HTTPCrypt: fast, opportunistic, proxyable HTTP encryption

Under double-blind review to Usenix Security 2015, Paper #307

Abstract

We introduce HTTPCrypt, a scheme of opportunistic encryption for the plain HTTP protocol designed to interoperate with the existing browser, server and proxy 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). Browser connections are secured by first establishing a TLS connection to the endpoint, and then issuing standard HTTP requests across this 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 [25] to optimise the process.

The introduction of the HTTP/2.0 standardisation process [20] – the first major HTTP protocol revision since 1999 – has brought up the possibility of supporting opportunistic encryption for *all* web browsing, 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.

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.

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

2 Protocol Overview

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 [24]. Middleboxes have hampered the adoption of many new Internet protocols; from full transport stacks such as

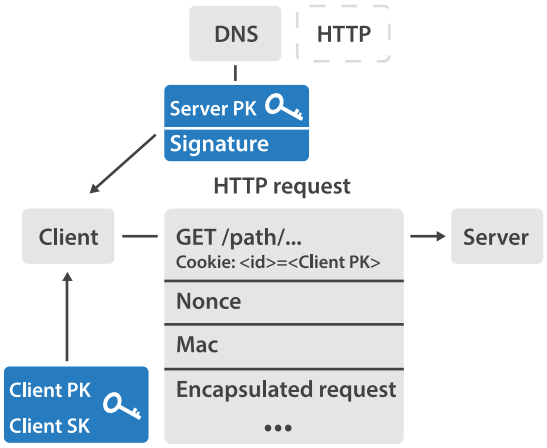


Figure 1: The overall structure of an HTTPCrypt connection. The server public key can be retrieved out-of-band via DNS, or directly over HTTP.

SCTP [34] or CurveCP [4], to extensions to existing protocols such as TCPCrypt [13] or MPTCP [3]. 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. The other 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 further protections. HTTPCrypt is thus 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 [16].

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.

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 [28].

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 Code size and embedded appliances

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 modern NaCL cryptobox primitives [6].

Cryptobox can be implemented using both a high-performance profile or as a small library within about a thousand lines of portable code. Unlike TLS, cryptobox does not rely on hardware accelerated cryptographic schemes such as AES-GCM [32], but provides a universal encryption standard that can benefit from the recent vectorised instructions in CPUs such as `AVX2` or `NEON`.

3 Architecture

We now describe describe the architecture of HTTPCrypt in detail, via the handshake procedure (§3.1), request structure (§3.2), cryptographic primitives (§3.3) and session resumption (§3.4).

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 [2] or DNSCurve [7].
- Obtain a DNS record with the current ephemeral public key with an Ed25519 [9] signature made by the long-term certificate of the server.
- Perform a plain-text `HTTP OPTIONS` request to the target HTTP server.

The Domain Name System is the preferred way to obtain the ephemeral public keys since DNS defines a clear model of trust: the domain owner is the the person of trust if we can verify the DNS reply by means of DNSSEC or DNSCurve. While both of these DNS extensions define a model of trusted anchors, only the trusted anchors of DNSSEC are currently widely deployed in the Internet. Nevertheless, DNSCurve grants greater security gain to providing confidential lookups and stronger cryptographic primitives.

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:

- DANE `TLSA` records [23] that contain the full ephemeral key and optionally the signature made by long-term certificate;
- `TXT` records can carry the keys and signatures encoded by using, for example, Base64 encoding;
- `AAAA` or `A` resource records can also encode keys and signatures that (described below)

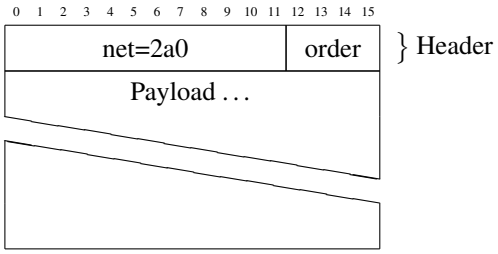


Figure 2: Key material encoding inside *AAAA* records

Restrictive DNS proxies HTTPCrypt is designed to pass through middleboxes, and so we cannot solely rely on DNSSEC or DNSCurve availability. In many networks, DNSSEC is filtered by the local recursive resolver clearing DNSSEC flags and filtering the relevant resource records. Therefore, the traditional PKI model is the only way to establish authenticity of the remote server. This model defines chains of trust organised via public key signatures: e.g., an ephemeral key is signed by server's long-term certificate that is in turn signed by some trusted third party certificate authority. The long term server's certificate is not used for key exchange and is only accessed for signing the current ephemeral public keys, minimising the risk of compromising it if a server is broken by an attacker.

A client obtains the remote long-term certificate by making a plain HTTP request (without a `Cookie` header) to the endpoint protected by HTTPCrypt. The server responds with its own public long-term certificate and all intermediate certificates in the chain.

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. Wee 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.

Deriving a session key After obtaining the server's public key, a client generates (or reuses) its own keypair and generates 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[6].

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 normally transformed by proxies. The rearrangement of chunked replies could be realistically be performed by HTTP proxies but HTTPCrypt cannot currently handle this situation with the selected cryptographic algorithms.

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, and 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))`

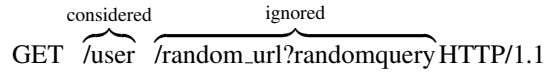
The server uses this ID when rotating ephemeral keys to select the correct key for a particular connection. Two ID collision probability is $1/2^{80}$ which is negligible when used for this purpose.

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, and so it is worth forging this header.
- **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.

HTTPCrypt uses a random alphanumeric string as the request URL and encrypts the real URL inside the payload. The unencrypted URL is ignored by a server and merely uses the path component where encryption starts. For instance, if HTTPCrypt is enabled for a `/user/` URI path, then all components after this path should be ignored by the server:



The encrypted payload has the following structure:

```
<Nonce>
<MAC>
<Encrypted HTTP request>
```

When HTTP chunked encoding is used [18] the structure changes slightly and each chunk has its own nonce and MAC tag. The chunk length includes the size of nonce and MAC. The structure of chunked HTTPCrypt request is as follows:

```
<Chunk length>
<Nonce>
<MAC>
<Encrypted chunk content>
<Next chunk>
Encrypted payload
...
<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 read the nonce, authentication tag and verify the encrypted content in the remaining data chunk. If verification succeeds then 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. If any peer receives unencrypted HTTP message then it should either ignore the message or terminate the connection.

3.3 Cryptographic primitives

HTTPCrypt encryption is based on the Cryptobox construction defined by Daniel Bernstein in the NaCL cryptographic library [6]. 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 are generated randomly and their length is chosen to be long enough to make the probability of repetition negligible. Berstein 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 $1/2^{128}$ [6].

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 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 [8].

3.4 Session Resumption

Since the generation of public key shared secrets is an expensive procedure, HTTPCrypto defines several approaches to skip it for already establishes sessions. HTTPCrypt uses the same common principles that are defined in TLS [14] by means of a session cache (§3.4.1) and ticket mechanism (§3.4.2).

3.4.1 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. Replay attacks for session renegotiation must be mitigated as described later (§4.1).

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 perfect forward secrecy can be preserved.

3.4.2 Sessions tickets

Session tickets are a mechanism for an HTTPCrypt server to support session resumption without having to store per-client state [33]. 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.

To use session tickets in HTTPCrypt, a server maintains a randomly generated symmetric key used for ticket encryption, and ensures that it is rotated regularly as with normal ephemeral public keys. To generate a ticket, the server encrypts the session key, the current pair of public keys, the random cookie of the session and the ticket maximum lifetime using a randomly chosen 24 byte nonce. The resulting ticket is passed within the encrypted payload using a special HTTP header:

```
Session-Ticket:␣<id>=<base64_ticket>
```

The ticket secret `id` is generated by taking the first 10 bytes of one-way hash function applied to the ticket's se-

cret key. This hash function must be resistant to preimage attacks to avoid key recovery [31], a property that is guaranteed by modern SHA2 and SHA3 hash functions.

To restore a session with a ticket, a client places the obtained ticket `id` and the encrypted payload in the `Cookie` header instead of the client's public key. The server interprets the `id` as a session ticket and decrypts the ticket and reuses it for the following messages within the session. When renegotiating a session using a ticket, both the server and client should establish a new random token to prevent sessions being replayed (§4.1). Storing of the previous random cookie inside session tickets helps to avoid replaying the first request when renegotiating.

4 Security Analysis

We now elaborate on the HTTPCrypt protocol's security properties (§4.1), how to cloak requests within normal HTTP traffic operates (§4.2), support for perfect forward secrecy (§4.3) and denial of service resistance (§4.4). 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 Ed25519 signatures [9] 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. In this case, the following session is protected from being replayed.

The server places the random cookie in the encrypted and authenticated payload, 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:

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

A server, in turn, generates the full authentication token addition by appending its own random cookie and pushes it inside the encrypted header:

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

The random strings should be long enough to make the probability of their repetition negligible.

Afterwards, all subsequent requests in the session include this random header to validate the HTTPCrypt session. In conjunction with the counter nonces policy described next (§4.1.2), this model provides effective replay attacksprotection. 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.

4.1.2 Nonce Counters

Nonces must never be repeated for subsequent requests during a single session's lifetime. This requires that both peers must maintain a list of nonces seen in the session and drop messages with repeated nonces. Maintaining this list requires additional memory and limits the number of requests inside a single keep-alive HTTP session. This limitation is not pressing in the vast majority of connections since a single session is usually limited to less than a hundred requests [1].

For longer sessions such as Websockets or long-lived HTTP connections via JavaScript, the nonce policy can be switched to a counter model where all nonces (after the first randomly generated one) are monotonically inreasing . In the case of a 24 byte nonce, the probability of counter repetition is negligible. The nonce policy can be set by means of an encrypted HTTP header sent by a server:

```
Nonce-Policy:␣counter
```

4.2 HTTPCrypt Request Cloaking

An HTTPCrypt request has a similar structure to a plain HTTP message and consists of the unencrypted HTTP request, and the real HTTP request encapsulated within an encrypted construction. However, a public key that is placed inside HTTP `Cookie` header typically represents some point on the appropriate elliptic curve for the cryptographic cipher in use. An adversary can thus easily distinguish this cookie from random data.

Cookie Hiding The Elligator type 2 construction [10] is used to cloak the presence of a valid HTTPCrypt cookie. This converts a point on an elliptic curve to a corresponding uniform hash (or almost uniform if computational resources are scarce). An adversary can still try to revert the Elligator algorithm and extract the public key, but this now requires a computationally expensive traffic analysis of all HTTP cookies in all requests.

Note that we do not aim to make it stenographically difficult to detect HTTPCrypt requests by middleboxes in the network path that can monitor the whole HTTP session and check HTTP headers and peer behaviour precisely. Instead, we aim to make large-scale passive monitoring to identify HTTPCrypt *vs* normal HTTP traffic to be expensive enough to be impractical to perform as a passive global adversary. Detecting HTTPCrypt requires deep packet inspection of HTTP traffic, which requires significantly more resources than, for example, the case of HTTP+TLS where anyone can detect HTTPS connection by capturing just a single handshake.

Metadata Exposure The major disadvantage of HTTP encapsulation is that the request lengths and boundaries can be observed clearly. For example, when HTTPCrypt is used to encrypt large file transfers, an adversary can capture the boundaries of the files, calculate their sizes and figure out what files are being downloaded. This sort of data leakage is also applicable to TLS connections, but in that case an adversary can only estimate timing of TLS messages to find the boundaries of messages. HTTPCrypt is weaker since the information about boundaries is leaked in clear-text. However, it is still possible to hide the real requests sizes and boundaries.

First, a server can add random and unpredictable padding to the end of each encrypted request. The real payload length is controlled by the `Content-Length` header that is hidden inside the encrypted part of the request, which an adversary cannot observe. This method adds considerable traffic bloat if large files are involved. In such a case, the server needs to make all replies to be of a similar size while the real content length might differ by several orders of magnitude. For other applications where all replies have almost equal size, the random padding method has low overhead and prevents an adversary from distinguishing replies solely by their sizes.

Another approach is to take advantage of range requests in the HTTP protocol, and split up the large request in multiple equal-sized chunks that are reassembled by the client. In this case, another peer needs to send HTTP requests to obtain the remaining portions of data, or using a P2P protocol such as Bittorrent that is specifically designed for this mode of operation. This can be achieved by placing the following HTTP header within the encrypted payload:

```
Remain-Length:␣<remaining_chunks_size>
Content-Length:␣<this_chunk_size>
```

To get the full reply, the peer concatenates the chunks unless Remain-Length − Content-Length = 0. More sophisticated schemes are also possible based on ranged requests. This approach imposes the overhead of extra HTTP requests that are required for continuation of the session. However, when transferring large files, this overhead is not significant comparing to the cost of bulk data transfer and encryption.

Combining these two methods allows for hiding the real request boundaries and cloaking the overall payload size transferred. Moreover, if just a single large file is transferred over HTTPCrypt, the combination of random padding and request splitting hides more information than when using a standard TLS+HTTP connection for the same purpose.

4.3 Perfect Forward Secrecy

HTTPCrypt uses slowly rotating ephemeral public keys for HTTP servers to provide perfect forward secrecy [15]. 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.

4.4 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 large 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 [30] 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

The original Cryptobox construction [6] defines `xsalsa20` as the stream cipher. However, the `salsa` family of ciphers is superseded by the `chacha` stream cipher [5]. We evaluated the performance of bulk data transfer speeds achieved by different combinations of stream ciphers and corresponding authentication algorithms to select the most appropriate set of cryptographic elements for HTTPCrypt. In particular, we compared authenticated encryption constructions based on `chacha20` using the `poly1305` authenticator, and AES ciphers in counter mode with the GMAC authenticator.

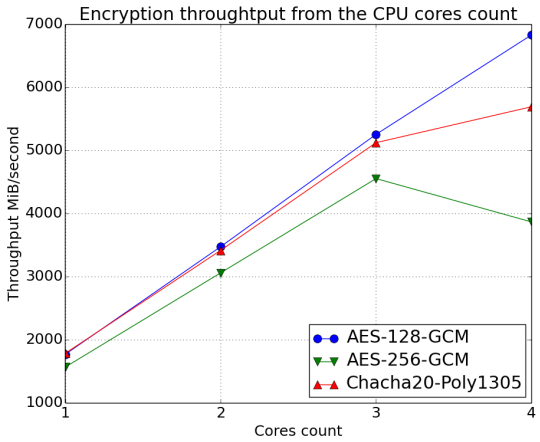


Figure 3: Authentication encryption bulk performance on Intel Core i7-4770K CPU @ 3.50GHz using large data chunks

The first CPU tested in Figure 3 (an Intel Core I7-4770K) supports both hardware AES acceleration and high performance vector operations via the AVX2 instructions set. In our benchmarks, we used both hardware AES and vectorisation for `ChaCha20-Poly1305`. In our tests, Chacha20 was significantly faster than an AES-256-GCM cipher with a comparable 256-bit key length.

The second CPU tested in Figure 4 (an Intel Core2 Quad-Core Q6600) does not support hardware AES and instead supports the SS3 vector operations. For this relatively old CPU, `ChaCha20-Poly1305` was more than twice as fast than comparable AES

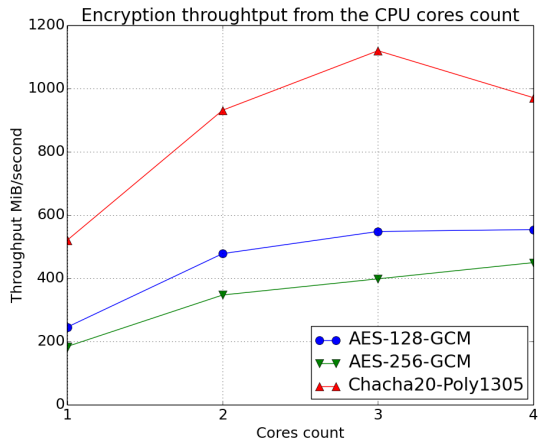


Figure 4: Authentication encryption bulk performance on Intel Core2 Quad CPU Q6600 @ 2.40GHz using large data chunks

ciphers. This motivates our final choice to use `ChaCha20-Poly1305`authenticated encryption construction in HTTPCrypt for bulk encryption.

The speed of key exchange is critically important for the overall performance of opportunistic encryption. In particular, key exchange is a bottleneck for establishing many short-lived HTTP connections. In order to find the best implementation to use in HTTPCrypt, we surveyed the results of the SUPERCOP[1] challenge [11]. We selected the implementation of `curve25519` by Adam Langley [26] that is optimised for using on 64-bit platforms. There are several variants suitable for 32-bit platforms based on SSE2 or NEON extensions, but not evaluated in this paper.

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` [36] 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.

HTTPCrypt can use the semantic of `sendfile` to send files encrypted without requiring copying through userspace. The `sendfile` interface needs to be extended to accept a session key and a generated nonce. Using this information, `sendfile` can encrypt and authenticate the subsequent HTTP payload chunks. In contrast, TLS encryption is far more complex to integrate with zero-copy operations in general due to the protocol complexity. Alerts, fragmentation and protocol extensions all require complex processing that is not easy (or wise) to put into the kernel.

5.3 Integration with Existing Software

HTTPCrypt is designed to be integrated with existing application easily. While it is relatively straightforward to migrate existing plaintext services to TLS by means of a proxy such as `stud`[2], it is more difficult to integrate it directly into an application not designed for it 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. 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 5 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 nonce and authentication tag from the encrypted payload, and parsing the following encapsulated HTTP request as defined earlier (§3.2).

To migrate to HTTPCrypt from plaintext HTTP, applications must also use a cryptographic quality random number generator. This is particularly important where there is not much entropy available, for example in embedded devices [21] or virtual machines [17]. To deal with this issue, modern operating systems provide fast pseudo-random numbers generators, for example, `arc4random` in BSD systems or `getentropy` call in Linux. On older operating systems, user space cryptographic random numbers generators can be used. Some of them, for example the `libottery` [27] library, can even utilize the same `ChaCha20` algorithm as the pseudo-random function.

[1] System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives

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

```
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 5: Sample code fragment illustrating HTTPCrypt request creation and writing to a socket in C

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 [12]. 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.

6 Evaluation

We have built the prototype of HTTPCrypt built on top of the `http-parser` library [35] that is in turn based on the popular Nginx HTTP server code. The goal of our tests was to estimate the influence of HTTPCrypt being enabled on the requests rate and connection latency for standard web workloads using TLS. We have compared our implementation against Nginx 1.7.1 built with OpenSSL 1.0.1f using `nistp256` curves (with the `int128` optimisations enabled) for both permanent and ephemeral key pairs for both ECDSA and ECDHE.

We used the following configuration for the Nginx benchmarks:

```
ssl_ciphers "ECDHE+ECDSA+AESGCM";
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` [19] HTTP benchmarking utility with a single testing thread and 10 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 HTTP keep-alive connections and 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 confirm the performance of specific algorithms used in the test runs:

```
256 bit ecdh (nistp256):     5391.8 op/s
256 bit ecdsa(nistp256):  9483.1 signs/s
aes-128-gcm:             1301785.26 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 loopback interface to exclude network influence, and all requests were successfully processed.

6.1 Performance Evaluation

Figure 6, shows the number of requests per second of the experimental runs with Nginx, and Figure 7 shows the same workload patterns obtained from the HTTPCrypt test suite.

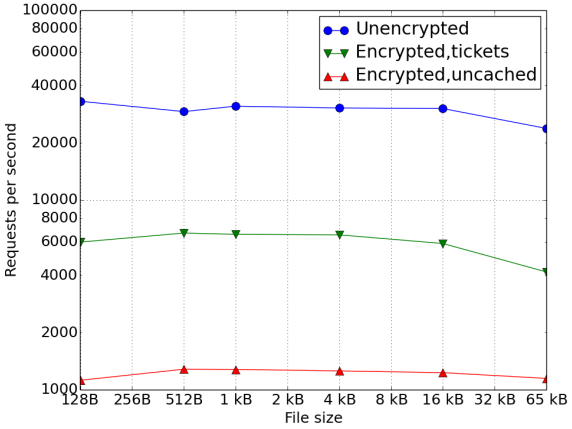


Figure 6: The performance of Nginx while serving static files using Intel Xeon E3 3.4 GHz

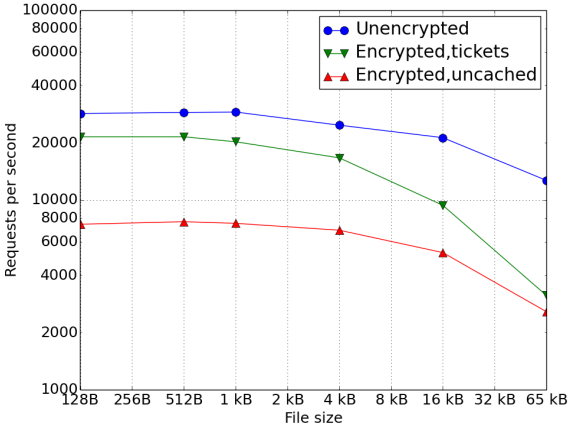


Figure 7: The performance of HTTPCrypt prototype while serving files using Intel Xeon E3 3.4 GHz

The request throughput of the HTTPCrypt test suite for the case of plain requests is significantly lower than the Nginx results as the file size increases. This is a consequence of inefficient IO operations executed by HTTPCrypt test suite: it reads the whole file into memory and writes the resulting reply within a single *writev* call. In contrast, Nginx uses the `sendfile` system call and switches to chunked encoding for large replies, allowing it to improve both unencrypted and encrypted throughput for large files.

However, 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 (e.g. 20000 requests per second *vs* 6000 in case of 1KB requests). 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. As shown in the Figure 3 the difference between bulk encryption performance between algorithms used in HTTPCrypt and TLS, namely `ChaCha20-Poly1305` and `AES-128-GCM`, is negligible.

The performance in the encrypted tests degraded with large files due to limitations of the IO implementation in our HTTPCrypt prototype, which does not yet take full advantage of the `sendfile` system call and asynchronous IO. This is an area we are improving in the next revision of our HTTPCrypt library.

6.2 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).

The results are shown in Figure 8 for Nginx and Figure 9 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 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 §4.4 that this difference does not make HTTPCrypt more vulnerable to denial-of-service attacks than TLS.

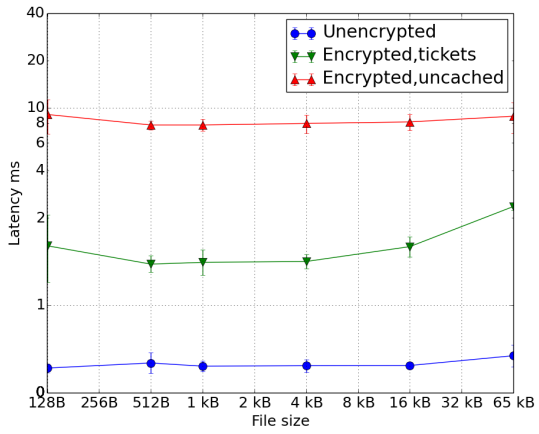


Figure 8: The request latency for Nginx while serving files using an Intel Xeon E3 3.4 GHz

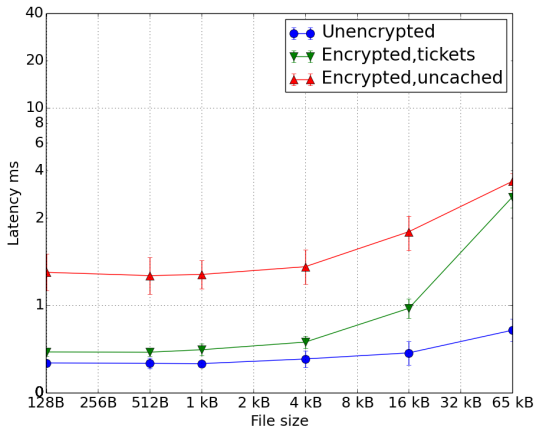


Figure 9: The request latency for the HTTPCrypt prototype while serving files using an Intel Xeon E3 3.4 GHz

7 Related work

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

7.1 Transport protocols

TCPCrypt TCPCrypt [13] 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 is significantly different from ordinary TCP and can hence be easily filtered (either deliberately or frequently accidentally due to protocol scrubbing).

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..

CurveCP CurveCP is a protocol designed by Daniel Bernstein as a complete replacement for the TCP protocol [4] with the following properties:

- Authentication and encryption of all payload
- Explicit peer authorization
- Own congestion control
- Replay attacks protection

CurveCP is also based on the NaCL cryptobox construction and defines a very fast key exchange procedure (namely, the same `curve25519` ECDH). CurveCP can be used to protect plain HTTP, but it requires that HTTP traffic be transported over the CurveCP UDP transport rather than conventional TCP. This is problematic from a deployment perspective.

For peer authentication, CurveCP uses long-term key encryption instead of signatures. This implies that the server has access to both the ephemeral and long-term keys, while in HTTPCrypt, there is no need for a server to have access to its long-term key. CurveCP suggest that static key material be stored in the DNS and its confidentiality protected by DNSCurve.

CurveCP is a good choice to protect plain HTTP opportunistically, since it is low latency (2 RTT), high performance and a large security improvement. However, applications have to be significantly changed to adopt CurveCP, and network middleboxes make avoid filtering difficult.

MinimalT MinimalT [30] 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.

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

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 [22, 29]. 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 [22], 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 [29] 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; for example, no additional handshake stages are needed that add complexity to existing web application stacks.

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

8 Conclusions

In this paper, we have proposed and evaluated HTTPCrypt – an opportunistic HTTP encryption scheme 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. In order to minimise middlebox meddling, HTTPCrypto is difficult to distinguish from plain HTTP traffic (§4.2). 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 and throughput for different payload types. The test results (§6) shows that HTTPCrypt provides significantly better performance and lower request latency than traditional HTTP/TLS due to its reliance on more efficient cryptographic primitives and 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 (§4.4). The source code to our implementation is open-source under a BSD license and available from *blinded for review*.

References

[1] AL-FARES, M., ELMELEEGY, K., REED, B., AND GASHINSKY, I. Overclocking the yahoo!: Cdn for faster web page loads. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 569–584.

[2] 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.

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

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

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

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

[7] BERNSTEIN, D. J. Dnscurve: Usable security for dns, 2009.

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

[9] 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.

[10] BERNSTEIN, D. J., HAMBURG, M., KRASNOVA, A., AND LANGE, T. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 967–980.

[11] BERNSTEIN, D. J., AND LANGE, T. ebacs: Ecrypt benchmarking of cryptographic systems, 2009.

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

[13] 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.

[14] DIERKS, T. The transport layer security (tls) protocol version 1.2.

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

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

[17] 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.

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

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

[20] GROUP, I. H. W., ET AL. Http 2.0 specifications, 2013.

[21] 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.

[22] 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`.

[23] 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.

[24] 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.

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

[26] LANGLEY, A. Implementations of a fast elliptic-curve diffie-hellman primitive. `https://github.com/agl/curve25519-donna`, 2015. Accessed: 2015-02-20.

[27] MATHEWSON, N. A fast secure userspace pseudorandom number generator. `https://github.com/nmathewson/libottery`, 2015. Accessed: 2015-02-20.

[28] MÖLLER, B., AND LANGLEY, A. Tls fallback signaling cipher suite value (scsv) for preventing protocol downgrade attacks. *InternetDraft draftietftls-downgradescsv00* (2014).

[29] 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`.

[30] 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.

[31] 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.

[32] SALOWEY, J., CHOUDHURY, A., AND MCGREW, D. AES galois counter mode (GCM) cipher suites for TLS. Tech. rep., RFC 5288 (Proposed Standard), 2008.

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

[34] STEWART, R., AND METZ, C. Sctp: new transport protocol for tcp/ip. *Internet Computing, IEEE 5*, 6 (2001), 64–69.

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

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