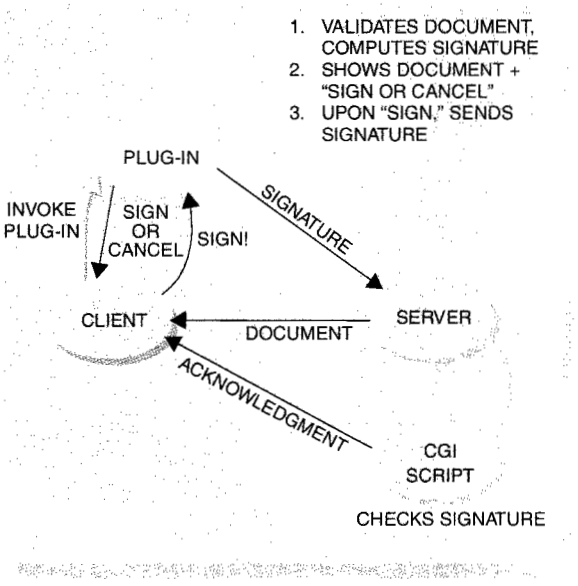


signature retrieval, which displays an HTML document that has been signed with Surf'N'Sign. The digital signature scheme employed is the RSA signature algorithm[2] and MD5 for computing the message digest (the One-Way Hash Function). All components are written in C or C++, developed with Microsoft Developer Studio 4.0** and running on Windows 95** or Windows NT**. The cryptographic operations (signing a message and verification of the signature) of Surf'N'Sign are implemented with the RSAREF 2.0** cryptographic library,[15] developed by RSA Laboratories.

Handling HTML forms. Surf'N'Sign is designed to sign a plain HTML text; however, it is often desirable to sign an HTML form after its fields have been filled by the client. The current implementation of Surf'N'Sign can be easily adapted to handle HTML forms in the following way: the HTML form is processed by a CGI program that dynamically creates a new, temporary HTML document. This document contains the user-specified values at the fields and is then signed by Surf'N'Sign.

Use of Surf'N'Sign. As indicated, Surf'N'Sign is designed to sign HTML documents at the client and send the signature back to the server (such as to a bank or to a car-rental company). The document is typically retrieved from the site of the server. The process of signing an HTML document with Surf'N'Sign

Figure 3 Architecture of the signature protocol



is very simple. The document to be signed is displayed in the window of the browser as shown in Figure 1. This window is split into three parts (frames): the first part (at the left) is the "Surf'N'Sign control window," the second part (at the right) displays the original document to be signed, and the bottom part contains two buttons, a *sign* button and a *cancel* button. Figure 1 shows a snapshot of the browser window and its three parts. Pressing the cancel button aborts the program, so that no signature is produced or sent to the server. When the sign button is pressed, a signature is produced and is sent to the server. In return, the server sends an acknowledgment that is displayed at the Surf'N'Sign control window frame, indicating that the signature has been received, verified, and is being archived at the client's local disk.

An archived signature is displayed by the browser as shown in Figure 2. The window of the browser is split into two parts: one that displays the document that has been signed, and the "control" frame that displays the date and time when it was signed and the signer's identity. Figure 2 shows a snapshot of the browser window displaying an archived signature.

Architecture of Surf'N'Sign. Netscape Navigator plug-ins are dynamically loaded code modules (dlls) that become part of the code of the browser. As such, they allow a simple integration between the application and the browser. Surf'N'Sign is implemented as a plug-in[16] in the client's browser using Netscape's application program interface (API) and interacts with the server via a CGI script. The signature protocol and the signature retrieval are implemented by the same plug-in program. The plug-in is registered by the Navigator browser using the MIME-type *application/x-sig* and is called whenever a file with the extension of ".sig" is being displayed by the Netscape browser (Netscape Navigator 3.0 or later). Instead of displaying it directly, the plug-in program checks the content of the file. If the header (first word) of the file is the keyword "ARCHIVE_Surf NSign," it is processed by the signature retrieval program. Otherwise, it treats the file as a standard HTML document that needs to be signed, and the file is processed by the signature protocol. The server's component of the signature program is a CGI script written in C.

The signature protocol. Surf'N'Sign implements a simple protocol that runs between the client and server. The protocol first displays the document at the client and asks for approval. When the client approves, the digital signature of the document is computed and sent to the server (the process is summarized in Figure 3). Upon receipt, the server sends an acknowledgment back to the client, and a record of the signature is kept at both. The message that passes from the client to the server contains the document's signature (signed with the client's private key), the client's public key and its certificate, date, and time of the signature, the browser's version, and the document's title (serves as identification for a document). We call it "The Signature Message."

The flow of the client's signature protocol is as follows:

- Filtering: Checks whether the HTML document can be signed (e.g., does not contain embedded texts such as images and frames). If not, an error message indicating the reason for not signing the document is displayed, and the program exits.
- User's approval: The valid document is displayed by the browser, along with a frame that asks for the client's approval (sign or cancel button).
- Signature generation and transmission: The client's public key is found and read (if a key pair does not exist, a new one is generated), and the signature of the text is computed. A signature message containing the signature, key, time and date, title, and browser used is sent to the server by the POST method. (The POST method could have been replaced by creating an HTML form with a submit

button. The reason for choosing the direct way of implementing POST is given later in the section on network and streams.)

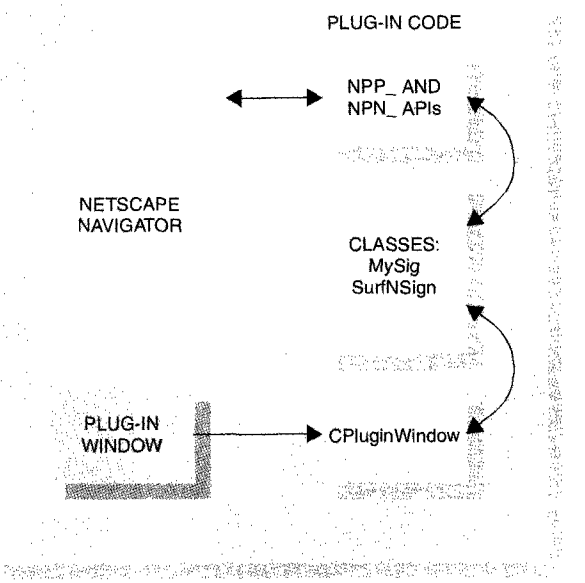
- Archiving: When the program is notified of successful transmission to the server, it writes the HTML document into a ".sig" file on the client's disk and attaches the message containing the signature and the other relevant information. From this file, the signed document can be retrieved in the future (see signature retrieval).

The flow of the server's signature program follows:

- Signature verification: The server accepts the message, decomposes its fields, identifies the appropriate HTML text by the title, and performs signature verification. This also verifies that the HTML document has not been corrupted in the midst of handling.
- Signature acceptance or rejection: If the signature has been successfully verified, the server writes the HTML document into a ".sig" file and attaches the message to it, so that the signed form can be retrieved in the future. An acknowledgment message is sent back to the client. Otherwise, the server sends an error message to the client, indicating that the signature cannot be verified and therefore has not been accepted.

The signature retrieval. The signature retrieval component of Surf'N'Sign is a utility for displaying a signed document that was produced by Surf'N'Sign. Recall that a successful signature protocol of Surf'N'Sign produces a file that contains a signature message on an HTML text. The signature retrieval does the "inverse" operation of Surf'N'Sign: it takes the file, decomposes it, verifies the signature using the public key, and if the signature verification has been successful, it displays the HTML document along with the rest of the information. It can be used for archival purposes both at the client and at the server. That is, the signed file is kept in some directory at the client and at the server, and these files can later be viewed. The input to this program is the name of a ".sig" file, produced by the signature program of Surf'N'Sign, containing the keyword "ARCHIVE_Surf NSign," followed by an encoded signature message (i.e., an HTML document, its signature, the signer's public key and its certification, date, and time of the signature, the browser version, and document title). The program reads the various fields from the file and verifies the signature (to avoid forgery of such files). If successfully verified, the program displays the HTML document along with the time and date of the signature, browser information, title, and the certification information. Otherwise, it displays an error message.

Figure 4 The plug-in code and Netscape Navigator



Software implementation

Recall that Surf'N'Sign consists of two components: the signature protocol, which lets the client sign a document and send its signature to the server, and the signature retrieval, which displays an HTML document that has been signed with Surf'N'Sign. The signature protocol is implemented by the plug-in NPsignature that recognizes the extension ".sig" and interacts with a CGI program, new_sig.exe, at the server. Signature retrieval is handled by the same plug-in.

NPsignature—the DLL program. The plug-in NPsignature is composed of three main classes: MySig, Surf NSign, and CPluginWindow. Table 1 shows the class definitions, and Figure 4 depicts the way by which they are integrated with Netscape's browser. The main class containing the data is the Surf NSign class. MySig is the main plug-in class, integrating the data with the window of the plug-in and with Netscape's information. CPluginWindow is the plug-in window—it is subclassed to the Navigator's window. These classes are embedded within the APIs provided by Netscape to produce the plug-in.

Table 1 The main classes SurfNSign, MySig, and CPluginWindow

```
class SurfNSign { // Signature message
public:
   SurfNSign (const Signature & HtmlSigEqto,
   const Public_key & PublicKeyEqto,
   const char *HtmlText, int HtmlTextLen,
   const char * HtmlTitle,
   const char *BrowserVersion);
   ...
private:
   Signature HtmlSignature;
   Public_key PublicKey;
   Certificate certificate;
   char HtmlText [MAX_DOC_SIZE];
   int HtmlTextLen;
   char HtmlTitle [TITLE_LEN];
   char BrowserVersion [VERSION_LEN];
   char TimeOfSignature [TIME_INFO_LEN];
};
```

```
class MySig {//integration of signature, window and browser
public:
   MySig (NPP instance);
   SurfNSign*      pRecord;
   CPluginWindow* pWindow;
   NPP             instance;
   void CreateSurfNSignRecord (
      const Signature & HtmlSigEqto,
      const Public_key & PublicKeyEqto,
      const char *HtmlText,
      int HtmlTextLen,
      const char * HtmlTitle,
      const char *BrowserVersion);
   void WriteError (char *error_message);
   void ArchiveSignature (char *FileName);
   int  UnArchive_File (char * text);
   void Display_Signature ();
};
```

```
class CPluginWindow : public CWnd
{ // Plugin Window
protected:
   MySig* pSig;
public:
   CButton * OKButton;
   CButton * CANCELButton;
   CPluginWindow (MySig *pSig);
   void CreateButtons();
   void CleanButtons();
   afx_msg void OnSign();
   afx_msg void OnCancel();
   DECLARE_MESSAGE_MAP()
};
```

Netscape's APIs. The use of Netscape's APIs[16] by the plug-in is now described.

Network and streams. The plug-in is invoked upon submitting a ".sig" file. It creates a new window in the function NPP_SetWindow and makes the new window a subclass of the browser's window. The content of the file is passed to the plug-in as a stream and is read in NPP_Write. The plug-in then checks whether the document can be signed. If so, the text is displayed in the Netscape browser using NPN_NewStream and NPN_Write, and two buttons (sign and cancel) are created in the plug-in window. When the sign button is activated, the plug-in posts the signature message to the CGI using NPN_PostURLNotify. In return, the plug-in is notified of the communication results in NPP_URLNotify.

A successful transmission results in keeping a record of the signature at the client, whereas any early termination of the plug-in (such as use of the cancel button or a nonvalid HTML file) results in a return of a negative number (return(−1)) in NPP_Write, which in turn destroys the plug-in. NPP_Destroy causes the plug-in window to unsubclass. An alternative, more direct, method for sending the signed message to the CGI of the server via an HTML form had been explored; namely, the plug-in will generate an HTML form containing the data to be signed. By using the submit button of the form, the data could be sent to the CGI of the server. However, this solution required the plug-in to be running just before the data of the form are sent to the server, activating a JavaScript procedure on submission from within the plug-in, which in turn required the plug-in to be "live-connected," namely to be able to call a JavaScript procedure from within the plug-in. Instead, we have chosen to implement the POST method directly from the plug-in (using the NPN_PostURL method), and in that way bypassing the live-connect requirement that is supported only by Netscape.

Frames and windows. Recall that the initial window of Surf'N'Sign (Figure 1) partitions the browser window into three parts. This partitioning is schematically depicted in Figure 5: the bottom part is the plug-in window that contains the sign and cancel buttons. The upper part is again partitioned into two frames: the Surf NSign control frame at the left that displays the message of the program, and the frame at the right that simply displays the document. For that, the browser window is partitioned into the two frames "Surf NSignWin" and "Surf NSigndummy" before the embedded call to the plug-in. The plug-in targets its output in NPN_Write to the frame Surf NSignWin. Surf NSigndummy, the plug-in frame, becomes the window containing the buttons.

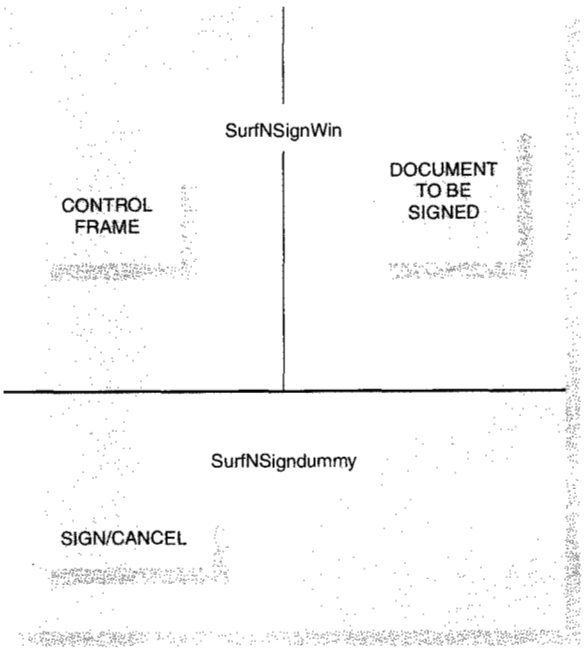
To support the setup of these frames, four files must be initially created: index.html, main.html, null.html, and Call_document.html. The index.html file defines the frames SurfNSignWin and SurfNSigndummy, where Call_document.html defines the plug-in window and its size by the EMBED command. Also, the document to be signed should have a ".sig" extension (for example, document.sig). The files are given in Table 2.

In principle, the file main.html should contain a number of links to Call_doc1.html, Call_doc2.html, etc., each containing an embedded call to a plug-in with a different ".sig" file (doc1.sig, doc2.sig, etc.). This corresponds to a situation where a server offers a number of documents that can be signed by the client (current implementation cannot handle more than one document to sign).

Archiving Surf'N'Sign signatures. An archived Surf'N'Sign signature is a file that begins with the string "ARCHIVE_SurfNSign" and contains an encoding of a Surf'N'Sign object (namely, the HTML text, browser version, date and time, signature, public key and its certificate). The file is read by the same plug-in NPsignature, hence its extension must be ".sig". When the plug-in receives the stream containing the file in NPP_Write, it checks whether it is an archived file. If so, the stream is decoded and read into a Surf'N'Sign object, the signature is verified, and the HTML text is displayed on the browser window in the SurfNSignWin frame, using NPN_NewStream and NPN_Write. Figure 2 illustrates how Surf'N'Sign displays an archived signature.

The CGI program new_sig.exe. The component of Surf'N'Sign at the server is the CGI program new_sig.exe that accepts the client's signature message. The plug-in sends to the server an encoded, "stripped" version of the object Surf'N'Sign—it contains all but the HTML text itself. It is expected that the server has the HTML text at hand. This is done in order to verify that the HTML document has not been corrupted and, in addition, to reduce communication. It requires a mechanism for identifying which HTML text has been signed. Currently this is done through the title of the document, which uniquely maps to the URL of the document at the server. Alternatively, the mapping between the document and its URL should preferably be done with a <METATAG>, namely the document should contain a <METATAG> that contains the URL of the document. Right now, the function "const char *GetFileName(const char *title)" returns the file (at the server) associated with the title. When the signature is received and the correct HTML document is attached to it, the signature is verified and archived at the server, and a message is sent back to the client's browser.

Figure 5 The schematic partition of window into frames



Cryptographic functions. The cryptographic operations (signing a text and verification of a signature) of Surf'N'Sign are implemented using the RSAREF 2.0 cryptographic library, developed by RSA Laboratories. RSAREF[15] is a free, portable implementation of public key cryptography. Surf'N'Sign uses the RSA signature algorithm for the signature scheme and key generation (using 512 bit-long keys) and MD5 for the message digest (hashing) algorithm. The four basic cryptographic classes are: Private_key, Public_key, Signature, and Certificate. The Private_key and Public_key classes are derived from the corresponding constructs of RSAREF. The certification class is currently unimplemented. A public and private key is generated when the application is first initiated by the static class *SignatureStaticData*. In subsequent applications, the key is read from a file.

Table 2 The four files to support setup of frames

```
<HTML>
<TITLE>Surf'N'Sign Sample - Index HTML File </TITLE>
<FRAMESET rows = "*,90">
<FRAME NAME = "SurfNSignWin" SRC = "main.html">
<FRAME NAME = "SurfNSigndummy" SRC = "null.html">
</FRAMESET>
</HTML>
```

```
<HTML>
<HEAD> <TITLE>Surf'N'Sign Sample - Null HTML File</TITLE>
<BODY></BODY>
</HTML>
```

```
<HTML>
<TITLE>Surf'N'Sign Sample - Main HTML File</TITLE>
<!—put here any text—>
To sign the document go to
<A HREF = "Call_document.html" TARGET = "SurfNSigndummy"> document Name To Sign</A>
</HTML>
```

```
<HTML><TITLE>Surf'N'Sign Sample - Call_document HTML File</TITLE>
<BODY>
<EMBED SRC = "document.sig" WIDTH = 200 HEIGHT = 50>
</BODY>
</HTML>
```

Further extensions

Surf'N'Sign is a protocol that was developed to allow clients to sign committing Web documents. It naturally integrates a digital signature mechanism at the client into the browsing process—its prototype has been implemented as a Netscape Navigator plug-in. The need for such a mechanism in numerous Web applications such as banking, reservations, and insurance is obvious. Several extensions can be called for. The most natural extension to this mechanism is a server-based signature mechanism that, when integrated together with Surf'N'Sign, will allow both the client and the server to commit to one another. Another extension that is called for is the semantic extension. We have chosen the "what you see is what you sign" approach for two reasons: first, since it simplifies the model and implementation greatly and second, since we have emphasized the requirement for a client trust in the system. This resulted in a rather restricted notion of a signature. The most obvious addition is to include content of the embedded parts (images, links, etc.) into the signed message. A few important components are still missing from the implemented system. These include, among others, the certification class (which is currently unimplemented), a directory-based interface for the archived signature, and the support for a number of documents that may be signed at once.

Acknowledgment

We would like to thank the anonymous referees for their useful comments on an earlier version of this paper.

**Trademark or registered trademark of Netscape Communications Corporation, Lotus Development Corporation, Sun Microsystems, Inc., Microsoft Corporation, or RSA Data Security.

Cited references

1. W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory* **IT-22**, No. 6, 644–654 (November 1976).
2. R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," *Communications of the ACM* **21**, 120–126 (February 1978).
3. Utah Digital Signature Development Program, http://www.commerce.state.ut.us/web/commerce/digsig/dsmain.htm, Division of Corporations and Commercial Code, State of Utah, Salt Lake City, UT.
4. Pretty Good Privacy: The International PGP Home Page, http://www.ifi.uio.no/pgp/.
5. S/MIME Central, http://www.rsa.com/rsa/S-MIME/, RSA Data Security, 100 Marine Parkway, Suite 500, Redwood City, CA 94065-1031.
6. B. Schneier, *E-Mail Security*, John Wiley & Sons, Inc., New York (1995).
7. A. Herzberg and H. Yochai, "Mini-Pay: Charging per Click on the Web," *Proceedings of the Sixth WWW Conference* (April 1997), pp. 239–256.
8. Secure Electronic Transactions, MasterCard and VISA, http:

//www.mastercard.com/set/, MasterCard International, Purchase, NY.
9. W3C Digital Signature Initiative, World Wide Web Consortium, http://www.w3.org/pub/WWW/Security/DSig/Overview.html.
10. The SSL Protocol, http://home.netscape.com/newsref/std/SSL.html, Netscape Communications Corporation, Mountain View, CA.
11. *Public Key Cryptography and Digital ID Certificates*, a White Paper by Cylink Corporation, Sunnyvale, CA, http://www.cylink.com/products/security/x509.htm.
12. The Secure HyperText Transfer Protocol, work in progress, http://www.terisa.com/shttp/current.txt, Terisa Systems, Inc., 4984 El Camino Real, Los Altos, CA 94022.
13. Shen: A Security Scheme for the World Wide Web, http://www.pku.edu.cn/on_line/w3html/Shen/ref/shen.html.
14. A. Ashokan, G. Tzudik, and M. Waidner, "Server Supported Signatures," *Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS)*, Number 1146 in *Lecture Notes in Computer Science*, Springer-Verlag, Berlin (September 1996), pp. 131–143. To appear in *Journal of Computer Security* (1997).
15. ftp://ftp.rsa.com/rsaref/, http://www.rsa.com/rsalabs/newfaq/worldreal.htm, Question 174, RSA Data Security, 100 Marine Parkway, Suite 500, Redwood City, CA 94065-1031.
16. Z. Oliphant, *Programming Netscape Plug-Ins*, Sams.net Publishing (1996). Also, in http://home.netscape.com/comprod/development_partners/plugin_api/index.html, Netscape Navigator LiveConnect/Plug-in Software Development Kit.

Accepted for publication September 8, 1997.

Amir Herzberg *IBM Research Division, Haifa Research Laboratory at Tel Aviv, IBM Building, 2 Weizmann Street, Tel Aviv 61336, Israel (electronic mail: amir@haifa.vnet.ibm.com).* Dr. Herzberg received the B.Sc. in computer engineering, the M.Sc. in electrical engineering, and the D.Sc. in computer science from the Technion-Israel Institute of Technology, in 1982, 1986, and 1991, respectively. In 1991, he joined the IBM Research Division, where he now manages the Network Computing and Security group. He established this group, as a Tel-Aviv annex of the Haifa Research Laboratory, in January 1996. His previous assignment was manager of the Network Security group in the IBM Thomas J. Watson Research Center. His research areas include network security, applied cryptography, electronic commerce, communication protocols, and fault tolerant distributed algorithms. Dr. Herzberg is the author of numerous publications and patents in these areas.

Dalit Naor *IBM Research Division, Haifa Research Laboratory at Tel Aviv, IBM Building, 2 Weizmann Street, Tel Aviv 61336, Israel (electronic mail: dalit@haifa.vnet.ibm.com).* Dr. Naor received her B.A. from the Technion-Israel Institute of Technology in 1985, and M.Sc. and Ph.D. from the University of California at Davis in 1988 and 1991, all in computer science. During the years 1991–95 she was involved in postdoctoral research at Stanford University and Tel Aviv University, working on bioinformatics topics. She joined the IBM Haifa Research Laboratory in 1996, where she is currently a research staff member in the Network Computing and Security group.

Reprint Order No. G321-5663.