# The Trusted Cloud Transfer Protocol

Mathias Slawik
Service-centric Networking, Department of Telecommunication Systems
Technische Universität Berlin, Germany
http://www.snet.tu-berlin.de/menue/team/mathias_slawik/

***Abstract*—Contemporary cloud computing solutions incorporate HTTP intermediaries, such as reverse proxies, load balancers, and intrusion prevention systems. These act as TLS server connection ends and access HTTP/TLS plaintext to carry out their functions. This raises many concerns: increased security efforts, the risk of losing confidentiality and integrity, and potentially unauthorized data access. Current HTTP entity-body encryption technologies address these concerns by providing end-to-end security between user agents and origin servers. However, they present disparate deficiencies, e.g., inefficient presentation languages, message-flow vulnerabilities, and the circumvention of HTTP streaming. This paper introduces the Trusted Cloud Transfer Protocol (TCTP), which presents a novel approach to entity-body encryption overcoming these deficiencies. The pivotal idea of TCTP are HTTP application layer encryption channels (HALECs), which integrate TLS functionality into the HTTP application layer. TCTP can be deployed immediately, as it is fully HTTP compliant, and rapidly implemented, as required TLS libraries are widely available. The reliance upon the mature TLS protocol minimizes the risk of introducing new security threats. Furthermore, TLS brings the benefit of relative efficiency, which is demonstrated on the basis of an example TCTP implementation.**

## I. Introduction

This introduction first gives a technology roundup of TLS and HTTP. The subsequent section outlines the motivation of TCTP to solve current issues within cloud computing environments. Thereafter, the protocol is presented at a glance and the remaining structure of this paper is outlined.

### A. Technology Roundup: TLS and HTTP

Transport Layer Security (TLS), which evolved from Secure Sockets Layer (SSL) [1], is a protocol which provides ”privacy and data integrity between two communicating applications”. [2] It includes the TLS Handshake Protocol, which provides reliable authentication of both communicating parties, and the secure negotiation of an encryption algorithm and cryptographic keys.

The RFC states, that TLS is ”application protocol independent”, and ”does not specify how protocols add security with TLS”. In effect, TLS is not confined to any conceptual layer of communication, e.g., the OSI presentation layer, as it transforms arbitrary data into encrypted and authenticated TLS records.

The Hypertext Transfer Protocol (HTTP) is defined as ”an application-level protocol for distributed, collaborative, hypermedia information systems” [3]. The HTTP specification designates two kinds of HTTP software: *user agents* (”the client which initiates a request”, e.g., browsers), and *servers*. Servers can act as *origin servers* (”The server on which a given resource resides or is to be created.”) or HTTP intermediaries, i.e., a *proxy*, *gateway*, or *tunnel*. HTTP intermediaries allow data exchange between user agents and origin servers that are not connected directly.

### B. Security challenges addressed by TCTP

Contemporary RESTful cloud computing solutions are often accessed through HTTP intermediaries, such as the Amazon Cloud Load Balancer [4], HAProxy [5], and the TRESOR distributed cloud proxy [6]. Intermediary functionality includes load balancing, reverse proxying, SLA monitoring, audit logging, intrusion prevention, and request-based billing.

If a cloud service is accessed through HTTP/S [7] these intermediaries have to act as the TLS server connection ends as they need to access the HTTP plaintext to operate on its contents. As current cloud management functions require access to URLs, HTTP methods, headers, and response codes, those could not be implemented if the intermediary would tunnel HTTPS traffic [8]. Repealing these intermediaries is also no viable option in contemporary cloud solutions, as this severely diminishes the capabilities for managing cloud consumption.

Even though TLS connections achieve security on the transport layer, there are many issues with them having access to message plaintext. It is important to recognize, that none of the following concerns is limited to public, private, single-, or multi-stakeholder clouds:

**Risk of losing confidentiality and integrity.** When intermediaries act as TLS server connection ends, the confidentiality and integrity of all HTTP communication is dependent on each of those intermediaries. As architectures become more complex and sophisticated, the risks rise with each additional intermediary.

**Increased security efforts.** Intermediaries, as well as origin servers have to be protected with similar effort. Every additional service which is accessed through an intermediary makes a security attack onto these systems more worthwhile. Thus, the overall efforts of securing such architectures rise.

**Legal obligations.** Within sensitive areas, provisions require end-to-end confidentiality of exchanged data records, such as those contained in an HTTP payload. Under German law, for example, disclosing health records contained in the HTTP payload to these intermediaries would be a criminal act [9].

**Risk of unauthorized access.** The risk of unauthorized access using session hijacking or reusing intercepted credentials rises as intermediaries process cookies or HTTP Basic authentication as plaintext.

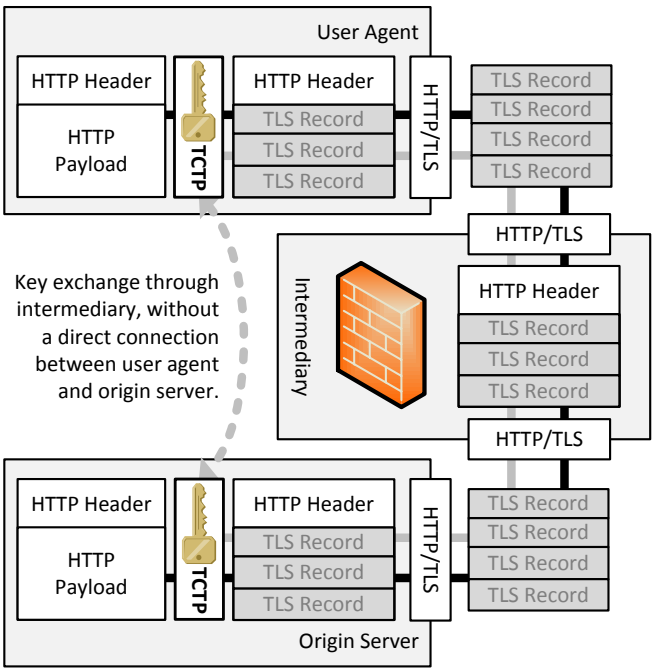


Fig. 1. TCTP secured communication

**Need-to-know principle.** The basic need-to-know principle means, that every component should only have access to the information, which it needs to carry out their function. This principle is not met by such cloud computing architectures, as only the origin servers need to have access to the entity-body and not the intermediaries. Most of the management functions carried out by intermediaries only need information contained in the HTTP headers and not the whole HTTP entity-body.

### C. TCTP at a glance

TCTP encrypts and authenticates the HTTP payload using TLS at the application layer, so that in effect all headers are still accessible by cloud computing intermediaries, while all of the issues mentioned in the preceding section are addressed by the entity-body encryption. Encryption keys and cipher suites are negotiated by wrapping the TLS Handshake Protocol into HTTP payload and sending it through the intermediaries. The reliance on this secure handshake minimizes the risk, that any intermediary intercepting these messages can act as a man-in-the-middle and compromise TCTP security. The term ”cloud” was included into the protocol designation in order to highlight the most compelling application domain, and not to express a cloud specificity of TCTP.

Figure I-C depicts TCTP secured communication. The payloads of all HTTP messages are transformed into encrypted and authenticated TLS records, whose communication paths are represented by the black line. This encryption employs keys exchanged between user agent and origin server through the intermediary, denoted by the grey lines. The HTTP messages containing an unencrypted HTTP header and TCTP encrypted payload are sent through HTTP/TLS connections, so that the headers are secured on the transport layer. The intermediary has access to the HTTP headers which allows it to perform management functions while the transmission of the entity body, e.g., sensitive patient data, from the origin server to the user agent stays private.

The next chapter analyses the state-of-the-art and demonstrates the improvements made by TCTP. After this, the protocol components are explicated. Before coming to a conclusion, an example TCTP implementation is evaluated, and its future development is outlined.

## II. Related work

In this section, four related technologies are presented, which can be used to achieve entity-body encryption. They are analyzed in the following, together with the proposed TCTP protocol.

**Secure/Multipurpose Internet Mail Extensions.** S/MIME [10] is a technology for signing and encrypting emails and other arbitrary data.

**XML Encryption and Signature** XML Encryption [11] and Signature [12] are W3C recommendations for cryptographic functions on XML and arbitrary data. Both are applied as SOAP [13] message encryption and signing within the WS-Security extensions [14]. As exemplified in the WS-Security based OSCI [15] protocol, TLS is often applied additionally in order to achieve transport layer security.

**HTTPSec.** The HTTPSec protocol realizes HTTP entity-body encryption, while being compatible to HTTP/1.1. The specification is not available on-line anymore, but still accessible through the Internet Archive [16].

**Secure HTTP.** S-HTTP [17] is a protocol encapsulating and encrypting an HTTP request in an S-HTTP request. This contradicts the basic TCTP prerequisite of giving intermediaries access to the HTTP plaintext. Nevertheless, as S-HTTP also considers any kind of data as an S-HTTP body, a modified version is conceivable.

### A. Requirements for entity-body encryption

All of the related technologies presented disparate deficiencies regarding to the requirements for entity-body encryption, which are shown in table 2. The symbols +, ○ and – specify the degree of requirement fulfillment, respectively ’full’, ’partial’ and ’none’.

**General considerations.** Any secure on-line exchange of keys requires at least one round-trip to the server. Furthermore, unnecessary round-trips for failed requests (e.g. requests where encryption is required but not applied or vice versa) can be reduced by an encryption capability discovery mechanism - at the cost of an additional one-time round-trip to the server. As the required number of round-trips for those functions is a conceptual constraint, further round-trip reduction is not considered as a requirement for HTTP entity-body encryption.

**R1 Efficient presentation.** Every decrease of transmitted or to be processed data can positively impact the performance of network applications. Thus, any entity-body encryption should employ the most efficient presentation language for the encrypted data.

**R2 Message-flow protection** TLS can detect and prevent replayed or reordered encrypted content by authenticating

| | TCTP | S/MIME | XML Enc/Sig | HTTPSec | (S-HTTP) |
|---|---|---|---|---|---|
| R1 | + | − | − | − | − |
| R2 | + | − | − | ○ | − |
| R3 | + | ○ | ○ | ○ | ○ |
| R4 | + | ○ | ○ | − | ○ |
| R5 | + | − | − | + | + |
| R6 | + | − | − | − | + |
| R7 | + | + | + | − | - |

Fig. 2. Requirements fulfillment overview

messages using keyed hashing. [1] Any entity-body encryption should likewise avert these kinds of behavior to prevent security vulnerabilities.

**R3 Encryption capability discovery** To prevent additional round-trips for messages declining or requiring encryption, the encryption capabilities of a server should be discoverable before exchanging application data. That makes it possible for a user agent to decide for which URLs it is acceptable, required, or forbidden to send an encrypted entity-body.

**R4 Streaming capabilities** Entity-body encryption should not prevent HTTP streaming, i.e., it should have the ability to authenticate and process fragments of an entity-body. This also permits handling messages as they arrive, which is more efficient than waiting until they are received completely.

**R5 Secure key exchange** The secure exchange of keys is a most basic requirement for an entity-body encryption.

**R6 Algorithm negotiation** Different security algorithms should be negotiable to allow distinct capabilities and prevent using algorithms, which are found to be insufficient at a later time.

**R7 Implementation support by existing software** Any mechanism for entity-body encryption would have to be added to user agents and origin servers. These implementation efforts can be considerably decreased, if supporting libraries and components are available for applying the technology.

The following paragraphs detail the information contained in the table in the form [RX+], where RX stands for the requirement and +, − or ○ specify the degree of fulfillment.

**TCTP** fulfills all of the requirements: The TLS presentation language employed by TCTP is the most space efficient of all investigated technologies [R1+]. HTTP application layer encryption channels protect the message-flow [R2+]. The TCTP discovery mechanism allows encryption capability discovery [R3+]. As the entity-bodies are fragmented by TLS, they can be processed in a streaming fashion [R4+]. The TCTP handshake provides secure key exchange with forward secrecy [R5+], and algorithm negotiation [R6+]. The implementation is supported by mature libraries (e.g., OpenSSL [19]), operating system components (e.g., Microsoft Windows Secure Channel [20]), and programming language integrations (e.g., Java Secure Socket Extension (JSSE) [21]) [R7+]. Building upon widely used software accelerates the implementation of TCTP, as shown by the TCTP prototype in Section IV.

[1] See [18] for reference of the HMAC scheme used in TLS.

**Secure/Multipurpose Internet Mail Extensions** The verbose headers and message structure of S/MIME are not as efficient as TCTP [R1−]. Binary representation is possible, but 7bit encoding is recommended by the RFC. Signing operates only on single messages [R2−]. A discovery mechanism is not specified, but could be realized by HTTP content negotiation, which would require additional round-trips to the server [R3○]. To support streaming, messages would have to be separated into MIME multipart segments incurring additional effort [R4○]. The RFC does neither specify secure key exchange [R5−] nor algorithm negotiation [R6−]. There are a number of S/MIME libraries, for example the JBoss RESTEasy framework [22] for entity-body encryption in RESTful applications [R7+].

**XML Encryption and Signature** XML Encryption and Signature embodies XML and either Base64 or XML-BOP MIME overhead [R1−]. Signing operates only on single messages [R2−]. Regarding Discovery the same conclusions as for S/MIME hold true [R3○]. As with S/MIME, XML data has to be costly separated to be processable in a streaming fashion [R4○]. "XML Encryption does not provide an online key agreement negotiation protocol" [11, Section 5.5, Para 2] [R5−]. Algorithms cannot be negotiated [R6−]. The implementation is supported by a number of libraries [R7+].

**HTTPSec** HTTPSec uses verbose HTTP headers containing Base64 encoded content [R1−]. The encrypted data can be in binary form. For message-flow protection only a weak message counter is fed to the MAC computation [R2○]. Only ad-hoc discovery is specified [R3○]. Streaming is not possible, as the MAC is part of the HTTPSec header [R4−]. Secure key exchange is possible [R5+], but not algorithm negotiation [R6−]. There are no existing HTTPSec implementations and supporting libraries [R7−].

**Secure HTTP.** The parenthesis in the table denote, that S-HTTP messages are not compatible to HTTP and therefore cannot be processed by any cloud computing intermediary, which prohibits its use for entity-body encryption. S-HTTP security headers are verbose [R1−] while encrypted data can also be in binary form. S-HTTP does not protect the message flow [R2−]. S-HTTP only allows for ad-hoc discovery [R3○]. S-HTTP specifies a MAC header computed over the whole entity, which prevents streaming, but the underlying CMS by now supports the fragmentation of data and could therefore allow streaming processing, if the RFC would be updated [R4○]. Secure key exchange [R5+] and algorithm negotiation [R6+] are supported. No implementation of S-HTTP, nor supporting libraries could be found [R7−].

## III. The Trusted Cloud Transfer Protocol

This chapter describes the elements of the Trusted Cloud Transfer Protocol and displays the corresponding HTTP messages. Within these examples the HTTP `Content-Length` headers and the chunked encodings are left out for simplification purposes.

### A. HTTP application layer encryption channels (HALECs).

HALECs are comparable to regular TLS connections: they are created by a handshake (i.e., the TCTP handshake), and

transform a data stream (i.e., the HTTP entity-body) into encrypted and authenticated TLS records. These are sent instead of a plaintext entity-body, and are authenticated and decrypted by the origin server. HALECs are identified by URLs, on which the TCTP handshake is performed.

As HTTP does not guarantee an unvarying order of parallel HTTP messages, an out-of-order TCTP response to parallel requests would invalidate the TLS HMAC and render the HALEC unusable. Parallel processing of TCTP traffic is possible nevertheless as more than one HALEC can be created.

HALECs are persisted by saving their TLS session state and security parameters and therefore do not have to be "closed". TCTP implementations may invalidate HALECs to release resources which were required for HALEC persistence. In this case, user agents would repeat the TCTP handshake to create new HALECs.

### *B. The TCTP discovery procedure.*

The TCTP discovery procedure allows TCTP clients to retrieve information on how to create HALECs for accessing protected resources. Every origin server establishes its own URLs for these purposes. It is performed by the HTTP `OPTIONS` method to the asterisk ("`*`") URL sending the `Accept: text/prs.tctp-discovery` request-header, as shown in the following listing:

Listing 1. TCTP discovery request
```
OPTIONS * HTTP/1.1
Host: www.example.com
Accept: text/prs.tctp-discovery
```

The TCTP discovery information consists of regular expressions of protected resource URLs, and another URL which provides means to perform a TCTP handshake to create HALECs to access these resources. Both are divided by a colon ("`:`"). URLs which are not protected are indicated by a line break (`CRLF`) following the colon. The expressions are matched against request URLs from top to bottom, where the first matched pattern decides the outcome of the discovery. In the example, the root URL, the service root URLs and all static assets are unprotected. Flexible discovery information allows different HALECs for different services (e.g. `/service-one/` and `/service-two/`). These could be operated by different parties, each having control over the secret TLS session states.

Listing 2. TCTP discovery response
```
HTTP/1.1 200 OK
Content-Type: text/prs.tctp-discovery

/:
/(service(.+?))?:
/(service(.+?)/)?static.*:
/(service(.+?)/)?.*:/\1/halecs
```

### *C. The TCTP handshake.*

The creation of an HALEC is performed by a `POST` request containing a TLS `client_hello` to the HALEC creation URL. When an HALEC is created, the origin server sends the response code `201 Created` and a `Location` response-header field referring to the URL of the new HALEC[2]. The HTTP response contains the TLS handshake reponse as an entity-body, which is shown in Listing 3.

Listing 3. TCTP HALEC creation
```
POST /halecs HTTP/1.1
Host: www.example.com

<TLS client_hello record>
```

```
HTTP/1.1 201 Created
Location: http://www.example.com/halecs/
 1kX28fAms

<TLS server_hello, certificate, server_key_exchange>
```

The remaining handshake is executed by sending an HTTP `POST` request to the HALEC URL containing TLS handshake records. Listing 4 illustrates this behavior.

Listing 4. TCTP Handshake
```
POST /halecs/1kX28fAms HTTP/1.1
Host: www.example.com

<TLS client_key_exchange, change_cipher_spec,
    finished>
```

```
HTTP/1.1 200 OK

<TLS change_cipher_spec, finished>
```

The HALEC URL can later be used to send TLS closure alerts, to perform a TLS renegotiation, and to close the HALEC by sending an HTTP `DELETE` request to the HALEC URL.

### *D. TCTP entity-body encryption.*

After the handshake established an appropriate TLS state, the HALEC is used to secure HTTP entity-bodies: the entity-body stream is transformed by TLS into encrypted and authenticated records. These are then sent instead of the plaintext payload to be authenticated and decrypted by the origin server.

The header field `Content-Encoding: encrypted` designates such messages. User agents request encrypted content through the `Accept-Encoding: encrypted` header. If an HTTP message contains a payload, the HALEC URL is sent as the first line of the payload, preceded by the encrypted entity-body. The user agent and origin server should be required to send a `Cache-Control` header of the value `no-cache`, so that no intermediary returns an encrypted server response from cache, as this would invalidate the HMAC.

Listing 5. TCTP entity-body encryption.
```
POST /patients/070386/details HTTP/1.1
Host: www.example.com
Accept-Encoding: encrypted
Content-Type: application/json
Content-Encoding: encrypted
Cache-Control: no-cache

http://www.example.com/halecs/1kX28fAms
<TLS application data records>
```

[2]This sequence closely follows Section 9.5 (`POST` method definition) of the HTTP specification.

| Req. Size | HTTPS | TCTP/S | Overhead |
|---|---|---|---|
| 1 kB | 35.49 ms | 37.14 ms | 4.63% |
| 2.5 kB | 34.41 ms | 36.11 ms | 4.94% |
| 5 kB | 35.12 ms | 35.65 ms | 1.50% |
| 7.5 kB | 33.81 ms | 37.65 ms | 11.38% |
| 10 kB | 34.72 ms | 35.45 ms | 2.08% |

Fig. 3. TCTP request and response processing overhead

```
HTTP/1.1 200 OK
Content-Encoding: encrypted
Cache-Control: no-cache

http://www.example.com/halecs/1kX28fAms
<TLS application data records>
```

The example shows how HTTP applications can be designed in such a way, that their HTTP communication shows *varying information confidentiality*: while the header signifies a general "update details of patient `070386`", the entity-body could convey sensitive medical information, which are protected by TCTP.

## IV. Implementation and Evaluation

This chapter presents the implementation of a TCTP enabled origin server and user agent. The server is evaluated regarding its performance characteristics. At the end of the chapter the further evolution of TCTP enabled software is illustrated as an outlook.

### A. The TCTP implementation

The TCTP implementation consists of a Rack middleware, which enables any Ruby origin server and application framework (e.g. Ruby on Rails) to support TCTP, and an HALEC implementation used to evaluate the performance overhead introduced by the middleware. The handshake and encryption primitives pivotal for TCTP rely on the standard Ruby OpenSSL[3] module. To reduce the effort of porting this TCTP implementation to other programming environments, it only relies on common classes, such as TCP and TLS sockets, and features a simple architecture.

### B. Performance evaluation

The TCTP evaluation consists of accessing different resources between 1kb and 10kb in size, as those represent the present average HTML transfer characteristics [23]. The HTTPS request/response processing times are compared to TCTP over HTTPS (denoted as TCTP/S in the table), averaged over 20 test runs. The benchmark client is an Intel Core i7-3520M Laptop running under Windows 8.1. The average processing times and the respective TCTP/S overhead are shown in Figure 3.

Figure 3 shows only fixed processing times for TCTP encryption and decryption, and does not include handshake and network round-trips. To assess the relative overhead of TCTP, the additional round-trips for the TLS and TCTP handshake have to be considered, as well as the handshake processing. In the benchmark setup, the average TCTP handshake took 129 ms, while the average TLS handshake took 23 ms. At last, network latency has also be taken into account when assessing the relative overhead of an end-to-end entity-body encryption. Figure IV-B shows the resulting relative overhead of TCTP/S with respect to different network latencies and a varying number of requests.

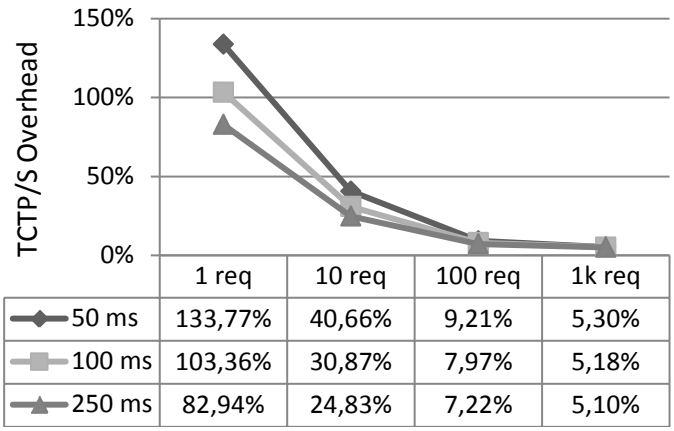


Fig. 4. Relative TCTP overhead

The TCTP overhead adds up to a considerable 133,77% when requesting only one resource over a fast connection. As the number of requests made through an established TCTP/S connection rises, the relative overhead approaches the average processing overhead of 4.86%, which is not substantial. Especially the non-linear increase of the processing overhead suggests improvement opportunities within the prototype implementation.

### C. Further development

This section outlines some of the next steps in the development of TCTP software, *user agents*, e.g., common browsers and HTTP libraries, *origin servers*, and *application frameworks*. Furthermore, *intermediaries* could also be extended by TCTP functionality. Such intermediaries could securely and transparently bridge clients from a protected company network to TCTP enabled cloud services with the additional benefit of not having to adjust any user agent on the company network.

*1) Streaming optimization:* TCTP processing overhead could be reduced by aligning the fragmentation introduced by the HTTP `chunked` transfer coding to the size of TLSPlaintext blocks, so that they are not separated into two or more HTTP chunks.

*2) HALEC Cookie Binding:* To prevent session hijacking using intercepted cookies, the validity of authentication cookies should be bound to a specific TLS session ID. This prevents the use of intercepted authentication cookies with self-created HALECs, as these HALECs would be associated with a different TLS session ID, than the one used to issue the cookie.

[3]The library name has been the same since 1998 and could be misleading. OpenSSL is still under development and supports every version of TLS and SSL.

*3) Peer Certificate Validation:* Besides securely validating peer certificates, as presented by Georgiev, et al. [24], TCTP implementations should issue a warning, if both the HTTPS connection and the TCTP entity-body encryption use the same certificate. Cloud computing intermediaries have access to the private key of the HTTP/TLS certificate and therefore could also access the plaintext of the entity-bodies secured by TCTP.

*4) TCTP discovery circumvention:* There are some measures to mitigate the circumvention of TCTP discovery: Preseeded discovery information, comparable to [25], DNS TXT records containing the discovery information or their cryptographic hash, and using historic data to detect important changes of TCTP discovery information, e.g., an origin server suddenly ceasing to offer TCTP discovery information.

*5) Support for imminent protocols.:* There are two notable transport protocol proposals for future RESTful cloud computing services: SPDY [26] and HTTP/2.0 [27]. Both allow the use of TLS, but do not consider any entity-body encryption. As the semantics of the `Content-Encoding` entity-header prevail, TCTP is equally applicable to these protocols. How TCTP would fit conceptually into these protocols is unclear, as both proposals do not specify communication semantics through intermediaries.

## V. Conclusions

This paper introduces a novel mechanism for providing end-to-end HTTP entity-body confidentiality, the Trusted Cloud Transfer Protocol. The mechanism overcomes deficiencies of related technologies and approaches issues found within common cloud computing environments. An evaluation shows promising performance characteristics and modest requirements for TCTP implementations. In the near future, TCTP will be developed further within the TRESOR project [28] in the areas outlined in Section IV-C. As TCTP is not limited to a specific set of use cases and is compatible to all existing cloud intermediaries, a wide adoption by researchers and practitioners is conceivable.

## Acknowledgment

This work was carried out within TRESOR, which is funded as a Trusted Cloud project by the Federal Ministry of Economy on the basis of a resolution passed by the German Bundestag.

## References

[1] A. Freier, P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101, IETF, 2011.

[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, IETF, 2008.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, IETF, 1999.

[4] Amazon Web Services, Inc., "Elastic Load Balancing," http://aws.amazon.com/elasticloadbalancing/, 2013. [Online]. Available: http://aws.amazon.com/elasticloadbalancing/

[5] W. Tarreau, "HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer," http://haproxy.1wt.eu/, 2013.

[6] D. Thatmann, M. Slawik, S. Zickau, and A. Küpper, "Deriving a distributed cloud proxy architecture for managed cloud service consumption," in *Proceedings of the 6th Intl. Conference on Cloud Computing (CLOUD 2013)*. Santa Clara, California, USA: IEEE, Jun 2013, pp. 614–620. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2013.56

[7] E. Rescorla, "HTTP Over TLS," RFC 2818, IETF, 2000.

[8] A. Luotonen, "Tunneling TCP based protocols through Web proxy servers," http://tools.ietf.org/html/draft-luotonen-web-proxy-tunneling-01, IETF, 1998. [Online]. Available: http://tools.ietf.org/html/draft-luotonen-web-proxy-tunneling-01

[9] German Federal Ministry of Justice, "StGB § 203 Verletzung von Privatgeheimnissen," http://www.gesetze-im-internet.de/stgb/__203.html, 2013, german.

[10] B. Ramsdell and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification," RFC 5751, IETF, 2010.

[11] T. Imamura, B. Dillaway, and E. Simon, "XML Encryption Syntax and Processing," http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/, 2002.

[12] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon, "XML-Signature Syntax and Processing," http://www.w3.org/TR/2001/PR-xmldsig-core-20010820/, 2001. [Online]. Available: http://www.w3.org/TR/2001/PR-xmldsig-core-20010820/

[13] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," http://www.w3.org/TR/soap12-part1, 2007.

[14] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker, "Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)," https://www.oasis-open.org/committees/download.php/21257/wss-v1.1-spec-errata-os-SOAPMessageSecurity.htm, 2006. [Online]. Available: https://www.oasis-open.org/committees/download.php/21257/wss-v1.1-spec-errata-os-SOAPMessageSecurity.htm

[15] Koordinierungsstelle für IT-Standards, "OSCI - Startseite," http://www.osci.de, 2013.

[16] S. Fowler, "Public key authentication for HTTP," http://www.httpsec.org, 2010, accessible through the Internet Wayback Machine: http://web.archive.org/web/20100926084623/http://www.httpsec.org/.

[17] E. Rescorla and A. Schiffman, "The Secure HyperText Transfer Protocol," RFC 2660, IETF, 1999.

[18] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology CRYPTO '96*, ser. Lecture Notes in Computer Science, N. Koblitz, Ed. Springer Berlin Heidelberg, 1996, vol. 1109, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1007/3-540-68697-5_1

[19] The OpenSSL project, "OpenSSL: The Open Source toolkit for SSL/TLS," http://www.openssl.org/, 2013.

[20] Microsoft, Inc., "Transport Layer Security Protocol (Windows)," http://msdn.microsoft.com/en-us/library/aa380516(v=vs.85).aspx, 2013.

[21] Oracle, "Java Secure Socket Extension (JSSE) Reference Guide for Java Platform Standard Edition 7," http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html, 2013.

[22] jBoss.org, "Body Encryption and Signing via SMIME - RESTEasy JAX-RS," http://docs.jboss.org/resteasy/docs/2.3.6.Final/userguide/html/ch38.html, 2013.

[23] S. Souders, "The HTTP archive," http://httparchive.org/, 2013.

[24] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating SSL certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382204

[25] Electronic Frontier Foundation, "HTTPS Everywhere," https://www.eff.org/https-everywhere, 2013.

[26] M. Belshe and R. Peon, "SPDY Protocol - Draft 1," 2012. [Online]. Available: http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft1

[27] M. Belshe, R. Peon, M. Thomson, and A. Melnikov, "Hypertext Transfer Protocol version 2.0," 2013. [Online]. Available: http://http2.github.io/http2-spec

[28] TRESOR consortium, "TRESOR project website," http://www.cloud-tresor.com/, 2012. [Online]. Available: http://www.cloud-tresor.com/