Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Cacheability of encrypted Internet Traffic

by

Armin Slopek

First supervisor:	Prof. Dr. Karl Jonas
Second supervisor:	Michael Rademacher, M.Sc.
Handed in:	June 9, 2016

Persönliche Erklärung

Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Bonn, (June 9, 2016) _____

(Armin Slopek)

Contents

List of Tables

List of Figures

List of Abbreviations

ACL Access Control List
CA Certificate Authority
CC Credit Card
CPU Central Processing Unit
DNS Domain Name System
FIFO First in First out
GiB Gibibyte (2^{30} bytes)
HTML Hyper Text Markup Language
HTTP Hyper Text Transfer Protocol
HTTPS Secure Hyper Text Transfer Protocol
IP Internet Protocol
ISP Internet Service Provider
JSON JavaScript Object Notation
KiB Kibibyte (2^{10} bytes)
LFU Least Frequently Used
LRU Least Recently Used
LZW Lempel-Ziv-Welch
MiB Mebibyte (2^{20} bytes)
MitM Man-in-the-Middle
OS Operating System
PKI Public Key Infrastructure
PMS Premaster Secret
RE Redundancy Elimination
RFC Request for Comments
SHA Secure Hash Algorithm
SSL Secure Sockets Layer
TCP Transmission Control Protocol
TCP/IP Transmission Control Protocol / Internet Protocol
TLD Top-level domain
TLS Transport Layer Security
UDP User Datagram Protocol
URL Uniform Resource Locator
XML Extensible Markup Language

1 Introduction and Motivation

The relevance of secure end-to-end encrypted connections between clients and servers has increased in the past years and will continue to do so in the future. As shown in [19], encrypted internet traffic through fixed (i.e. not mobile) connections in North America rose from 29.10% to 37.50% from April 2015 to January 2016. In Europe, two thirds of non-mobile internet traffic are already encrypted. As far as mobile connections are concerned, 64.52% of North American and 61.25% of European internet traffic are encrypted.

While the encryption of internet traffic results in security and privacy benefits, encryption is also associated with some disadvantages: Intermediate caches attached to a network-architecture are incapable of decrypting and storing transmitted objects for future requests. Even if objects were held in storage in an encrypted state, they would not be reusable as the session-key used to encrypt the objects changes every time a new connection is established. Furthermore, requests sent from the client to an internet-server are also encrypted. Hence, a cache would not be able to find and serve the appropriate object without decrypting the request [20].

Load reduction plays an important role on low-performance or cost-intensive network links, e.g. when supplying remote areas with internet connections via satellite-links[20]. With the introduction of 5G cellular networks, intermediate caches and caching technologies will become even more relevant [1, 8]. Thus, analyzing the impact of encryption on the effectiveness of intermediate caches is a paramount concern.

In previous studies [11, 20], caching allowed to achieve a load reduction by 7%. Therefore, it may be possible to reduce the load on a given network link by allowing a cache to examine encrypted traffic.

The focus of this study is on an intermediate Man-in-the-Middle (MitM)-enabled cache, which is provided with the means to examine encrypted traffic. Using this cache, the experiment conducted in [20]is repeated under otherwise identical conditions to examine whether network load can be reduced by MitM-enabled caches. The main questions this study aims to answer are:

- How is a MitM-enabled cache correctly implemented and established?
- Does the deployment of a MitM-enabled cache yield any gains in terms of load reduction compared with a standard cache?
  If so, how much improvement can be expected?
- What are the consequences for users' privacy and security?

The MitM-enabled cache will be implemented using the squid proxy-cache [26]. It supports the caching of objects transmitted via a SSL-secured connection [22, 23].

This paper is structured as follows: The second chapter discusses the background of secure internet connections, the underlying Public Key Infrastructure (PKI) and current means of thwarting secure connections. Then, a short introduction to caches and caching technologies is given and the principles are applied to the cryptographic background of secure connections. Given this information, the current state of the art concerning the caching of encrypted data is abstracted and summarized. The following chapter presents the methodology, experimental setup and tools used to analyze the collected data, followed by an interpretation and discussion of the results. Finally, the conclusions and a perspective on future work are given.

2 State of the Art

This section gives an overview on the technical background of secure HTTP (HTTPS) connections. Firstly, the underlying PKI is explained in section 2.1 and its importance for HTTPS is emphasized in section 2.2. These principles are used to explain a serious threat jeopardizing secure connections: Man-in-the-Middle-Attacks (section 2.3).

Section 2.4 introduces the most important aspects of caching. Finally, research on reducing bandwidth usage is presented in section 2.5.

2.1 Public Key Infrastructures

PKIs are vital to today's information and communication networks. Their purpose is to testify the identity of communication partners before the information flow between them begins (also see Section Establishing a secure HTTP Connection). To achieve this goal the Certificate Authorities (CAs) provide certificates which attach identities to their respective public keys.

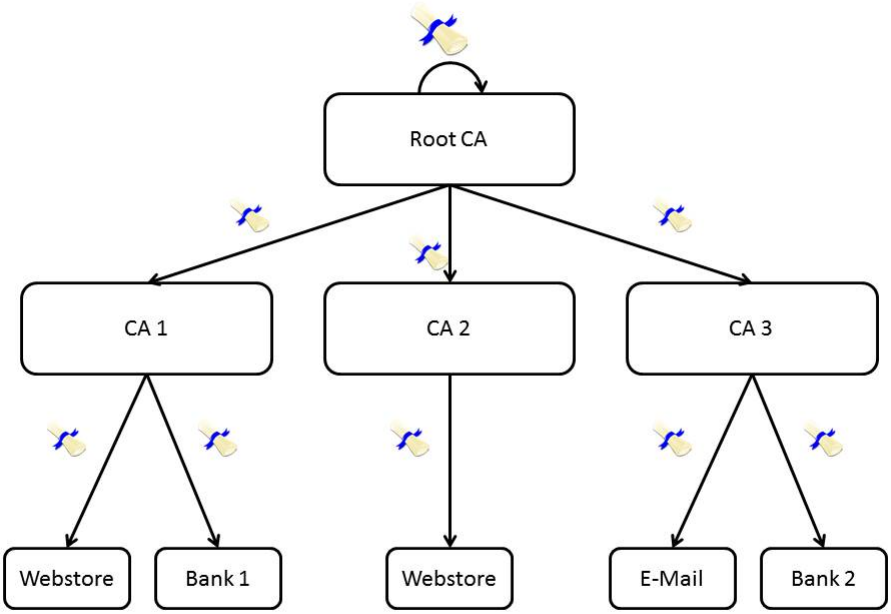

Figure 1: Generic Public Key Infrastructure (in dependence on [12])

A PKI is a structure consisting of multiple CAs as illustrated in Figure 1. The Root-CA uses a self-signed (root-)certificate that is typically deposited in web browsers and Operating Systems (OSs). Its sole purpose is to create certificates for other CAs by

signing their public keys. Hereby the other CAs are able to issue certificates to service providers such as banks and e-mail providers. A more detailed introduction to PKI can be found in related literature [5, 7, 12].

2.2 Establishing a secure HTTP Connection

When data and information transmitted via the Hyper Text Transfer Protocol (HTTP) are confidential or at least one communication partner needs to be identified with absolute certainty, Transport Layer Security (TLS)[10] – more specifically, the TLS handshake – is used to authenticate the respective partner(s) and to create an encrypted channel between them. This secure extension of HTTP is also referred to as HTTPS. The TLS handshake protocol is carried out as follows[10]:

1. Exchange of Client-Hello and Server-Hello messages to
   - negotiate algorithms and
   - exchange cryptographic nonces (random numbers).
2. Exchange cryptographic parameters for the algorithms negotiated in the first step. This results in the Premaster Secret (PMS).
3. Optional: Exchange certificates to let client and server authenticate each other as needed.
4. Generate the master secret from the PMS and cryptographic nonces.
5. Verify the validity and integrity of security parameters.

A typical scenario for the use of HTTPS and hence, the TLS handshake protocol is a user browsing websites where confidential data, like passwords, Credit Card (CC) numbers and other credentials, are transmitted. There, when a user (client) initiates a secure connection to a service provider (server), in the third step the server sends its certificate along with all other CA and Root-CA certificates. By verifying this *chain of trust*, the client can confirm the server's identity and verify that no MitM-Attack occured (see Section Man-in-the-Middle-Attack).

Instead of executing the full TLS handshake protocol, a previous session can be resumed or an existing session can be duplicated by using a modified version of the protocol abstracted above. The client then includes the session ID of the session to be resumed or duplicated in the Client-Hello message. If the server finds a match for the ID in its session cache *and* if it concedes to resume/duplicate the session, it sends a Server-Hello message containing the same session ID.

See Section 7 of RFC 5246 [10] for more details.

2.3 Man-in-the-Middle-Attack

The MitM-Attack is a form of an *impersonation-attack*[2]. An attacker controlling the communication channel between two (or more) parties can perform such an attack by "intercepting and forwarding the traffic that would normally flow directly between the client and the server" [2], as outlined in Figure 2.

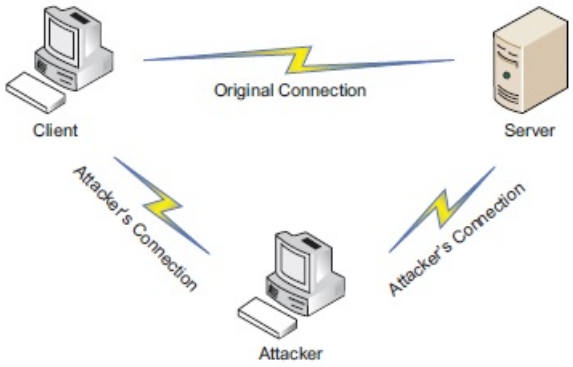

Figure 2: Man-in-the-Middle-Attack [2]

In the context of establishing an HTTPS session, the attacker would interfere when the server sends its public key to the client. He substitutes the server's public key with his own and forwards the altered message to the client. The client then encrypts all subsequent messages with the attacker's public key instead of the server's public key, enabling the attacker to decrypt and comprehend the flow of information between client and server. The use of digital certificates (see Establishing a secure HTTP Connection) allows the client to verify the validity of a public key and helps to prevent MitM-Attacks. A (technical) background on how a MitM-Attack is performed and how one can defend against such an attack [18].

2.4 Caching

Regarding caching three important questions are answered briefly in this section, based on the findings in [20]:

- Where to cache?
- What to cache?
- How to cache?

Where to cache?

The strategic positioning of an intermediate cache determines the number of users connected to it. If an intermediate cache is placed closer to the user and his devices, there will be little to no cost savings for the Internet Service Provider (ISP). However, low-bandwidth connections may gain a boost in latency and throughput. On the other hand, the ISP will benefit from a cache that is placed farther away from the users. This is because more users are connected to one cache, allowing it to serve duplicate requests for the same object requested by many users and hence, reduce bandwidth usage on backhaul links. The deployment of a cache can lead to cost savings, for example when it is placed at the base station of a satellite link.

What to cache?

The most common content types transmitted via HTTP are text, image, video and application data. The latter also consists of text, for example JSON or XML formats. The hit ratios[1] quantified in the experiment in [20] indicate an excellent cacheability of image contents (40% hit ratio), and a good cacheability of text and application data (both with hitratios of 20%). Even though video contents caused the highest bandwidth consumption, they had a very poor hit ratio (almost 0%).

Another aspect is the size of the requested objects. The measured 7% reduction of bandwidth usage were distributed as follows: Only objects with a size between 1 KiB and 1 GiB contributed to the reduction of bandwidth usage. The most effectively cacheable objects were sized 1 MiB to 10 MiB (1%), 100 KiB to 1 MiB (1.5%) and 10 KiB to 100 KiB (1%).

How to cache?

A variety of strategies exists for recording and organizing data and objects within the cache. Firstly, there are two approaches for recording the objects. Traditionally, they are recorded after they are requested and transmitted for the first time; this approach is referred to as *passive* or *reactive caching*. Newer concepts include the use of machine learning and profiling to store objects inside the cache even before they are requested. This does not directly result in bandwidth savings, since the object still needs to be transmitted at least once. However, this technique allows the mitigation of peak loads within a given network.

The recorded objects can essentially be organized in two ways: *Web-based* and *URL-based* caching respectively, where the objects are stored along with their Uniform Resource Locator (URL) and a request counter. Another approach is Redundancy Elimination (RE) caching. Two separate hardware platforms communicate with each other and transmit hash values of bytestreams instead of the corresponding bytestreams themselves. If the hash value and its bytestream are stored in one of the stations, the original

[1] Relative number of requests that could be served from the cache.

bytestream is recovered by that cache. This technique is protocol-independent as it operates on the raw bytestream.
If the total size of objects exceeds the available space, cache replacement strategies come into effect. They are similar to cache-replacement-strategies used in other contexts, e.g. CPU caches. A few examples are Least Frequently Used (LFU), Least Recently Used (LRU), First in First out (FIFO) or the replacement of a randomly selected object.

A more detailed introduction to caching is given in [20].

2.5 Caching of encrypted Data

In 2001, Mraz proposed a server-side architecture "to offload SSL set-up protocol activity [...] to a scalable array of SSL Handshake Protocol specific servers" [16] that would allow proxies to cache the encrypted objects. The problematic relationship between encryption and cacheability mentioned in the introduction of this work and in [20] was also recognized by Mraz.

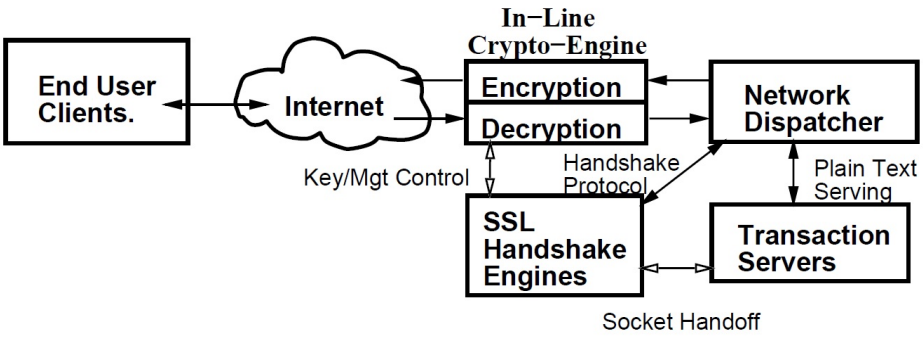

Figure 3: Overview of the Secure Blue architecture [16]

The *Secure Blue* architecture consists of several components (see Figure 3). In this architecture, the *Transaction Servers* represent the web servers used to host securely accessible websites such as e-commerce or banking sites. When a client invokes a new SSL connection, the *SSL Handshake Engines* "provide the initial handshake [...] masquerading as the original TCP/IP address" [16]. The decryption and encryption of incoming/outgoing traffic is processed at the *In-Line Crypto-Engine*; both the *Transaction Server* and *Network Dispatcher*[2] send and receive data in plain text. From the client's point of view, the Secure Blue architecture is transparent.

This architecture allows the actual web server (transaction server) to process and serve all requests in plain text and thus relieving its CPU and reducing load by transferring

[2]Load-balancing mechanism.

resource-demanding, cryptographic computations to the stand-alone handshake servers and crypto-engines.

Caches cannot effectively be deployed beyond the crypto-engines, for caching encrypted traffic without taking further measures is not feasible as previously discussed. When deploying Secure Blue as intended on the server-side, only the server may benefit from a lower bandwidth usage by positioning a cache between server and crypto-engine.

However, it would not make sense to deploy Secure Blue as is on the client-side. This is because rather than *responding* to many concurrent TLS handshakes, a client usually *invokes* only few TLS handshakes at the same time, if any. More commonly, quite large time slots (compared to a server) may occur between two TLS handshakes on one client.

In contrast to this procedure, Boneh et al. introduced a different approach in 2004 [6]. Their *fast-track* mechanism consists of two extensions for the TLS handshake protocol. A first extension is used "in an ordinary TLS handshake" [6] which is intended to "negotiate the future use of fast-track" [6]. A second extension then carries out the actual fast-track mechanism, after the client and server agree to use it in the future.

Fast-track reduces bandwidth consumption during the TLS handshake without reducing computational burden on the CPU. This is achieved by a client-side cache that contains *determining parameters* about the server's configuration (certificate chain, Diffie-Hellman group[3][4], client-side authentication[4]), preferred cipher suite and compression method. The collection of this information is carried out by the first extension described above.

Once the first, ordinary handshake is completed, a following fast-track handshake is processed in three steps (instead of four) [6]:

1. The client sends a Client-Hello message including the fast-track extension and hash value of the determining parameters[5]. Hereby client and server are enabled to confirm they both work with the same set of determining parameters.

2. "The server builds its own version of the parameters, and ensures that the hashes match" [6]. If this is the case, it responds to the client with *ChangeCipherSpec* and *Finished* messages.

3. The client also sends *ChangeCipherSpec* and *Finished* messages to the server.

[3] More information on DH groups can be found in section 15.3.2 in [12].

[4] if applicable

[5] A fixed algorithm (SHA-1) is used for this step to avoid an undesirable handshake on which algorithm to use.

Performance gains depend on several factors: Cipher suite, compression method and whether or not a client is required to authenticate him or herself. If no client authentication is required, up to 72% of the traffic caused by TLS handshakes can be saved; otherwise, the bandwidth savings are only 28%. However, the *absolute* maximum gain in bandwidth savings is only 1,108 bytes when the cipher suite *TLS_DHE_RSA_WITH_3DES_SHA* is used and client authentication is mandatory [6]. The use of fast-track has no effect on bandwidth savings and cacheability of subsequently transmitted data.

Cooley et al. define a mechanism (2010) [9] which also focuses on reducing TLS handshake overhead on TCP [13] connections. Unlike fast-track it is not implemented as an extension for the TLS protocol. Therefore clients and servers do not need to be aware of or specifically support it.

Two caches are placed between or directly on the client and server. These caches communicate with each other in a way quite similar to RE caching (see Section Caching). When either cache observes the transmission of a certificate or certificate chain, the certificates are replaced by identifiers (*compression*). The other cache then receives the TLS handshake packets including the aforementioned identifiers. Now, two cases may occur:

1. The cache can associate the correct certificate to the identifier (*cache-hit*)
2. There is no identifier/certificate pair available in the cache's storage (*cache-miss*)

In the first case, where a cache-hit occurs, the identifier is simply replaced again by the original certificate (*expansion*) and the packet is forwarded to its destination. When the certificate cannot be successfully restored, the cache drops the packet. This causes the underlying TCP connection to re-transmit it. The other cache recognizes the re-transmission and forwards the packet including the original certificate(s) and added identifier(s). The cache that dropped the packet in the first place adds the certificate/identifier pair(s) to its storage for future use and forwards the packet. A successfully performed *proxied* handshake is sketched and compared to a standard handshake in Figure 4.

Cooley et al. evaluate the performance of their technique using three different certificate sizes and observe a reduction of bandwidth usage of "at least 50%" [9]. Similar to Boneh et al.'s fast-track, the saved bandwidth per handshake ranges between one and two KiB and also does not allow further reduction of bandwidth usage when transmitting application data.

Monica and Raluca propose an architecture in [15] (2012) which resembles Cooley et al.'s mechanism [9] for the most part. Similar to [9], two caches that are placed some-

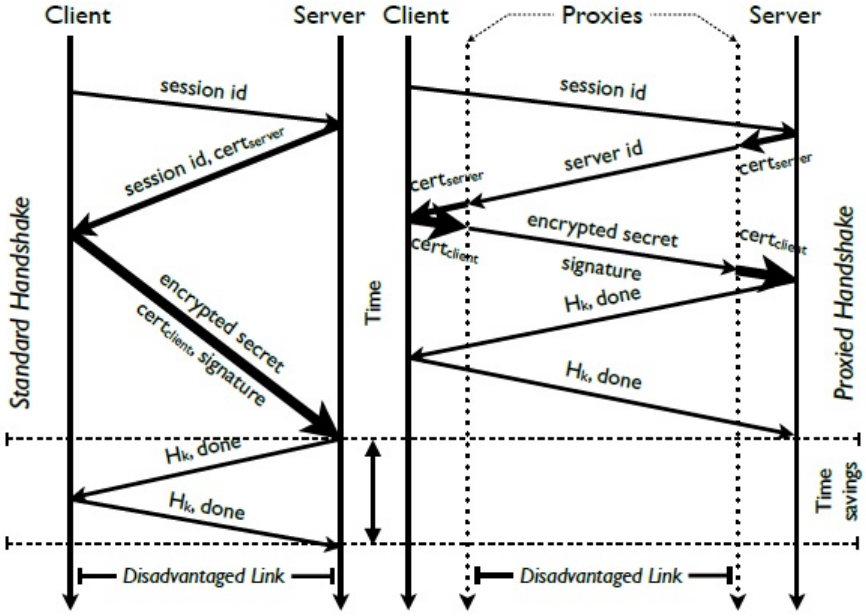

Figure 4: Standard and proxy-based TLS handshake with mutual authentication [9]

where on the link between client and server communicate with one other. As a key difference, Monica and Raluca implemented their method "as an extension to the ServerHello message" [15]. Where Cooley et al. use identifiers to replace certificates, this method uses the Lempel-Ziv-Welch (LZW) algorithm [28]: When the server sends its certificate to the client, it is compressed by the server-side proxy and de-compressed by the client-side proxy. Compression and de-compression only work in one direction; the case where a client is required to authenticate him or herself using certificates is neglected. The proposed mechanism is evaluated with a certificate of size 928 bytes and bandwidth usage reduction is claimed to be nearly 50% [15].

3 Methodology

The publications discussed in section 2.5 concentrate on relieving a web-server's (computational) resources [16] or speeding up the TLS handshake [6, 9, 15]. While the latter do not allow for the traffic flow occurring after the TLS handshake to be cached, Mraz's approach [16] supports caching to a certain degree: A cache can be placed at any point between the crypto engine and the transaction server (see Figure 3), enabling the reduction of bandwidth usage within the web-server host's network.

In this work a mechanism for load reduction *after* an HTTPS-secured connection has been established is described and analyzed. The mechanism involves a MitM-enabled proxy-cache which creates a connection *on behalf* of a user to a web-server. If the user wants to access resources via an HTTPS-secured connection, the secure channel is set up between the cache and web-server ((2) in Figure 5); optionally, also the connection between client and cache may be secured by HTTPS ((1) in Figure 5).

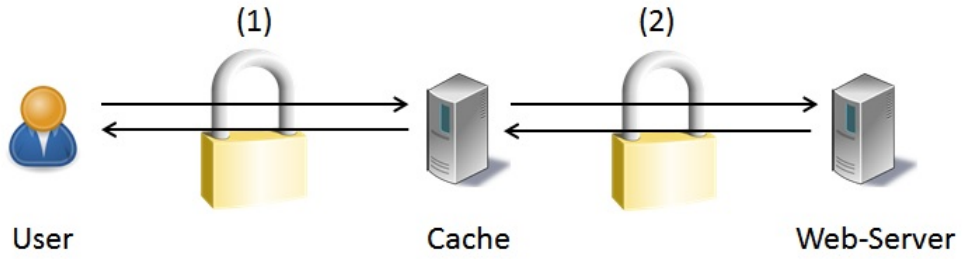

Figure 5: Man-in-the-Middle-enabled cache architecture (own figure)

3.1 The Squid Proxy Server

To accomplish the goals of this study, a squid proxy-cache [26] is deployed and configured to create a network architecture as illustrated in Figure 5.

3.1.1 Introduction to Squid

Squid is an open-source proxy-cache which emerged from the Harvest Cache Demon [14]. It is used by many companies to reduce traffic and improve web-object delivery times and browsing experience [21]. More specifically, among the several well-known organizations and projects which deploy squid [25] are: Wikipedia operator Wikimedia, the image and video hosting platform Flickr and the distributed computing framework FroNTier.

3.1.2 Configuration

The setup and configuration of the MitM-Cache is done as described in [17, 27]. Precompiled binaries of squid do not support the *SSLBump*-feature which enables squid to act as a Man-in-the-Middle. Hence the sources are compiled with the SSLBump-feature enabled (using squid version 3.2.14):

```
./configure --enable-ssl
make
make install
```

Squid's configuration file also needs adjustment to allow SSLBump. The following lines are added or altered[6]:

```
always_direct allow all
ssl_bump allow all
sslproxy_cert_error allow all
sslcrtd_program /usr/lib/squid3/ssl_crtd -s /var/lib/ssl_db -M 4MB
sslcrtd_children 5
http_access allow all
http_port 3128 ssl-bump generate-host-certificates=on
    dynamic_cert_mem_cache_size=4MB cert=/usr/local/squid/ca/cert.
    pem key=/usr/local/squid/ca/key.pem
```

The impact of these options, according to the documentation included in squid's configuration file and [24], is:

always_direct: In a larger cache architecture, where multiple caches, e.g. parent and sibling caches, are connected, the requests are sent *directly* to the destination server without consulting any siblings or parents. For this experiment, this option has no effect for only a single squid exists. Setting this option to *allow all* makes sense in large cache architectures if not all caches are MitM-enabled.

ssl_bump: This is squid's main feature to investigate encrypted traffic. It allows requests from all origins to be 'bumped'. Its mode of operation is equivalent to that of a MitM-Attacker (Section Man-in-the-Middle-Attack). Access Control List (ACL)s can be used to define which clients' requests will be bumped or not bumped, respectively. To do this, the Internet Protocol (IP)-address ranges or network addresses are assembled in one or more ACL(s):

[6] /usr/local/squid is squid's installation directory

```
acl ALLOWED_NETWORK src 192.168.0.3/24
acl DENIED_NETWORK src 192.168.0.4/24

ssl_bump deny DENIED_NETWORK
ssl_bump allow ALLOWED_NETWORK
```

It is also possible to use 'all' instead of an ACL to allow or deny all clients' requests to be bumped.

Another important function is to prevent bumping of requests sent to certain websites, e.g. banking and webmail:

```
acl NO_CACHING "/etc/squid3/no_caching.txt"
cache deny no_caching
```

In this example, the ACL is defined in a separate file outside of squid's configuration. The rules associated with 'allow' and 'deny' (e.g. http_access, ssl_bump, cache) are prioritized by the order in which they appear in the configuration file.

sslproxy_cert_error: Server certificates may be invalid for a variety of reasons; for example when they are revoked or issued by an unknown CA. Like a *whitelist*, this ACL defines exceptions for servers which are trusted but operate with an invalid certificate. If a server has an invalid certificate and is not part of these exceptions, squid terminates the connection.

sslcrtd_program: defines the path and command line options for the program that generates the certificates which squid sends to the user.

sslcrtd_children: The number of processes that serve the generated certificates.

http_access: This option allows or denies HTTP access from networks and computers to squid.

http_port: specifies the port number on which squid will listen for HTTP requests. Squid's standard port number is 3128.

In the last line, there are a few other options necessary for SSLBump:

generate_host_certificates If set to *on*, this option allows squid to dynamically create certificates for 'bumped' requests.

dynamic_cert_mem_cache_size Size of the cache for dynamically created certificates.

cert Path to the root certificate which squid will use to sign the dynamically created certificates.

key Path to the private key that belongs to the public key in the root certificate.

The root certificate for squid is created in three steps: Generating the key pair, creating a request for a certificate – in this step all identity information must be provided – and signing the certificate. For this experiment a self-signed root certificate is used, but more commonly a CA would sign the certificate. These three steps correspond to the following three commands:

```
openssl genrsa -out key.pem 2048
openssl req -new -key key.pem -out csr.pem
openssl req -x509 -days 365 -key key.pem -in csr.pem -out cert.pem
```

3.2 Evaluation of the Data

To efficiently analyze the performance of the MitM-enabled squid proxy-cache, a report based on squid's log file and the analyzer "calamaris" [4] is created. Processing and evaluating the information they provide is done using the performance indices specified in [20]:

Hitratio: Share of requests that were served by data held inside the cache.

$$\frac{requests\ served\ by\ cache}{total\ number\ of\ requests} \cdot 100\% \tag{1}$$

Effectiveness: Ratio between data volume served by data held inside the cache and overall requested data volume.

$$\frac{data\ volume\ served\ by\ cache}{overall\ requested\ data\ volume} \cdot 100\% \tag{2}$$

Efficiency: Compares the saved traffic to the amount of occupied storage within the cache.

$$\frac{volume\ of\ saved\ traffic}{occupied\ space\ within\ cache} \cdot 100\% \tag{3}$$

Cacheability: Share of the incoming data volume that was able to be cached. Note that the incoming data volume does not include data volume served by the cache to prevent the underlying objects from being counted twice.

$$\frac{size\ of\ all\ objects\ within\ the\ cache}{incoming\ data\ volume} \cdot 100\% \tag{4}$$

with

$$\begin{aligned} &\textit{incoming data volume} \\ &= \textit{overall requested data volume} - \textit{data volume served by cache} \end{aligned} \tag{5}$$

In order to gain a better understanding of a MitM-enabled cache's performance Hitratio and Effectiveness are calculated separately for encrypted requests and the respective data volume. Further, the percentage of encrypted objects served by the cache in comparison to the total data volume served by the cache is calculated:

Hitratio_Encrypted:

$$\frac{\textit{encrypted requests served by cache}}{\textit{total number of encrypted requests}} \cdot 100\% \tag{6}$$

Effectiveness_Encrypted:

$$\frac{\textit{encrypted data volume served by cache}}{\textit{overall requested encrypted data volume}} \cdot 100\% \tag{7}$$

Share in saved traffic:

$$\frac{\textit{encrypted data volume served by cache}}{\textit{overall data volume served by cache}} \cdot 100\% \tag{8}$$

3.3 Implications on User's Security and Privacy

Deploying a MitM-enabled cache degrades the level of security and privacy of all users connected to that cache. The problems faced by all users connected to such a cache and the group of users participating in this experiment in particular are discussed in this section.

3.3.1 General Security and Privacy Issues implied by Man-in-the-Middle Caches

The weak point from the security perspective of the architecture shown in Figure 5 is the cache. When data are transmitted using HTTPS, the underlying connection is expected to provide *end-to-end encryption*. However, on the MitM-enabled cache the data are no longer protected by either HTTPS connection's encryption. Additionally, the confidential data may be stored for a certain time period and the user has no control over

if and *when* the data will be made unavailable (deleted). Hence, the deployment of a MitM-enabled cache is a direct contradiction to the security goal *confidentiality*.

Since the cache contains many confidential data it is a highly attractive target for hackers and two basic, yet dangerous attacks are possible: If attackers gain (root) access to the cache they could install a backdoor to sniff all traffic flowing through the cache in real time or steal all data currently stored within the cache.

It is rather unlikely that CC numbers and other credentials like passwords can be stolen, because the cache only stores incoming but no outgoing traffic. Only the answers to HTTP/HTTPS requests are stored, but *never* the requests themselves.

Nonetheless this does not mean that these data are safe despite the deployment of a MitM-enabled cache. If a website displays those information directly to the user, they may be part of an HTML document that can be cached. Furthermore, e-mails transmitted via HTTPS will be saved inside the cache; this is usually the case for webmail environments. Login credentials, CC numbers and other confidential data that are typically only sent from the user to a web-server can be obtained by other means: *Phishing*.

This leads to another attack vector provided by MitM-enabled caches: Rather than reading data from the cache, an attacker might try to inject a phony website into the cache[7]. For example, a counterfeit online banking website which sends all data entered by the user to the attacker. Similar to DNS poisoning[8], the URL of a website would be stored together with the phony website inside the cache instead of the original website.

The attack proceeds as shown in Figure 6: After the attacker injects a phony website together with the URL of the original website into the cache (1) and the user makes a request for the original website (2), the cache delivers the phony website (3). It is hard for the user to detect this type of attack, because even if a genuine website is delivered to the user, the original certificates are replaced with the cache's certificate. This has to be the case because the user (client) never talks directly to the web-server; from the user's (client's) point of view the secure channel is built up between him and the cache.

For the same reason it is impossible for a user to detect if an ongoing Man-in-the-Middle-Attack takes place between the cache and the server. To prevent any third party from successfully launching a Man-in-the-Middle-Attack, the cache's certificate acceptance policy needs to be set up in a way that rejects HTTPS connections when the web-server's certificate is invalid for any reason.

[7] This possible attack is also briefly discussed in [20].

[8] An attacker messes with the victim's DNS-cache to alter the referenced IP address of a domain name.

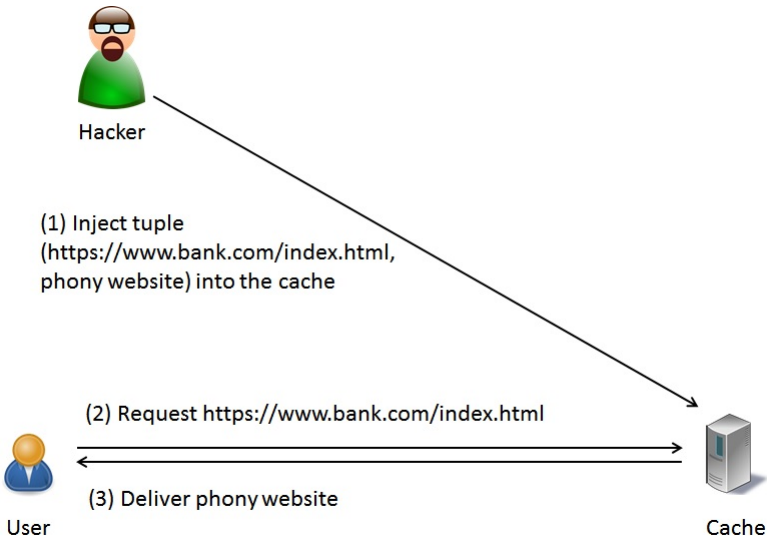

Figure 6: Cache-poisoning attack (own figure)

3.3.2 Experimental Data Acquisition

The group of users participating in this experiment faces an additional privacy issue. The cache's log file has to be examined in order to obtain the necessary information to calculate the performance indices described in section 3.2. In fact, it is not necessary for any human being to directly inspect the log file. It suffices to have a log analyzer create a report based on the log file as previously described. A demo report of the calamaris log analyzer can be found in [3].

Calamaris reports contain some information that may impair a participant's privacy:

1. Outgoing requests by destination
2. Request-destinations by 2nd-level domain
3. Request-destinations by Top-level domain (TLD)
4. Requested content-type and Requested extensions
5. Incoming UDP/TCP-requests by host
6. Performance in 1 hour steps

Under (1), (2) and (3) the destinations of incoming and outgoing requests are depicted along with the number of requests and data volume associated to them. If the number of (active) participants is small, these statistics impose the greatest threat to the participant's privacy, because they allow to determine who requested objects from which websites by linking IP addresses to the computers of the participants of the experiment. On the other

hand, the more the number of (active) participants increases, the harder it is to trace back these information.

Requested content types and file extensions in statistic (4) come with similar downsides , but these now apply to the type of content rather than the content's location.

The hosts are only incorporated in statistic (5). This part of the report only shows the amount of traffic caused per host (IP-address) and protocol, not the actual contents requested per host. It is important do determine how many of the participants pass a certain threshold, e.g. one percent of the total traffic. This information is important to determine the sample size of the experiment.

Information provided by statistic (6) is irrelevant to this experiment, but it would allow ISPs to determine peak hours and to take them into account in their plans. The threat imposed on each participant's privacy is the monitoring of start and end points of their activities. Since the experiment is performed at a University's laboratory, these times will unsurprisingly be during day time. If the experiment were to be carried out with participants using their internet connection at home, it would be easy to derive when a user wakes up, goes to bed, works and so on.

4 Results

The data of nine participants were collected for the experiment that lasted ten days. Four of them were *active*[9] participants. All participants were students working in a University laboratory. In this ten day period the cache handled a total of 152,678 HTTP and HTTPS requests resulting in a total traffic of 16,216 MiB. Out of the generated traffic 162 MiB (1%) were served by the cache. To achieve this level of *effectiveness* 1,661 MiB of the cache's storage were occupied. These general results of the experiment are illustrated in Figure 7.

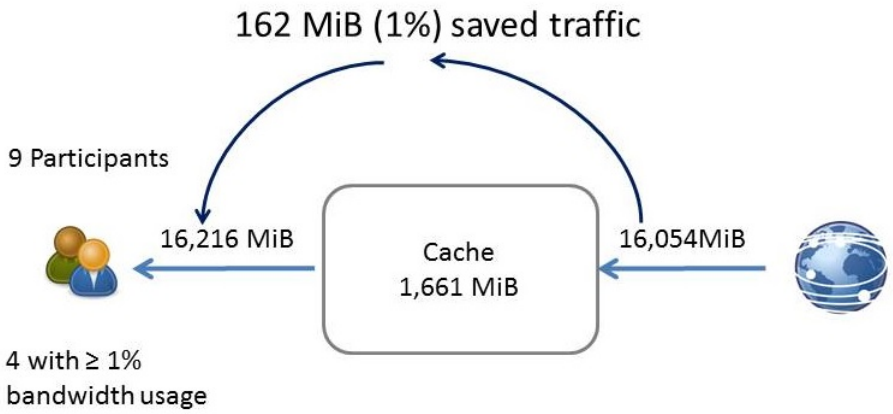

Figure 7: Overview of the experimental results (own figure)

All main performance indices of the MitM-enabled cache (plotted in Figure 8) are considerably worse than those of a non-MitM cache that was evaluated in a previous study[20]:

Hitratio and effectiveness
The hitratio measured in this experiment is 5%, compared to 20% in [20]. Effectiveness also decreased: Previously, 7% of the requested data volume could be served by the cache. Now, this figure is reduced to 1% despite the cache being able to examine encrypted traffic and hence, being able to cache *more* objects. This reduction can be explained with the composition of the traffic: While 39% of all requests in [20] were requests for image files, only 33% of the requests sent to the MitM-enabled cache were targeted at images. This is important because images both had an excellent hitratio (40%) and generated the second-highest amount of traffic after videos (16,19%) according to [20] and thus, were the main boost for both hitratio and effectiveness. However, the image hit ratio for this experiment was a tenth (4%) compared to that in [20], resulting in an overall decreased hitratio and effectiveness.

[9]Participants that generated at least 1% (162.16 MiB) of the total traffic.

	Previous Experiment	Current Experiment
Hitratio	20%	5%
Effectiveness	7%	1%
Efficiency	50%	10%
Cacheability	20%	10%

Table 1: Key performance indices observed in [20] and this study

Cacheability and efficiency

With decreasing effectiveness it is evident that the cache's *efficiency* – bandwidth relief vs. disk usage – decreases as well. With the previously mentioned 162 MiB of saved bandwidth and 1,661 MiB of disk usage the MitM-enabled cache's efficiency is nearly 10%. In [20] an efficiency ratio of 50% was established over the whole course of the experiment though. Cacheability – the percentage of incoming traffic stored in the cache – is 10% for the MitM-enabled cache vs. 20% for the non-MitM cache. This is a surprising outcome because a cache able to examine encrypted traffic is expected to cache a larger portion of the incoming traffic, rather than a smaller portion. Another interesting fact is the course of the cacheability over the experiment duration. It started at a rate of 100% on the first day and decreased to a stable 30% during the first seven days. This figure is higher than the 20% measured previously. In the last three days of the experiment, cacheability decreased to 10%, indicating that most of the traffic in this time period was not cacheable. The key performance indices of this experiment and the previous experiment are listed in Table 1.

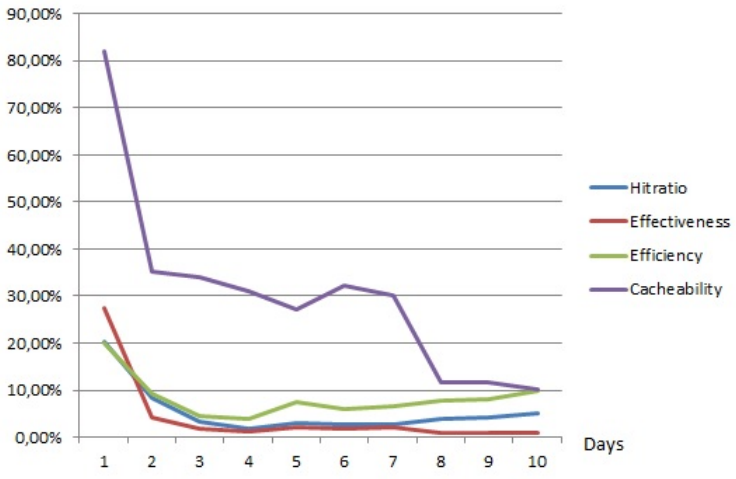

Figure 8: Main performance indices of the cache (own figure)

Hitratio and effectiveness of HTTPS-protected traffic are plotted separately in Figure 9 and the share of HTTPS-protected objects in the traffic served by the cache in Figure 10 to investigate the gains of caching encrypted web-objects. They both are relatively close

to the combined hitratio and effectiveness of HTTP and HTTPS traffic; hitratio is 3.87% – roughly one percent less than the combined hitratio – and effectiveness (1.22%) is even *slightly above* the combined effectiveness (1%). This means, despite lower overall results, caching of HTTPS traffic yields similar results to caching HTTP traffic.

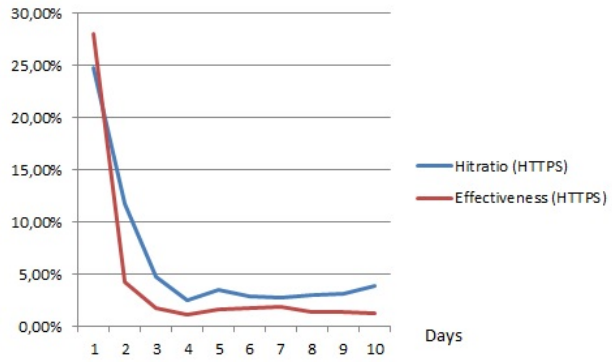

Figure 9: Characteristic numbers for HTTPS-secured traffic (own figure)

This conclusion is also supported by the share of HTTPS traffic in the saved bandwidth: At the end of the experiment, 44.4% of the data volume served by the cache were originally requested via HTTPS (refer to Figure 10).

Summing up the outcome of the experiment it can be said that, despite poor results compared to the non-MitM cache, allowing a cache to examine encrypted traffic can increase its effectiveness roughly by factor two, possibly even more (see Figure 10). An explanation for the decrease in hitratio, effectiveness and efficiency is the reduced number of active participants and the apparently changed composition of the traffic. The latter is indicated by smaller cacheability, less requests for images and more importantly, considerably less cache hits for images.

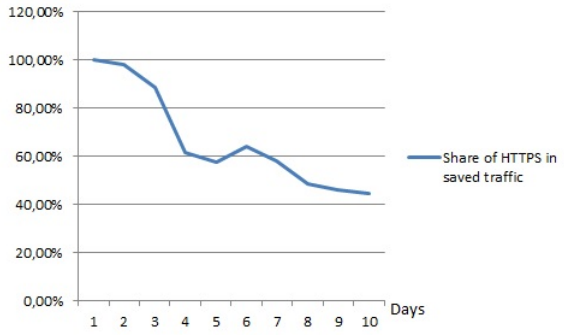

Figure 10: Percentage of HTTPS-secured objects in the saved bandwidth (own figure)

Another change in the composition of the traffic is the percentage of encryption: In [20] 45% of the traffic were encrypted. However, in this experiment the amount of encrypted

traffic has reduced to 36,4%. The Canadian network company Sandvine observes that two thirds of the internet traffic are encrypted [19].

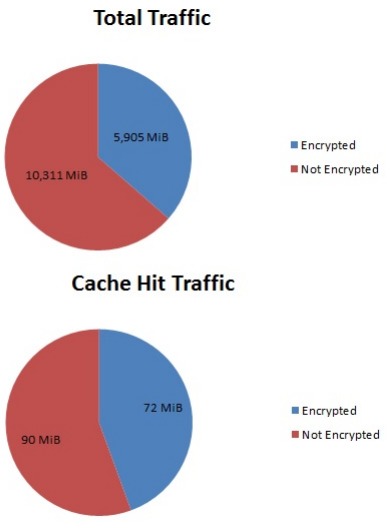

Figure 11: Percentage of HTTPS-secured objects in the total traffic (top) and in the saved bandwidth (bottom) (own figure)

It is important to note that the portion of encrypted objects in the saved bandwidth (traffic served by cache) is higher than in the total traffic, (Figure 11) as it underlines the potential of a MitM-enabled cache and the good reusability of encrypted objects.

5 Conclusions and Future Work

The motivation for this work is the increase of encrypted internet traffic [19] and the general contradiction between encryption and cacheability [20]. Among other measures, for example compression, caching allows the reduction of traffic and hence, bandwidth usage. Reducing the load on a network link may not be a relevant issue in all networks, but when a backhaul link is expensive to use or provides only limited bandwidth like cellular networks and satellite links it helps to improve performance and cost control.

In previous studies negative effects of encryption were considered. Mraz [16] proposes a mechanism to relieve a web-server's computational resources. Encryption and Secure Sockets Layer (SSL) (or TLS respectively) handshakes are performed by separate entities, allowing the web-server to work with plain text at all times. Other studies [6, 9, 15] focus on reducing bandwidth usage in TLS protected sessions. However, the gains are limited to reduction of bandwidth usage in the setup phase of the TLS protected connection (the TLS handshake protocol) by compressing certificates or replacing them with identifiers. These approaches do not consider any reduction of bandwidth usage once a connection is successfully set up.

This study aims to fill this gap by investigating the potential of a Man-in-the-Middle-enabled cache – a cache that can decrypt traffic of TLS or HTTPS secured connections. To achieve this goal, a squid proxy-cache is set up in a way that lets it act as a Man-in-the-Middle. It creates two connections: One connection (on behalf of the client) to the web-server, another one to the client himself. Unlike a conventional cache it does not simply forward the encrypted traffic between client and server but actively decrypts and encrypts everything. While this approach allows the cache to store objects for later use and hence, reduce bandwidth usage, it also imposes privacy issues on the clients. A real attacker could try to install a backdoor on the proxy-cache and sniff all traffic in real time, steal the stored objects or even insert phony websites as a new form of cache poisoning and phishing. Users participating in the experiment face an additional privacy issue: As the log files need to be examined to evaluate the MitM-enabled cache, requested content types like images, videos or text and the respective request destinations (websites) may be linked to the clients who requested them. This was prevented by using a log analyzer.

Compared to the previous study [20] where a conventional cache is considered, the overall results for hitratio, effectiveness, efficiency and cacheability decreased considerably. Hitratio went down from 20% to 5%, effectiveness was only 1% compared to 7% in the previous study. Hence, efficiency also decreased and is now 10%, in [20] it is 50%. There are two hypotheses to explain the decline of the results: Firstly, there were only

four active participants in this experiment, nine people actively participated in the previous experiment, which means the potential to reuse cached objects decreased. Additionally the type of traffic differs from the traffic in the previous experiment. Indications for this are the reduced cacheability (10% vs. 20%), reduced amount of encrypted traffic (36% vs. 45%) and a considerably lower hitratio for images. It is also apparent that with a decrease in cacheability, there are less objects inside the cache that can be served when handling future requests. However, for the first seven days of the experiment cacheability was at 30% or above.

On the other hand, the outcome of the experiment shows that despite the results described above a MitM-enabled cache can further reduce bandwidth usage compared to a conventional non-MitM cache. Even though only 1% of the traffic was saved (served by objects from the cache), 44% of the saved bandwidth comes from objects requested via HTTPS – during the experiment this portion was even 60% and higher. When compared to the 36% of encryption when in comes to the total traffic, this shows that objects transmitted via HTTPS have a quite good reusability. Also, the hitratio and effectiveness are calculated separately for HTTPS traffic. Hitratio is nearly the same for HTTPS as for HTTP and HTTPS combined (1% difference). Effectiveness was even slightly higher than for combined traffic (1.22% vs. 1.0%).

Even though the overall results decreased because of different composition of the underlying traffic, this study was able to show that caching of encrypted traffic can improve a cache's performance. Since the size of the sample is relatively small as only four users actively participated in the experiment, it will be necessary to investigate the performance of a MitM-enabled cache in larger networks and over a longer time period in order to determine the full potential of such a cache.

References

[1] A. Abboud, E. Bastug, M. Debbah, and K. Hamidouche. Distributed Caching in 5G Networks: An Alternating Direction Method of Multipliers Approach. In *2015 IEEE 16th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 171–175, June 2015.

[2] J. Andress. *The Basics of Information Security*, chapter 2, pages 17 – 31. Syngress, 2011.

[3] C. Beermann. Calamaris demo report. http://cord.de/files/calamaris/calamaris-2.html. Online as of Mar 14, 2016.

[4] C. Beermann. Calamaris (english). http://cord.de/calamaris-english. Online as of Mar 14, 2016.

[5] V. Bindschaedler, J. Freudiger, J.-P. Hubaux, and N. Vratonjic. *Economics of Information Security and Privacy III*, chapter The Inconvenient Truth About Web Certificates, pages 79 – 117. Springer, 2013.

[6] D. Boneh, E. Rescorla, and H. Shacham. Client Side Caching for TLS. *AACM Transactions on Information and System Security (TISSEC)*, 7(4):553–575, November 2004.

[7] J. A. Buchmann, E. Karatsiolis, and A. Wiesmaier. *Introduction to Public Key Infrastructures*. Springer, 2013.

[8] M. Chen, A. Ksentini, V. C. M. Leung, T. Taleb, and X. Wang. Cache in the Air: Exploiting Content Caching and Delivery Techniques for 5G Systems. *IEEE Communications Magazine*, 52(2):131–139, February 2014.

[9] J. A. Cooley, R. I. Khazan, and S. McVeety. Secure Channel Establishment in Disadvantaged Networks. In *The 2010 Military Communications Conference (MILCOM) - Unclassified Program - Cyber Security and Network Management*, pages 32–37, October 2010.

[10] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. https://tools.ietf.org/html/rfc5246, August 2008. Online as of Feb 19, 2016.

[11] L. M. Drabeck, M. Haner, T. E. Klein, N. Nithi, B. A. Ramanan, and C. Sawkar. Cacheability analysis of HTTP traffic in an operational LTE network. In *Wireless Telecommunications Symposium (WTS), 2013*, pages 1–8, April 2013.

[12] D. Gollmann. *Computer Security*. Wiley, 3rd edition, 2011.

[13] Information Sciences Institute, University of Southern California (Ed.). TRANSMISSION CONTROL PROTOCOL. https://tools.ietf.org/html/rfc793, September 1981. Online as of Mar 5, 2016.

[14] G. Mitchell. The Harvest Cache - A Heirarchical (sic!) Internet Object Cache suited to HTTP. http://grahammitchell.com/harvest/. Online as of Mar 14, 2016.

[15] B. Monica and D. Raluca. TLS Protocol: Improvement Using Proxies. In *2012 10th International Symposium on Electronics and Telecommunications (ISETC)*, pages 151–154, November 2012.

[16] R. Mraz. Secure Blue: An Architecture for a Scalable, Reliable High Volume SSL Internet Server. In *17th Annual Computer Security Applications Conference (ACSAC)*, pages 391–398, December 2001.

[17] mydlp.com (Ed.). How to configure Squid 3.2 for SSL Bumping and Dynamic SSL Certificate Generation. https://www.mydlp.com/how-to-configure-squid-3-2-ssl-bumping-dynamic-ssl-certificate-generation/. Online as of Apr 4, 2016.

[18] G. N. Nayak and S. G. Samaddar. Different Flavours of Man-In-The-Middle Attack, Consequences and Feasible Solutions. In *2010 3rd IEEE International Conference on Computer Science and Information Technology(ICCSIT)*, pages 491 – 495, July 2010.

[19] Sandvine, Inc. (Ed.). 2016 Global Internet Phenomena SPOTLIGHT: ENCRYPTED INTERNET TRAFFIC. https://www.sandvine.com/trends/encryption.html, February 2016. Online as of Feb 11, 2016.

[20] A. Slopek and T. Thaeren. *Bandbreitenreduktion durch (lokale) Caches*. Term Paper, Hochschule Bonn-Rhein-Sieg, Grantham-Allee 20 53757 Sankt Augustin, January 2016.

[21] squid-cache.org (Ed.). About Squid. http://www.squid-cache.org/Intro/. Online as of Mar 14, 2016.

[22] squid-cache.org (Ed.). Feature: Squid-in-the-middle SSL Bump. http://wiki.squid-cache.org/Features/SslBump. Online as of Feb 15, 2016.

[23] squid-cache.org (Ed.). Feature: SSL Peek and Splice. http://wiki.squid-cache.org/Features/SslPeekAndSplice. Online as of Feb 15, 2016.

[24] squid-cache.org (Ed.). Squid configuration directives. http://www.squid-cache.org/Doc/config/. Online as of Apr 4, 2016.

[25] squid-cache.org (Ed.). Squid Deployment Case Studies. http://www.squid-cache.org/Library/. Online as of Mar 14, 2016.

[26] squid-cache.org (Ed.). squid: Optimising Web Delivery. http://www.squid-cache.org/. Online as of Feb 15, 2016.

[27] D. Vassallo. SQUID transparent SSL interception. http://blog.davidvassallo.me/2011/03/22/squid-transparent-ssl-interception/. Online as of Apr 4, 2016.

[28] T. A. Welch. A Technique for High-Performance Data Compression. *Computer*, 17(6):8–19, June 1984.