The Design and Implementation of Datagram TLS

Nagendra Modadugu
Stanford University
nagendra@cs.stanford.edu

Eric Rescorla
RTFM, Inc.
ekr@rtfm.com

Abstract

A number of applications have emerged over recent years that use datagram transport. These applications include real time video conferencing, Internet telephony, and online games such as Quake and StarCraft. These applications are all delay sensitive and use unreliable datagram transport. Applications that are based on reliable transport can be secured using TLS, but no compelling alternative exists for securing datagram based applications. In this paper we present DTLS, a datagram capable version of TLS. DTLS is extremely similar to TLS and therefore allows reuse of pre-existing protocol infrastructure. Our experimental results show that DTLS adds minimal overhead to a previously non-DTLS capable application.

1. Introduction

TLS [7] is the most widely deployed protocol for securing network traffic. TLS is used to protect Web traffic (HTTP [9] [25]) and e-mail protocols such as IMAP [6] and POP [23]. The primary advantage of TLS is that it provides a secure, transparent channel; it is easy to provide security for an application protocol by inserting TLS between the application layer and the network layer—where the session layer is in the OSI model. TLS, however, requires a *reliable* transport channel—typically TCP—and therefore cannot be used to secure datagram traffic.

When TLS was developed, this limitation was not considered particularly serious because the vast majority of applications then ran over TCP. While this is still largely true today, the situation is changing. Over the past few years an increasing number of application layer protocols, such as Session Initiation Protocol (SIP) [26], Real Time Protocol (RTP) [28], the Media Gateway Control Protocol (MGCP) [1], and a variety of gaming protocols have been designed to use UDP transport.

Currently, designers of such applications are faced with a number of unsatisfactory choices for providing security. First, they can use IPsec [18]. However, IPsec is not well suited for client-server application models and is difficult to package with applications since it runs in the kernel. Section 2.1 has a detailed discussion of why IPsec has been found to be a less than satisfactory option. Second, they can design a custom application layer security protocol. SIP, for instance, uses a variant of S/MIME [2] to secure its traffic. Grafting S/MIME into SIP took vastly more effort than did running the TCP variant of SIP over TLS. Third, one can rehost the application on TCP and use TLS. Unfortunately many such applications depend on datagram semantics and have unacceptable performance when run over a stream protocol such as TCP.

The obvious alternative is to design a generic channel security protocol that will do for long lived applications using datagram transport what TLS did for TCP. Such a protocol could be implemented in user space for portability and easy installation but would be flexible and generic enough to provide security for a variety of datagram-oriented applications. Despite initial concerns that this solution would be a large and difficult design project, constructing a working protocol was fairly straightforward, especially with TLS as a starting point and IPsec as a reference. This paper describes the new protocol, which we call “Datagram TLS”. DTLS is a modified version of TLS that functions properly over datagram transport. This approach has two major advantages over the alternatives. First, since DTLS is very similar to TLS, preexisting protocol infrastructure and implementations can be reused. To demonstrate, we implemented DTLS by adding to the OpenSSL [30] library; in all, we added about 7000 lines of code, about 60% of which were cut and pasted from OpenSSL. Second, since DTLS provides a familiar interface to a generic security layer, it is easy to adapt protocols to use it. Experience with TLS has shown that this ease of adaptation is a key to wide deployment.

The basic design principle of DTLS is “bang for the buck.” We wished to minimize both our design and implementation effort and that of the designers and implementors who are potential DTLS users. Thus, in our design of DTLS we choose not to include any features as “improvements" over TLS; all the features additional to DTLS are included for the sole purpose of dealing with unreliable datagram traffic. This design point simplifies the security analysis of DTLS.

2. Design Overview

The target applications for DTLS are primarily of the client-server variety. These are the kinds of applications for which TLS was designed and for which it works well. The present security model of such applications is that the server is authenticated by its DNS name and IP address but the client is either anonymous or authenticates via some form of credential, typically in the form of a username and password handled by the application layer protocol.

This practice is not especially secure. However, application protocol designers, want to maintain as much of their protocol and implementation infrastructure as possible while adding security. This makes a channel security protocol such as TLS or IPsec very attractive since changes are minimized. From this perspective, ideally a datagram channel security protocol would substitute strong cryptographic authentication of the server for DNS and IP-based authentication but leave client authentication to the application layer protocol.

Our design is not the only possible one that can be used in this scenario. In the following sections we consider several alternative approaches and argue that they fit these requirements less well than does DTLS.

2.1. Why not use IPsec?

IPsec was designed as a generic security mechanism for Internet protocols. Unlike TLS, IPsec is a peer-to-peer protocol. For many years IPsec was expected to be a suitable security protocol for datagram traffic generated by client-server applications. In practice, however, there are a number of problems with using IPsec for securing such traffic. These problems stem directly from IPsec residing at the network layer rather than the session or application layer.

Review of IPsec architecture Unlike TLS, IPsec is not one protocol but rather three: Authentication Header (AH) [16] and Encapsulating Security Payload (ESP) [17] are used for traffic security and Internet Key Exchange (IKE) [12] is used for the establishment of keying material and other traffic security parameters. These parameters are collectively referred to as Security Association (SA). In host implementations, AH and ESP are typically implemented in the kernel as part of the IP stack, while IKE is implemented as a user daemon. In network gateways the architecture varies based on the device programming model.

IPsec security policy is controlled using the Security Policy Database (SPD). SPD entries can be created in two ways. First, administrators can directly create entries in the SPD. In addition, many host-based implementations allow applications to set per-socket policies, for instance using the PF_KEY API [20], thus allowing finer control of policy.

When a socket is created in a host-based IPsec implementation, the SPD is consulted to determine the correct security policy. If IPsec processing is required and an appropriate SA does not exist, IKE is invoked to create one. Future packets sent using that socket are protected using that SA. In network gateway-based IPsec implementations the stack performs a SPD lookup for each outgoing packet.

In the remainder of this section, we discuss several aspects of IPsec that make it less than ideal for the kind of applications we are concerned with.

Server Authentication Client-server applications typically identify endpoints in terms of domain names. This is the scenario for which TLS is optimized. In such an environment, the client has an identifier for the server, typically of the form of a DNS name or a URL. When the client connects to the server, it wants the server to authenticate using a credential that matches that identity.

IPsec security policies (as defined in the SPD) are usually expressed in terms of IP addresses, although there is nominal support for symbolic names, including DNS names. IKE supports use of symbolic names, including DNS names in certificates analogous to TLS. However, the primary motivation for support of these sorts of identifiers in IKE was for road warriors, whose IP address could not be known in advance. Thus use of a DNS name to securely identify a server, for example, is not supported by most host IPsec implementations. In principle, IPsec could provide verification by DNS name in two ways. First, DNSSEC [8] could be used to securely map the server's DNS name to its IP address. However, DNSSEC deployment has so far been minimal, making this option problematic. Second, IPsec certificates could contain DNS names and the client could use an IPsec API to verify that the correct certificate was used. Unfortunately, not all IPsec APIs allow certification information to be determined and so this verification cannot be done reliably or portably.

Residence in the kernel Because IPsec operates at the IP layer, it generally must be implemented in the operating system kernel, either directly compiled in or linked in as a loadable module. This makes IPsec fairly inconvenient to install on non-IPsec systems. This is no longer as large a problem as it once was, since most modern operating systems contain IPsec stacks. However, a large number of legacy operating systems still are not IPsec-capable and installing IPsec on them is generally a major operation.

A related problem is the lack of standardized IPsec APIs. An IPsec using application which wishes to control keying policy has no way to portably do so. While TLS APIs are not standardized either, an application developer can easily ship a TLS toolkit along with their application, thus achieving portability. Increased developer control does introduce the possibility that the developer will use the toolkit insecurely. Developers have, however, historically been willing to bear this risk.

2.2. Key Exchange over TCP?

Key negotiation over an unreliable connection is more complicated than with a reliable connection. One alternative is to complete key negotiation on a TCP connection and use the negotiated parameters to secure a *separate* datagram channel. This split design is similar to that used by IPsec but has a number of problems.

The primary virtue of a split design is that it releases DTLS from having to implement a reliable handshake layer. In exchange, an application must now manage two sockets (one TCP, and one UDP). Synchronizing these sockets is a significant application programming problem. In particular, session renegotiation is complicated by this architecture. With the TCP connection closed once key negotiation is complete, renegotiation messages must be communicated over the unreliable datagram channel, requiring the implementation of a retransmission mechanism.

If the TCP connection is left open once key negotiation is complete, unnecessary system resources are consumed. This is a problem because operating system kernels often exhibit problems when programs have a large numbers of sockets open [14]. In particular, select() performs poorly (if at all) with large numbers of open sockets and replacements are often not portable. In addition, some older operating systems have tight limits on the number of open files per process (in older Linux kernels this limit was 1024.)

An ordinary UDP server expects to read and write on only a single socket. Thus, the use of a TCP handshake channel could force significant rewriting of server code. Additionally, error case handling becomes complicated: say the TCP connection is reset, does that imply that the bulk transfer channel should be closed?

These considerations lead us to conclude that it is better to have the handshake and data transfer occur over the same channel from the beginning. As we shall see, DTLS's reliability requirements are quite primitive, allowing us to make do with a protocol much simpler than TCP.

2.3. Design Requirements

Once we decided on a user-space protocol that runs over a single channel, the direct course of action was to make TLS datagram capable. Although DTLS must be somewhat different from TLS, in keeping with our basic principle we have kept TLS unchanged wherever possible. Where we have had to make changes to TLS, we have attempted to borrow from preexisting systems such as IPsec. Similarly, DTLS is explicitly designed to be as compatible as possible with existing datagram communication systems, thus minimizing the effort required to secure one's application.

Datagram Transport DTLS must be able to complete key negotiation and bulk data transfer over a single datagram channel. This property allows applications to simply replace each datagram socket with a secure datagram socket managed by DTLS.

Reliable Session Establishment DTLS must provide a mechanism for authenticating endpoints, reliably establishing keying material and negotiating algorithms and parameters. Since DTLS must run entirely over unreliable datagram transport, it must implement a retransmission mechanism for ensuring that handshake messages are reliably delivered. However, the retransmission mechanism should be simple and lightweight, ensuring that DTLS is as portable as possible. Note that the requirement to create a session means that DTLS is primarily suited for long-lived "connection-oriented" protocols as opposed to totally connectionless ones like DNS. Connectionless protocols are better served by application layer object-security protocols.

Security Services DTLS must provide confidentiality and integrity for the data transmitted over it. It should optionally provide the ability to detect replayed packets.

Ease of Deployment The ability to implement TLS entirely in user space without changing the kernel has been a major contributor to TLS deployment. This feature allows developers to bundle a TLS implementation with their application without dependence on operating system vendors. DTLS should similarly be implementable solely in user space.

Semantics For many TCP based applications it has been very simple to implement a security layer by using TLS. One of the main reasons is that TLS semantics mimic those of TCP. Thus, a TLS API can mimic the well known socket interface, making network connections appear to be read-write streams. DTLS semantics should mimic

UDP semantics thus allowing DTLS implementations to mimic the UDP API.

Minimal Changes DTLS must be as similar to TLS as possible. Over the years, TLS has become more robust and has been refined to withstand numerous attacks. Our goal is for DTLS to be equally robust by inheriting all the tested and popular features of TLS. By minimizing changes we reduce the likelihood of introducing any unforeseen weaknesses.

Additionally, minimizing changes has the benefit that DTLS can be easily implemented based on TLS implementations such as OpenSSL [30]. Hardware implementations of TLS are optimized to speed up asymmetric and symmetric cryptographic operations. DTLS should not introduce new cipher suites or make changes to the key derivation algorithms. Hence DTLS implementations can leverage hardware implementations of TLS.

2.4. Non-Requirements

DTLS is not intended to provide any congestion control functionality. Congestion control needs to be addressed by a datagram transport using application regardless of whether a security layer is in place, and hence is beyond the scope of DTLS. Applications that do not implement congestion control can use the Datagram Congestion Control Protocol (DCCP) [19] as the underlying transport protocol with DTLS providing the security layer.

3. TLS Overview

Since DTLS is based on TLS, it is useful for the reader to be familiar with TLS. In this section we provide a brief overview of TLS.

3.1. TLS Features

TLS is a generic application layer security protocol that runs over reliable transport. It provides a secure channel to application protocol clients. This channel has three primary security features:

1. Authentication of the server.
2. Confidentiality of the communication channel.
3. Message integrity of the communication channel.

Optionally TLS can also provide authentication of the client.

In general, TLS authentication uses public key based digital signatures backed by certificates. Thus, the server authenticates either by decrypting a secret encrypted under his public key or by signing an ephemeral public key. The client authenticates by signing a random challenge. Server certificates typically contain the server's domain name. Client certificates can contain arbitrary identities.

3.2. Protocol

TLS is a layered protocol consisting of four pieces, shown in Figure 1.

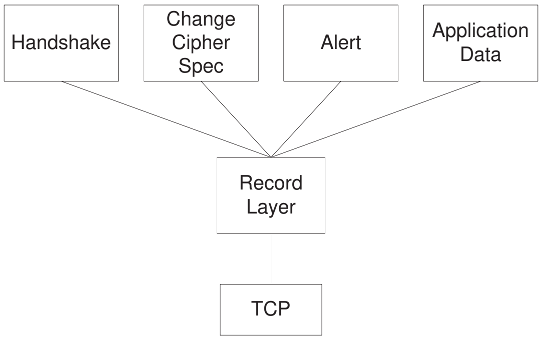

Figure 1. The Structure of TLS

At the bottom is the TLS Record Layer which handles all data transport. The record layer is assumed to sit directly on top of some reliable transport such as TCP. The record layer can carry four kinds of payloads:

1. Handshake messages—used for algorithm negotiation and key establishment.
2. ChangeCipherSpec messages—really part of the handshake but technically a separate kind of message.
3. Alert messages—used to signal that errors have occurred
4. Application layer data

We focus on describing the record and handshake layers since they are of the most relevance to DTLS.

3.3. Record Protocol

The TLS record protocol is a simple framing layer with record format as shown below (see RFC 2246 [7] for a description of the specification language):

```
struct {
    ContentType     type;
    ProtocolVersion version;
    uint16          length;
    opaque          payload[length];
} TLSRecord;
```

Each record is separately encrypted and MACed. In order to prevent reordering and replay attacks a sequence number is incorporated into the MAC but is not carried in the record itself. Since records are delivered using a reliable transport, the sequence number of a record can be obtained simply by counting the records seen. Similarly, encryption state (CBC residues or RC4 keystream)

```
ClientHello              -------->
                                          ServerHello
                                          Certificate
                         <--------    ServerHelloDone
ClientKeyExchange
[ChangeCipherSpec]
Finished                 -------->
                                   [ChangeCipherSpec]
                         <--------           Finished
```

Figure 2. The simple RSA TLS handshake

is chained between records. Thus, a record cannot be independently decrypted if for some reason the previous record is lost.

3.4. Handshake Protocol

The TLS handshake is a conventional two round-trip algorithm negotiation and key establishment protocol. For illustration, we show the most common RSA-based variant of the handshake in Figure 2.

A TLS client initiates the handshake by sending the ClientHello message. This message contains the TLS version, a list of algorithms and compression methods that the client will accept and a random nonce used for anti-replay.

The server responds with three messages. The ServerHello contains the server's choice of version and algorithms and a random nonce. The Certificate contains the server's certificate chain. The ServerHelloDone is simply a marker message to indicate that no other messages are forthcoming. In more complicated handshakes other messages would appear between the Certificate and the ServerHelloDone messages.

The client then chooses a random PreMasterSecret which will be used as the basis for each side's keying material. The client encrypts the PreMasterSecret under the server's RSA public key and sends it to the server in the ClientKeyExchange message. The client then sends the ChangeCipherSpec message to indicate that it is changing to the newly negotiated protection suite. Finally, the client sends the Finished message which contains a MAC of the previous handshake messages. Note that the Finished message is encrypted under the new protection suite. The server responds with its own ChangeCipherSpec and Finished messages.

As with the record layer, the handshake protocol assumes that data is carried over reliable transport. The order of the messages is precisely defined and each message depends on previous messages. Any other order is an error and results in protocol failure. In addition, no mechanism is provided for handling message loss. Retransmission in case of loss must be handled by the transport layer.

4. DTLS Design

DTLS reuses almost all the protocol elements of TLS, with minor but important modifications for it to work properly with datagram transport. TLS depends on a subset of TCP features: reliable, in-order packet delivery and replay detection. Unfortunately, all of these features are absent from datagram transport. In this section we describe the DTLS protocol and how it copes with the absence of these features. Note that although we believe that IPsec is the wrong tool for providing this type of security, many of its techniques for handling these effects are quite useful and are borrowed for DTLS.

4.1. Record Layer

As with TLS, all DTLS data is carried in records. In both protocols, records can only be processed when the entire record is available. In order to avoid dealing with fragmentation, we require DTLS records to fit within a single datagram. There are three benefits to this requirement. First, since the DTLS layer does not need to buffer partial records, host memory can be used more efficiently, which makes the host less susceptible to a DoS attack. Second, it is quite possible that datagrams carrying the remaining record fragments are lost, in which case the received fragments are useless and cannot be processed. Third, it is not clear how long received fragments should be buffered before being discarded. Buffering record fragments would unnecessarily complicate a DTLS implementation without providing any obvious benefits. Note that DTLS will still operate correctly with IP fragmentation and re-assembly, since IP re-assembly is transparently handled by the kernel.

The DTLS record format is shown below. The boxed fields are introduced by DTLS and are absent from TLS records.

```
struct {
    ContentType     type;
    ProtocolVersion version;
    uint16          epoch;
    uint48          sequence_number;
    uint16          length;
    opaque          payload[length];
} DTLSRecord;
```

Epoch Epoch numbers are used by endpoints to determine which cipher state has been used to protect the record payload. Epoch numbers are required to resolve ambiguity that arises when data loss occurs during a session renegotiation. To illustrate, consider a client transmitting data records 7, 8 and 9, followed by ChangeCipherSpec message in record 10. Suppose the server receives records 7 and 9 (8 and 10 are lost). From the server's point of view, record 8 could have been the ChangeCipherSpec message, in which case record 9 is (incorrectly) assumed to be associated with the pending cipher state. Since epoch numbers are incremented upon sending a ChangeCipherSpec message, the server can use the epoch number to resolve the ambiguity. In this case, records 7 and 9 have the same epoch, implying that record 8 must have been a data record.

An alternative to epoch numbers would be to simply use random initial sequence numbers for records. The sequence numbers are sufficiently large that the chance of collision of active sequence number ranges is vanishingly small. However, this would probably require slightly more code to implement than the epoch strategy and is less in keeping with the style of TLS, which uses zero-based sequence numbers.

Sequence Number TLS employs implicit record sequence numbers (RSN) for replay protection. RSNs play a similar role in DTLS, but must be explicitly specified since records can get lost or be delivered out of order. As with TLS, RSNs are incremented by 1 for each record and are reset to zero whenever the cipher state is rolled over due to a session renegotiation. Note that DTLS sequence numbers are 48 bits (TLS's are 64 bits) and therefore the total space occupied by epoch and sequence number is the same as the sequence number in TLS.

Replay detection is performed using the replay window mechanism of RFC 2401 [18]. If datagrams always arrived in order, it would be sufficient for a DTLS end point to keep track of the most recent record seen in order to detect replays. But since datagrams may also arrive out of order, a replay window mechanism is required. This is most easily implemented as a bitmap where the set bits represent the most recently received records. RSNs that are too old to be checked against the bitmap are discarded.

Note, however, that replay detection can be undesirable in some applications since packet duplication may be an unintentional network effect. If replay detection is turned off, then sequence numbers are not of any significance in MAC computation, but can be useful for counter mode ciphers.

Payload Length DTLS requires that a record fit entirely within a single datagram. This means that DTLS records will often be smaller than TLS records. The largest packet that can be transmitted between two hosts—the Path Maximum Transmission Unit (PMTU)—is typically less than the maximum size of a TLS record.

4.2. Ciphering Modes

DTLS cannot use any of the TLS 1.0 cipher modes, since they all maintain residual state between records requiring records to be processed in order without loss. However, the CBC mode proposed for TLS 1.1 is compatible with DTLS, as we describe in this section. We also explain why RC4 is unsuitable for use in DTLS.

DTLS can also make use of counter mode AES, once this mode has been standardized.

CBC Mode An attack [31] against CBC mode ciphers as employed by TLS 1.0 has resulted in the use of a slightly modified version of CBC that requires *explicit* initialization vectors (IVs). The new version is likely to be a feature of TLS 1.1 and is well suited for use in DTLS.

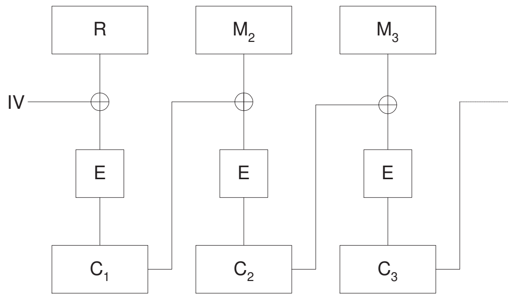

Figure 3. CBC Encryption with Explicit IV

As shown above, in explicit IV mode a random data block is prepended to record data. All the encrypted blocks are transmitted, and the receiver simply discards the first plaintext block to retrieve record data. With an explicit IV, each record can be separately decrypted. Triple-DES and AES can be used with DTLS in this mode.

RC4 RC4 has been the cipher of choice for securing TLS 1.0 connections due to its computational efficiency. Unfortunately, RC4 is not easily applied to lossy datagram traffic: random access implies that the key stream needs to be buffered. Alternatively, the RC4 engine can be reseeded for each incoming record, which is also fairly inefficient especially considering that work by Mironov [21] recommends that the first 512 bytes of RC4 keystream be discarded due to a weakness in the RC4 key scheduling algorithm [10].

We conclude that RC4 is an unsuitable cipher for use in DTLS.

4.3. Handshake Protocol

The DTLS handshake, shown in Figure 4, is nearly identical to that of TLS. There are two major changes:

1. Stateless cookie exchange to prevent denial of service.

2. Message fragmentation and re-assembly

We begin by describing the modifications to protect the handshake exchange from denial of service.

Handshake Exchange Because the DTLS handshake takes place over datagram transport, it is vulnerable to two denial of service attacks that TLS is not. The first attack is the standard resource consumption attack. The second attack is an amplification attack where the attacker sends a ClientHello message apparently sourced by the victim. The server then sends a Certificate message—which is much larger—to the victim.

To mitigate these attacks, DTLS uses the *cookie exchange* technique that has been used in protocols such as Photuris [13]. Before the handshake proper begins, the client must replay a "cookie" provided by the server in order to demonstrate that it is capable of receiving packets at its claimed IP address.

Figure 4 shows the DTLS protocol.

$\boxed{\text{ClientHello}}$	$\longrightarrow$	
	$\longleftarrow$	$\boxed{\text{HelloVerifyRequest}}$
$\boxed{\text{ClientHello}}$	$\longrightarrow$	
		ServerHello Certificate
	$\longleftarrow$	ServerHelloDone
ClientKeyExchange [ChangeCipherSpec] Finished	$\longrightarrow$	
		[ChangeCipherSpec]
	$\longleftarrow$	Finished

Figure 4. The simple DTLS RSA handshake

The DTLS ClientHello message contains a cookie field. The initial ClientHello contains an empty (zero-length) cookie or potentially one cached from a prior exchange. A server that is unable to verify the incoming cookie and wishes to establish the liveness of the DTLS client sends a HelloVerifyRequest message. Servers that are more sensitive to overall handshake latency can skip the HelloVerifyRequest message and instead respond with ServerHello messages, in which case the protocol flow is the same as in TLS. Note that servers which choose to make this optimization can still be used as denial of service amplifiers and should therefore only do so in environments where amplification attack is known not to be a problem.

The HelloVerifyRequest message contains a cookie. This cookie should be generated in such a way that it does not require keeping state on the server, thus avoiding memory consumption denial of service attacks. For example, the cookie can be generated from a keyed hash of the client IP address, using a global key. Techniques for generating and verifying this kind of stateless cookie are well known, see for instance Photuris [13].

Servers that are willing to resume sessions can skip the cookie exchange phase if a valid session ID is presented by the client, since the identity of the client must have been previously established. One possible optimization for servers that do not support session resumption is to maintain a cache of recent (client, cookie) pairs, so that cookie exchange can be skipped if a match is made on the first ClientHello .

The formats of the ClientHello and HelloVerifyRequest messages are provided below.

```
opaque Cookie<0..32>;

struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    Cookie cookie;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod comp_meth<1..2^8-1>;
} ClientHello;
struct {
    ProtocolVersion server_version;
    Cookie cookie;
} HelloVerifyRequest;
```

Unlike application data, handshake messages (including the ChangeCipherSpec message) must be reliably delivered since all handshake messages are necessary for successful session negotiation. This creates three problems. First, messages may be lost on the network. Second, they may be reordered, confusing the receiving peer. Third, some handshake messages are too large to fit in a single DTLS record and therefore must be fragmented across multiple records. The DTLS handshake layer is responsible for reassembling these records into a coherent stream of complete handshake messages. This necessitates the addition of retransmission as well as a more complicated message format.

4.4. Timeout and Retransmission

Because DTLS handshake messages may be lost, DTLS needs a mechanism for retransmission. DTLS implements retransmission using a single timer at each end-point. Each end-point keeps retransmitting its last message until a reply is received. The state machine that implements the timer and resulting retransmissions is shown in Figure 5. In the balance of this section, we describe the operation of the timer state machine and explain how timer expiry values are picked.

State Machine Once in the *Read Message Fragment* state, transitions are triggered by the arrival of data fragments or the expiry of the retransmission timer. If a data

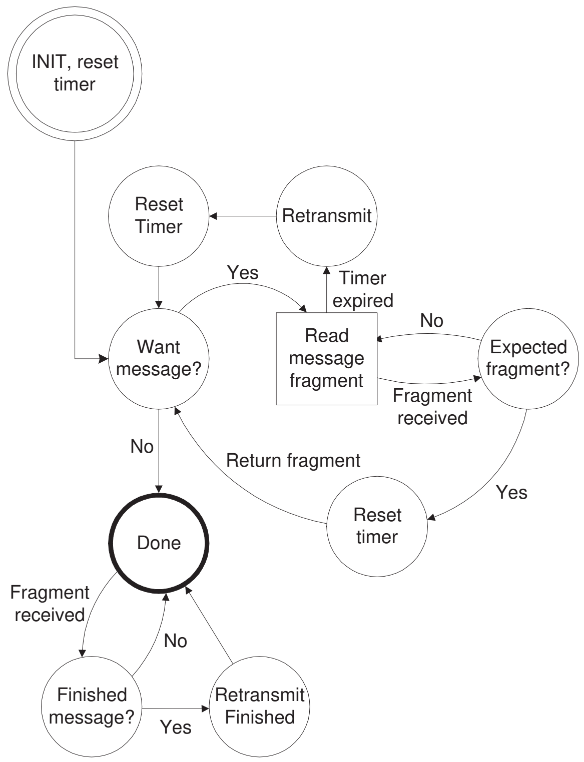

Figure 5. Timer state machine

fragment is the expected next handshake message then the fragment is returned to the higher layers and the timer is cancelled. Otherwise, the fragment is buffered or discarded as appropriate and the timer is allowed to continue ticking. When the retransmit timer expires, the implementation retransmits the last flight of messages that it transmitted.

Timer Values Picking appropriate timer values is a difficult problem due to the heterogeneous nature of the Internet and the wide variance in round trip times (RTT). While estimating RTT would allow for estimating a timer value, requiring that DTLS estimate RTT is an unnecessary burden, given the simplicity of the handshake protocol. Deciding on the exact timer value is especially tricky because the peer is often doing some kind of cryptographic computation, which can take a substantial fraction of the RTT. Thus, one wishes to set one's timer values conservatively to avoid unnecessary retransmissions.

We recommend that DTLS implementations use timer values between 500 to 1000ms. In general, well-behaving implementations should back off their retransmission timers.

Are ACKs necessary? When a retransmission event happens, the entire flight of un-answered messages is retransmitted. If that flight is large, like a Certificate message, a nontrivial amount of network bandwidth (though probably less than 5k) is wasted. In addition, the desire to avoid unnecessary retransmission motivates large timer values which result in high latency. An alternative strategy would be to allow receivers to transmit an ACK value that indicated that they have received the message and were processing it. This would allow timers to be set lower as well as reducing the number of packets that have to be retransmitted (since the sender would know that some had already been received.) In the interest of simplicity, we decided not to add an ACK feature to DTLS, but future measurement may indicate that ACKs provide a large enough improvement to be worthwhile adding.

4.5. Handshake Message Ordering and Fragmentation

Because handshake messages may be too large to fit into a single DTLS record, we need to modify the handshake messages to be able to span records. The new format is shown below.

```
struct {
    HandshakeType msg_type;
    uint24 length;
    uint16 message_seq;
    uint24 frag_offset;
    uint24 frag_length;
    HandshakeMessage msg_frag[frag_length];
} Handshake;
```

Message Length The handshake message header contains the overall message length. This makes it easy to allocate buffer space for the message regardless of which fragment is received first.

Message Sequence Number Handshake (and Change Cipher Spec) messages include their own message sequence numbers (MSN), independent of record sequence numbers (RSN). Since the record layer assigns unique sequence numbers to each record, it is possible that a DTLS end-point receives a handshake message and its retransmitted version under different RSNs. In the absence of the MSN, it is not possible for the handshake layer to detect duplicates. All fragments of a handshake message carry the same MSN.

It is worthwhile considering whether retransmits can reuse the original RSN, and hence make do without the MSN. As it turns out, there are two problems with reusing RSNs. First, it is a layering violation: the handshake layer is a client of the record layer, just like the application layer, and should not receive different treatment. Second, the original handshake message may have been dropped due to the packet size exceeding PMTU. In this case the handshake message needs to be fragmented, which implies that it spans multiple records, each with their own unique RSN.

Fragment Offset and Length As previously mentioned, handshake messages may be fragmented when they are larger than PMTU. In fact such fragmentation is fairly likely since certificates can easily be a couple of kilobytes in size. We chose to use fragment offset and length rather than fragment sequence numbers to aid in handling messages which are fragmented twice in two different ways. With this scheme, it is easy to reassemble the original message provided at least one copy of each byte is received.

Finished Message The purpose of Finished messages is to verify that parties have correctly negotiated keys and algorithms. In TLS, the Finished message contains MD5 and SHA1 hashes of all the handshake messages, sequentially appended to each other (including their message headers). The DTLS algorithm for computing finished hashes has to be slightly different due to the presence of message fragmentation headers. Since the message might have been fragmented multiple times with different fragment sizes, this creates a potential inconsistency. In order to remove this inconsistency, the handshake hashes are computed as if handshake messages had been received as a single fragment.

Alert Messages DTLS reuses all of the TLS alerts. Most TLS alerts signal the end of the connection–either graceful or abortive–and therefore no data should come after them. Under no circumstances should a record be accepted with a sequence number postdating that of an alert which closed the connection.

There is, however, a complication introduced by a sender transmitting data followed by an alert but have them arrive in the reverse order. We have not analyzed this situation, but believe that it is safer for implementations to reject such data records.

5. Security Analysis

Considering the complexity of modern security protocols and the current state of proof techniques, it is rarely possible to completely prove the security of a protocol without making at least some unrealistic assumptions about the attack model.

Instead of attempting to rigorously prove the security of DTLS, one of our main goals in the design of DTLS is to follow the TLS specification as closely as possible. As a result, DTLS does not offer any "improvements" over TLS. All the features introduced into DTLS are for the sole purpose of dealing with unreliable datagram transport.

We argue that DTLS does not reveal any additional information beyond TLS during the handshake or bulk transfer phase—all the additional information in a DTLS stream can be derived by passively monitoring a TLS stream. To justify this argument, consider the additional information that is available from a DTLS stream.

Record Layer The DTLS record layer reveals the current epoch and sequence number. This is public information to an adversary monitoring a TLS session: the sequence numbers are implicit in TLS, but nonetheless may be inferred, and epoch numbers may also be derived from the stream since session renegotiations may be detected (by observing Handshake records being exchanged during an established session.)

Handshake Layer Handshake messages reveal message number, fragment length and fragment offset. Once again, this information is easily derived by an eavesdropper monitoring a TLS session. Message number is obtained by counting exchanged messages, fragment length is obtained from record length and fragment offset is derived from the length of preceding message fragments. Only the Finished message is encrypted during the initial handshake phase, and since it is of a fixed format, its fragment length and offset are obvious.

Handshake messages exchanged due to session renegotiation are completely encrypted in both DTLS and TLS.

Timing information Recently, timing information has been used as the basis for attacks on TLS [4][5]. Therefore it is critical to consider what information is revealed by timing.

DTLS receive record processing is essentially the same as that of TLS. On reception, records and handshake messages are not processed until available in entirety, and therefore the processing of DTLS records and messages is identical to the processing procedure of TLS.

DTLS transmit processing leaks a small amount of timing information when compared to TLS. In general, when applications issue TLS or DTLS writes, this causes a single DTLS/TLS record to be generated. The time when the packet is delivered to the network potentially reveals information about the plaintext [29]. With TLS, TCP congestion and flow control hides this information to some extent, especially if the Nagle algorithm [24] is used. With DTLS, however, records are likely to be transmitted as soon as they are generated. Users who wish to prevent this kind of traffic analysis should buffer writes.

Implementation We implemented DTLS based on the OpenSSL toolkit and reuse much of the code already used in production TLS servers. As a result, DTLS inherits well tested and stable code.

6. Implementation

We implemented DTLS based on the popular OpenSSL library [30]. OpenSSL is the de facto standard open source TLS/SSL implementation. Additionally, OpenSSL has proven to be stable and is used by numerous production quality servers such as the Apache Web server.

We modified the demo server and client applications that are part of the OpenSSL distribution to be UDP capable. We also implemented a UDP proxy application that is capable of dropping, delaying and duplicating packets. Results from our experiments are listed in Section 8. Our implementation was tested and run on the Linux 2.4.21 kernel.

Our implementation required adding about 7000 lines of additional code to the OpenSSL base distribution. Considering that this line count includes libraries, data structures and socket management needed for DTLS, our code makes up only a small portion of the 240,000 line OpenSSL package. Conveniently, we were able to leverage a number of OpenSSL features that were designed for different use. For example, OpenSSL provides an I/O buffering layer that causes TLS to only make `send()` system calls when it has serialized all data to be sent on a particular round of the handshake. We are able to reuse the buffering code to maximize handshake packet payload size.

In the remainder of this section we describe some details of our implementation.

OpenSSL Architecture OpenSSL implements SSLv2, SSLv3 and TLSv1. Each of these protocols are implemented by sharing as much code as possible, with virtual functions handling protocol differences. From the library's standpoint, DTLS appears to be another version of the TLS protocol.

As a result of implementing DTLS in this way, we can reuse much of the utility, state machine and record/message generation code of OpenSSL. In a number of cases we found it was inconvenient to write special cases into TLS processing code, and as a result we copied many functions and modified them appropriately. Roughly 60% of the 7000 lines of additional code were actually copied from the other protocol implementations in OpenSSL. With some effort it should be possible to reduce the amount of duplicated code substantially.

One of the nice side effects of implementing DTLS this way is that DTLS can be accessed through the same functions used by TLS, for example `SSL_connect()`, `SSL_read()`, `SSL_write()` and, `SSL_close()`.

Below we describe some issues encountered in our implementation.

PMTU Path Maximum Transmission Unit (PMTU) is the maximum sized packet that can travel on a path without requiring fragmentation. In general, paths consist of heterogeneous networks that have links with varying limits on maximum packet size. Therefore the PMTU for a given path is set by the limiting link on the path. Previous work [15] shows that fragmentation is undesirable. Fragmentation results in inefficient use of network and routing resources, and lost fragments cause degraded performance. Additionally, IP fragments interact poorly with firewalls and NAT devices, which often discard fragments. Therefore it is useful to know the PMTU.

RFC 1191 [22] specifies the process by which PMTU is discovered. In short, hosts send out IP packets with the DF (Don't Fragment) bit set, iteratively reducing the size of packets until the host is reached. Therefore, it is difficult for the kernel to know a priori what the appropriate PMTU is without incurring a significant probing cost–though it can guess it after enough traffic has been transmitted. In general, kernel support for PMTU is quite poor. On the Linux system, where we developed our implementation, the kernel keeps track of its PMTU estimate and returns an error if an application attempts to send a larger packet.

DTLS needs to be agnostic about such kernel behavior so as to not get caught using an excessive PMTU value. Unless an application explicitly sets a PMTU value we turn on the DF bit in outgoing datagrams via `setsockopt()` and query the kernel for the MTU. If the PMTU is unavailable, we guess a PMTU starting with 1500 (the ethernet MTU), successively reducing the PMTU estimate if the current setting happens to be too large. We can detect that PMTU has been exceeded if `send()` returns -1 and sets errno to EMSGSIZE.

On some operating systems, even this level of PMTU support is unavailable and the only feedback that the PMTU has been exceeded is packet loss. It's not clear what the best approach for dealing with such an environment is, but our intention is to start with a large packet size and then back off the packet size with each successive retransmit.

Note however, that performance sensitive datagram applications are generally PMTU aware anyway, in which case DTLS can be relieved of having to guess PMTU.

During the handshake phase, DTLS attempts to send the largest packets possible, which includes packing multiple records into a single packet.

Buffering Because retransmits may be necessary, we buffer a copy of outbound handshake messages. Optionally, handshake messages may be reconstructed whenever a retransmit request is received, but this is unnecessarily computation intensive, especially when memory is available. Buffered messages need only be buffered until the next expected handshake message is received. This is because the handshake protocol is executed in lock-step and the incoming message provides an implicit acknowledgment for all the buffered messages. Our implementation of DTLS also buffers out-of-order handshake messages, since the handshake layer expects messages to be delivered in order.

Retransmit Timer Our implementation uses a timer value of 750ms, which is more than sufficient given that our experiments were run on a LAN. When using blocking sockets, the timeout (set via `setsockopt()`) causes `recv()` to return with an explicit timeout error if data is not received during the time period. While we chose a value suitable to our environment, our DTLS API allows applications to set their own read and write timeout values.

Sockets that run in non-blocking mode cause DTLS to return either `SSL_ERROR_WANT_READ` or `SSL_ERROR_WANT_WRITE` which are effectively equivalent to `EAGAIN`, signalling that data was not immediately available for reading or writing (this is the same behavior as the TLS API). Non-blocking DTLS applications are required to call `DTLS1_get_timeout()` to determine when the next DTLS I/O call should be invoked and use their own timers to arrange for the call at that time.

7. Programmer Experience

The DTLS API is very similar to the API provided by OpenSSL for operating TLS connections. The only additional calls provided by DTLS are related to datagram transport: setting and getting PMTU, timer values and datagram socket connection options. For testing purposes, we ported the `s_server` and `s_client` programs that are part of the OpenSSL distribution to use DTLS. Almost all the effort required to port these applications to DTLS was concentrated on making them UDP-capable.

At a high level, one can take an ordinary UDP application and render it DTLS-capable by simply replacing all calls to `send()` and `recv()` with calls to `SSL_write()` and `SSL_read()`, the default I/O calls of the OpenSSL library. As with OpenSSL's ordinary behavior, the first call to the read or write functions attempts to negotiate a DTLS connection. This simple approach works well for applications which use a blocking I/O discipline but does not work well for those which want to operate in non-blocking mode. Thus, applications that wish to have a more complicated I/O control discipline need to either use threads or non-blocking mode.

Thread-based I/O discipline In case of threaded applications, calls to the DTLS library are blocking, and the library is fully responsible for handling timer expiry and dispatching retransmits. Thus, the application can essentially be oblivious to DTLS being in use, provided that it uses a separate thread for each DTLS "connection."

One consequence of protocol logic being abstracted from applications is a slight break from blocking-socket convention. In the case of blocking datagram sockets, `recv()` either returns -1 on error, or a non-zero number of bytes read. However, `SSL_read()` can return 0. This happens when the data available on the incoming socket is not application data, but control information, an Alert message for example. This behavior of `SSL_read()` interface is not specific to DTLS. The TLS programmer has a similar experience when using OpenSSL.

Non-blocking I/O discipline When DTLS is used in the context of a non-blocking event driven application, the application needs to be prepared for timeouts during handshake processing. Effectively, any I/O call to DTLS can return with `SSL_ERROR_WANT_READ` or `SSL_ERROR_WANT_WRITE`, signalling that an I/O operation blocked. An application receiving such an error needs to determine the current DTLS timeout by calling `DTLS1_get_timeout()` and restart the I/O call when the timer expires. Once the handshake is complete, DTLS returns a value of 0 for the timer, signalling that it does not have any pending I/O events. For simplicity, applications may choose to call `DTLS1_get_timeout()` regardless of whether the handshake is in progress.

8. Experiments and Results

Our results from comparing network traffic generated by TLS and DTLS are listed in Tables 1 and 2. The cipher negotiated in these tests was `EDH-RSA-DES-CBC3-SHA`. This cipher results in a total of 10 records being exchanged between client and server for TLS. The DTLS negotiation had at least two more records due to the cookie exchange phase and the rest due to message fragmentation.

Each DTLS handshake message fragment has 25 bytes of overhead from headers (13 for record header and 12 for message fragment), compared to 9 bytes for TLS. In all, the headers contribute to most of the overhead in DTLS (the remainder comes from the the extra padding block required by CBC with explicit IV). Even though the overhead for DTLS is close to 35%, the actual size of the overhead is quite small, since even exchanges with large certificates generate less than 3 KB of data. It should also be noted that the results provided are only for the handshake phase; overhead for data records is lower due to the absence of the fragment header.

	DTLS		TLS	
	Packets	Bytes	Packets	Bytes
Client	3	446	2	228
Server	3	1015	2	857
Total	6	1461	4	1085

Table 1. Bytes and Packets transferred with PMTU 1500, Certificate size 562 bytes

	DTLS		TLS	
	Packets	Bytes	Packets	Bytes
Client	3	446	2	228
Server	4	2313	3	2105
Total	7	2759	5	2333

Table 2. Bytes and Packets transferred with PMTU 1500, Certificate size 1671 bytes

8.1. Latency

We measured latency of the TLS and DTLS handshakes on a local machine. DTLS and TLS handshakes took 42.9 ms and 41.5 ms respectively. The difference between these results is small due to the negligible RTT. In order to differentiate the two protocols, we introduced a 150 ms delay element, after which the DTLS handshake took 927 ms and the TLS handshake took 627 ms. This is exactly as expected, since DTLS results include one extra RTT for cookie exchange. Our measurements do not include the time taken for TCP connection establishment. Since session establishment requires a minimum of one RTT, this virtually eliminates the latency difference.

9. Related Work

9.1. IPsec

The design of DTLS is probably closest to that of IPsec. A number of the techniques that we used to make DTLS records safe for datagram transport were borrowed from IPsec. However, DTLS differs from IPsec in two important respects. First, DTLS is an application layer protocol rather than a network layer protocol. Thus, it is far easier to incorporate DTLS into an application since the DTLS implementation can simply be delivered with the application. This ease of deployment is to a great extent responsible for the wide use of TLS.

Second, DTLS uses the familiar TLS programming model in which security contexts are application controlled and have a one-to-one relationship with communication channels. By contrast, there is no standard IPsec API or programming model and the widely deployed IPsec implementations are all extremely difficult to program to. As previously noted, this is primarily a result of the fact that the IPsec key management model is extremely complex compared to that of TLS.

9.2. WTLS

There has been at least one previous attempt to add datagram capability to TLS: the Wireless Application Protocol Forum's WTLS [11]. However, WTLS made a large number of other changes, including integrating network transport with the security protocol, thus making it unsuitable for deployment on the Internet. In addition, WTLS does not appear to handle small path MTUs. Finally, the WTLS designers appear to have made a number of optimizations that lead to security flaws not in TLS [27] and is therefore not widely used.

9.3. SRTP

The Real Time Protocol (RTP) is widely used to carry multimedia traffic such as voice and video. RTP has no support for security. The IETF is currently considering standardization of the Secure Real Time Protocol (SRTP) [3] which is an application-specific security protocol for RTP. SRTP is substantially more limited than DTLS. First, it cannot be used to protect traffic other than RTP. Second, it relies on an external signaling protocol such as SIP to set up the keying material. By contrast, DTLS can be used to set up its own channel. However, in extremely bandwidth constrained applications SRTP has advantages over DTLS because its tight integration with RTP allows it to have lower network overhead. In situations where bandwidth is less limited DTLS would be a potential substitute for SRTP.

10. Future Work

Future work on TLS focuses mostly on integration with other protocols. Currently, we have an implementation of DTLS at the early toolkit stage. Our next step is to integrate it with some common datagram-based applications, which will give us feedback as to the suitability of our design. Our initial target is SIP. Since SIP already uses TLS in TCP mode, integrating DTLS in UDP mode is an attractive design choice and open source SIP implementations are readily available. Following SIP, we are considering integrating DTLS with a number of gaming and multimedia protocols. Moreover, integrating DTLS with a variety of other protocols will give us an opportunity to observe its performance behavior and make changes as appropriate.

We would also like to perform additional performance tuning on DTLS. Although TLS works well, subsequent performance analysis has uncovered some unfortunate interactions with TCP, especially with the Nagle algorithm [24]. As DTLS allows finer control of timers and record sizes, it is worth doing additional analysis to determine the optimal values and backoff strategies. Finally, we intend to do further analysis in an attempt to more tightly define the security bounds of DTLS.

11. Summary

We have described Datagram Transport Layer Security, a generic channel security protocol designed for use in datagram environments. DTLS is based on the well understood TLS protocol and like TLS is designed to provide a secure channel that mimics the semantics expected by existing application protocols. Due to simplicity and ease of deployment, DTLS provides an attractive alternative to IP security or custom application layer protocols. We have implemented DTLS as part of the popular OpenSSL cryptographic library and find that it provides acceptable performance and is relatively easy to program to.

12. Acknowledgements

The authors would like to thank Dan Boneh, Eu-Jin Goh, Constantine Sapuntzakis, and Hovav Shacham for discussions and comments on the design of DTLS. Thanks to the anonymous reviewers for their comments, which helped us improve the paper. Also thanks to Steve Kent for feedback that helped clarify many points. Dan Boneh, Lisa Dusseault, and Eu-Jin Goh provided comments on the paper.

The first author is supported by the NSF.

References

[1] F. Andreasen and B. Foster. Media Gateway Control Protocol (MGCP). RFC 3435, January 2003.

[2] E. B. Ramsdell. S/MIME Version 3 Message Specification. RFC 2633, June 1999.

[3] M. Baugher, D. McGrew, D. Oran, R. Blom, E. Carrara, M. Naslund, and K. Norrman. The Secure Real-time Transport Protocol. `draft-ietf-avt-srtp-08.txt`, May 2003.

[4] D. Boneh and D. Brumley. Remote Timing Attacks are Practical. *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[5] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. *In Proceedings of the Crypto*, August 2003.

[6] M. Crispin. Internet Message Access Protocol - Version 4rev1 (IMAP). RFC 3501, March 2003.

[7] T. Dierks and C. Allen. The TLS Protocol, Version 1.0. RFC 2246, January 1999.

[8] D. Eastlake. Domain Name System Security Extensions (DNSSEC). RFC 2535, March 1999.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol (HTTP). RFC 2616, June 1999.

[10] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the Scheduling Algorithm of RC4. *In Proceedings of SAC*, August 2001.

[11] W. A. P. Forum. WAP WTLS. WAP Forum protocol standard, November 1999.

[12] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409, November 1998.

[13] P. Karn and W. Simpson. Photuris: Session-Key Management Protocol. RFC 2522, March 1999.

[14] D. Kegel. The C10K Problem. `http://www.kegel.com/c10k.html`.

[15] C. A. Kent and J. C. Mogul. Fragmentation considered harmful. *In Proceedings of ACM SIGCOMM*, August 1987.

[16] S. Kent and R. Atkinson. IP Authentication Header (AH). RFC 2402, November 1998.

[17] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). RFC 2406, November 1998.

[18] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol (IPsec). RFC 2401, November 1998.

[19] E. Kohler, M. Handley, S. Floyd, and J. Padhye. Datagram Congestion Control Protocol (DCCP). `draft-ietf-dccp-spec-04.txt`, June 2003.

[20] C. Metz and B. Phan. PF_KEY Key Management API, Version 2. RFC 2367, May 1998.

[21] I. Mironov. (Not So) Random Shuffles of RC4. *In Proceedings of Crypto*, August 2002.

[22] J. Mogul and S. Deering. Path MTU Discovery. RFC 1191, November 1990.

[23] J. Myers and M. Rose. Post Office Protocol - Version 3 (POP). RFC 1939, May 1996.

[24] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

[25] E. Rescorla. HTTP Over TLS. RFC 2818, May 2000.

[26] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, and M. H. E. Schooler. SIP: Session Initiation Protocol. RFC 3261, June 2002.

[27] M.-J. O. Saarinen. Attacks against the WAP WTLS protocol. *CMS 99*, 1999.

[28] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003.

[29] D. Song, D. Wagner, and X. Tian. Timing Analysis of Keystrokes and SSH Timing Attacks. *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[30] The OpenSSL Project. `http://www.openssl.org/`.

[31] S. Vaudenay. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS .... *In Proceedings of Eurocrypt*, April 2002.