HTTPCrypt: Fast and Simple Secure Protocol

Paper **#273**

12 pages

Abstract

We introduce HTTPCrypt, a scheme of opportunistic encryption for the plain HTTP protocol designed to build web services compatible with the existing clients, servers and proxies ecosystem. HTTPCrypt uses state-of-art cryptographic primitives to provide both high performance throughput and low latency connection establishment, as well as complete interoperability with any middlebox that supports plain HTTP traffic. Unlike other opportunistic encryption schemes, HTTPCrypt defines procedures to establish name-based authenticity of a server even in the case of networks with restrictive access policies. We evaluate the practicality of deploying HTTPCrypt by integrating it into a popular HTTP server stack and evaluating it against alternative opportunistic encryption schemes.

1. Introduction

Transport Layer Security (TLS) has long been the standard choice for HTTP data encryption (to prevent passive snooping) and the validation of peer identities (to prevent active attacks). Clients connections are secured by first establishing a TLS connection to the endpoint, and then issuing standard HTTP requests across TLS tunnel. This separation has significantly increased the latency of encrypted web browsing due to the multiple network round-trips required, and motivated the proposal of TLS protocol extensions [24] to optimise the process.

The introduction of the HTTP/2.0 standardisation process [19] – the first major HTTP protocol revision since 1999 – has brought up the possibility of supporting opportunistic encryption for *all* web services, including those connections whose identities cannot be verified due to server misconfiguration, self-signed certificates, or expired signatures. All such proposals (§7.2) have involved upgrading the connection to TLS, thus maintaining the dependency to the complex suite of libraries that implement the full TLS protocol.

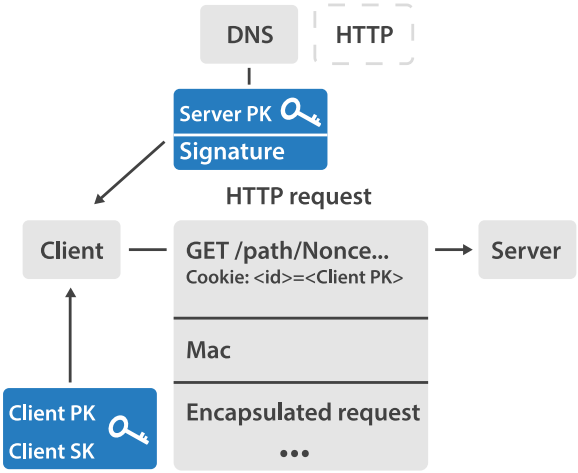


Figure 1. The overall structure of an HTTPCrypt connection. The server public key can be retrieved out-of-band via DNS, or directly over HTTP.

This paper introduces HTTPCrypt: an alternative application level HTTP extension that enables HTTP payload encryption with the following desirable properties:

- Transparency for existing HTTP caches and proxies, meaning that it is expensive to distinguish HTTPCrypt requests from plain HTTP requests;
- No significant latency increase of opportunistically encrypted connections *vs* plain HTTP;
- Removes the dependency on TLS and replaces it with cryptography suitable for embedded devices;
- Optional end-to-end identity checking;
- Low performance overhead for server software, including support for using the same efficient kernel system calls to transfer large files.
- Easy migration to the encrypted connections for the existing software

The remainder of this paper will first outline the high-level protocol properties (§2), followed by a detailed explanation (§3), protection against common attacks (§4), and discussion of some of the quirks unique to HTTP (§5), and con-

clude by evaluating our implementation of HTTPCrypt *vs* a popular application stack (§6).

2. Design Goals

HTTPCrypt is designed to work with the HTTP protocol and its many quirks, and avoids forcing a dependency on TLS as the sole method of opportunistic encryption. We will next explain our approach to HTTP compatibility (§2.1), the impact on performance and latency (§2.2), and finally how our approach is easy to embed (§2.3).

2.1 HTTP Compatibility

The HTTP protocol has explicitly supported proxying since its inception, and the influence of middleboxes can no longer be ignored when updating protocols [23]. Middleboxes have hampered the adoption of many new Internet protocols; from full transport stacks such as SCTP [31] or CurveCP [3], to extensions to existing protocols such as TCPCrypt [10] or MPTCP [2]. It is also common to encounter monitored gateways that provide Internet access purely through HTTP proxies that actively intercept or block HTTPS traffic.

The logical choice to avoid the influence of middleboxes while adding opportunistic encryption is to make HTTP requests that are difficult to distinguish from ordinary requests. In particular, the `Cookie` HTTP header is well suited to carrying cryptographic data, as it usually contains an encrypted payload that is indistinguishable from random data. Request URL is another suitable place for putting cryptographic data by the same reason as cookies.

Another main difference with the use of opportunistic encryption *vs* a fully established secure transport is that peer identities need not be fully verified. HTTPCrypt supports establishing the remote peer's identity via side channels such as the local DNS service, and we explain later (§3.1) why this is a reasonable approach in the modern Internet.

The basic scheme of an HTTPCrypt request is depicted in Figure 1. A client obtains the server public key out-of-band, and makes a normal HTTP request with the session key contained in the cookie. The encrypted contents then follow containing nonce and authentication tag.

In plain HTTP protocol, there is no payload within `GET` requests. However, for HTTPCrypt any request requires encrypted content. The first way to solve this incompatibility is to use `POST` requests for all HTTP messages. Another option is to encode the complete encrypted HTTP request within the request URL. In this case, the limit of HTTP request that could be encoded is about 2K due to URL length restriction (that is 2083 bytes). Since encrypted payload contains a nonce, all request URLs will be unique preventing thus caching on proxies. HTTP pragmas could also reduce chances to be cached by middleboxes for HTTPCrypt request but this does not matter for the protocol itself merely helping to reduce unnecessary caching by intermediate proxies.

HTTPCrypt is compatible with all the major HTTP transfer encodings, such as chunked encoding, and can maintain keep-alive connections just as normal HTTP does. Moreover, HTTPCrypt natively supports name based virtual hosts without the need for protocol extensions such as the TLS Server Name Indication [15].

Unlike TLS, where there is a ciphersuites agreement phase, we claim that it can be skipped for the vast majority of REST based HTTP services: in case of ciphersuite migration, it is possible to specify a new scheme explicitly, for example by creating a new set of access URL's or even by creating a dedicated host for the new ciphers. Moreover, the current encryption scheme used by a specific server could be explicitly stated in the DNS.

2.2 Latency and performance

HTTPCrypt requests follow the HTTP model of starting without any preliminary handshake phases. The client is responsible for obtaining the server public key (§3.1), and the only extra information that needs to be passed to a client connection is the client's own public key. The client can therefore encrypt an HTTP request immediately using its own private key and the server's public key, and assume that the server can decrypt the request using its own private key and the client's public key.

One drawback of this approach is the vulnerability to replay attacks, since a server cannot send its own random data before the initial client request. We discuss later how developers can mitigate this problem at the application level (§4.1) to prevent replaying the whole session. The initial request can always still be replayed, but this is not a security flaw if mitigated at the application level or if used to access read-only data.

HTTPCrypt is designed to establish HTTP sessions by skipping intermediate phases of key exchange or capability agreement for a connection. The disadvantage of this approach is that it reduces the flexibility of the connections, but it is also simpler and prevents downgrade attacks, such as the recent TLS vulnerability [25].

The performance and latency benefits from the above simplifications are significant, and make this is a viable approach for widespread implementation of opportunistic encryption. The data passed over HTTPCrypt is encrypted in-place, with a small authentication tag for a data chunk placed prior to each the encrypted payload. This allows using of multi-buffer kernel system calls such as `writev` to avoid data copying and improve performance. It also permits implementations to transfer arbitrary sized chunks of payload to optimize the connection utilization, and not be limited to a window of the maximum intermediate buffer size.

It is very common to encounter many short requests in HTTP, particularly when dealing with proxies that do not fully support HTTP/1.1 keep-alive. Opportunistic encryption schemes that require TLS handshakes are costly both

in terms of connection setup latency and the request rate. In contrast, HTTPCrypt skips the full handshake-term connections, thus supporting the asynchronous requests to advertising networks or loggins counters that are commonplace in modern websites.

2.3 Integration with the existing code

The TLS protocol stack is very complex, and even embedded implementations contain tens of thousands of lines of code. One reason for the code bloat is that all TLS implementations have to implement multiple cipher suites, key exchange and signing schemes for the purposes of backwards compatibility.

Rather than propagate this complexity into opportunistic encryption (which we want deployed as widely as possible), HTTPCrypt specialises its encryption to be based on the NaCL cryptobox primitives [5]. Cryptobox can be implemented using both a high-performance profile or as a small library within about a thousand lines of portable code [9].

The design of HTTPCrypt proposes the minimal changes to the existing applications: just use any suitable cryptobox library (choosing either performance or code size), get any HTTP parser library and encrypt the payload using cryptobox construction. Unlike TLS, HTTPCrypt does not interfere with the IO processing logic nor require some intermediate buffering.

3. Protocol Description

We now describe describe the architecture of HTTPCrypt in detail, via the handshake procedure (§3.1), request structure (§3.2), cryptographic primitives (§3.4) and session resumption (§3.5).

3.1 Handshake Procedure

The client initially obtains and checks the ephemeral public key of a server. Since we are dealing with opportunistic encryption, it is not necessary to protect this phase against active attacks. Methods of retrieving the public key include:

- Obtain an ephemeral public key from the corresponding DNS record, and check the authenticity of this reply using DNSSEC [1] or DNSCurve [6].
- Obtain a DNS record with the current ephemeral public key as a SPKI or x.509 certificate using, for example EdDSA [8] signature scheme to fit the whole certificate in a DNS record (which is typically allowed to be not larger than 512 bytes for many constrained networks).
- Perform a plain-text `HTTP OPTIONS` request to the target HTTP server and obtain the current certificate.

The Domain Name System is the preferred way to obtain the ephemeral public keys since DNS is used to resolve names and to verify domains ownership. If DNSSEC or DNSCurve are enabled for name resolution, it is also possible to validate the published key via DNS without PKI system. On the other hand, DNSCurve breaks intermediate DNS caching and DNSSEC can significantly increase requests latencies for clients as each name must be validated using multiple requests for each name component when using DNSSEC. DNS caching could reduce the negative effect of validation but it is still required to ask a caching server for each name component for a client. Moreover, DNSSEC and DNSSCurve can be filtered in restrictive networks, where plain DNS requests are the last resort available for the clients. Therefore, HTTPCrypt enabled services should publish the full PKI certificate in the DNS record to be compatible with all clients. Using of the modern signature schemes, such as EdDSA, allows storing a full x509 certificate within 512 bytes that is typical limit for a DNS record.

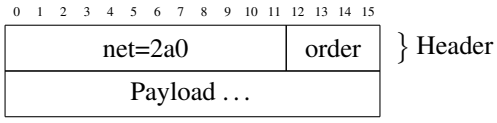


Figure 2. Key material encoding inside *AAAA* records

Storing ephemeral keys in the DNS defines their expiration time as equal to the DNS TTL value. HTTPCrypt defines these methods to store ephemeral keys in DNS:

- DANE `TLSA` records [22] that contain the full ephemeral key signed by a trusted key within x.509 or SPKI certificate;
- `TXT` records can carry the keys and signatures encoded with Base64 encoding;
- `AAAA` or `A` resource records can also encode keys and signatures that (described below)

AAAA record tunnelling Not all types of DNS records are allowed by some network policies; for instance, a common practice is to filter all unknown DNS resource records. `TLSA` records have been proposed relatively recently, and so are treated as unknown by many DNS recursive forwarders. `TXT` queries are forbidden in some networks since such records are frequently used to construct IP-over-DNS tunnels. This restriction also denies DNSCurve requests that use the `TXT` compatibility mode for their encrypted payloads.

In such a constrained network, HTTPCrypt can use IPv6 *AAAA* records to encode public key material. We store the relative order of the key material record inside the first 16 bits of the address and use the remaining 120 bits to encapsulate the payload in network-endian order. This encoding is depicted in Figure 2.

The first 12 bits are used to define IPv6 addresses. The `order` field is required to reconstruct the payload when a DNS server performs round-robin rotation of resource records. With 4 bits of order available, it is possible to encode up to 16 addresses with 112 bits of useful payload. Therefore, to encode a 256-bit key and 512-bit signature, we need 8 IPv6 addresses: three for the public key and five for

the signature. An additional address can be used to publish when this signature has been issued and the validity interval.

Deriving a session key After obtaining the server's public key, a client generates (or reuses) its own key pair and derives the shared secret using the `curve25519` scalar multiplication procedure on the server's public key and its local private key. An additional round of pseudo-random permutation then produces the final session key[5].

The resulting session key is subsequently used to encrypt and authenticate the HTTP payload. The authentication tag only covers the encrypted payload, since middleboxes can modify HTTP headers or the structure of a plain HTTP request. In contrast, the HTTP payload is not altered by proxies.

After receiving the HTTPCrypt request from a client, a server derives the shared secret using its local secret key and the client's public key from the `Cookie` HTTP header. The cookie also contains the ID of the server's public key (§3.2).

All further requests down the TCP connection (including HTTP `Keep-Alive` sessions) are encrypted using the established shared key. The keep-alive timeout must thus be less than the ephemeral key lifetime to guarantee forward secrecy for all long term connections.

3.2 Request structure

HTTPCrypt requests are designed to be difficult to distinguish from plain HTTP requests. Therefore, HTTPCrypt uses the HTTP `Cookie` header to provide the public key of a client to the server. The structure of this header is:

`Cookie:` ID_{sk}`||pad=`K_{client}`||pad`

All the values are encoded with `base64` that is commonly used in HTTP cookies. The padding prevents middleboxes from detecting the specific length of the public key and ID (if desired). Paddings should have unpredictable length and content for that purposes. The server key ID field applies a SHA3 cryptographic hash function to the server's public key and takes the first 10 bytes of its output:

`ID =` $take$`(10,` PRF`(pubkey))`

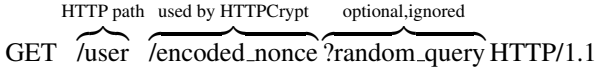
The server uses this ID when rotating ephemeral keys to select the correct key for a particular connection. Two ID collision probability is $1/2^{40}$ which is suitable for the purpose of public keys distinguishing on a server's side in conjunction with `Host` header.

In HTTPCrypt, the encrypted payload encapsulates the real HTTP request, but some HTTP headers are used from the unencrypted part, for instance the:

- **Host** is used to distinguish name-based virtual hosts and is required if those hosts have different public keys. Alternatively, the HTTP server may distinguish hosts by the ID_{sk} value and ignore this header.
- **User-Agent** header may be checked by middleboxes.
- **Content-Length**, **Content-Range** headers are used by the HTTP session as-is.
- **Pragma**, **Expires** and **Cache-Control** prevents proxies from caching encrypted requests.

While most of the HTTP headers could be sent unencrypted to save computational resources, some headers such as **Cookie** or **Referer** must remain encrypted (the former because its use is overloaded by the HTTPCrypt request structure). We do not define the exact list of headers that must be encrypted as this depends on the applications' architecture.

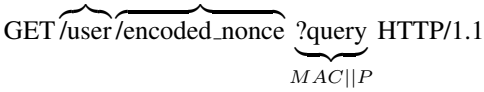
If the payload is placed within HTTP body, HTTPCrypt uses the unencrypted URL to store the initial nonce for the request. We propose to use the first path component after the HTTP path as nonce whilst the initial path could be used by a web server to find HTTPCrypt protected resource. The overall URL structure is depicted as following:



The encrypted payload has the following structure (|| denotes concatenation):

MAC||Encrypted HTTP request

In the case when request is completely encoded within URL, the encrypted payload is placed (base64 encoded) into the query parameters component of URL. The URL structure changes in this case to the following one:



3.3 Chunked encoding

In HTTP standard, chunked encoding [17] must be supported by any client, therefore any proxy can split the original request into chunks. Hence, HTTPCrypt provides support to transfer chunked HTTP messages securely by means of the internal encrypted chunks. To distinguish between chunked and plain encoding, HTTPCrypt uses the special query `?chunked=1` to tell a client that the request is split to a set of encrypted chunks. The traditional `Content-Transfer-Encoding` header could not be used since it can be altered by middleboxes (for example, when a proxy performs gzip encoding). However, encrypted chunks support requires more complicated procedure to decode, so it might be avoided when implementing simple services. In this case, a client always reconstruct a server's reply as a single chunk; even if the plain request contains chunks they are merged into a single buffer for decryption and verification.

When HTTP chunked encoding is used, each chunk has its own nonce, MAC tag and additional length of the encrypted content. The second length specifies the size of data

that is authenticated by this MAC with the specific nonce. It is needed when a middlebox or a proxy performs chunks resegmentation. In this case, a client will still be able to restore the original chunks sent by a server, decrypt and authenticate them properly. To avoid reordering of chunks, the subsequent nonces after the first one should be incremented as counter. Additionally, each authentication tag should be calculated using the previous authentication tag in addition to the current encrypted chunk content as following:

$$M(i) = M_n(M_{n-1}(i-1)||P)$$

where P is the current encrypted payload and n is the current nonce value. In this scheme, an attacker cannot remove any chunk of data from a message: if he or she removes the first chunk, then MAC won't be valid for the subsequent chunk as the original MAC is calculated using the first chunk MAC. The same logic is valid for all subsequent chunks. Even in the case if chunks have the same content, the fact that they are encrypted using different nonces and fact that the MAC tag is calculated over the encrypted payload (encrypt-then-mac method) provide negligible chances that MAC tags for these equal chunks will be the same.

Nonetheless, the last chunk is special. In plain HTTP, the last chunk has always zero length. However, re-using of this plain scheme opens a security breach: an attacker can simply remove some chunks from the end of a message adding unencrypted zero-length chunk. Therefore, HTTPCrypt defines a special encrypted zero-chunk which contains a next nonce, zero encrypted length and authentication tag calculated as following:

$$M = M_{n+1}(M_n(i-1)||0)$$

After this special tag, all following payload could be used for compatibility with HTTP (namely, the last zero chunk) and should be ignored by HTTPCrypt protocol.

The structure of chunked HTTPCrypt request is depicted as following:

```
<Chunk length>
┌──────────────────────────────┐
│<MAC>                         │
│<Encrypted payload length>    │
│<Encrypted chunk content>     │
└──────────────────────────────┘
...
<Chunk length>
┌──────────────────────────────┐
│<Nonce>                       │
│<MAC>                         │
│0                             │
└──────────────────────────────┘
<HTTP zero chunk>
```

To reconstruct the final HTTP message HTTPCrypt defines the following procedure:

1. Parse the original request and append all unencrypted headers except for the `Cookie` header.
2. For each encrypted chunk or the whole message increment the previous nonce, read the authentication tag and verify the encrypted content using the previous MAC tag as additional data for authentication (for the second and subsequent chunks). If the verification succeeds, decrypt the contents and parse the encapsulated request.
3. The request URI in the encapsulated request replaces the original URI and all encrypted headers replace the corresponding unencrypted headers.

If MAC verification fails for any chunk, the peer sends an HTTP 500 error code inside the encrypted payload to prevent connection termination by an adversary who can inject arbitrary HTTP messages to the peers.

3.4 Cryptographic primitives

HTTPCrypt encryption is based on the Cryptobox construction introduced by Daniel Bernstein in the NaCL cryptographic library [5]. This construction defines both secrecy and integrity for the messages based on public key cryptography, and is a conjunction of *(i)* public key exchange procedure; *(ii)* a stream cipher; and a *(iii)* one-time authentication algorithm.

The wire format of Cryptobox specifies a cryptographic nonce, an authentication tag (MAC) and the encrypted payload.

Nonces Nonces for the first message in an HTTP session is generated randomly by both a client and a server. Nonces length is chosen to be long enough to make the probability of repetition negligible. Bernstein defines a nonce length of 24 bytes which is 2^{192} of possible nonces values and the probability of nonces collision as low as at least $1/2^{128}$ [5]. For the subsequent chunks or messages within the same keep-alive HTTP session, client and server should monotonically increase their initial nonces in counter like matter. Switching to counters allows to skip nonces in all but the first chunks.

Algorithms selection The original NaCL implementation suggests the following components to be used in the cryptobox:

- `curve25519` and `hsalsa20` as the public key exchange operation
- `xsalsa20` for symmetric encryption
- `poly1305` for message authentication

At present, the `salsa` cipher family is superseded by `chacha` ciphers that have the same internal architecture but are optimised for the modern hardware and vectorized operations. We later evaluate the `chacha` cipher *vs* other state-of-art ciphers such as `AES` (§6). The most significant advantage of `chacha` is good performance on the commodity hardware, including embedded systems based on ARM or MIPS processors as well as x86.

Hence, our final selection of symmetric encryption algorithm is the `chacha20` stream cipher. Moreover, the `hsalsa`

and `xsalsa` ciphers are replaced with the corresponding `chacha` variants as described in [7].

3.5 Session Resumption

Since the generation of public key shared secrets is an expensive procedure, HTTPCrypt defines several approaches to skip it for already establishes sessions. HTTPCrypt uses the same common principles that are defined in TLS [12] by means of a session cache (§3.5.1) and ticket mechanism (§3.5.2).

3.5.1 Session Cache

The session cache stores some of the client's state on the server side to avoid recalculating it. The state in HTTPCrypt includes the hashes of the client's and server's ephemeral public keys, and the resulting shared secret. When reestablishing a session, a server finds the cached state based on the public keys fingerprints and reuses the shared secret instead of regenerating it using public key cryptography. Session expiration is based on the server's ephemeral public key lifetime, and a least-recent-use expiration strategy if there are more clients than session cache space available. The server must destroy sessions associated with an expired ephemeral key in order to ensure that forward secrecy is preserved.

3.5.2 Sessions tickets

Session tickets are a mechanism for an HTTPCrypt server to support session resumption without having to store per-client state [30]. This method implies that a client supports and enables tickets when resuming an HTTPCrypt connection. When using session tickets, the shared state is encrypted by a symmetric secret key known only by the server and subsequently passed to the client. The client can then reconnect by including a previously obtained session ticket in the initial handshake. The server decrypts and verifies this ticket and restores the session without an additional key exchange procedure being required. Session tickets in HTTPCrypt have the same structure and definition as in TLS with the only difference that an HTTP header is used to store and pass a session ticket.

4. Security Analysis

We now elaborate on the HTTPCrypt protocol's security properties (§4.1), support for forward secrecy (§4.2) and denial of service resistance (§4.3). We define the threat model for HTTPCrypt as follows:

- HTTPCrypt should be resistant to passive, active and denial-of-service attacks;
- HTTPCrypt should provide both secrecy and integrity for transmitted payload;
- There should be no easy way to distinguish HTTPCrypt requests from plain HTTP traffic.

4.1 HTTPCrypt Security Model

To defend against active attacks such as Man-in-the-Middle, HTTPCrypt uses the traditional model of peer validation using public key signatures and trusted 3rd party authorities. When using DNS-based signatures the authorities are defined as trusted DNS anchors, while in the case of a certificate chain HTTPCrypt uses the traditional PKI model where a peer's key can be signed by any trusted authority.

HTTPCrypt recommends the use of DNS chains of trust granted by means of DNSSEC or DNSCurve anchors. Nevertheless, for embedded appliances or difficult-to-change infrastructure the cost of a complete DNSSEC validation might be too expensive. Furthermore, the cryptographic algorithms and standards recommended by DNSSEC (e.g. 1024-bit RSA), are more expensive than state-of-art cryptography.

In contrast, DNSCurve provides secure and efficient cryptographic primitives but is not widely deployed in the Internet for various reasons – both historical and more practical since DNSCurve does not interoperate with intermediate DNS caching. Therefore, the traditional PKI model based on EdDSA signatures [8] is a reasonable fall-back choice for HTTPCrypt validation.

4.1.1 Replay Protection

Protecting against reply attacks is more complicated than in HTTPS, since HTTPCrypt does not require a server handshake with a random cookie provided by the server. In HTTPCrypt, the first request can always be replayed by an adversary for the duration of the server's ephemeral key it is purely client initiated.

It is possible (and recommended) to implement replay protection at the application level, for example by providing a unique authentication token from server to a client before granting access to the restricted area. However, a session before the first server reply is not protected against replay attacks.

To protect the subsequent session, the server places the random cookie in the encrypted and authenticated reply, for example inside a predetermined HTTP header. If the first request sent over HTTPCrypt is limited to an idempotent `GET` method, an adversary can capture and replay the first request, but will not be able to gain any advantage since the server's reply then includes a random element. Therefore, an attacker cannot replay any subsequent messages within a session, in particular side-effecting operations such as HTTP `POST` or `DELETE` methods.

Here is the practical example of replay protection applied to an HTTPCrypt session. Initially, a client sends a `GET` request that contains a client's random cookie within the header inside the encrypted payload to provide protection from replayed server's messages:

```
Client-Random: <24 bytes of random data>
```

A server, in turn, generates the full authentication token (that should be long enough to make repetition probability negligible) by appending its own random cookie and pushes it inside the encrypted header:

`Random:␣<client␣random><server␣random>`

This model provides effective replay attacks protection. For example, if an adversary can repeat the server's replies, then a client will not be able to match its own random part. Similarly, a server will fail to verify its own cookie and will drop the replayed requests if the client is replayed. Placing random cookies inside the authenticated and encrypted payload prevents an adversary from both observing or modifying the tokens.

4.2 Forward Secrecy

HTTPCrypt uses slowly rotating ephemeral public keys for HTTP servers to provide forward secrecy [14]. Clients generate a new keypair for each unique HTTPCrypt session. If DNS is used to store and validate public keys, the rotation of ephemeral servers keys is implicitly defined by the DNS time-to-live (TTL) property used as the lifetime value for the ephemeral server's key. To avoid time synchronisation issues, servers should generate new ephemeral keys at each period of time equal to the DNS resource record's TTL value, and publish keys material to the DNS server.

HTTPCrypt does not mandate an exact procedure for updating the DNS, since any of the standard methods used all serve; e.g. AXFR, an LDAP directory or by executing scripts via SSH. Servers just need to be able to store ephemeral keys for the time equal to **two** DNS TTL values to be able to interact with the clients that have previous keys cached in some DNS cache Hence, the real ephemeral key lifetime is two DNS TTL periods. The servers should destroy the keys from persistent storage once they have expired.

4.3 Denial-of-Service Protection

Availability is an important property of HTTPCrypt if it is to achieve wide deployment. Unlike TLS, the HTTPCrypt server computes the shared key one the first stage of a connection. This operation is expensive in terms of CPU resources, whereas in TLS all computationally complex procedures are performed at the later stages of the handshake (starting from the second message received from a client).

At first glance, this is a disadvantage of the HTTPCrypt design. However, we observe that the TCP three-way handshake protects a HTTPCrypt server from promoting spoofed requests, to the established state. On the other hand, if an adversary is able to establish a valid TCP connection (for instance, via a distributed botnet) then there are no obstacles to continue to the additional stages of a TLS negotiation and force the server to execute CPU expensive computations.

Therefore, HTTPCrypt is no more vulnerable to denial-of-service attacks than HTTP+TLS. Moreover, since the random response cookies are signed by the server in TLS, it requires more resources to perform signing and shared secret generation than the HTTPCrypt mechanism of merely generating a shared secret and encrypting the cookie. HTTPCrypt could also upgrade its scope to include cryptographic puzzles as part of its handshake, for example as defined in the MinimalT [28] protocol. The concrete definition and evaluation of crypto puzzles are beyond the scope of this paper.

5. Discussion

We now describe the implementation peculiarities used by our HTTPCrypt prototype, beginning with low-level cryptographic optimisations (§5.1), operating system acceleration (§5.2), and integration with existing application stacks (§5.3).

5.1 Cryptobox Optimizations

As a default ciphers suite HTTPCrypt proposes chacha20 [4] as stream cipher and pseudo-random function, poly1305 as one time authenticator, and curve25519 as key exchange function. However, these primitives could be easily switched to another ones (for example, further we demonstrate openssl cryptography used by HTTPCrypt). In our experiments, we use optimized versions of chacha20-poly1305 [26] and optimized version of curve25519 ECDH implementation derived from Sandy2x [11]. Both bulk encryption and key exchange algorithms benefit from AVX instructions set implemented in Intel processors.

We have also extended the original NaCL cryptobox primitive to allow vectorization of the encryption, for example, to encrypt headers and body in the same call to the API. This change still allows to encrypt and authenticate data in-place and avoid intermediate buffering.

5.2 Operating System Optimizations

Contemporary operating systems provide various high performance systems calls to optimise the I/O handling for serving HTTP requests with a high throughput. For example, an HTTP server running on Linux or FreeBSD can utilise the `sendfile` [33] system call to transfer a file to a socket directly via the kernel without requiring intermediate copying through user-space buffers. Some variants of this system call (e.g. the FreeBSD `sendfile`) also accept arbitrary prefixes as arguments.

Unlike TLS that requires intermediate buffering, HTTPCrypt can use the semantic of `sendfile` call to send files encrypted without requiring copying through userspace. To support HTTPCrypt, the `sendfile` interface needs to be extended to accept a session key and a generated nonce. Bulk data can be encrypted directly in the kernel. In contrast, in TLS other protocol components, such as TLS Alerts or other extensions all require complex processing that is not easy (or wise) to put into the kernel.

```
int http_crypt_write(
  char *plain, size_t plainlen,
  char *payload, size_t paylen,
  char *pk, char *sk) {
  struct iovec iov[4];
  unsigned char n[NONCELEN], m[MACLEN];

  randombytes(n, sizeof(n));
  cryptobox_encrypt_inplace(
    payload, paylen, n, pk, sk, m);
  iov[0].iov_base = plain;
  iov[0].iov_len = plainlen;
  iov[1].iov_base = n;
  iov[1].iov_len = sizeof(n);
  iov[2].iov_base = m;
  iov[2].iov_base = sizeof(m);
  iov[3].iov_base = payload;
  iov[3].iov_len = paylen;
  return (writev(fd, iov, 4));
}
```

Figure 3. Sample code fragment illustrating HTTPCrypt request creation and writing to a socket in C

5.3 Integration with Existing Software

HTTPCrypt is designed to be integrated with existing application easily. While it is relatively straightforward to migrate the already written plaintext services to TLS by means of a proxy such as `stud`[1], it is more difficult to integrate TLS stack directly into an application that is not specifically designed for to support TLS for the following reasons:

- TLS alerts and handshakes change the connection processing logic significantly, especially for asynchronous or non-blocking applications;
- TLS uses intermediate buffering for all data transfers that leads to additional latency and performance penalties.

In contrast, in HTTPCrypt, there are no protocol alerts or additional handshake stages to complicate integration. Applications thus can send or receive data without any intermediate steps, leaving the event processing logic in the client and server unchanged. In TLS, read operation might require writing and vice-versa: write operation might request reading. Moreover, an application can create messages in HTTPCrypt without copying data to an intermediate buffer since all data is encrypted and authenticated in-place. Listing 3 demonstrate how simple HTTPCrypt request creation using multiple buffers operations is (namely, `writev` and `readv`).

Reading and processing of HTTPCrypt requests is implemented by extracting the authentication tag from the encrypted payload, and parsing the following encapsulated HTTP request as defined earlier (§3.2).

[1] `https://github.com/bumptech/stud`

To migrate to HTTPCrypt from plaintext HTTP, applications must also use a cryptographic quality random number generator to generate nonces and key pairs. This is particularly important where there is not much entropy available, for example in embedded devices [20] or virtual machines [16]. However, this requirement is held for TLS as well.

5.4 Embedded usage

HTTPCrypt is particularly well suited to embedded devices where including the full TLS suite is too large or too slow to use. The CPUs used in embedded appliances are often not able to drive encrypted connections at a reasonable rate as they have neither hardware cryptographic acceleration nor optimised instructions cores.

HTTPCrypt with the static key model is a better choice to protect communications on such embedded devices. Despite the fact that this scheme does not guarantee forward secrecy, it is still better than plaintext HTTP connections by providing stronger confidentiality and authentication properties.

There are several optimised embedded implementations of the Cryptobox construction elements used in HTTPCrypt; for example, ARM NEON specific optimisations to speed up `ChaCha20-Poly1305`. There is also a generic implementation of Cryptobox optimized for code size and memory consumption called TweetNaCl [9]. This library supports the digital signatures created by the Ed25519 algorithm, which can be used to check the identity of ephemeral keys via the PKI chain-of-trust model.

We have evaluated the performance of generic version of our HTTPCrypt prototype on Cortex A20 ARM board. And even with the generic unoptimized C versions of all cryptographic primitives it has shown highes requests per second rate then TLS stack (50 complete encrypted requests per second against 30 requests per second for TLS). However, the detailed evaluation on embedded platforms is beyond the scope of this paper.

6. Evaluation

We have built the prototype of HTTPCrypt built on top of the `http-parser` library [32] that is in turn based on the popular Nginx HTTP server code. The goal of our tests was to compare HTTPCrypt with the standard web workloads using TLS. We have compared our implementation against Nginx 1.9.5 built with OpenSSL 1.0.2d using TLS v1.2 protocol with `nistp256` curves for both ECDSA and ECDHE.

We used the following configuration for the Nginx benchmarks:

```
ssl_ciphers "ECDHE-ECDSA-AES256-GCM-SHA384";
ssl_session_cache off;
ssl_session_tickets off;
ssl_ecdh_curve prime256v1;
keepalive_timeout 0;
```

For HTTPCrypt testing, we wrote our HTTP server and benchmarking tool based on the same principles as `wrk` (non-blocking IO) and the same HTTP parser.

We ran a sequence of experiments using both Nginx and the HTTPCrypt prototype. The HTTP client is the `wrk` [18] HTTP benchmarking utility with a single testing thread and 50 parallel connections in the test runs.

We first ran the servers in plain HTTP mode with no encryption enabled at all ("Unencrypted"). When then disabled SSL sessions cache/tickets in order to evaluate the performance of complete TLS handshakes ("Encrypted, uncached"). In the last experiment, we turned on the SSL session cache to evaluate the performance of session resumption ("Encrypted, tickets").

The selection of cipher suites and the ECDHE curve was based on the assumption that on the tested CPU with hardware AES support (via AES-NI instructions) and vectorised operations (AVX instructions), the speed of these particular primitives was optimal. We used the `openssl speed` command to ensure that the performance of specific algorithms was optimal on the tested hardware:

```
256 bit ecdh (nistp256):   8582.6 op/s
256 bit ecdsa(nistp256):   15000.0 signs/s
aes-256-gcm:               731623.42 kB/s
```

In all cases, we evaluated serving static files using a single process on the client and one on the server to estimate latency and the number of requests per second that were processed. The client and server were connected over the 10G network interface, and all requests were successfully processed.

We have also evaluated HTTPCrypt proxying using both forward proxies, such as squid or tinyproxy, and reverse proxies, such as nginx or lighttpd. We have not found any compatibility issues when traversing HTTPCrypt requests over these proxies.

6.1 Performance Evaluation

Figure 4, shows the number of requests per second of the experimental runs with Nginx, and Figure 5 shows the same workload patterns obtained from the HTTPCrypt test suite.

The important results are the tests with encryption enabled, and in this case HTTPCrypt demonstrates significantly superior requests per second than Nginx/TLS for the transfer of small files. For larger request the throughput benefits of HTTPCrypt are not so clear, since the handshake cost is negligible comparing to the cost of bulk encryption and network transfer.

The better performance of HTTPCrypt is achieved by use of the faster ECDH crypto primitives, the elimination of the handshake stages, and skipping encryption of the unnecessary HTTP headers in favour of the payload.

Furthermore in the Figure 6, we demonstrated the difference between OpenSSL and HTTPCrypt comparing them with nginx+TLS as the baseline. OpenSSL mode used the

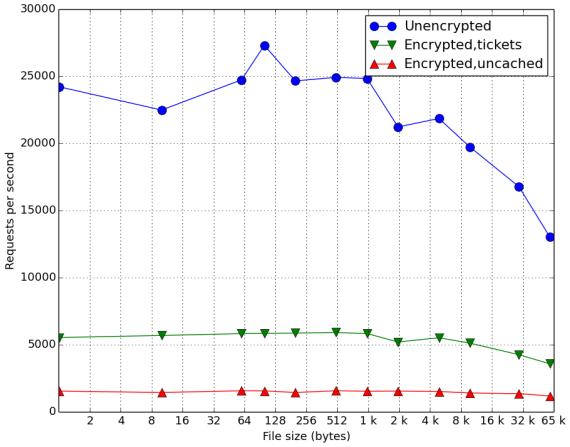


Figure 4. The performance of Nginx while serving static files using 1 process on Intel Xeon E5 2.4 GHz

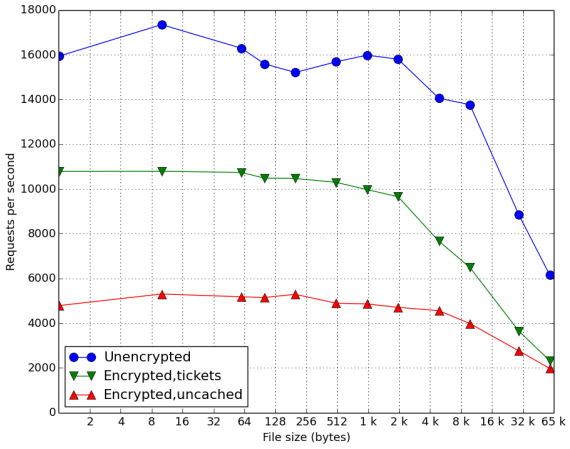


Figure 5. The performance of HTTPCrypt prototype with ChaCha20-Curve25519 crypto while serving files using 1 process on Intel Xeon E5 2.4 GHz

following set of crypto primitives: NIST p256 curve for ECDH, AES-256-GCM for authenticated encryption and hchacha20 as PRF (PRF selection does not influence the overall performance since its execution time is negligible comparing to ECDH procedure). This test shows that even with the equal cryptography model HTTPCrypt is significantly faster than TLS for small requests that are common for the web RPC services.

In the Figure 7, we demonstrated scaling of HTTPCrypt prototype from the number of independent processes running on the same machine comparing to TLS. For both nginx/TLS and HTTPCrypt we used the equal number of processes for both client and server. This experiment illustrates

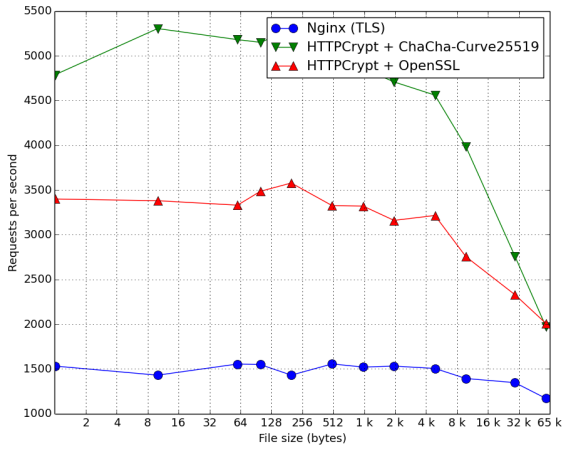


Figure 6. The performance of HTTPCrypt prototype with OpenSSL crypto while serving files using 1 process on Intel Xeon E5 2.4 GHz

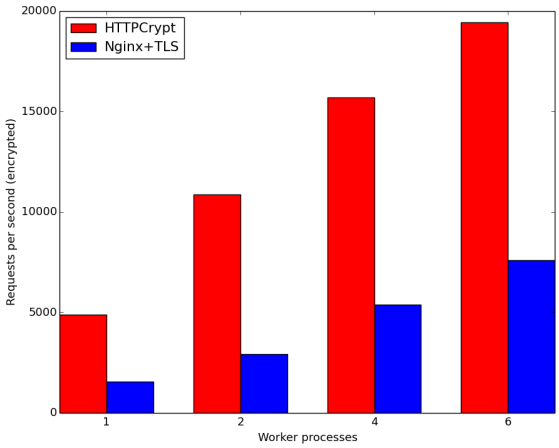


Figure 7. HTTPCurve and Nginx+TLS scaling from the number of worker processes

the capabilities of HTTPCrypt to scale with the CPU cores amount increasing.

6.2 Overhead evaluation

The traffic overhead is another important property of an encryption protocol. Therefore, we have compared the extra data required for HTTPCrypt and TLS connections depending on the payload size. We have chosen the plain HTTP request of the same size as the baseline and estimated both incoming and outcoming traffic on the client side. The results depicted in the Figure 8 shows that HTTPCrypt introduces less overhead than TLS, especially for small requests, that could reduce the overall traffic for the encrypted connections comparing to the TLS case.

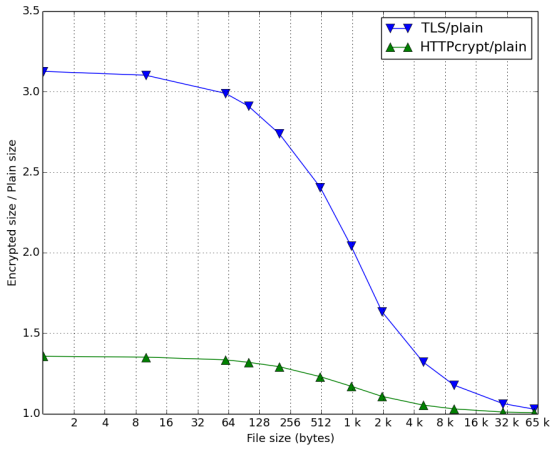


Figure 8. HTTPCurve and Nginx+TLS traffic overhead comparing to the plain HTTP

6.3 Latency evaluation

Request latency is an important property for opportunistic encryption, since we want this to be deployed widely and with no user-visible impact. We compared the request latency for different encryption methods using the unencrypted latency as the baseline. We evaluated the same three payload models used earlier (§6). We have first ensured that all latencies are distributed normally by building Q-Q plot demonstrated in the Figure 11.

The evaluation results are shown in Figure 9 for Nginx and Figure 10 for HTTPCrypt. We measured the latency between connecting to the HTTP server, sending a request and receiving the reply that concludes the complete HTTP session (keep-alive was disabled). Connection latency is included as well since the socket connection time introduces a small and constant delay.

The latency tests are consistent with the earlier throughput evaluation: HTTPCrypt provides lower delay than TLS, but latency degrades when serving large files due to the current lack of IO optimisations in our prototype. However, once again the important result is that for encrypted connections, the benefit from HTTPCrypt compared to HTTPS is very significant, especially for the common case of small HTTP response sizes.

Table 1 summarises the work performed by a server to establish a TLS connection, whilst Table 2 depicts the corresponding messages for HTTPCrypt. In TLS, the length of the initial handshake is at least 4 round trips and can be extended to even longer in some situations (such as when using a long certificate chain).

The most expensive computational operations for TLS are generating a shared secret and signing the random cookie. In contrast, HTTPCrypt does not require the server to do any signing, since the random cookie is cheaply placed

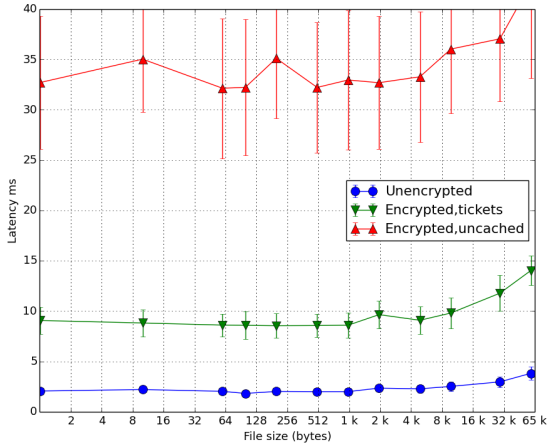


Figure 9. The request latency for Nginx while serving files using 1 process on Intel Xeon E5 2.4 GHz

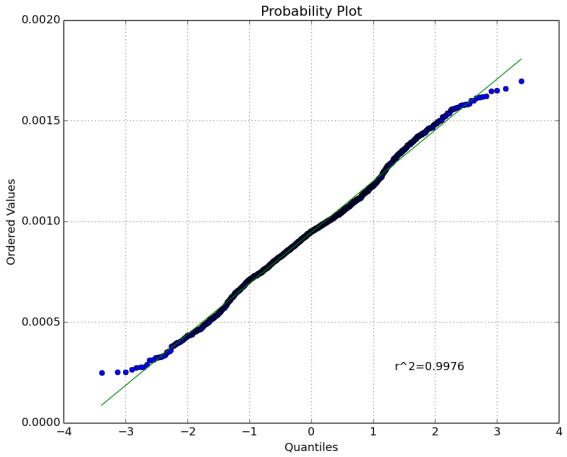


Figure 11. Q-Q plot of latencies distribution for HTTPCurve serving 500 bytes payload with 1 worker process

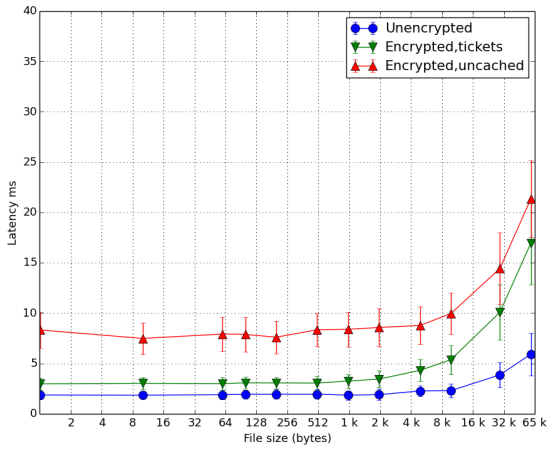


Figure 10. The request latency for the HTTPCrypt prototype while serving files using 1 process on Intel Xeon E5 2.4 GHz

RTT	Payload	Work done
1	CHello	Generate server random
2	SHello Certificate	Send certificates
2(3)	SKeyEx	Sign random and ephemeral key
3(4)	CKeyEx	Generate shared secret
4(5)	CCSpec	-

Table 1. Computation executed by a server at each stage of TLS connections

RTT	Payload	Work done
1	Request Cookie Payload	Generate shared secret
2	Reply Payload	Create server random addition

Table 2. Computation executed by a server at each stage of HTTPCrypt connections

within the encrypted and authenticated payload. The work in the first stage is significant comparing to the first stage of TLS; however we discussed in §4.3 that this difference does not make HTTPCrypt more vulnerable to denial-of-service attacks than TLS.

7. Related work

There have been several proposals related to improving opportunistic encryption support in HTTP, which we now discuss.

7.1 Transport protocols

TLS 1.3 TLS version 1.3 [13] is the next version of TLS protocol that defines various features and improvements that are similar to HTTPCrypt properties. For example, it introduces Zero-RTT connection establishment which involves obtaining of a server's credentials prior to connection creation. This feature looks very similar to HTTPCrypt and allows to skip several stages of TLS connection handshake and skip online signing stage. However, even in this case, replay protection either requires additional stages or TLS states that the first client request payload is not replay protected which could lead to future misuse of Zero-RTT connection feature in applications. On the other hand, this TLS version does not change the application interfaces meaning that there is still the requirement of data copying and changing the IO logic when enabling TLS encryption. Proxies interaction is

not defined as well, meaning that an incompatible proxy could force protocol downgrade to a previous version. Furthermore, if TLS communications are disabled in some constrained network this filter would apply to TLS 1.3 connections. HTTPCrypt connections are more hard to detect and to block by such policies.

TCPCrypt TCPCrypt [10] defines a TCP extension that implements opportunistic encryption. TCPCrypt does not define a method to authenticate peers, and extends the TCP handshake with a certificate exchange phase (performed by extra packets with the TCP `PUSH` flag enabled).

TCPCrypt adds one additional RTT over vanilla TCP in order to establish an encrypted connection, but this handshake procedure requires special TCP packets that could be filtered by middleboxes.

The lack of peer authentication allows active attacks such as man-in-the-middle. Nevertheless, TCPCrypt requires no modifications to existing applications and thus improves the status quo of TCP not having any encryption by default.

MinimalT MinimalT [28] is designed to create low latency encrypted tunnels. The authentication and encryption model in MinimalT are very similar to the HTTPCrypt ones proposed in this paper. For instance, MinimalT uses Cryptobox for payload encryption, and suggests storing ephemeral keys in the directory service, assuming that there is a way to establish the authenticity of directory replies (e.g. via DNSCurve trusted anchors if DNS is used as a directory).

Another interesting feature defined in MinimalT is the use of crypto-puzzles instead of the traditional TCP-like handshake to reduce connection latency significantly while remaining resistant to denial-of-service attacks. However, MinimalT also uses a UDP transport and assumes that higher level protocols will provide reliability delivery, reordering and congestion control facilities. In contrast, HTTPCrypt avoids reinventing these aspects of the transport in order to preserve compatibility with existing middlebox infrastructure where possible.

7.2 HTTP/2.0 opportunistic encryption

HTTP 2.0 is an effort within the *httpbis* working group in the IETF to develop a standardised successor to the HTTP 1.1 protocol. While initial revisions of the specification mandated the use of TLS encryption with HTTP 2.0, this was subsequently made optional in later drafts due a lack of consensus as to the practicality of this approach. There are now two proposals for HTTP/2.0 that suggest different schemes of opportunistic encryption [21, 27]. Both of these schemes aim to provide protection from passive attacks and do not define concrete methods to encrypt the HTTP payloads.

In the first Internet draft by Paul Hoffman [21], the minimal unauthenticated encryption scheme is proposed. It defines an additional agreement stage to establish a shared secret and to select the ciphers suite for a connection. During these stages, peers can efficiently establish a random cookie to prevent session replay attacks.

In the second draft, Nottingham and Thomson [27] propose the of use of the `http` URI to indicate those HTTP/2.0 connections that are encrypted using *unauthenticated* TLS. Services can indicate their support for this mode via an `HTTP-TLS` header in their responses.

HTTPCrypt proposes an incremental deployment scheme that can also include peer authentication. To simplify deployment, HTTPCrypt can first be used as a purely opportunistic encryption scheme, providing only secrecy in this mode without any peer authentication. Unlike the above proposed schemes, HTTPCrypt used in unauthenticated mode requires minimal modifications to existing applications.

8. Conclusions

In this paper, we have proposed and evaluated HTTPCrypt – an opportunistic HTTP encryption scheme for web service that is based on modern state-of-art cryptographic primitives. HTTPCrypt is designed specifically to interoperate with the existing HTTP ecosystem of middleboxes, and be relatively easy to integrate into clients and servers. Unlike related protocols and extensions (§7), HTTPCrypt also extends beyond pure opportunistic encryption to defines server authentication methods using the DNS or HTTP to fetch peer public keys.

We have built a prototype of HTTPCrypt client and server and evaluated latency, throughput, scaling and overhead for different payload types. The test results (§6) shows that HTTPCrypt provides better performance and lower request latency than traditional HTTP/TLS due to a more lightweight handshake mechanism. The integration of HTTPCrypt to existing HTTP servers requires few fewer dependencies than introducing the full TLS stack just for the purposes of opportunistic encryption (§5.3).

The major drawback of HTTPCrypt is the absence of builtin protection from replay attacks. However, we have discussed application-level methods to resolve this (§4.1.1) and believe that the resulting simplicity and performance improvements increase the viability of wider deployment of the protocol. Furthermore, the majority of web applications designed for operation over plain HTTP have already replay attacks protection which can be reused in HTTPCrypt.

We believe that HTTPCrypt is a timely contribution as the importance of ubiquitous opportunistic encryption is becoming ever more important, and early proposals have begun to be discussed in standards bodies (§7.2). We are continuing to work on HTTPCrypt by building a more efficient implementation using asynchronous IO, a browser plugin and HTTP proxy cache, and defining more fine-grained denial-of-service attacks protections (§4.3). The source code to our implementation is open-source under a BSD license and available from *blinded for review*.

References

[1] ATENIESE, G., AND MANGARD, S. A new approach to DNS security (DNSSEC). In *Proceedings of the 8th ACM conference on Computer and Communications Security* (2001), ACM, pp. 86–95.

[2] BARRE, S., PAASCH, C., AND BONAVENTURE, O. Multipath TCP: from theory to practice. In *NETWORKING 2011*. Springer, 2011, pp. 444–457.

[3] BERNSTEIN, D. J. CurveCP: Usable security for the internet, dec 2010. *http://curvecp. org*.

[4] BERNSTEIN, D. J. ChaCha, a variant of salsa20. Tech. rep., The University of Illinois at Chicago, 2008.

[5] BERNSTEIN, D. J. Cryptography in NaCl. *Networking and Cryptography library* (2009).

[6] BERNSTEIN, D. J. DNSCurve: Usable security for DNS, 2009.

[7] BERNSTEIN, D. J. Extending the salsa20 nonce. In *Workshop record of Symmetric Key Encryption Workshop* (2011), vol. 2011.

[8] BERNSTEIN, D. J., DUIF, N., LANGE, T., SCHWABE, P., AND YANG, B.-Y. High-speed high-security signatures. *Journal of Cryptographic Engineering 2*, 2 (2012), 77–89.

[9] BERNSTEIN, D. J., VAN GASTEL, B., JANSSEN, W., LANGE, T., SCHWABE, P., AND SMETSERS, S. TweetNaCl: A crypto library in 100 tweets, 2014.

[10] BITTAU, A., HAMBURG, M., HANDLEY, M., MAZIERES, D., AND BONEH, D. The case for ubiquitous transport-level encryption. In *USENIX Security Symposium* (2010), pp. 403–418.

[11] CHOU, T. Sandy2x: New curve25519 speed records.

[12] DIERKS, T. The transport layer security (TLS) protocol version 1.2.

[13] DIERKS, T., AND RESCORLA, E. The transport layer security (tls) protocol version 1.3. draft-ietf-tls-tls13-07, 2015.

[14] DIFFIE, W., VAN OORSCHOT, P., AND WIENER, M. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography 2*, 2 (1992), 107–125.

[15] EASTLAKE, D., ET AL. Transport layer security (TLS) extensions: Extension definitions.

[16] EVERSPAUGH, A., ZHAI, Y., JELLINEK, R., RISTENPART, T., AND SWIFT, M. Not-so-random numbers in virtualized linux and the whirlwind RNG. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 559–574.

[17] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext transfer protocol–HTTP/1.1, 1999.

[18] GLOZER, W. Modern HTTP benchmarking tool. `https://github.com/wg/wrk`, 2015. Accessed: 2015-02-20.

[19] GROUP, I. H. W., ET AL. HTTP 2.0 specifications, 2013.

[20] HENINGER, N., DURUMERIC, Z., WUSTROW, E., AND HALDERMAN, J. A. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, 2012), USENIX, pp. 205–220.

[21] HOFFMAN, P. Minimal unauthenticated encryption (MUE) for HTTP/2. Internet-Draft draft-hoffman-httpbis-minimal-unauth-enc-01, IETF Secretariat, December 2013. `http://www.ietf.org/internet-drafts/draft-hoffman-httpbis-minimal-unauth-enc-01.txt`.

[22] HOFFMAN, P., AND SCHLYTER, J. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. Tech. rep., RFC 6698, August, 2012.

[23] HONDA, M., NISHIDA, Y., RAICIU, C., GREENHALGH, A., HANDLEY, M., AND TOKUDA, H. Is it still possible to extend TCP? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 181–194.

[24] LANGLEY, A. Transport layer security next protocol negotiation extension. Tech. rep., draft-agl-tls-nextprotoneg-04, May 2012.

[25] MÖLLER, B., AND LANGLEY, A. TLS fallback signaling cipher suite value (SCSV) for preventing protocol downgrade attacks. *InternetDraft draftietftlsdowngradescsv00* (2014).

[26] MOON, A. Optimized implementations of poly1305, a fast message-authentication-code. `https://github.com/floodyberry/poly1305-opt/`, 2015. Accessed: 2015-10-23.

[27] NOTTINGHAM, M., AND THOMSON, M. Opportunistic encryption for HTTP URIs. Internet-Draft draft-nottingham-http2-encryption-03, IETF Secretariat, May 2014. `http://www.ietf.org/internet-drafts/draft-nottingham-http2-encryption-03.txt`.

[28] PETULLO, W. M., ZHANG, X., SOLWORTH, J. A., BERNSTEIN, D. J., AND LANGE, T. MINIMALT: Minimal-latency networking through better security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 425–438.

[29] ROGAWAY, P., AND SHRIMPTON, T. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption*, B. Roy and W. Meier, Eds., vol. 3017 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 371–388.

[30] SALOWEY, J., ERONEN, P., AND TSCHOFENIG, H. Transport layer security (TLS) session resumption without server-side state. Tech. rep., RFC 5077, Jan. 2008.

[31] STEWART, R., AND METZ, C. SCTP: new transport protocol for TCP/IP. *Internet Computing, IEEE 5*, 6 (2001), 64–69.

[32] SYSOEV, I., AND JOYENT. HTTP request/response parser for c. `https://github.com/joyent/http-parser`, 2015. Accessed: 2015-02-20.

[33] TRANTER, J. Exploring the sendfile system call. *Linux Gazette 91* (2003).