FORMAL VERIFICATION OF TLS HANDSHAKE AND EXTENSIONS FOR WIRELESS NETWORKS

Llanos Tobarra, Diego Cazorla, Fernando Cuartero and Gregorio Díaz
Departamento de Sistemas Informáticos. EPSA. Universidad de Castilla-La Mancha
Av. Campus Universitario s/n. 02071. Albacete
{mtobarra,dcazorla,fernando,gregorio}@info-ab.uclm.es

ABSTRACT

Transport Layer Security (TLS) is a security protocol widely used in e-commerce in recent years. This protocol has been extended in order to deal with clients connecting from mobile devices (PDAs, cellular phones) through a wireless network. The main goal of this paper is to prove, using model checking techniques, that TLS Handshake and its wireless extensions are secure enough to ensure that a client and a server are able to exchange secret data in a secure way when they communicate over a non trusted wireless network.

KEYWORDS

Evaluation and Assessment, Security, Wireless Applications, E-commerce Theory and Practice

1. INTRODUCTION

Security has become one of the most important topics in the Internet in recent years. The quick development of e-commerce, that needs the exchange of sensitive information (credit card numbers, account numbers, …), has given great importance to the study of communication protocols that allow the exchange of secret data between two or more agents in a non trusted environment.

Moreover, the quick development of wireless technologies is leading us to a scenario in which more and more people will be using mobile devices (PDAs, cellular phones, notebooks) in order to access services in the world wide web. Thus, security over (wired and wireless) networks is becoming nowadays one of the most important issues in e-commerce.

Several general purpose cryptographic protocols have been developed in order to guarantee integrity, reliability and authentication. In particular, wireless networks add some constraints to these security properties. Wireless clients usually have less memory, they cannot perform complex operations and some technologies have a limited bandwidth. Thus, security protocols must be adapted to these restrictions.

One of the most popular security protocols used in e-commerce is Transport Layer Security (TLS) (Dierks et al,1999), a standard protocol considered as the new version of the well known Secure Sockets Layer protocol (SSL) (Freier et al,1996). TLS was mainly developed in order to give the customers confidence in their interactions with on-line shops, electronic banks, etc, when they access these services from their desktop computers. Nevertheless, some extensions have been defined in order to give the same protection against attackers when these customers use their mobile devices.

In parallel to the development of communication and security protocols, some analysis techniques have also been developed in order to verify that these protocols work as they are supposed to, and they do not have bugs that allow intruders to hear or alter the information. One of the most promising techniques in this line is model checking. Model checking (Clarke et al,1999) is a formal methods based technique for verifying finite state concurrent systems that has been implemented in several tools. One of the main advantages of this technique is that model checking is automatic and allows us to know whether a system works properly or not. In case the system does not work as expected, the model checking tool gives us a trace that leads to the source of error.

In this paper we present a formal verification of the TLS Handshake protocol and its extensions for wireless networks using the Casper/FDR2 toolbox (G.Lowe; 1998 and manual FDR2). Casper is a compiler that accepts a syntax very similar to the syntax used to specify protocols, and translates a model into CSP (C.A.R. Hoare, 1985) code which is verified using the model checker FDR2. Casper/FDR2 has been used successfully to verify several security protocols, and we consider it is also appropriate for the verification of TLS and extensions.

In literature we have found several papers that deal with the TLS protocol. In G. Díaz et al. (2004) a formal specification and analysis of TLS using UPPAAL is presented. Nevertheless, only functional properties are considered and no security properties are taken into account. In L. C. Paulson (2003) an inductive analysis of TLS has been performed using the theorem prover Isabelle. In both cases, wireless extensions have not been taken into account.

Other papers deal with the analysis of the SSL protocol. In J.C. Mitchell et al. (1998) SSL Handshake is analyzed using a general purpose finite-state enumeration tool called Murφ. The tool Murφ (Dill, 1996; Dill et al.;1992) was designed for hardware verification and related analysis, but has also been used to analyze some security protocols (J.C. Mitchell et al.; 1997). Finally, in D. Wagner et al. (1996) an informal analysis of SSL Handshake and SSL Record is presented. This analysis explores several possible attacks on the SSL protocol and also gives several ways in which the robustness of the SSL protocols can be improved.

Therefore, as far as we know, this is the first paper that deals with the formal verification of TLS and its extensions for wireless networks.

The paper is organized as follows. In Section 2 TLS handshake and the general mechanism in order to extend its functionality is presented. In Section 3 basic TLS Handshake is verified. In Section 4 three wireless networks related extensions are verified: URL certificate, trusted CA, and request of server certificate status. Finally in Section 5 we give our conclusions and some lines of future work.

2. TRANSPORT LAYER SECURITY

TLS is a layered protocol which was developed by the Internet Engineering Task Force (IETF) in 1999 (Dierks et al,1999). The main goal of TLS is providing a secure connection between two communicating applications. It is a protocol based on SSL (Freier et al;1996), but there are enough differences between them in order to be considered as different protocols.

TLS is composed of four subprotocols: *Handshake*, which allows two agents to establish a session; *Record*, which exchanges application data; *Alert*, which reports about errors in the session; and *Change Cipher Spec*, which copies the negotiated settings to the actual settings. In this paper we will focus on TLS Handshake.

2.1 TLS Handshake

TLS Handshake negotiates the elements that configure a session between a client and a server: a session identifier, the agents certificate, a compression method, some secret data (*MasterSecret*) for generating session keys, a cipher mode *CipherSpec,* and a flag that indicates if the session can be resumable.

TLS Handshake guarantees three security properties:

1. Each connection extreme is authenticated by symmetric and asymmetric algorithms. This is optional but, generally, at least one extreme must be authenticated.
2. The secret data negotiation is secure. It is not possible that an intruder guesses some secret data.
3. The negotiation is trusted. An intruder is not able to modify any message without being detected.

We consider the case when a client wants to establish a new session with the server, and they have to negotiate the session settings. This situation is represented in Figure 1, where the messages in brackets represent optional messages. The protocol performs the following actions:

- The agents exchange hello messages in order to negotiate the best combination of settings. These settings are the agent protocol version, a session identifier, a *CipherSuite* and a compression method. Two random values are generated: *ClientHello.random* and *ServerHello.random*.

- The agents exchange some secret data in a secure way. The client generates some secret data that is sent to the server in the *ClientKeyExchange* message.
- The agents authenticate each extreme if it is required by the selected *CipherSuite*. Then, the agent must send a *Certificate* message. If a certificate allows a digital sign and the owner is the client, it sends a *CertificateVerify* message. If the owner is the server, it sends a *ServerKeyExchange* message.
- They finish the handshake with a *ChangeCipherSpec* message and a *Finished* message. The *Finished* message includes all the previous handshake messages and allows an agent to verify whether the session parameters are correct or not.

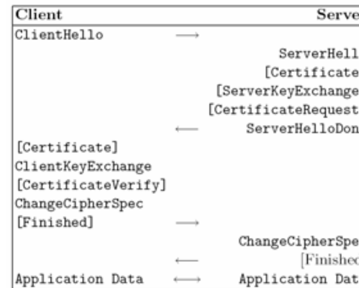


Figure 1. TLS Handshake messages for establishing a new session

2.2 TLS Extensions

TLS provides a general mechanism in order to extend its functionally. This extension mechanism is described in detail in Slake-Wilson et al.(2003). Some specific extensions for wireless networks are described in that document. These extensions focus on the constraints of memory and bandwidth of wireless clients as mobile phones or personal digital assistants (PDAs). See Table 1 for more information about specific extensions.

Table 1. Summary of the specific TLS extensions

Extension type	Code	Description
Server_name	0	Indicate server name in a virtual network
Max_fragment_length	1	Negotiate a maximum fragment size
Client_certificate_url	2	Use a URL certificate instead of a certificate
Trusted_ca_keys	3	Report to the server about the client CA root
Truncated_hmac	4	Use truncated HMAC
Status_request	5	The client request the server certificate status

The general extension mechanism only modifies *Hello* messages. An *Extended Hello* message includes a new field, a list of extensions. Each extension is a pair composed of an identifier of the extension type and some extension data.

When a client requests an extended function from a server, it sends an *Extended Hello* message. If the server does not support the client request, it can decide to finish the handshake. Otherwise the server sends an *Extended Hello* message which includes the same type of extension without extension data.

A client finishes the handshake when it receives an *Extended Hello* which contains an extension type different from the one requested, or when it receives an *Extended Hello* message when client did not request any extension.

Only the client is allowed to request for extensions, and it can request as many extensions as it can include into $2^{16} - 1$ bytes.

3. VERIFICATION OF TLS HANDSHAKE

In this section we focus on the study of the TLS protocol in an environment where there are three kinds of agents: clients, servers and intruders. A group of clients are connected to a server and an intruder may interfere in the message stream.

The intruder is allowed to:

- Overhead and intercept all the messages over the network.
- Modify the messages. He can add bytes, delete bytes or change the value of several bytes.
- Generate new messages using its initial knowledge or parts of the overheard messages.
- Send a new or captured message to another entity of the system.

Nevertheless, we consider that it cannot perform cryptanalysis.

We assume that all the cryptographic functions and algorithms are secure, and there are trusted certificate authorities (CA) that generate secure and non-modified certificates.

3.1 Establishing a New Session

When a client and a server establish a new session, the following steps are performed:

1. The Agents exchange two *Hello* messages, *ClientHello* and *ServerHello*, to negotiate the cipher algorithm (*CipherSuite*), the compression method, a session identifier, agent protocol version and some random data. These messages allow the agents:
   - To agree on security parameters and algorithms, such as cipher algorithm, hash function, which agents have to authenticate, etc.
   - To exchange some cryptographic data needed to generate session keys (random data and session identifier).
   - To provide security parameters to record layer, such as the compression method.
2. Then, the authentication phase starts. The server sends its certificate in a *Certificate* message. If the client has to authenticate himself with the server, the server sends a *CertificateRequest* message. The client answers with its *Certificate* message. Sometimes, the client certificate is not enough to guarantee its identity. In this case it sends a *ClientVerify* message with a digital signature calculate from some session parameters.
3. Afterwards the agents exchange some secret data and both of them generates session data.
4. Both agents verify that all the exchanged parameters are correct and they were not modified. Finally they exchange *Finished* messages.

A description of the system and the Casper protocol specification when a client wants to establish a new session with a server is depicted in Figure 2 .

We do not consider in our protocol specification some messages because they are only constants than agents use to indicate they are ready to start the following phase. In order to simplify the protocol specification, we consider that the session identifier is included in the *CipherSuite* constant because both variables have a very similar treatment. We will use these simplifications through all the verifications.

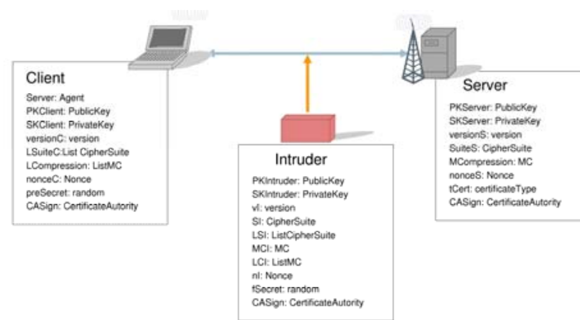
In our study, we consider three security properties:

A. Secret(C, preSecret,[S])
B. Agreement(C,S, [preSecret, SuiteS, MCompression, PKServer, NonceC, NonceS, verS, verC])
C. Agreement(S,C, [preSecret, SuiteS, MCompression, PKServer, NonceC, NonceS, verS, verC])

The property A indicates that the client exchanges some secret data with the server with success and the intruder is not able to guest the secret. Our description of the protocol satisfies it.

Properties B and C are similar. In the first one we check if the client completes a TLS Handshake run with the server and both of them have the same value for a list of variables: the secret data *preKey*, the selected CipherSuite *SuiteS*, the compression method *MCompression*, the server public key *PKServer*, the random data *NonceC* and *NonceS*, and the agents protocol version *verC* and *verS*.

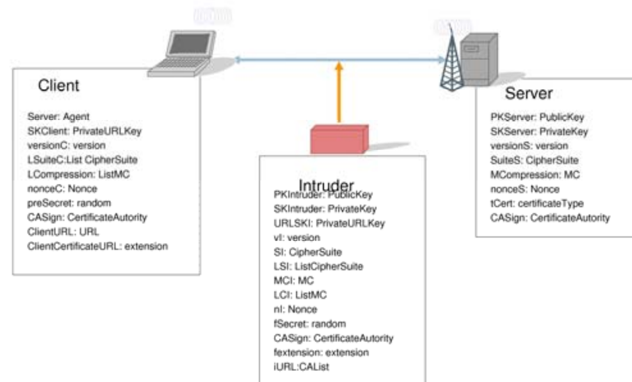
The property C is similar but now is the server which completes a TLS Handshake run with the client. These properties verify whether the intruder is able to take the place of an honest agent or it can modify some session variables. Our protocol description meets both properties.



Protocol Description:

```
ClientHello.   C -> S : C, verC, NonceC, SuiteC, LCompression
[C!=S]
ServerHello.  S -> C : verS, NonceS, SuiteS, MCompression
CertificateS.  S -> C : {S,PKServer}{CASign}
[ValidCertificate(S,PKServer)]
CertificateRequest. S -> C : tCert
CertificateC.  C -> S : {C, PKClient}{CASign}
[ValidCertificate(C,PKClient)]
ClientKeyExchange. C -> S : {preSecret}{PKServer}
CertificateVerify.  C -> S : {MD5(preSecret, NonceC,NonceS)}{SKClient}
FinishedC1.   C -> S : {MD5(SuiteC, verC, LCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedC2.  C -> S : {MD5(SuiteS, verS, MCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedS1.  S -> C : {MD5(SuiteC, verC, LCompression)}{SWS(preSecret, NonceC, NonceS)}
FinishedS2.  S -> C : {MD5(SuiteS, verS, MCompression)}{SWS(preSecret, NonceC, NonceS)}
```

Figure 2. Representation of the system model and protocol specification for establishing a new session



Protocol Description:

```
ClientHello.   C -> S : C, verC, NonceC, SuiteC, LCompression, ClientCertificateURL
[C!=S]
ServerHello.  S -> C : verS, NonceS, SuiteS, MCompression, ClientCertificateURL
CertificateS.  S -> C : {S,PKServer}{CASign}
[ValidCertificate(S,PKServer)]
CertificateRequest. S -> C : tCert
CertificateC.  C -> S :  {ClientURL}{CASign}
[CheckURLCorrect(C,ClientURL)]
ClientKeyExchange. C -> S : {preSecret}{PKServer}
CertificateVerify.  C -> S : {MD5(preSecret, NonceC,NonceS)}{SKClient}
FinishedC1.   C -> S : {MD5(SuiteC, verC, LCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedC2.  C -> S : {MD5(SuiteS, verS, MCompression, ClientCertificateURL)}{CWS(preSecret, NonceC, NonceS)}
FinishedS1.  S -> C : {MD5(SuiteC, verC, LCompression)}{SWS(preSecret, NonceC, NonceS)}
FinishedS2.  S -> C : {MD5(SuiteS, verS, MCompression, ClientCertificateURL)}{SWS(preSecret, NonceC, NonceS)}
```

Figure 3. Representation of the system and protocol specification for the URL certificate extension

4. VERIFICATION OF TLS EXTENSIONS

In the previous section we have verified that basic TLS Handshake is a secure protocol. Now we want to verify if extended TLS for wireless networks is as secure as the basic version.

We are going to focus on wireless extensions of TLS which are related to security. Furthermore, we only include extensions and verify the situation when a new session is established. The extensions are negotiated when the agents establish a new session and they are applied through the session and the resume of the

session. We consider three wireless extensions: URL certificate, trusted CA, and request of server certificate status.

4.1 URL Certificate

Some constrained wireless clients do not have enough memory to store its certificate. It would be desirable that when a TLS client has to sent its certificate to a server, instead of sending a full certificate, the client could send an URL address where its certificate is stored.

First, a client sends an *Extended Hello* message with this extension type. Afterwards, it includes the certificate URL instead of the certificate in the *Certificate* message. When the server receives it, it must check the certificate URL and it must retrieve the certificate chain. We represent this process with the function *CheckURLCorrect,* which accepts an agent identifier and an URL, and it returns true when the agent is the URL owner.

A description of the system and the Casper protocol specification when a client uses the URL certificate extension is depicted in Figure 3.

We have verified the same three properties that we checked with basic TLS (see Section 3). In the case of property B we have added to the variable list the extension type field. All the properties are guarantied.

4.2 Trusted CAs

Some wireless clients can only store a small number of trusted CA due to their memory limitations. In order to avoid failures during TLS Handshake it may be recommendable that constrained clients could send to the server a list of its trusted CAs.

In TLS, clients can send an *Extended Hello* message with the extension type *trusted_ca_authority* and a list of the trusted CAs that the client knows. Each item of this list has two parameters: the type of CA and the identifier of the CA. This list can be empty. When a server receives one of this type of *Extended Hello* messages, it selects the more appropriate certificate chain from the CAs list. Then it sends it back to the client.

A description of the system and the Casper protocol specification when a client uses the URL certificate extension is depicted in Figure 4.

We check the properties A, B and C. In this case we modify property B in order to add the CAs list and the extension type to the variables list in the agreement property. This way we check if the intruder can alter them. We did not find any attack.

4.3 Request of Server Certificate Status

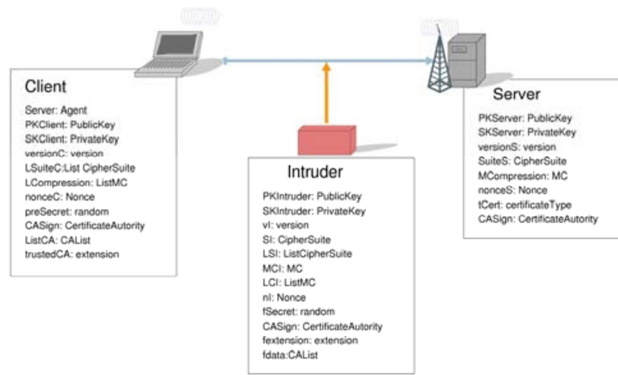
Finally, we consider the third extension. Some constrained clients use some protocols that verify a certificate status instead of checking it itself. Online Certificate Status Protocol (OCSP) (M. Myers; 1999) is one of these protocols. TLS includes an extension that allows a client to verify a certificate status without more resources.

First, a client sends an *Extended Hello* message where it includes an extension type *status_request* and a request field for the certificate status. Nowadays, we only have one type of request, OCSP. In this kind of request, a client includes a list of trusted responders, and a field for extensions request. If the list of responders is empty, the responders are implicit.

We consider that the responders, in this situation the server, are implicit and the client will not ask for any extension. Thus, the *Extended Hello* message only includes the extension type. The server sends the *Certificate* and the *CertificateStatus* messages. As in the extension, there is only one type of *CertificateStatus* response, a OCSP response.

It includes the protocol version, responder identifier, certificate identifier, certificate status (unknown, invalid or valid), validity interleave, extensions, and a digital signature of some message parameters.

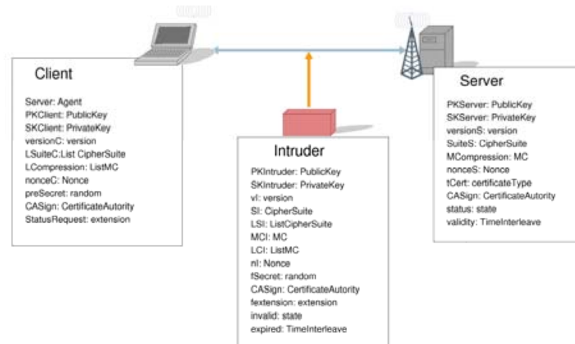
We simplify OCSP response considering that a server only includes the sever identifier, the certificate status, a validity interleave and a digital signature calculated across the response. Afterwards a client receives the status response and it verifies it with the function *Verify*. This function returns false if the certificate is invalid or it has expired.



Protocol Description:

ClientHello. C -> S : C, verC, NonceC, SuiteC, LCompression, trustedCA , ListCA
[C!=S]
ServerHello. S -> C : verS, NonceS, SuiteS, MCompression, trustedCA
CertificateS. S -> C : {S,PKServer}{CASign}
[ValidCertificate(S,PKServer)]
CertificateRequest. S -> C : tCert
CertificateC. C -> S : {C, PKClient}{CASign}
[ValidCertificate(C,PKClient)]
ClientKeyExchange. C -> S : {preSecret}{PKServer}
CertificateVerify. C -> S : {MD5(preSecret, NonceC,NonceS)}{SKClient}
FinishedC1. C -> S : {MD5(SuiteC, verC, LCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedC2. C -> S : {MD5(SuiteS, verS, MCompression, ListCA)}{CWS(preSecret, NonceC, NonceS)}
FinishedS1. S -> C : {MD5(SuiteC, verC, LCompression)}{SWS(preSecret, NonceC, NonceS)}
FinishedS2. S -> C : {MD5(SuiteS, verS, MCompression, ListCA)}{SWS(preSecret, NonceC, NonceS)}

Figure 4. Representation of the system and protocol for Trusted CA extension



Protocol Description:

ClientHello. C -> S : C, verC, NonceC, SuiteC, LCompression, StatusRequest
[C!=S]
ServerHello. S -> C : verS, NonceS, SuiteS, MCompression, StatusRequest
CertificateS. S -> C : {S,PKServer}{CASign}
[ValidCertificate(S,PKServer)]
CertificateStatus1. S -> C : S, status, validity
[Verify(status, validity)]
CertificateStatus2. S -> C : {MD5(S, status, validity)}{SKServer}
CertificateRequest. S -> C : tCert
CertificateC. C -> S : {ClientURL}{CASign}
[CheckURLCorrect(C,ClientURL)]
ClientKeyExchange. C -> S : {preSecret}{PKServer}
CertificateVerify. C -> S : {MD5(preSecret, NonceC,NonceS)}{SKClient}
FinishedC1. C -> S : {MD5(SuiteC, verC, LCompression)}{CWS(preSecret, NonceC, NonceS)}
FinishedC2. C -> S : {MD5(SuiteS, verS, MCompression, ClientCertificateURL)}{CWS(preSecret, NonceC, NonceS)}
FinishedS1. S -> C : {MD5(SuiteC, verC, LCompression)}{SWS(preSecret, NonceC, NonceS)}
FinishedS2. S -> C : {MD5(SuiteS, verS, MCompression, ClientCertificateURL)}{SWS(preSecret, NonceC, NonceS)}

Figure 5. Representation of the system and protocol specification for the Certificate Status extension

A description of the system and the Casper protocol specification when a client uses the URL certificate extension is depicted in Figure 5.

As we did in previous extensions, we have verified the three security properties and we have checked that there are no attacks.

5. CONCLUSION

Our analysis has shown that TLS and its extensions for wireless networks allow a client and a server to exchange some secret data in a secure way when they communicate over an insecure wireless network. Agents can generate secure session keys that guarantee that all data application encrypted with them is secret, and the connection is private. Thus, extended TLS is as secure as basic TLS. The main advantage of TLS extensions over basic TLS is that clients may save bandwidth and memory when they connect from mobile devices.

Although TLS has been shown to be secure, it is worth noting that we have focused on the verification of TLS and extensions specifications as published in Dierks et al.,1999 and Slake-Wilson et al.,2003. This does not mean that an attack over a real implementation of the protocol can be performed (Canvel, B. et al;2003).

Our future work is concerned with extending our analysis of the TLS protocol to other security protocols and e-commerce protocols, like Single Sign On protocols with Secure TLS channels. With respect to e-commerce protocols, we are planning to deal with the verification of security mechanisms related to payment for wireless clients.

REFERENCES

Clarke, E.M.; Grumberg, O. and. Peled, D.A; 1999. *Model Cheking*. The MIT Press.

Hoare, C.A.R.; 1985. *Communicating Sequential Processes*. Prentince Hall.

Lowe, G.;1998. Casper: A Compiler for the Analysis of Security Protocols. *Journal of Computer Security*, 6:53–84, 1998.

Canvel, B.; Hiltgen, A; Vaudenay, S.; and Vuagnoux,M.; 2003. Password Interception in a SSL/TLS Channel. *In Proc. of Advances in Cryptology (CRYPT'03)*, LNCS 2729, pages 583–599. Springer, 2003.

Díaz, G.; Cuartero, F.; Valero, V.; and Pelayo, F.L., 2004. Automatic Verification of the TLS Handshake Protocol. *In Proc. of the 19th ACM Symposium on Applied Computing (SAC'04)*. ACM, 2004.

Dill, D. L.;1996 . The Murφ Verification System. *In Proc. of 8th International Conference on Computer Aided Verification (CAV'96)*, LNCS 1102, pages 390–393. Springer, 1996.

Dill, D. L.; Drexler, A. J;. Hu, A. J.; and Yang, C. H.; 1992. Protocol Verification as a Hardware Design Aid*. In Proc. of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'92)*, pages 522–525. IEEE Computer Society Press, 1992.

Mitchell, J. C.; Mitchell, M.; and Stern, U.; 1997. Automated analysis of cryptographic protocols using Murφ. *In Proc. of IEEE Symposium on Security and Privacy*, pages 141–151. IEEE Computer Society Press, 1997.

Mitchell, J. C.; Shmatikov, V.; and Stern, U.; 1998. Finite-State Analysis of SSL 3.0. In *Proc. Of 7th USENIX Security Symposium*, pages 201–216. USENIX Press, 1998

Paulson, L. C;1999. Inductive Analysis of the Internet Protocol TLS. *ACM Transactions on Information and System Security*, 2:332–351, 1999.

Wagner, D.; and Schneier, B.; 1996. Analysis of the SSL 3.0 Protocol. *In Proc. of 2nd USENIX Workshop on Electronic Commerce*, pages 29–40. USENIX Press, November 1996.

Dierks, T. and Allen, C. 1999. *The TLS Protocol Version 1.0*. Internet Standards, RFC 1999. http://www.ietf.org/rfc/rfc2222.txt.

Formal Systems (Europe) Limited. *FDR Manual*. http://www.fsel.com/fdr2_manual.html.

Freier, O. A.; Karlton, P. and Kocher, P. C.; 1996. *The SSL Protocol Version 3.0*. Netscape Communications. http://wp.netscape.com/eng/ssl3/ssl-toc.html

Myers, M.; Ankney, R.; Malpani, A.; Galperin, S.; and Adams, C.; 1999. *Internet X509 Public Key Infracstructure: Online Certificate Status Protocol (OCSP). RFC 2560.* InternetStandards, RFC 2246, 1999. http://www.ietf.org/rfc/rfc2560.txt.

Slake-Wilson, S.; Nystrom, M.; Hopwood, D.; Mikkelsen, J. and Wright, T.; 2003. *Transport Layer Security (TLS) Extensions. RFC 3546.* Technical Report, Network Working Group.