UC Santa Cruz
UC Santa Cruz Previously Published Works

Title
GroupSec: A New Security Model For The Web

Permalink
https://escholarship.org/uc/item/1dt2k843

Authors
Sevilla, Spencer
Garcia-Luna-Aceves, J.J.
Sadjadpour, Hamid

Publication Date
2017-05-21

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

GroupSec: A New Security Model For The Web

Spencer Sevilla, J.J. Garcia-Luna-Aceves, Hamid Sadjadpour
{spencer, jj, hamid}@soe.ucsc.edu
UC Santa Cruz, Santa Cruz, CA

***Abstract*—The de facto approach to Web security today is HTTPS. While HTTPS ensures complete security for clients and servers, it also interferes with transparent content-caching at middleboxes. To address this problem and support both security and caching, we propose a new approach to Web security and privacy called GroupSec. The key innovation of GroupSec is that it replaces the traditional session-based security model with a new model based on content group membership. We introduce the GroupSec security model and show how HTTP can be easily adapted to support GroupSec without requiring changes to browsers, servers, or middleboxes. Finally, we present results of a threat analysis and performance experiments which show that GroupSec achieves notable performance benefits at the client and server while remaining as secure as HTTPS.**

I. Introduction

Today, Web publishers wishing to securely deliver their content must do so using HTTPS. HTTPS refers to HTTP layered on top of Transport Layer Security (TLS); TLS is a session-oriented protocol that ensures data confidentiality, integrity, and authentication by encrypting all communication between the client and the server. HTTPS is used today for slightly over half of all traffic flows [1] and adoption is rapidly increasing, led by movements such as the Electronic Frontier Foundation (EFF)'s call for "HTTPS Everywhere" [2].

Unfortunately, the strong security guarantees of TLS *also* inhibit middleboxes from acting on HTTPS flows - this is because from the perspective of TLS, any middlebox operation is a man-in-the-middle attack! This results in significant consequences for end-users, publishers, and network operators alike, including increased latency, bandwidth consumption, CPU usage, and significant financial impact: Netflix alone estimates that switching from HTTP to HTTPS will cost upwards of $100M *per year* [3]!

Though middleboxes provide a wide range of services, [1] identifies that the primary cost of TLS stems from preventing transparent content caching. Transparent caching is a technique used to reduce network bandwidth consumption, and is incredibly effective because it exists at a powerful intersection of motives and ability: First, transparent caches can be deployed by ISPs and network operators without requiring coordination with clients or publishers. Second, these operators are the same entities with a true "birds eye" view as to where network congestion occurs, and therefore best equipped to combat congestion effectively. Finally, market forces mean that these operators are *also* the most incentivized to combat congestion in their networks.

To address this challenge, we introduce a radically new approach to Web security and privacy that we call **GroupSec**. GroupSec introduces a new, fine-grained security model based on *content group membership*; this model also separates the security requirements of publishers from those of clients. Subsequently, GroupSec combines this model with a novel mechanism for "offloading" Web content from an existing HTTPS session to plaintext HTTP, thereby enabling transparent content caching to occur without compromising a client or publisher's security interests.

Section II surveys related work and discusses the shortcomings of prior approaches. Section III explains the new security model of GroupSec in detail, including the assumptions we make regarding the motives and interests of clients and publishers. Section IV provides specific implementation details of GroupSec, which consists of minimal modifications to HTML and no modifications to HTTP, TLS, or middleboxes. Section V provides a comprehensive threat analysis of the GroupSec security model, Section VI provides the results of a preliminary performance evaluation, and Section VII concludes the paper.

II. Related Work

The challenge of integrating HTTPS with middleboxes has attracted a substantial amount of attention in the research community. Recent works [4], [5] propose altering TLS itself to support specific middlebox operations on traffic flows. However, these works appear to have focused exclusively on adding support for *qualitative* middlebox features such as intrusion-detection or content-filtering, and do not support transparent content caching. This limitation stems from the fact that such works enable middleboxes to alter existing TLS sessions, whereas the goal of transparent caches is to prevent an end-to-end flow from ever being established in the first place.

Meanwhile, policy-based solutions [6], [7], [8] propose leaving TLS unchanged and "splitting" a TLS session into two separate connections: one from the client to the middlebox, and another from the middlebox to the server. However, since these proposals require the caching entities to be trusted either by root CAs and/or browsers, they completely break end-to-end authentication, and have thus met widespread resistance by the Web community [9], [10], [11].

There exists a set of "secure content delegation" proposals [12], [13], [14] that enable publishers to host content at a (presumably untrusted) CDN by providing clients with keys needed to verify and/or decrypt the content over a separate (out-of-band) HTTPS session. However, these approaches incur significant overhead (both in bandwidth and latency) by

relying on redirection to obtain the location of the resource at the CDN. More importantly, this approach only supports explicitly configured CDNs and proxies, not transparent or opportunistic caches.

The most crucial limitation of all these works is that they treat middleboxes as trustworthy entities that can and should be authorized to read and write content. Remarkably absent from prior work are discussions regarding new approaches to security, new models that reexamine or redefine the security requirements of Web users and publishers, or models that support transparent caching *without* trusting middleboxes.

III. Content Group Security

The key innovation of GroupSec is a new security model based on *group membership*. In this model, clients are defined as being in a "content group" together if they are authorized by the publisher to view the same content object. Content groups exist separately for each content object, and may overlap. Figure 1 provides a simple example, where users a and b are in two content groups together (groups $f1$ and $f2$), and user b is also in a third content group ($f3$) with user c.

Compared to TLS, GroupSec relaxes the security restraints on content groups in two key ways. First, nodes within the same content group are allowed to infer each other's membership with respect to that particular group (i.e. if a node is authorized to view a content object, it can also deduce when other clients are viewing the same content object). Second, nodes *outside* the group may see that clients are in a unique group together, but cannot deduce anything about the nature of the group. Continuing the example in Figure 1, user a can see that user b is able to access files $f1$ and $f2$. Likewise, user c can see that users a and b are in two distinct content groups together, but cannot access either file or its filename.

While content groups represent a new conceptual model, they do *not* require extra storage or processing power. Publishers simply respond to client requests for Web objects, identically to the current model, and the corresponding privacy rules and relaxations (described in Section IV) serve to create the model of the client group.

A. Asymmetric Privacy Models

The group membership security model is carefully designed to meet the concerns of publishers distributing a file to multiple clients. Specifically, GroupSec is designed around the observation that in this scenario, publishers and clients have asymmetric privacy concerns! From the perspective of content publishers, privacy refers to the nature of the file itself (i.e. its

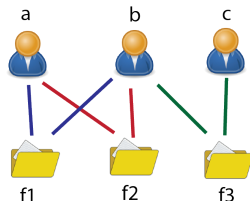

Fig. 1. Content Group Membership

name and contents) that the publisher is serving. Conversely, from the perspective of clients, privacy refers to the nature of the file, but *also* the fact that the client requested that specific file.

Subdividing privacy in this manner is a crucial part of the GroupSec design, because it enables minimal leaking of information while still supporting transparent caching: for a transparent cache to operate, it only needs to know when multiple clients request the same content object. The core goal of the GroupSec privacy model is to expose this information, and *only* this information, while still protecting the name and content of the cached files *even from the caches themselves!*

1) Publisher Privacy: From the perspective of content publishers, GroupSec achieves a level of privacy equivalent to HTTPS: the only clients that can decrypt and view a specific content object are those that are specifically authorized by the content publisher. All other nodes are completely unable to discern any information about the content object, including its name.

2) Client Privacy: In contrast to the publishers, GroupSec *clients* see a significant privacy downgrade when compared to HTTPS. Whereas HTTPS offers clients assurance that no one but the publisher knows anything about their request, GroupSec relaxes this restriction in two ways: (1) other nodes in a client's content group can see that the client has requested the file, and (2) nodes outside the content group can identify the members of a unique content group by IP address.

This relaxation raises significant client-privacy concerns. Because of these concerns, GroupSec represents itself to clients as a completely *non-private* connection. That is to say, end clients using GroupSec can be assured of the *integrity* of the content, yet are given no assurances at all about the *confidentiality* of their request.

Defining a connection as *non-private* at a single side of the connection is a significant departure from all prior security models. However, such a definition is not only a good fit for the Web, it is remarkably easy to convey. Whereas publishers manage the security of their website via the technologies they implement, clients must look for the "lock" icon in their browser. This asymmetry, combined with the observation that Web users most often use "secure" to mean "private" [15], [16], means that GroupSec can achieve such asymmetry by simply identifying GroupSec content as insecure with this icon.

B. HTTP-Centric Security

Instead of layering plaintext HTTP traffic on top of a TLS session, GroupSec enacts security within HTTP itself, by encrypting the filename and contents separately though an out-of-band process similar to [17]. Once encrypted, the HTTP requests and responses are layered directly over plaintext TCP. Figure 2 illustrates this process and shows which specific fields of the HTTP request and response are encrypted.

The decision to shift security from TLS into HTTP is an absolutely crucial part of our design. By encrypting the Request-URL and Message Body fields separately, GroupSec

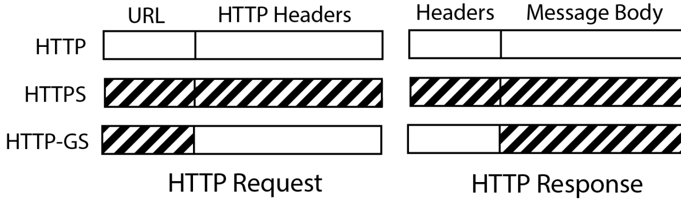

Fig. 2. Encrypted Requests and Responses

allows intermediate nodes to (1) identify HTTP traffic and (2) view and modify HTTP header fields. While this can be seen as "relaxing" the TLS security model, this shift comes with minimal drawbacks and many key benefits. First, even when TLS is used, an attacker can infer that two nodes are exchanging HTTP traffic simply by observing the port numbers and communication pattern. Second, we note that HTTP Request header modification is a powerful bandwidth-saving technique employed by transparent caches to encourage content reencoding for mobile devices [1].

C. Middleboxes and Trust

A vital drawback of prior work [4], [5] is that they explicitly authorize certain middleboxes to view and/or modify content. This design is fundamentally at odds with a common observation that middleboxes are not trustworthy, and in some cases (e.g. ISPs injecting additional advertisements) should be considered malicious. Thus, rather than opening debate or proposing mechanisms to separate "good" middleboxes from "bad" ones, GroupSec assumes a much simpler trust model wherein clients trust content publishers and no one else.

IV. HTTP-GroupSec

For the GroupSec privacy model to be feasibly deployable in the Internet today, it must be implementable with minimal changes to browsers and servers, and must *not* depend on alterations to middleboxes themselves. Under these constraints, we found that four key requirements dictated our protocol design. Formally, a GroupSec content object must (1) be decryptable by the intended clients, (2) not be decryptable by other nodes, (3) be cacheable by intermediate entities, and (4) fully mask the name of the object.

A. Content Object Encryption

Our solution, which we call HTTP-GroupSec (HTTP-GS), starts with the assumption that a HTTPS session exists between a client and publisher, and that this session was used to load a preexisting page, which we call the *linking page*. Through the inclusion of two new HTML attributes, `http-gs-key` and `http-gs-salt`, the linking page indicates to the browser that certain elements (either embedded or linked) are HTTP-GS powered. The browser then retrieves these specific elements over HTTP (*not* HTTPS), decrypts them, and renders them in the page accordingly.

Before they are linked or embedded in a page, each HTTP-GS object is encrypted with its own public-private keypair. In keeping with recommended best practices [18], [19], [20], [21], [22], [23], [24] we use a 2048-bit RSA key, but stress that HTTP-GS can support any form of asymmetric encryption. Each object is encrypted with its own key to support fine-grained object-level security and enable different groups to emerge for each individual object. The public key is transmitted in the linking page under the `http-gs-key` attribute; this enables clients to be assured of both the integrity and confidentiality of the key itself.

B. Request URL Hashing

HTTP URLs are comprised of two parts: the Hostname (e.g. `www.example.com`) and the Path (e.g. `/videos/v1.mpg`). Both fields are transmitted in a HTTP request, and therefore must be disguised in order to ensure client privacy. We use two different techniques to encrypt each component separately.

The Path requested by the client is generated by hashing the URL provided by the linking page; we call this value the *name-hash*. In generating this hash, we have two goals: First, nodes that are not members of the content group must not be able to reverse the name-hash to the original URL. Second, the name-hash must be *consistent* so that multiple clients requesting the same content object generate the same name-hash.

The name-hash is generated by including a second attribute, `http-gs-salt`, in the HTML of the linking page. This attribute has two values, the salt itself (a randomly-generated number provided by the publisher) and an expiration date.[1] The browser verifies that the salt is within the expiration date, and then creates the name-hash by adding the salt, the key, and the URL together, and then calculating an md5 hash of this value as in Equation 1. After creating this hash, the client then issues a HTTP GET request for it, as illustrated in Figure 3.

$$name_hash = md5(url + key + salt) \quad (1)$$

Hashing the URL in this manner ensures request consistency across all clients, since every client receives the same URL, key, and salt from the publisher. Meanwhile, the salt ensures forward-secrecy by effectively placing an expiration date on the name-hash. If an eavesdropper possesses the content object's key, either by compromising a previous HTTPS session or having been previously authorized to receive the content, it still cannot create the name-hash without the current salt. This enables sufficient client privacy, performance optimization, DRM, and key revocation.

C. Hostname Stripping

HTTP-GS requests replace the Hostname field with the host's IP address; this decision comes from several motives. First, the hostname cannot be included in plaintext, since it leaks valuable information about the content the client is requesting. However, hashing the hostname separately for each web object, the way the name-hash is generated, risks a cross-domain hash collision at the caches (e.g.

[1] Choosing a good salt refresh-rate is left to the discretion of the publisher, since it dictates a tradeoff between client privacy and cache efficiency; this value should be closely coordinated with the `Cache-Control` header.

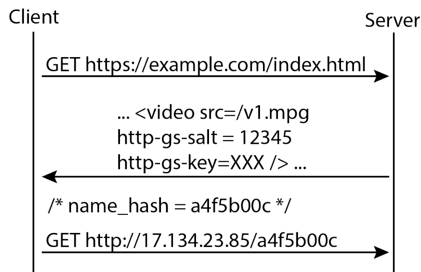

Fig. 3. HTTP-GS URL Hashing

`domain1.com/fileX` and `domain2.com/fileY` both hash to the same value). Such a collision poses a serious problem, because it disrupts client access to the content (i.e. a client requests one file and the cache provides the other), yet this behavior would be completely undetectable by the content publishers themselves.

To prevent these cache collisions, each domain must have a unique namespace to generate their hashes under; this makes hash-collisions trivial to identify and correct. However, this design poses an alternate security leak: if a hash used to mask a domain name is consistent for every object named under the domain, an attacker can discover the Hostname hash value simply by visiting any public-facing page under the domain.

By replacing a publisher's hostname with it's IP address, we resolve both problems at once. The consistent value of the IP address removes the threat of hash collisions at caches, yet does not expose any information that was not already visible in the IP header. Furthemrore, in the case of colocation facilities that host multiple domain names under a single physical IP address, these collisions are just as trivial to identify and correct, and actually serve to further obfuscate the name of the content object requested: a request of the form `ip:content_hash` leaks no information at all if `ip` is the address of a server known to host several different websites.

D. Transparent Caching

Transmitting requests and responses over plaintext HTTP enables Web caches to consistently identify, store, and serve HTTP-GS content in response to future requests. Additionally, leaving the HTTP headers unprotected allows these same caches to (1) read and act on relevant cache-specific HTTP headers such as `Cache-Control` and (2) add specific headers to outgoing HTTP requests, such as requesting a mobile-specific version of the object if one exists.

E. Cross-Domain Linking

One of the key benefits of public-private keypairs, as opposed to symmetric keys, is to support HTTP-GS linking and re-hosting across domains. Specifically, because the `http-gs-key` attribute is a public key, a page loaded over HTTPS can securely embed or link to elements outside of its domain. This enables integrated support for personalized content or aggregator sites (e.g. Reddit or Google News) and highlights the strength of HTTP-GS. While the initial personalized or aggregated site must be loaded over HTTPS, every subsequent linked or embedded object can be loaded over cacheable HTTP-GS! Given the rising trend of such aggregators, we anticipate this specific use model to account for a large portion of the network benefits of HTTP-GS.

For cross-domain linking to work, the `http-gs-salt` attribute must either be (1) set to a sufficiently large value or (2) updated by the publisher every time it changes. However, a simple update service (e.g. RSS) could support this feature automatically for established relationships across known websites. More importantly, similar to key revocation for clients, this design puts an "expiration date" on cross-linking websites, since a publisher can revoke access by simply denying a linker's request for the current salt.

V. Threat Model Analysis

We examined GroupSec and compared it to HTTPS with respect to a range of common attack vectors. We primarily consider two attack models: an on-path attacker that is *not* part of a client's content group, and an on-path attacker that is part of the client's content group (i.e. the attacker possesses the current salt and key). We explicitly do not consider attack vectors that lie orthogonally or out-of-band with respect to HTTP-GS and HTTPS (e.g. an attacker gaining physical access to a server or breaking public-key cryptography) since such vectors are out of the scope of our approach.

A. Unauthorized File Access

The primary privacy concern of *publishers* is that unauthorized clients will access their content. However, since the content is encrypted with the publisher's private key prior to distribution, and the public key is only distributed over HTTPS, an attacker without the public key will be unable to decrypt HTTP-GS content. To claim otherwise is to say that either (1) the attacker was able to obtain the key by hijacking an HTTPS session or (2) the attacker was able to break public key encryption.

An attacker that was previously able to access the content (i.e. its access was revoked) will already have the content name and key, but *not* the current salt. Even with both the name and key, the attacker will still not be able to create the correct name-hash to request the content object. Additional techniques may be necessary to further obfuscate the name hash and protect it from brute-force attacks, but we leave such techniques (and further analysis of this attack) to a future work.

B. Client Requests

The primary concern of *clients* is that an attacker will learn that the client requested a specific file. In these cases, the attacker's ability to do so hinges on membership in the content group.

In the cases where the attacker is *not* a member of the content group, we can assume that the attacker is not in possession of the filename, key, or salt. It follows that without at least two of these values, the process of reversing a name hash to a {filename, salt, key} tuple is fundamentally

impossible, even if the hash function used is broken! This is because even if the attacker can break the hash function, the attacker will simply obtain the sum of these three variables, without any information as to which value is which.

In the case where the attacker *is* a member of the content group, the process of identifying the file requested by the client is trivial: by creating the current name hashes for each file it can access, an on-path attacker can immediately detect whenever such a file is requested by a client. However, this case is *explicitly* allowed by the security model, and therefore not a violation.

More abstractly, GroupSec addresses this threat by portraying GroupSec content to clients as non-private, and relying on prior works [25], [26], [27] which have found that Web users alter their behavior based on privacy indicators. We anticipate that publishers will use GroupSec primarily to "upgrade" the security of relatively non-private content (e.g. movies, news articles, etc.) in a way that protects the publisher's interests (i.e. DRM and client authentication). We stress that HTTP-GS is a poor fit and not intended for private or sensitive communication (e.g. email), in that it does not guarantee the same degree of client privacy as TLS.

C. Content Spoofing

In a content spoofing attack, on-path attackers respond to intercepted content requests with a fake piece of content. For such an attack to work on HTTP-GS content, the fake content must have been encrypted with the correct key, or else client-side deencryption will fail. Since HTTP-GS uses asymmetric keys, and the publisher's *private* key is never even transmitted over the network, this attack cannot succeed unless the attacker either breaks public-key encryption or obtains the publisher's private key through some form of offline attack.

D. Cache Poisoning

A cache poisoning attack is similar to a content spoofing attack, except that the goal of the attacker is simply to disrupt client access to content by populating a cache with incorrect or false objects. Even if clients detect that the content is spoofed, cache poisoning can still occur because if a cache stores this incorrect object, it will respond to all subsequent requests with the same incorrect content.

HTTP-GS protects against cache poisoning attacks that seek to disrupt access to a specific content object by ensuring that attackers cannot generate the name-hash for a specific content object; without access to the name-hash, attackers cannot pick out the specific piece of content to attack. Wide-range cache poisoning attacks (wherein an on-path attacker replies to every HTTP request with a fake content object) are still possible, yet unlikely: for far less resources, attackers can achieve the same result by silently dropping all HTTP request packets (i.e. executing a traditional DoS attack).

VI. Performance Evaluation

We implemented a GroupSec prototype in client-side Javascript, designed a basic webpage that uses this code to serve ten pictures, and deployed it on a simple testbed, consisting of two laptop computers connected over ethernet. We then used this testbed to compare GroupSec to HTTP and HTTPS in two key metrics: *sustainable load* at the content server and *page load latency* at the client. We chose these two metrics and evaluation because (1) load and latency are the two metrics most important to Web publishers and (2) they compare GroupSec at its absolute worst (i.e. no transparent caching).

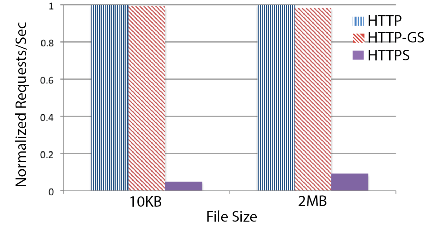

Fig. 4. Server Load

A. Sustainable Load

Though the key benefit of GroupSec is to enable transparent content caching, GroupSec must not incur additional load when caches are not on the network path. We recorded the sustainable load at the server, measured in requests-per-second, by running a standard Apache Benchmark test from the client to the server; this test repeatedly requests the same URL over HTTP, HTTPS, or HTTP-GS. Figure 4 contains our results for two filesizes: the Apache2 default page ($\sim$10 KB), and a larger file ($\sim$2 MB) chosen to reflect the current average Web object size [28]. To provide a platform- and filesize-independent comparison metric, we normalized the results by the baseline values collected over regular HTTP (8084 RPS for the 10K file and 1161 RPS for the 2M file).

These results show that the load of serving a HTTP-GS object is roughly equivalent to serving a regular HTTP object; this is unsurprising, given that HTTP-GS objects are transmitted over regular HTTP. More importantly, Figure 4 *also* shows that HTTP (and HTTP-GS) both vastly outperform HTTPS, by 20x and 10x, respectively! We therefore conclude that HTTP-GS can dramatically increase the sustainable load on a publisher's server even when transparent caching is not employed.

B. Latency

Client-perceived latency is generally considered the most important metric for content publishers [29]. We explored the effects of HTTP-GS on page load latency by creating a webpage with ten separate images, hosting it on the server, and migrating the images one-by-one from HTTPS to HTTP-GS. Figure 5 compares the GroupSec total latency (i.e. loading the initial page over HTTPS, fetching the elements over GroupSec, and decrypting the content out-of-band) to the "flat" cases where the same page was loaded entirely over HTTPS or HTTP.

Unsurprisingly, the downward trend of HTTP-GS content is explained by the fact that the initial HTML (and therefore

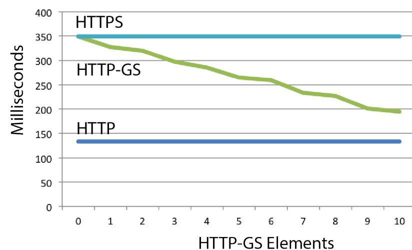

Fig. 5. Page Load Time

all non-migrated elements) is served over HTTPS; as a result, migrating an element to HTTP-GS decreases the latency at the client. Notably, after all elements on a page are migrated from HTTPS to HTTP-GS, the end page load latency closely resembles the latency of loading the entire page over HTTP! As above, this shows that migrating content from HTTPS to HTTP-GS results in substantial performance benefits even when transparent caching is not accounted for.

VII. Conclusion

In this paper we introduced a new security model for Web content delivery, *GroupSec*. GroupSec redefines the Web security model from session-based to group-based, and is the first security model to separate the privacy needs of *clients* from those of *publishers*. We provided strong arguments for why the GroupSec model better fits Web content delivery today and enables transparent caching while still meeting the privacy needs of both clients and publishers. We explained how HTTPS content can be easily adapted to GroupSec with minimal protocol changes, and showed that GroupSec is resilience against a wide range of attack-vectors. Finally, preliminary performance evaluations show that GroupSec outperforms HTTPS and approaches HTTP performance across multiple metrics, even in cases where transparent caching is not employed.

GroupSec has the potential to redefine Web content delivery in a wide range of cases. More importantly, it represents a fundamental shift in how Web security is perceived. This shift invites future work and debate on a wide range of topics, including additional analysis on GroupSec-specific threats, GroupSec-related performance optimizations and integration within HTTP(S), and additional security models inspired by the GroupSec approach.

References

[1] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste. The cost of the S in HTTPS. *Proc. ACM CoNEXT*, 2014.

[2] HTTPS everywhere. https://www.eff.org/https-everywhere.

[3] It wasn't easy, but Netflix will soon use HTTPS to secure video streams. http://arstechnica.com/security/2015/04/it-wasnt-easy-but-netflix-will-soon-use-https-to-secure-video-streams/.

[4] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. López, K. Papagiannaki, P. Rodriguez, and P. Steenkiste. multi-context TLS (mcTLS): Enabling secure in-network functionality in TLS. *Proc. ACM SIGCOMM*, 2015.

[5] J. Sherry, C. Lan, R. Popa, and S. Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. *Proc. ACM SIGCOMM*, 2015.

[6] S. Loreto, J. Mattsson, R. Skog, H. Spaak, G. Gus, D. Druta, and M. Hafeez. Explicit trusted proxy in HTTP/2.0. *IETF Standards-Track Internet-Draft*, 2014.

[7] R. Peon. Explicit proxies for HTTP/2.0. *IETF Informational Internet-Draft*, 2012.

[8] D. McGrew, D. Wing, Y. Nir, and P. Gladstone. TLS proxy server extension. *IETF Informational Internet-Draft*, 2013.

[9] HackerNews discussion: explicit trusted proxy in HTTP/2.0. https://news.ycombinator.com/item?id=7296128.

[10] Explicit trusted proxy in HTTP/2.0 or... not so much. https://isc.sans.edu/forums/diary/Explicit+Trusted+Proxy+in+HTTP20+ornot+so+much/17708/.

[11] Evil or benign? 'Trusted proxy' draft debate rages on. http://www.theregister.co.uk/2014/02/25/evil_or_benign_trusted_proxy_draft_debate_rages_on.

[12] M. Thomson, G. Eriksson, and C. Holmberg. An architecture for secure content delegation using HTTP. *IETF Standards Track Internet-Draft*, 2016.

[13] J. Reschke and S. Loreto. 'Out-Of-Band' content coding for HTTP. *IETF Standards Track Internet-Draft*, 2016.

[14] M. Thomson, G. Eriksson, and C. Holmberg. Caching secure HTTP content using blind caches. *IETF Standards Track Internet-Draft*, 2016.

[15] C.W. Turner, M. Zavod, and W. Yurcik. Factors that affect the perception of security and privacy of e-commerce web sites. *International Conference on Electronic Commerce Research*, 2001.

[16] D.M. Kline, L. He, and U. Yaylacicegi. User perceptions of security technologies. *Privacy Solutions and Security Frameworks in Information Protection*, 2012.

[17] M. Thomson. Encrypted content-encoding for HTTP. *IETF Standards Track Internet-Draft*, 2016.

[18] RSA laboratories: What key size should be used? http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/key-size.htm.

[19] So you're making an RSA key for an SSL certificate. What key size do you use? https://certsimple.com/blog/measuring-ssl-rsa-keys.

[20] How big an RSA key is considered secure today? http://crypto.stackexchange.com/questions/1978/how-big-an-rsa-key-is-considered-secure-today.

[21] OpenSSL. https://www.openssl.org/.

[22] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for key management part 1: General. *NIST Special Publication*, 2006.

[23] E. Barker and A. Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication*, 2011.

[24] W. Ford and Y. Poeluev. An efficient certificate format for ECC. http://csrc.nist.gov/groups/ST/ecc-workshop-2015/presentations/session2-ford-warwick.pdf.

[25] R. LaRose and N. Rifon. Promoting i-safety: effects of privacy warnings and privacy seals on risk assessment and online privacy behavior. *Journal of Consumer Affairs*, 2007.

[26] N. Rifon, R. LaRose, and S. Choi. Your privacy is sealed: Effects of web privacy seals on trust and personal disclosures. *Journal of Consumer Affairs*, 2005.

[27] M. Eltoweissy, A. Rezgui, and A. Bouguettaya. Privacy on the web: Facts, challenges, and solutions. *IEEE Security and Privacy*, 2003.

[28] The HTTP Archive. http://httparchive.org.

[29] J. Hamilton. The Cost of Latency. http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/.