Authenticated Network Time Synchronization

Benjamin Dowling
Queensland University of Technology
b1.dowling@qut.edu.au

Douglas Stebila
Queensland University of Technology
stebila@qut.edu.au

Greg Zaverucha
Microsoft Research
gregz@microsoft.com

Abstract

The Network Time Protocol (NTP) is used by many network-connected devices to synchronize device time with remote servers. Many security features depend on the device knowing the current time, for example in deciding whether a certificate is still valid. Currently, most services implement NTP without authentication, and the authentication mechanisms available in the standard have not been formally analyzed, require a pre-shared key, or are known to have cryptographic weaknesses. In this paper we design an authenticated version of NTP, called ANTP, a generic construction which protects against desynchronization attacks. To make ANTP suitable for large-scale deployments, it is designed to minimize server-side public-key operations and requires no server-side state. Additionally, ANTP ensures that authenticity does not degrade accuracy. Authentication adds no latency to server responses, by using the fact that authentication information can arrive in a separate, subsequent message. We define a novel provable security framework involving adversary control of time, and use the framework to analyze ANTP. The framework may also be used to analyze other secure time synchronization protocols.

Keywords: time-synchronization, Network Time Protocol (NTP), provable security, network security

1 Introduction

The Network Time Protocol (NTP) is one of the Internet's oldest protocols dating back to RFC 958 [17], published in 1985. In the simplest NTP deployment, a client device sends a single UDP packet to a server (the request), who responds with a single packet containing the time (the response). The response contains the time the request was received by the server, as well as the time the response was sent, allowing the client to estimate the network delay and set their clock. If the network delay is *symmetric*, i.e., the travel time of the request and response are equal, then the protocol is perfectly accurate. *Accuracy* means that the client correctly synchronizes its clock with the server (regardless of whether the server clock is accurate in the traditional sense, e.g.,, synchronized with UTC).

The importance of accurate time for security. There are many examples of security mechanisms which often implicitly rely on having an accurate clock:

- *Certificate validation in TLS and other protocols.* Validating a public key certificate requires confirming that the current time is within the certificate's validity period. Performing validation with a slow or inaccurate clock may cause expired certificates to be accepted as valid. A revoked certificate may also validate if the clock is slow, since the relying party will not check for updated revocation information.
- *Ticket verification in Kerberos.* In Kerberos, authentication tickets have a validity period, and proper verification requires an accurate clock, to prevent authentication with an expired ticket.
- *HTTP Strict Transport Security (HSTS) policy duration.* HSTS [10] allows website administrators to protect against downgrade attacks from HTTPS to HTTP by sending a header to browsers indicating that HTTPS must be used instead of HTTP. HSTS policies specify the duration of time that HTTPS must be used. If the browser's clock jumps ahead, the policy may expire re-allowing downgrade attacks.

For clients who set their clocks using NTP, these security mechanisms and more can be attacked by a network-level attacker who can intercept and modify NTP traffic, such as a malicious wireless access point or an insider at an ISP. In practice, most NTP servers do not authenticate themselves to clients, so a network attacker can intercept responses and set the timestamps arbitrarily. Even if the client sends requests to multiple servers, these may all be intercepted and modified to present a consistently incorrect time to a victim. Such an attack on HSTS was demonstrated by Selvi [32], who provided a tool to advance the

clock of victims in order to expire HSTS policies.

In terms of security, a time-synchronization protocol must authenticate server responses to the client, even in the presence of an active network adversary. Confidentiality is not a concern (thus forward secrecy is not an issue), since all time-synchronization information is public.

NTP security today. Early versions of NTP (NTP, NTPv1 and NTPv2) had no standardized authentication method. NTPv3 added an authentication method using pre-shared key symmetric cryptography. An extension field in the NTP packet added a cryptographic checksum, computed over the packet. NTPv3 negotiation of keys and algorithms must be done out-of-band. For example, NIST offers a secure time server, and (symmetric) keys are transported from server to client by postal mail [23]. Establishing pre-shared symmetric keys with billions of client PCs and mobile phones seems impractical. NTPv4 introduced a public-key authentication mechanism called Autokey, that did not see widespread adoption; Autokey uses small 32-bit seeds that can be easily brute forced to then forge packets. A more recent proposal is the Network Time Security (NTS) protocol [33], which does not have zero-cryptographic latency and as we show in Section 2.3 is potentially vulnerable to a downgrade attack.

Most NTP servers do not support NTP authentication, and NTP clients in desktop and laptop operating systems will set their clocks based on unauthenticated NTP responses. On Linux and OS X, by default the client either polls a server periodically, or creates an NTP request when the network interface is established. In both cases the system clock will be set to any time specified by the NTP response. On Windows, by default clients will synchronize their clock weekly (using `time.microsoft.com`), and ignore responses that would change the clock by more than 15 hours. These two defaults reduce the opportunity for a man-in-the-middle (MITM) attacker to change a victim clock, and the amount by which it may be changed. In Windows domains (a network of computers, often in an enterprise), the domain controller provides the time with an authenticated variant of NTPv3 [16].

1.1 Contributions

This paper presents ANTP, a protocol for authenticated time-synchronization. ANTP protocol messages are transported in the extension fields of NTP messages. ANTP allows a server to authenticate itself to a client, and provides cryptographic assurance that no modification of the packets has occurred in transit. ANTP uses public-key encryption for key exchange, and certificates to authenticate the server public key. Like other authenticated time synchronization protocols using public keys, we assume an out-of-band method for certificate validation exists, as certificate validation requires an accurate clock.

Protocol	**Auth. type**	**Security**	**Server operations per time sync.**
NTPv0–v2	—	—	—
NTPv3 sym. key	sym. key	no proof	1 hash
NTPv4 Autokey	pub. key	multiple flaws (§2.3)	$\frac{2}{n}$ pub. key, $\frac{1}{n}+1$ sym. key
NTS [33]	pub. key	downgrade attack (§2.3)	$\frac{3}{n}$ pub. key, $\frac{2}{n}+2$ sym. key
ANTP (Fig. 2)	pub. key	proof (§5)	$\frac{1}{n}$ pub. key, $\frac{6}{n}+2$ sym. key

Table 1: Comparison of time synchronization protocols. $\frac{a}{n}+b$ denotes a operations that can be amortized over n time synchronizations plus b operations per time sync.

ANTP performance. Performance constraints on time-synchronization protocols are driven by the fact that time servers are heavily loaded, and must provide responses promptly. To make ANTP suitable for large deployments the design meets the following performance goals. First, the server does not need to keep state for each client but instead encrypts a small amount of state and stores it with each client. Second, to generate a shared secret key and authenticate the connection, the server must only do a single public-key operation per client. Once a shared secret key is negotiated, it can be used for multiple time-synchronization attempts by the same client, amortizing the cost of negotiation over multiple time-synchronizations. For example, clients may re-negotiate a new key monthly, but synchronize daily. The cost of the daily synchronizations is only a few symmetric-key operations more expensive than unauthenticated NTP. A comparison of the number of operations for ANTP and existing time synchronization protocols appears in Table 1.

Additionally, ANTP supports delayed authentication, preventing the added latency of cryptographic computations from degrading accuracy. This works by having the server respond immediately with an unauthenticated response, then replying shortly after with authentication information for the response. The client measures the roundtrip time based on the unauthenticated response, but does not update its clock until authenticating the response. We refer to this throughout the paper as the "no-cryptographic-latency" feature.

ANTP security. We include a thorough analysis of the cryptographic security of ANTP using the provable security paradigm. To do so, we extend existing frameworks for key-exchange and secure channels [1, 11] to handle protocols where *time* plays a central role. The adversary in our security analysis is a network attacker capable of deleting, reordering and editing messages between parties. In addition, the adversary is given complete control over

the initialization of all clocks, as well as the ability to increment the time of parties not involved in a protocol run. This allows us to model the ability of an adversary to delay packet transmission: this is particularly important in the case of NTP, where delaying packets asymmetrically can cause the client to synchronize to an inaccurate time. Our security model also considers the "amortized public key operations" of ANTP where the same session is used for multiple time-synchronizations.

We then show that ANTP satisfies these properties, by making standard assumptions about the primitives used to construct the protocol (public-key encryption, a hash function, authenticated encryption, a message authentication code and a key-derivation function). We use game-hopping proof techniques commonly used in formal security treatments of protocols such as TLS and SSH [11, 2, 13] in order to prove authentication for ANTP, and thus time-synchronization security.

Client		**Server**
$t_1 \leftarrow \mathsf{Now}()$		
$\mathtt{req} \leftarrow t_1$	$\xrightarrow{\mathtt{req}}$	$t_2 \leftarrow \mathsf{Now}()$
		$\vdots$
		$t_3 \leftarrow \mathsf{Now}()$
$t_4 \leftarrow \mathsf{Now}()$	$\xleftarrow{\mathtt{resp}}$	$\mathtt{resp} \leftarrow t_2 \Vert t_3$
$RTT \leftarrow (t_4 - t_1) - (t_3 - t_2)$		
$\tilde{\theta}_3 \leftarrow RTT/2$		
$\mathit{offset} \leftarrow \frac{1}{2}(t_2 + t_3 - t_1 - t_4)$		
$\mathit{time} \leftarrow \mathsf{Now}() + \mathit{offset}$		

Figure 1: Simple Network Time Protocol (SNTP). $\mathsf{Now}()$ denotes the procedure that outputs the local machine's current time. RTT denotes the total round-trip delay the client observes and $\tilde{\theta}_3$ denotes the approximation of the propagation time from server to client. The time of the server receiving `req` is denoted t_2 and sending `resp` is t_3. Note that $\mathit{offset} = t_3 + \tilde{\theta}_3 - t_4$, which we will use in our correctness analysis of ANTP.

2 Existing Network Time Protocols

Here we review the two most commonly deployed time-synchronization protocols, NTP and SNTP. We also review the security of time protocols, and discuss a recent proposal called Network Time Security [33].

2.1 The Network Time Protocol

The Network Time Protocol (**NTP**) was developed by Mills in 1985 [17], and revised in 1988, 1989, 1992 and 2010 (NTPv1 [9], NTPv2 [26], NTPv3 [18] and NTPv4 respectively [19]). NTP is designed to synchronize the clocks of a subnet of machines directly connected to hardware clocks (known as *primary servers*) to a network of machines without hardware clocks (known as secondary servers). NTP protects against Byzantine traitors by querying multiple servers, selecting a majority clique and updating the local clock with the majority offset. This assumes that the attacker can only influence some minority subset of the queried servers – an assumption that may not be realistic. The clock update procedure and calculations for determining a single server's offset is found in Figure 1 and the NTP packet format can be found in Appendix A. Though the format for NTP packets are identical for both client and server NTP messages, we use `req` to indicate a NTP packet in client mode, and `resp` to indicate a NTP packet in server mode.

2.2 The Simple Network Time Protocol

The Simple Network time Protocol (**SNTP**) is a variant of NTP that uses an identical message format but only queries a single server when requesting time-synchronization. Windows and OSX by default synchronize using a single time source (`time.windows.com` and `time.apple.com` respectively). Our construction lends itself well to SNTP, as it authenticates time samples from a single server. Security analysis is also easier as we can avoid the more complex sorting and filtering algorithms of NTP, and client and server behaviors are simpler. Note that SNTP and NTP client request messages are the same.

SNTP has three distinct stages: the creation and transmission of `req` by the client; the processing of `req` by the server, and transmission of `resp`; and the processing of `resp` and clock update by the client. An abstraction of the protocol behavior can be found in Figure 1 along with the clock update procedure.

1. The client creates a SNTP `req` packet, sets `transmit_timestamp` to $\mathsf{Now}()$ and sends the message.
2. The SNTP Server creates a `resp` with all fields identical to the received `req`, but signaling Server mode. The server then sets `originate_timestamp` with the value `transmit_timestamp` from `req`. The server will also set `receive_timestamp` to $\mathsf{Now}()$ when receiving `req`, and will set the `transmit_timestamp` to $\mathsf{Now}()$ when sending the message to the client.
3. Upon receiving `resp`, the client notes the current time (and saves it as t_4), and if `resp.originate_timestamp` is not equal to `req.transmit_timestamp`, the client aborts the protocol run. The client sets t_1, t_2 and t_3 to the values `originate_timestamp`, `receive_timestamp` and `transmit_timestamp` from `resp`, respectively. Then the client calculates the total round-trip time RTT and the local clock offset offset as above.

From this, we can compute a bound of the amount of error that is introduced to the clock update procedure via asymmetric packet delay when the packets are unmodified. Asymmetric packet delay is the scenario where the propagation time from client to server is not equal to the propagation time from server to client. Let θ_1 be the propagation time from client to server, θ_2 the server processing time and θ_3 the propagation time from server to client. θ_3 is approximated in SNTP is approximated by $\tilde{\theta_3} = \frac{RTT}{2}$, where $RTT = (t_4 - t_1) - (t_3 - t_2) = \theta_1 + \theta_3$.

The actual offset is $offset_{actual} = t_3 + \theta_3 - t_4$. The approximated offset is computed by $offset = \frac{1}{2}(t_2 + t_3 - t_1 - t_4)$. It is straightforward to show that when $\theta_1 = \theta_3$, then $offset = t_3 + \tilde{\theta_3} - t_4$ and $offset = offset_{actual}$. In the worst possible case, packet delivery is instantaneous, and the entire roundtrip time is asymmetric delay. The client approximates the offset as above, and thus the error introduced this way is $\frac{1}{2}|(\theta_1 - \theta_3)| \leq RTT$.

Thus in SNTP, the error that a passive adversary with the ability to delay packets is able to introduce does not exceed the RTT. Clients can abort the protocol run when RTT grows too large, giving them some control over the worst-case error. Cryptographically speaking, the rest of the fields in the NTP packets (see Appendix A) is irrelevant for calculating the local clock offset and correcting the local clock for a single-source time-synchronization protocol. These extra fields in the NTP packet are used primarily for ranking multiple distinct time sources.

2.3 NTP Security and Other Related Work

In terms of security, early versions of NTP (NTP to NTPv2) had no standardized authentication method.

NTPv3 symmetric-key authentication. NTPv3 presented a method for authenticating time synchronization – using pre-shared key symmetric cryptography. NTPv3's added additional extension fields to the NTP packet, consisting of a 32-bit key identifier, and a 64-bit cryptographic checksum. The distribution of keys and negotiation of algorithms was considered outside the scope of NTP. NTPv4 introduced a method for using public-key cryptography for authentication, known as the Autokey protocol. Autokey is designed to prevent inaccurate time-synchronization by authenticating the server to the client, and verifying no modification of the packet has occurred in transit. NTPv4 has additional extension fields to be used in the Autokey protocol, and works with the key identifier and checksum extension fields added in NTPv3.

NTPv4 Autokey public key authentication. Autokey uses MD5 and a variety of Schnorr-like [30] identification schemes to prevent malicious attacks, but as an analysis of Autokey by Röttger shows [27], there are multiple weaknesses inherent in the Autokey protocol, including use of small seed values (32 bits) and allowing insecure identification schemes to be negotiated. The size of the seed allows a MITM adversary with sufficient computational power to generate all possible seed values and use the cookie to authenticate adversarial-chosen NTP packets. This weakness alone allows an attacker in control of the network to break authentication of time-synchronization, thus NTP with the Autokey protocol is not a secure time-synchronization protocol. Mills describes his experiments on demonstrating reliability and accuracy of network time-synchronization using NTPv2 implementations [20], but does not offer a formal security analysis of NTP. Mills does show that honest deployment of NTP in networks can offer time-synchronization accuracy to within a few tens of milliseconds after only a few synchronizations. ANTP was originally intended as a means to addressing the vulnerabilities in the Autokey protocol, but with many changes to minimize public-key and symmetric-key operations, message bandwidth. While inspiration for ANTP is the Autokey protocol, the design diverged significantly enough to consider it a separate protocol design.

Network Time Security. The Network Time Security protocol (NTS) [33] is an alternative security protocol that uses public-key infrastructure in order to secure time-synchronization protocols such as NTP and the Precise-Time Protocol (PTP). However, NTS is costly in terms of server-side public-key operations, potentially vulnerable to downgrade attacks in the negotiation phase, and does not have an equivalent to the zero-cryptographic latency feature of ANTP. NTS has had no formal security treatment at the time of this writing.

NTS initially inherited many design choices from the Autokey protocol, in particular protocol flow, but has since been updated as a three round-trip protocol. Similarly to the Autokey protocol, NTS servers reuse the randomness *server_seed* used to generate a shared secret key (referred to as a *cookie*) for each client by $cookie = \mathsf{HMAC}(server\_seed, \mathsf{Hash}(\text{client public-key certificate}))$, encrypting this value and a client-chosen nonce with the client public-key, authenticating the server by digitally signing the *cookie* with the server private key. Note that the client public-key certificate in NTS serves to protect the confidentiality and ensures uniqueness of the *cookie* for each client using a different public-key certificate. It does not serve to authenticate the client to the server. In ANTP clients do not need a certificate, only the server.

In addition, NTS requires the server send (in the `server_assoc` message) a signature over the other data in the message. The information in the `server_assoc` is static for each client, except the negotiated hash and encryption algorithms (a finite combination of algorithms). Then a NTS server must either store digital signatures for each combination of supported hash and public-key encryption algorithms, or compute costly public-key op-

erations over these values. NTS also requires a server encrypt the *cookie* value with the client public-key and sign this value with the server private-key. As a result, a NTS server requires two to three public-key operations per client to establish a shared secret *cookie*.

NTS does not authenticate all information in the `client_assoc` message, specifically the list of client-supported hash and encryption algorithms. Since the signature in the `server_assoc` is not computed over the client supplied lists in the `client_assoc` message, a MITM attacker could alter the `client_assoc` to force a server to select sub-optimal combinations of hash and encryption algorithms, which would be undetectable to the client.

Finally we note that NTS is a work-in-progress and a future revision may be udpated to address some of these issues. We reviewed draft version `-06` for this paper, and communicated our findings to the authors.

Provable Security. There is a small amount of literature analyzing time-synchronization protocols and security frameworks implementing time. Schwenk recently proposed a framework for the purposes of modeling time in provable security analysis, very similar to our own work but he models time as a global parameter [31]. Each party may query a *time oracle* $\mathcal{T}$ to receive the time from the global time counter. The security framework is subsequently used to analyze one round-trip cryptographic protocols, such as a one-time-password protocol, and a Kerberos-like protocol. In IETF RFC 7384 [21], Mizrahi discusses security properties time-synchronization protocols such as NTP and ANTP should achieve, in particular categorizing attackers with respect to the position of the attacker (internal or external) and ability (MITM, Packet Injector). Mizrahi also ranks security threats, and categorizes based on impact of threat and type of attacker.

3 Authenticated Network Time Protocol

In this section we present the *Authenticated Network Time Protocol (ANTP)*: a new variant of NTP designed to allow an SNTP client to authenticate a single NTP server and output a time counter within some accuracy margin of the server time counter. Our new protocol ANTP allows an ANTP server to authenticate itself to an ANTP client, as well as provide cryptographic assurances that no modification of the packets has occurred in transit. ANTP messages, much like Autokey and NTS, are included in the extension fields of NTP messages. We summarize the novel features of ANTP below:

- The client is capable of authenticating the server, and all messages from the server. Replay attacks are explicitly prevented.
- The server does not need to keep state for each client.
- The server does only one public-key operation per client in order to generate a shared secret key.
- The shared secret key can be used for multiple time-synchronization attempts by the same client.
- The client has a "no-cryptographic-latency" option to avoid additional error in the approximation of $\tilde{\theta}_3$ due to cryptographic operations.

ANTP is divided into four separate phases. A detailed protocol flow can be found in Figure 2.

- *Setup:* The server chooses a long term key s the authenticated encryption algorithm. This is used to encrypt and decrypt server state between phases.
- *Negotiation Phase*: The client and server communicate supported algorithms, and the server sends his certificate, and C_1, an encryption of the hash of the message flow (using s). The value C_1 will be used to authenticate negotiation later in the protocol.
- *Key-Exchange Phase*: The client encrypts a premaster secret with the server public-key and sends the ciphertext and C_1 to the server. The server derives the shared key k , then encrypts it with s. Call this value C_2. The server replies with a MAC (for key confirmation) and C_2 (for use in the next phase).
- *Time-Synchronization Phase*: The client requests a synchronization and sends C_2. The server recovers k from C_2 and uses it to derive a fresh key to authenticate the response. The client can also request a high-accuracy time-synchronization, where the server will immediately reply without authentication, and then send a second message with authentication.

Appendices B and D contain excerpts of the full ANTP specification [6], and detail the ANTP message formats and cryptographic algorithms.

3.1 Design Rationale and Discussion

In SNTP, the accuracy is bounded by the total roundtrip time of the time-synchronization phase. If we build a secure authentication protocol over SNTP, then the total accuracy of the new authenticated protocol is also bound by the total round-trip time of the time-synchronization phase, assuming secure authentication. Note that a client must not pull synchronization from any NTP packets that have not been authenticated via ANTP. Doing so would allow downgrade attacks from ANTP to NTP and lose all security benefits of ANTP.

Of the security properties discussed in RFC 7384 [21], ANTP achieves the following: *protection against manipulation, spoofing, replay and delay attacks; authentication of the server* (if ANTP is applied in a chain, implicit authentication of primary server); *key freshness; avoids degradation time-synchronization; minimizes computational load; minimizes per-client storage requirements of the server*. The following properties from [21] are

Client		**Server**
supported algorithms $\vec{algs}_C$		supported algorithms $\vec{algs}_S$ long-term secret s certificate $cert_S$ for the PKE keypair (pk_S, sk_S)
	Negotiation phase	
$\alpha \leftarrow$ `in-progress` $n_c \leftarrow\!\!\$\ \{0,1\}^{256}$ $m_1 \leftarrow \vec{algs}_C \| n_c$	$\xrightarrow{m_1}$	$(\mathsf{KDF}, \mathsf{Hash}, \mathsf{PKE}, \mathsf{MAC}) \leftarrow \text{negotiate}(\vec{algs}_C, \vec{algs}_S)$ $h \leftarrow \mathsf{Hash}(m_1 \| \vec{algs}_S \| cert_S)$ $C_1 \leftarrow \mathsf{AuthEnc}_s(h \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{PKE} \| \mathsf{MAC})$
Verify $cert_S$ $pk_S \leftarrow \text{parse}(cert)$	$\xleftarrow{m_2}$	$m_2 \leftarrow \vec{algs}_S \| cert_S \| C_1$
	Key exchange phase	
$(\mathsf{KDF}, \mathsf{Hash}, \mathsf{PKE}, \mathsf{MAC}) \leftarrow \text{negotiate}(\vec{algs}_C, \vec{algs}_S)$ $h \leftarrow \mathsf{Hash}(m_1 \| \vec{algs}_S \| cert_S)$ $pms \leftarrow\!\!\$\ \{0,1\}^{128}$ $e \leftarrow \mathsf{PKE.Enc}(pk_s, pms)$		
$m_3 \leftarrow C_1 \| e$	$\xrightarrow{m_3}$	$h \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{PKE} \| \mathsf{MAC} \leftarrow \mathsf{AuthDec}_s(C_1)$ $pms \leftarrow \mathsf{PKE.Dec}(sk_S, e)$
$k \leftarrow \mathsf{KDF}(pms)$		$k \leftarrow \mathsf{KDF}(pms)$ $C_2 \leftarrow \mathsf{AuthEnc}_s(k \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{PKE} \| \mathsf{MAC})$ $\tau_1 \leftarrow \mathsf{MAC}(k, h \| m_3 \| C_2)$
Verify $\tau_1 = \mathsf{MAC}(k, h \| m_3 \| C_2)$ if verify fails, $\pi_i^s.\alpha \leftarrow$ `reject`	$\xleftarrow{m_4}$	$m_4 \leftarrow C_2 \| \tau_1$
	Time synchronization phase $p = 1, \ldots, n$	
$\alpha \leftarrow$ `in-progress` $n_{c_2} \leftarrow\!\!\$\ \{0,1\}^{256}$ $t_1 \leftarrow \mathsf{Now}()$ $m_5 \leftarrow t_1 \| n_{c_2} \| C_2$	$\xrightarrow{m_5}$	$t_2 \leftarrow \mathsf{Now}()$ $k \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{PKE} \| \mathsf{MAC} \leftarrow \mathsf{AuthDec}_s(s, C_2)$ $t_3 \leftarrow \mathsf{Now}()$
	$\left[\xleftarrow{m_6^*}\right]$	$m_6^* \leftarrow t_1 \| t_2 \| t_3$
		$\tau_2 \leftarrow \mathsf{MAC}(k, m_5 \| t_1 \| t_2 \| t_3)$
$t_4 \leftarrow \mathsf{Now}()$ $RTT \leftarrow (t_4 - t_1) - (t_3 - t_2)$ If $RTT > E$ then $\alpha \leftarrow$ `reject` Verify $\tau_2 = \mathsf{MAC}(k, m_5 \| t_1 \| t_2 \| t_3)$ if verify fails, $\pi_i^s.\alpha \leftarrow$ `reject` $offset = \frac{1}{2}(t_3 + t_2 - t_1 - t_4)$ $time_p \leftarrow \mathsf{Now}() + offset$ $\alpha \leftarrow \texttt{accept}_p$ If $p = n$, terminate	$\xleftarrow{m_6}$	$m_6 \leftarrow t_1 \| t_2 \| t_3 \| \tau_2$

Figure 2: Authenticated NTP (ANTP_E), where E is a fixed upper bound on the desired accuracy. The pre-determined negotiation function *negotiate*, takes as input two ordered lists of algorithms and returns a single algorithm. n denotes the maximum number of synchronization phases, and p denotes the current synchronization phase. $[m_6^*]$ indicates an optional message sent based on a "no-cryptographic-latency" flag present in m_5, omitted in this figure. Note that if PKE.Dec or AuthDec fails for any ANTP server, the server simply stops processing the message and allows the client to time-out. If certificate validation fails, the client aborts the protocol run.

only partly addressed by ANTP, which we explain in further detail below: resistance against the *rogue master*, *cryptographic DoS* and *time-protocol DoS* attacks.

Stateless server. A single server might provide time-synchronization to hundreds or thousands of clients, thus keeping state for each client may be impossible for highly queried servers. Multiple round-trips allow for simpler constructions, but a stateless server is a feature ANTP achieves by allowing the server to regenerate per-client state upon request. Thus our construction uses *authenticated-encryption (AE)* schemes in a similar manner to TLS Session Tickets [28], where the server authenticates and encrypts its per-client state using a long-term symmetric key, then sends the ciphertext to the client for storage. The client responds with the ciphertext in order for the server to decrypt and recover state. The server periodically refreshes the long-term secret key for the AE scheme (the intervals are dependent on the security requirements of the AE scheme).

No-cryptographic-latency. Since cryptographic processing adds asymmetrically to propagation time, they can introduce error in the approximation of propagation time $\tilde{\theta}_3$, authentication operations degrade the accuracy of the `transmit_timestamp` in the `resp`. We introduce a method for zero additional error due to authentication: during the Time-Synchronization Phase, at the client's option, the server will immediately process a `resp` as in Figure 1 and sends it to the client, without authentication. The server subsequently creates an ANTP `ServerResp` message, and sends the `resp` with `ServerResp` in the NTP extension fields of the saved `resp`. A client can then use the time when receiving the initial `resp` and verify authentication with the ANTP `ServerResp`, aborting if authentication fails, if either message wasn't received, or if messages were received in incorrect order.

Efficient cryptography. Public-key operations are computationally expensive, especially in the case of a server servicing a large pool of NTP clients. ANTP only requires a single public-key operation per-client to ensure authentication and confidentiality of the premaster secret key material. The client can reuse the shared secret key on multiple subsequent time-synchronization requests with that server. ANTP uses *public-key encryption (PKE)* for establishing the shared premaster secret; while some protocols avoid using PKE for key exchange since it does not provide forward secrecy, this is not a concern for ANTP since we do not need confidentiality.

Key freshness and reuse. ANTP is also designed to allow multiple time-synchronization phases for each session, using a new nonce each Time-Synchronization Phase (preventing replay attacks and ensuring uniqueness of the protocol flow), until either the client restarts the negotiation phase, or the server rotates public-keys or authenticated-encryption keys.

Denial of service attacks. Against a MITM adversary, some types of denial-of-service (DoS) attacks are unavoidable, as the adversary may always drop messages. Unauthenticated SNTP has a roughly 1:1 ratio of attacker work to server work, in that one attack packet causes one packet in response, and a small computational effort is required by the server. In ANTP, the cryptographic operations do allow some amplification of work. First, the server performs a public key operation during key exchange while a malicious client may not; but a server under attack can temporarily stop responding to key exchange requests, and since most honest clients will perform key exchange infrequently, their service will not be denied. The second amplification is caused the by the zero cryptographic latency feature, since two response packets are sent for each request. This feature can also be turned off during attack, and in this case the server should indicate with a flag that it does not support this feature. Finally, in the negotiation phase the server's response is also considerably larger than the client request (because it may include a certificate chain), but like the key exchange phase, the negotiation phase may be temporarily disabled without denying service to clients who already have established a premaster secret. Another option is to replace the server certificate chain with a URL where the client can download it. Depending on the size of the certificate(s) this could reduce the bandwidth amplification considerably. This last mitigation requires detailed analysis, which we leave to future work.

Certificate validation. When using digital certificates to authenticate public keys, the synchronization of the issuer and the relying party is an underlying assumption. This serves to highlight a significant problem – *how do you securely authenticate time using public-key infrastructure without previously having time-synchronization with the issuer?* For our construction we assume that the client has some out-of-band method for establishing the trustworthiness of public-keys, perhaps using OCSP [29] with nonces to ensure freshness of responses, or by the user manually setting the time for first certificate validation. Since certificate validity periods typically range from months to years, if the user is assured of time-synchronization with the issuer to be within range of hours or days and that range sits comfortably within the certificate validation period, this is a viable solution.

4 Security Framework

In this section we introduce our new time-synchronization provable security framework for analyzing time synchronization protocols such as ANTP, NTP and the Precise Time Protocol. It builds on both the Bellare-Rogaway model [1] for *authenticated key-exchange* and the Jager et al. framework for *authenticated and confidential channel establishment* [11]. Neither of those models however includes time. Schwenk [31] recently proposed a framework for the purposes of modeling time in provable security analysis which models time as a global parameter. Our framework however models time as a counter that each party separately maintains, as the goal of the protocol is to synchronize these disparate counters. Additionally, the adversary in our execution environment has the ability to initialize each protocol run with a new time counter independent of the party's own counter, and controls when protocol runs can increment their counter, effectively giving the adversary complete control of both the latency of the network and the computation time of the parties.

4.1 Execution Environment

There are n_p parties $P_1, \ldots, P_{n_p}$, each of whom is a protocol participant. Each party generates a long-term key-pair (sk_i, pk_i), and can run up to n_s instances of the protocol which are referred to as sessions. We denote the sth session of a party P_i as π_i^s. Note that each session π_i^s has access to the long-term key pair of the party P_i.

Per-Session Variables. The following variables maintained by each session:

- $\rho \in \{\texttt{client}, \texttt{server}\}$: the role of the party in the session.
- $id \in \{1, ..., n_p\}$: the identity of the party running the session.
- $pid \in \{1, \ldots, n_p\}$: the believed identity of the partner of the session.
- $\alpha \in \{\texttt{accept}, \texttt{reject}, \texttt{in-progress}\}$: the status of the session.
- k: the session key.
- T: the transcript of messages sent and received in the session.
- $time$: a counter maintained by the session.

Adversary Interaction. The adversary schedules and controls all interactions between protocol participants. The adversary is in complete control of all communication, able to create, delete, reorder or modify messages at will. The adversary can compromise long-term and session keys. Additionally, the adversary is able to set the clock of a party to an arbitrary time when beginning a session and control the rate at which time progresses during the execution of a session. The following queries model normal execution with adversary control of time:

- $\mathsf{Create}(i, r, t)$: The adversary actives a new session with party P_i, initializing it with $\pi_i^s.\rho = r$ and $\pi_i^s.time = t$. Note that if $\pi_i^s.\rho = \texttt{client}$, then π_i^s responds with the first message of the protocol run.
- $\mathsf{Send}(i, s, m, \vec{\Delta})$: The adversary sends a message m to a session π_i^s. Party P_i processes the message m and responds according to the protocol specification, updating per-session variables and outputting some message m^* if necessary. During message processing, the party may execute multiple calls to a distinguished $\mathsf{Now}()$ procedure, modeling the party reading its current time from memory; immediately before the ℓth such call to the $\mathsf{Now}()$ procedure, the session's $\pi_i^s.time$ variable is incremented by Δ_ℓ.

The next queries model compromise of secret data:

- $\mathsf{Reveal}(i, s)$: The adversary receives the session key k of the session π_i^s.
- $\mathsf{Corrupt}(i)$: The adversary receives the long-term secret-key sk_i of the party P_i.

The following query allows additional adversary control of the clock:

- $\mathsf{Tick}(i, s, \Delta)$: The adversary increments the counter $\pi_i^s.time$ by Δ.

The vector $\vec{\Delta}$ in Send is necessary due to subtleties in the security framework: An adversary cannot issue Tick queries to a session during the processing of a Send query, but a party may read its clock multiple times while processing a message and thus expect to receive different clock times. The vector $\vec{\Delta}$ in the Send query allows adversary control of this clock rate.

Note that our model assumes that during execution of a session, the clocks between two parties advance at the same rate, otherwise it does not make sense for two parties to try to synchronize their clocks at all. This implicitly assumes that the parties are in the same reference frame. Additionally, while computer clocks may progress at different rates, we are assuming that, over a relatively short period of time, like the few seconds for an execution of the protocol, the difference in clock rate will be negligible. This will be formalized in Definitions 3 and 4 with the condition that the adversary advances the $time$ of matching sessions symmetrically: a $\mathsf{Tick}(j, t, \sum_{i=1}^{\ell} \Delta_l)$ must be issued if session π_j^t matching π_i^s exists when $\mathsf{Send}(i, s, m, \vec{\Delta})$ is issued.

Security Experiment. The *time-synchronization security game* is played between a challenger $\mathcal{C}$ who implements all n_p parties according to the execution environment and protocol specification, and an adversary $\mathcal{A}$. After the challenger generates the long-term key pairs, the

adversary receives the list of public keys and interacts with the challenger using the queries described above. Eventually the adversary terminates.

4.2 Security Definitions

The goal of the adversary, formalized in this section, is to break time-synchronization security by causing any client session to complete a session with a time counter such that $|\pi_i^s.time - \pi_j^t.time| > \delta$, (where π_j^t is the partner of the session π_i^s such that $\pi_j^t.id = \pi_i^s.pid$, and δ is an accuracy margin) or cause a session π_i^s to accept a protocol run without having a matching conversation with another session π_j^t. The adversary controls the initialization of the party's clock in each session, and the rate at which the clock advances during each session, with the restriction that during execution of a session the adversary must advance the party and its peer at the same rate.

4.2.1 Matching Conversations and Authentication

Authentication is defined similarly to the approach of Bellare and Rogaway [1], by use of matching conversations. We use the variant of matching conversations employed by Jager *et al.* [11].

Definition 1 (Matching Conversations)**.** *We say a session π_i^s has a* matching conversation *with a session π_j^t if $\pi_i^s.\rho \neq \pi_j^t.\rho$ and $\pi_i^s.T$ prefix-matches $\pi_j^t.T$. For two transcripts $\pi_i^s.T$ and $\pi_j^t.T$, we say that $\pi_i^s.T$ is a* prefix *of $\pi_j^t.T$ if $|\pi_i^s.T| \neq 0$ and $\pi_j^t.T$ is identical to $\pi_i^s.T$ for the first $|\pi_i^s.T|$ messages in $\pi_j^t.T$. Two transcripts $\pi_i^s.T$ and $\pi_j^t.T$* prefix-match *if $\pi_i^s.T$ is a prefix of $\pi_j^t.T$, or $\pi_j^t.T$ is a prefix of $\pi_i^s.T$.*

Prefix matching prevents an adversary from trivially winning the game by dropping the last protocol message after a session has accepted.

Definition 2 (Authentication)**.** *We say that a session π_i^s* accepts maliciously *if:*

- *$\pi_i^s.\alpha =$ `accept`;*
- *no $\mathsf{Reveal}(i,s)$ or $\mathsf{Reveal}(j,t)$ queries were issued before $\pi_i^s.\alpha \leftarrow$ `accept` and π_j^t prefix-matches π_i^s;*
- *no $\mathsf{Corrupt}(j)$ query was ever issued before $\pi_i^s.\alpha \leftarrow$ `accept`, where $j = \pi_i^s.pid$;*

but there exists no session π_j^t such that π_i^s has a unique matching conversation with π_j^t.

We define $\mathrm{Adv}^{\mathrm{auth}}_{\mathcal{T}}(\mathcal{A})$ as the probability of $\mathcal{A}$ forcing any session π_i^s to accept maliciously.

The Reveal query restriction stops $\mathcal{A}$ from trivially winning the game by accessing the session key of the Test session. The $\mathsf{Corrupt}$ query restriction also stops $\mathcal{A}$ from trivially winning by decrypting the premaster secret with the session peer's public-key.

4.2.2 Correct and Secure Time Synchronization

The goal of a time synchronization protocol is to ensure that the difference between the two parties' clocks is within a specified bound. A protocol is δ-correct if that difference can be bounded in honest executions of the protocol, and δ-accurate secure if that difference can be bounded even in the presence of an adversary.

Definition 3 (δ-Correctness)**.** *A protocol $\mathcal{T}$ satisfies* δ-correctness *if, in the presence of a passive adversary that faithfully delivers all messages and increments in each partner session symmetrically, then the client and server's clocks are within δ of each other. More precisely, in the presence of a passive adversary, for all sessions π_i^s where*

- *$\pi_i^s.\alpha =$ `accept`;*
- *$\pi_i^s.\rho =$ `client`;*
- *whenever $\mathcal{A}$ queries $\mathsf{Send}(i,s,m,\vec{\Delta})$ or $\mathsf{Send}(j,t,m',\vec{\Delta}')$, $\mathcal{A}$ also queries $\mathsf{Tick}(j,t,\sum_{i=1}^{\ell}\Delta_\ell)$ or $\mathsf{Tick}(i,s,\sum_{i=1}^{\ell}\Delta'_\ell)$, respectively; and*
- *whenever $\mathcal{A}$ queries $\mathsf{Tick}(i,s,\Delta)$, or $\mathsf{Tick}(j,t,\Delta')$, $\mathcal{A}$ also queries $\mathsf{Tick}(j,t,\Delta)$ or $\mathsf{Tick}(j,t,\Delta')$, respectively;*

we must also have that $|\pi_i^s.\mathrm{time} - \pi_j^t.\mathrm{time}| \leq \delta$.

Definition 4 (δ-Accurate Secure Time-Synchronization)**.** *We say that an adversary $\mathcal{A}$* breaks the δ-accuracy *of a time-synchronization protocol if when $\mathcal{A}$ terminates, there exists a session π_i^s with partner id $\pi_i^s.pid = j$ such that:*

- *$\pi_i^s.\alpha =$ `accept`;*
- *$\pi_i^s.\rho =$ `client`*
- *$\mathcal{A}$ did not make a $\mathsf{Corrupt}(j)$ query before $\pi_i^s.\alpha \leftarrow$ `accept`;*
- *$\mathcal{A}$ did not make a $\mathsf{Reveal}(i,s)$ or $\mathsf{Reveal}(j,t)$ query before $\pi_i^s.\alpha \leftarrow$ `accept` and π_j^t has a matching conversation with π_i^s;*
- *while $\pi_i^s.\alpha =$ `in-progress` and $\mathcal{A}$ queried $\mathsf{Send}(i,s,m,\vec{\Delta})$ or $\mathsf{Send}(j,t,m',\vec{\Delta}')$, then $\mathcal{A}$ also queried $\mathsf{Tick}(j,t,\sum_{i=1}^{\ell}\Delta_\ell)$ or $\mathsf{Tick}(i,s,\sum_{i=1}^{\Delta}\Delta'_\ell)$, respectively;*
- *while $\pi_i^s.\alpha =$ `in-progress` and $\mathcal{A}$ queried $\mathsf{Tick}(i,s,\Delta)$, or $\mathsf{Tick}(j,t,\Delta')$, then $\mathcal{A}$ also queried $\mathsf{Tick}(j,t,\Delta)$ or $\mathsf{Tick}(j,t,\Delta')$, respectively; and*
- *$|\pi_i^s.\mathrm{time} - \pi_j^t.\mathrm{time}| > \delta$.*

The probability an adversary $\mathcal{A}$ has in breaking δ-accuracy of a time-synchronization protocol $\mathcal{T}$ is denoted $\mathrm{Adv}^{\mathrm{time}}_{\mathcal{T},\delta}(\mathcal{A})$.

4.3 Multi-Phase Protocols

Our construction in Section 3 has a single run of the negotiation and key-exchange phases, followed by multiple time-synchronization executions reusing the negotiated cryptographic algorithms and shared secret key. To model

the security of such *multi-phase* time-synchronization protocols, we further extend our framework so that a single session can include multiple time-synchronization phases. The differences from the model described in the previous section are detailed below.

4.3.1 Execution Environment

Per-Session Variables. The following variables are added or changed:

- $n \in \mathbb{N}$: the number of time-synchronization phases allowed in this session.
- $time_p$, for $p \in \{1,\ldots,n\}$: the time recorded at the conclusion of phase p.
- $\alpha \in \{\texttt{accept}_p, \texttt{reject}, \texttt{in-progress}\}$, for $p \in \{1,\ldots,n\}$: the status of the session. Note that, when phase p concludes and $\alpha \leftarrow \texttt{accept}_p$ is set, the party also sets $time_p \leftarrow time$.

Adversary Interaction. The adversary can direct the client to run an additional time-synchronization phase with a new Resync query, and the client will respond according the protocol specification. The Create query in this setting is also changed:

- $\mathsf{Create}(i,r,t,n)$: Proceeds as for $\mathsf{Create}(i,r,t)$, and also sets $\pi_i^s.n \leftarrow n$.
- $\mathsf{Resync}(i,s,\vec{\Delta})$ - The adversary indicates to a session π_i^s to begin the next time-synchronization phase. Party P_i responds according to protocol specification, updating per-session variables and outputting some message m^* if necessary. During message processing, immediately before the ℓth call to the $\mathsf{Now}()$ procedure, the session's $\pi_i^s.time$ variable is incremented by Δ_ℓ.

4.3.2 Security Definitions

The goal of the adversary is also slightly different to account for the possibility of breaking time-synchronization of any given time-synchronization phase: the adversary's goal is to cause a client session to have *any* phase where its time is desynchronized from the server's. In particular, for there to be some client instance π_i^s and some phase p such that $|\pi_i^s.time_p - \pi_j^t.time_p| > \delta$ where π_j^t is the partner of session π_i^s. Again the adversary in general controls clock ticks and can tick parties at different rates, however must tick clocks at the same rate when phases have switched back to being `in-progress`.

Definition 5 (δ-Accurate Secure Multi-Phase Time-Synchronization)**.** *We say that an adversary $\mathcal{A}$* breaks the δ-accuracy *of a multi-phase time-synchronization protocol if when $\mathcal{A}$ terminates, there exists a phase p session π_i^s with partner id $\pi_i^s.pid = j$ such that:*

- $\pi_i^s.\rho = \texttt{client}$
- $\pi_i^s.\alpha = \texttt{accept}_q$ *for some $q \geq p$;*
- *$\mathcal{A}$ did not make a* $\mathsf{Corrupt}(j)$ *query before* $\pi_i^s.\alpha \leftarrow \texttt{accept}_p$ *was set;*
- *$\mathcal{A}$ did not make a* $\mathsf{Reveal}(i,s)$ *or* $\mathsf{Reveal}(j,t)$ *query before* $\pi_i^s.\alpha \leftarrow \texttt{accept}_p$ *was set and π_j^t has a matching conversation with π_i^s;*
- *while* $\pi_i^s.\alpha = \texttt{in-progress}$ *and $\mathcal{A}$ queried* $\mathsf{Send}(i,s,m,\vec{\Delta})$ *or* $\mathsf{Send}(j,t,m',\vec{\Delta}')$*, then $\mathcal{A}$ also queried* $\mathsf{Tick}(j,t,\sum_{i=1}^{\ell}\Delta_\ell)$ *or* $\mathsf{Tick}(i,s,\sum_{i=1}^{\ell}\Delta'_\ell)$*, respectively;*
- *while* $\pi_i^s.\alpha = \texttt{in-progress}$ *and $\mathcal{A}$ queried* $\mathsf{Tick}(i,s,\Delta)$*, or* $\mathsf{Tick}(j,t,\Delta')$*, then $\mathcal{A}$ also queried* $\mathsf{Tick}(j,t,\Delta)$ *or* $\mathsf{Tick}(j,t,\Delta')$*, respectively; and*
- $|\pi_i^s.time_p - \pi_j^t.time_p| > \delta$.

The probability an adversary $\mathcal{A}$ has in breaking δ-accuracy of multi-phase time-synchronization protocol $\mathcal{T}$ is denoted $\mathrm{Adv}^{\text{multi-time}}_{\mathcal{T},\delta}(\mathcal{A})$.

5 Security of ANTP

In this section we present our correctness and security theorems for demonstrating that ANTP satisfies a E-accurate secure time-synchronization protocol, where E represents the maximum error an attacker can induce. We explain our intuition by using the bound on the possible error that an $\mathcal{A}$ can introduce without altering packets introduced in Section 3. It follows then that if all messages are securely authenticated, and the only inputs to the clock-update procedure are either:

- authenticated via messages; or
- the round trip delay RTT,

then any attacker can only introduce at most E error into the clock-update procedure (where $E \geq RTT$). We say that such a protocol is an E-accurate secure time-synchronization protocol. We formalize this as Theorem 2, below:

5.1 Correctness

Theorem 1 (Correctness of ANTP)**.** *Fix $E \in \mathbb{N}$. ANTP_E is an E-correct time-synchronization protocol as defined in Definition 3.*

Proof. When analyzing ANTP in terms of correctness, we can restrict analysis to data that enters the clock-update procedure as input, as the rest of the protocol is designed to ensure authentication and does not influence the session's *time* counter. This allows us to narrow our focus to SNTP, which is the time-synchronization core of ANTP.

We first focus on a single time-synchronization phase. At the beginning of the time-synchronization phase of ANTP, the client will sent an NTP request (`req`) which contains t_1, the time the client sent `req`. Note that the adversary is restricted to delivering the messages faithfully as a passive adversary, and also must increment the

time of each protocol participant symmetrically. The adversary otherwise has complete control over the passage of time. Thus θ_1, θ_2, θ_3 are non-negative but otherwise arbitrary values selected by the adversary (where θ_1 is the propagation time from client to server, θ_2 is server processing time and θ_3 is propagation time from server to client). Thus the client computes the round-trip time of the protocol as: $RTT = (t_4 - t_1) - (t_3 - t_2) = \theta_1 + \theta_3$ and approximates the server-to-client propagation time as $\tilde{\theta}_3 = \frac{1}{2}(\theta_1 + \theta_3)$.

When the client-to-server and server-to-client propagation times are equal ($\theta_1 = \theta_3$) then $\tilde{\theta}_3 = \theta_3$, and the values t_3 and t_2 allow the client to exactly account for θ_2. The *time* counter is updated by $time + offset = t_3 + \tilde{\theta}_3 - t_4$, and upon completion the client's clock is exactly synchronized with the server's clock.

When $\theta_1 \neq \theta_3$, we have that $\theta_3 - \tilde{\theta}_3 = \frac{1}{2}(\theta_3 - \theta_1)$, so the statistics t_1, ..., t_4 do not allow the client to exactly account for client-to-server propagation time θ_3; the client's updated time may be off by up to $\frac{1}{2}(\theta_3 - \theta_1)$. Fortunately, we can bound this value by E: we know that $\frac{1}{2}(\theta_3 - \theta_1) \leq \frac{1}{2}(\theta_1 + \theta_3)$, and furthermore we know that ANTP_E will only accept time-synchronization when $\frac{1}{2}(\theta_1 + \theta_3) \leq E$, so in sessions that accept (assuming a passive adversary) we have that the client's clock is at most $\frac{1}{2}(\theta_3 - \theta_1) \leq E$ different from the server's clock.

Now moving to the multi-phase setting, we note that this analysis of the correctness of ANTP applies to each separate time-synchronization phase: since the client's (t_1, t_4) values are only used to calculate the total round-trip time of the time-synchronization phase, thus if the rate-of-time for both client and server during the phase is the same, each phase is also E-accurate in the presence of a passive adversary, even if the adversary dramatically changes the rate-of-time for partners between time-synchronization phases. □

5.2 Security

Theorem 2 (Security of ANTP)**.** *Fix $E \in \mathbb{N}$. Assuming the public key encryption scheme* PKE *is IND-CCA-secure, the message authentication code* MAC *is eUF-CMA-secure, the hash function* Hash *is collision-resistant, and the key derivation function* KDF *and authenticated encryption scheme* AE *are secure, then* ANTP_E *is a E-accurate secure time-synchronization protocol as in Definition 4. In particular, there exist algorithms $\mathcal{B}_3, \ldots, \mathcal{B}_8$, described in the proof of the theorem, such that, for all adversaries $\mathcal{A}$, we have that*

$$\begin{aligned}\mathrm{Adv}^{\mathrm{time}}_{\mathsf{ANTP}_E,E}(\mathcal{A}) \leq \frac{n_p^2 n_s^2}{2^{254}} + n_p^2 n_s^2 \Big(& \mathrm{Adv}^{\mathrm{coll}}_{\mathsf{Hash}}(\mathcal{B}_3^{\mathcal{A}}) \\ & + \mathrm{Adv}^{\mathsf{AE}}_{\mathsf{AuthEnc}}(\mathcal{B}_4^{\mathcal{A}}) + \mathrm{Adv}^{\mathrm{ind-cca}}_{\mathsf{PKE}}(\mathcal{B}_5^{\mathcal{A}}) \\ & + \mathrm{Adv}^{\mathrm{kdf}}_{\mathsf{KDF}}(\mathcal{B}_6^{\mathcal{A}}) + \mathrm{Adv}^{\mathsf{AE}}_{\mathsf{AuthEnc}}(\mathcal{B}_7^{\mathcal{A}}) \\ & + \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{B}_8^{\mathcal{A}}) \Big)\end{aligned}$$

where n_p and n_s are the number of parties and sessions created by $\mathcal{A}$ during the experiment.

The standard definitions for security of the underlying primitives and the corresponding advantages $\mathrm{Adv}^{\mathsf{AuthEnc}}_{\mathsf{AE}}(\mathcal{A}), \mathrm{Adv}^{\mathrm{ind-cca}}_{\mathsf{PKE}}(\mathcal{A}), \mathrm{Adv}^{\mathrm{coll}}_{\mathsf{Hash}}(\mathcal{A}), \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{A})$, and $\mathrm{Adv}^{\mathrm{kdf}}_{\mathsf{KDF}}(\mathcal{A})$ are given in Appendix C.

Proof. From Theorem 1, ANTP_E is an E-correct time-synchronization protocol in the sense of Definition 3. Thus all passive adversaries have probability 0 of breaking E-accuracy of ANTP_E. If we show that the advantage $\mathrm{Adv}^{\mathrm{auth}}_{\mathsf{ANTP}_E}(\mathcal{A})$ of any adversary $\mathcal{A}$ of breaking authentication security (i.e., to accept without matching conversations) of ANTP_E is small, then it follows that the advantage of any active adversary $\mathcal{A}$ in breaking E-accuracy of ANTP_E is similarly small. In other words, it immediately is the case that $\mathrm{Adv}^{\mathrm{time}}_{\mathsf{ANTP}_E,E}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{auth}}_{\mathsf{ANTP}_E}(\mathcal{A})$.

We now focus on bounding $\mathrm{Adv}^{\mathrm{auth}}_{\mathsf{ANTP}_E}(\mathcal{A})$. In order to show that an active adversary has negligible probability in breaking ANTP_E authentication, we use a proof structured as a sequence of games. We let $\Pr(break_i)$ denote the probability that the adversary causes some session to accept maliciously in game i. We iteratively change the security experiment, and demonstrate that the changes are either failure events with negligible probability of occurring or that if the changes are distinguishable we can construct an adversary capable of breaking an underlying cryptographic assumption. Since the client will only accept synchronization if all three phases are properly authenticated, the advantage of an active adversary is negligible given our cryptographic assumptions.

Game 0. This is the original time-synchronization game described in § 4: $\mathrm{Adv}^{\mathrm{auth}}_{\mathsf{ANTP}_E}(\mathcal{A}) = \Pr(break_0)$.

Game 1. In this game, we add an abort rule for non-unique nonces. Specifically, if any nonce is used by two different sessions by client instances, we abort the simulation. This is a transition based on failure events. Recall that n_p is the number of parties, and n_s is the number of sessions per party. Then there are at most $2n_s n_p$ nonces used by client instances, each of 256 bits. The probability that a collision occurs among these values is $(2n_s n_p)^2/2^{256}$, so: $\Pr(break_0) \leq \Pr(break_1) + \frac{n_s^2 n_p^2}{2^{254}}$.

Game 2. In this game, we try to guess which client session will be the first to accept maliciously, and abort if we guess in correctly. We choose a random session π_i^s where $(i,s) \leftarrow\$ \{1,\ldots,n_p\} \times \{1,\ldots,n_s\}$. If π_i^s is not the first session to accept maliciously, then the challenger aborts the game. Note that from Game 2 forward, the challenger answers $\mathsf{Reveal}(i,s)$ queries before $\pi_i^s.\alpha = \texttt{accept}$ by aborting the game, as it follows that the guessed session cannot accept maliciously. This is a transition based on failure events. There are at most $n_p n_s$ client sessions, and we guess the first session to accept maliciously with probability at least $1/n_p n_s$, so $\Pr(break_1) \leq n_p n_s \Pr(break_2)$.

Game 3. In this game, we try to guess which server session will be the partner to the first session to accept maliciously. We choose a random session π_j^t where $(j,s) \leftarrow\$ \{1,\ldots,n_p\} \times \{1,\ldots,n_s\}$. If π_j^t is not the partner of the first session π_i^s to accept maliciously, then the challenger aborts the game. Note that from Game 3 forward, the challenger answers $\mathsf{Corrupt}(j)$ and $\mathsf{Reveal}(j,t)$ queries before $\pi_i^s.\alpha \leftarrow \texttt{accept}$ by aborting the game, as it follows that the guessed session cannot accept maliciously. This is a transition based on failure events. There are at most $n_p n_s$ server sessions, and we guess the partner of the first session to accept maliciously with probability at least $1/n_p n_s$, so $\Pr(break_2) \leq n_p n_s \Pr(break_3)$.

Game 4. In this game, we exclude hash collisions. We construct a simulator $\mathcal{B}_4^{\mathcal{A}}$ that runs the security game and computes all hash values $h = \mathsf{Hash}(m_1 \| \vec{alg}_2 \| \mathit{ServerCert})$ honestly, and maintains a list *Coll* where all inputs and outputs of the hash function are stored. $\mathcal{B}_3^{\mathcal{A}}$ aborts if there exists two entries $(in, \mathsf{Hash}(in)), (\hat{in}, \mathsf{Hash}(\hat{in}))$ such that $in \neq \hat{in}$ but $\mathsf{Hash}(in) = \mathsf{Hash}(\hat{in})$. Whenever the adversary $\mathcal{A}$ wins the game, $\mathcal{B}_3^{\mathcal{A}}$ inspects *Coll* to see if a collision occurred and if so, outputs the collision. Thus $\Pr(break_3) \leq \Pr(break_4) + \mathrm{Adv}^{\mathrm{coll}}_{\mathsf{Hash}}(\mathcal{B}_3^{\mathcal{A}})$.

Game 5. In this game, we abort if in server session π_j^t the ciphertext received in m_3 is not equal to the ciphertext sent in m_1 but the output of $\mathsf{AuthDec}_s$ is not $\perp$.

We construct an algorithm $\mathcal{B}_4^{\mathcal{A}}$ that interacts with an AE challenger in the following way: $\mathcal{B}_4^{\mathcal{A}}$ acts exactly as in game 4 except for sessions run by party P_j. When P_j needs to run AuthEnc or AuthDec, $\mathcal{B}_4^{\mathcal{A}}$ uses its AE challenger's oracles to compute the required value. This is a perfect simulation of game 4. In server session π_j^t, when $\mathcal{B}_4^{\mathcal{A}}$ receives a ciphertext in m_3 that was not equal to the ciphertext sent in m_1 but the output of the AuthDec oracle is not $\perp$, this corresponds to a ciphertext forgery, and thus $\mathcal{B}_4^{\mathcal{A}}$ has broken the integrity of AE. Thus, $\Pr(break_4) \leq \Pr(break_5) + \mathrm{Adv}^{\mathsf{AE}}_{\mathsf{AuthEnc}}(\mathcal{B}_4^{\mathcal{A}})$.

Game 6. In this game, sessions π_i^s and π_j^t compute the session key k by applying KDF to a random secret $pms' \leftarrow\$ \{0,1\}^{128}$, rather than the *pms* that was encrypted using PKE.Enc and transmitted in ciphertext e. Any algorithm used to distinguish Game 5 from Game 6 can be used to construct an algorithm capable of distinguishing PKE encrypted values via plaintext, thus breaking IND-CCA security of the public-key encryption scheme.

We construct a simulator $\mathcal{B}_5^{\mathcal{A}}$ that interacts with a PKE challenger. $\mathcal{B}_5^{\mathcal{A}}$ activates party P_j with the public key pk received from the challenger. $\mathcal{B}_5^{\mathcal{A}}$ responds identically to queries from $\mathcal{A}$ as in Game 5, except as follows:

- $\mathcal{B}_5^{\mathcal{A}}$ computes the public-key encrypted premaster secret e for the session π_i^s by querying the challenger's Encrypt oracle with (pms, pms') where $pms' \leftarrow\$ \{0,1\}^{128}$.
- $\mathcal{B}_5^{\mathcal{A}}$ computes $\pi_i^s.k \leftarrow \mathsf{KDF}(pms)$
- In any P_j session where m_3 contains the challenge ciphertext above, $\mathcal{B}_5^{\mathcal{A}}$ computes the session key as $k \leftarrow \mathsf{KDF}(pms)$.
- In any other P_j session where m_3 does not contain the challenge ciphertext above, $\mathcal{B}_5^{\mathcal{A}}$ queries the ciphertext to its Decrypt oracle to obtain the premaster secret and uses that as its input to KDF to compute the session key k.
- $\mathcal{B}_5^{\mathcal{A}}$ never needs to answer a $\mathsf{Corrupt}(j)$ query because of Game 3.

When the random bit b sampled by the challenger is 0, $e = \mathsf{PKE.Enc}(pms, pk_j)$ and $\mathcal{B}_5^{\mathcal{A}}$ perfectly simulates of Game 5. When $b = 1$, $e = \mathsf{PKE.Enc}(r_3, pk_j)$, and $\mathcal{B}_5^{\mathcal{A}}$ perfectly simulates Game 6. $\mathcal{B}_5^{\mathcal{A}}$ never asks the challenge ciphertext to its decryption oracle.

An adversary capable of distinguishing Game 5 from Game 6 can therefore be used to break IND-CCA security, so $\Pr(break_5) \leq \Pr(break_6) + \mathrm{Adv}^{\mathrm{ind-cca}}_{\mathsf{PKE}}(\mathcal{B}_5^{\mathcal{A}})$.

Game 7. In this game, we replace the secret key k in sessions π_i^s and π_j^t with a uniformly random value k' from $\{0,1\}^{l_{\mathsf{KDF}}}$ where l_{KDF} is the length of the KDF output, instead of being computed honestly via $k \leftarrow \mathsf{KDF}(pms)$.

In Game 6, we replaced the premaster secret value *pms* with a uniformly random value from $\{0,1\}^{128}$. Thus, any algorithm that can distinguish Game 6 from Game 7 can distinguish the output of KDF from random. We explicitly construct such a simulator $\mathcal{B}_6^{\mathcal{A}}$ that interacts with a KDF challenger, and proceeds identically to Game 6, except:

- When computing k for π_i^s, $\mathcal{B}_6^{\mathcal{A}}$ queries the KDF challenger with *pms*.
- When computing k for π_j^t, $\mathcal{B}_6^{\mathcal{A}}$ sets $\pi_j^t.k = \pi_i^s.k$.

When the random bit b sampled by the KDF challenger is 0, $k = \mathsf{KDF}(pms)$, and $\mathcal{B}_6^{\mathcal{A}}$ provides a perfect simulation of Game 6. When $b = 1$, $k \leftarrow\$ \{0,1\}^{l_{\mathsf{KDF}}}$ and $\mathcal{B}_6^{\mathcal{A}}$ provides a perfect simulation of Game 7.

An adversary capable of distinguishing Game 6 from Game 7 can therefore distinguish the output of KDF from

random, so $\Pr(break_6) \leq \Pr(break_7) + \mathrm{Adv}^{\mathrm{kdf}}_{\mathsf{KDF}}(\mathcal{B}_6^{\mathcal{A}})$.

Game 8. In this game, in session π_j^t we replace the contents of the ciphertext C_2 sent in m_3 with a random string of the same length, and abort if the ciphertext received in m_5 is not equal to the ciphertext sent in m_3 but the output of the $\mathsf{AuthDec}_s$ algorithm is not $\perp$.

We construct an algorithm $\mathcal{B}_7^{\mathcal{A}}$ that interacts with an AE challenger in the following way: $\mathcal{B}_7^{\mathcal{A}}$ acts exactly as in game 7 except for sessions run by party P_j. In session π_j^t, for the computation of C_2, $\mathcal{B}_7^{\mathcal{A}}$ picks a uniformly random binary string z' of length equal to $z = k\|\mathsf{KDF}\|\mathsf{Hash}\|\mathsf{PKE}\|\mathsf{MAC}$ and submits (z, z') to its $\mathsf{AuthEnc}$ oracle. For all other computations that P_j involving $\mathsf{AuthEnc}_s$ or $\mathsf{AuthDec}_s$, $\mathcal{B}_7^{\mathcal{A}}$ submits the query its respective $\mathsf{AuthEnc}$ or $\mathsf{AuthDec}$ oracle.

When the random bit b sampled by the AE challenger is 0, C_2 contains the encryption of z, so $\mathcal{B}_7^{\mathcal{A}}$ provides a perfect simulation of Game 7. When $b = 1$, C_2 contains the encryption of z', so $\mathcal{B}_7^{\mathcal{A}}$ provides a perfect simulation of Game 8. An adversary capable of distinguishing Game 7 from Game 8 can therefore break the confidentiality of AE and guess b. Additionally, if $\mathcal{B}_7^{\mathcal{A}}$ receives a ciphertext in m_5 that was not equal to the ciphertext sent in m_3 but the output of the $\mathsf{AuthDec}$ oracle is not $\perp$, this corresponds to a ciphertext forgery, and thus $\mathcal{B}_7^{\mathcal{A}}$ has broken the integrity of AE. Thus, $\Pr(break_7) \leq \Pr(break_8) + \mathrm{Adv}^{\mathsf{AE}}_{\mathsf{AuthEnc}}(\mathcal{B}_7^{\mathcal{A}})$.

The effect of Game 8 is that, in the target session and its partner, the key used in the MAC computations is independent of the values transmitted.

Game 9. In this game, we abort when the session π_i^s accepts maliciously. We do this by constructing a simulator $\mathcal{B}_8^{\mathcal{A}}$ that interacts with the MAC challenger, but computes τ_1 and τ_2 for π_j^t by querying $h\|m_3\|C_2$ and $m_5\|t_1\|t_2\|t_3$ to the MAC challenger. $\mathcal{B}_8^{\mathcal{A}}$ verifies MAC tags for π_{i*}^{s*} by again querying $h\|m_3\|C_2$ and $m_5\|t_1\|t_2\|t_3$ to the MAC challenger and ensuring the MAC challenger's output is equal to the tag to be verified. Note that now that the key k is substituted for the key maintained by the MAC challenger: k was already uniformly random and independent of the protocol run, and by Game 2 and Game 3, the simulator already responds to Reveal queries to π_i^s and π_j^t by aborting the security experiment. Thus these changes to the game are indistinguishable. When $\pi_i^s.\alpha \leftarrow \mathtt{accept}$, $\mathcal{B}_8^{\mathcal{A}}$ checks P_j to see if there is a session with a matching conversation. Since by Game 1 all protocol flows are unique (by unique nonces), if P_j has no session with a matching conversation the adversary must have produced a valid MAC tag $\hat{\tau}_1$ or $\hat{\tau}_2$ such that $\mathsf{MAC.Tag}(k, h\|m_3\|C_2) = \hat{\tau}_1$ or $\mathsf{MAC.Tag}(k, m_5\|t_1\|t_2\|t_3) = \hat{\tau}_2$ and (by Game 8) the key k is uniformly random. $\mathcal{B}_8^{\mathcal{A}}$ submits the appropriate pair $(h\|m_3\|C_2, \hat{\tau}_1)$, $(m_5\|t_1\|t_2\|t_3, \hat{\tau}_2)$ to the MAC challenger and aborts. Thus, $\Pr(break_8) \leq \Pr(break_9) + \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{B}_8^{\mathcal{A}})$.

Analysis of Game 9. We now show that an active adversary has probability 0 of forcing a client session π_{i*}^{s*} to accept maliciously in Game 9.

π_i^s is a target session where: no $\mathsf{Reveal}(i,s)$ or $\mathsf{Reveal}(j,t)$ queries were issued before $\pi_i^s.\alpha \leftarrow \mathtt{accept}$; no $\mathsf{Corrupt}(j)$ query was ever issued before $\pi_i^s.\alpha \leftarrow \mathtt{accept}$, where $\pi_i^s.pid = j$; and π_i^s only accepts if $\tau_1 = \mathsf{MAC}(k, h\|m_3\|C_2)$ and $\tau_2 = \mathsf{MAC}(k, m_5\|t_1\|t_2\|t_3)$. By unforgeability these tags cannot be generated by $\mathcal{A}$ and by Game 1 the protocol flow of each session is unique. τ_1 and τ_2 verification will thus only occur if $\pi_i^s.T = \pi_j^t.T$, as τ_1 is over all messages in the negotiation and key-exchange phase, and τ_2 is over all messages in the time-synchronization phase and thus π_i^s will only accept if $\pi_j^t.T$ prefix-matches $\pi_i^s.T$. Thus, no client session accepts maliciously in Game 9: $\Pr(break_9) \leq 0$.

Summing all of the probabilities yields the desired bound, showing that ANTP_E is a E-accurate secure Time-Synchronization Protocol. □

5.3 Proof of Multi-Phase Security of ANTP

A similar argument shows multi-phase security of ANTP_E, with minor changes to the games in the proof to enable guessing of the first *phase* session to accept maliciously.

Theorem 3 (Multi-Phase Security of ANTP)**.** *Fix $E, n \in \mathbb{N}$. Under the same assumptions as in Theorem 2, ANTP_E is a E-accurate secure multi-phase time-synchronization protocol as defined in Definition 5. In particular, there exist algorithms $\mathcal{B}_3, \ldots, \mathcal{B}_8$ described in the proof of Theorem 2, such that, for all adversaries $\mathcal{A}$, we have that*

$$\begin{aligned}\mathrm{Adv}^{\text{multi-time}}_{\mathsf{ANTP}_E,E}(\mathcal{A}) \leq \frac{n_p^2 n_s^2}{2^{254}} + n_p^2 n_s^2 n \Big(& \mathrm{Adv}^{\mathrm{coll}}_{\mathsf{Hash}}(\mathcal{B}_3^{\mathcal{A}}) \\ & + \mathrm{Adv}^{\mathsf{AE}}_{\mathsf{AuthEnc}}(\mathcal{B}_4^{\mathcal{A}}) + \mathrm{Adv}^{\text{ind-cca}}_{\mathsf{PKE}}(\mathcal{B}_5^{\mathcal{A}}) \\ & + \mathrm{Adv}^{\mathrm{kdf}}_{\mathsf{KDF}}(\mathcal{B}_6^{\mathcal{A}}) + \mathrm{Adv}^{\mathsf{AE}}_{\mathsf{AuthEnc}}(\mathcal{B}_7^{\mathcal{A}}) \\ & + \mathrm{Adv}^{\text{euf-cma}}_{\mathsf{MAC}}(\mathcal{B}_8^{\mathcal{A}}) \Big)\end{aligned}$$

where n_p, n_s, n are the maximum number of parties, sessions and phases created by $\mathcal{A}$ during the experiment.

Proof. The proof for Theorem 3 is identical to the proof to Theorem 2 except as follows.

A new game is inserted between Game 3 and Game 4 that guesses the first time-synchronization phase $p \in \{1, \ldots, n\}$ that the target session π_i^s will accept maliciously: by Theorem 2, we know that a session π_i^s will not accept maliciously for time-synchronization phase

$p = 1$, so by this step we know that π_i^s has a matching conversation with π_j^t up to and including phase $p-1$.

We also edit the final game (MAC challenger) so that $\mathcal{B}$ aborts if π_i^s accepts maliciously in phase p. We do this by editing the final game in the following way: When processing m_5 for π_j^t in the guessed phase p (we indicate this with m_{5p}) $\mathcal{B}$ will also compute τ_{2p} by querying the MAC challenger with $m_{5p}\|t_{1p}\|t_{2p}\|t_{3p}$, and verifies the τ_{2p} for π_i^s by querying the MAC challenger with $m_{5p}\|t_{1p}\|t_{2p}\|t_{3p}$ and accepting only if the output from the MAC challenger matches the τ_p in m_{6p}. Following the same structure as the proof to Theorem 2, we have that k is a uniformly random key generated independently from the protocol run and this change is indistinguishable. Verification of τ will only occur if $\pi_i^s.T = \pi_j^t.T$ up to phase p, as τ_1 is over all messages in the negotiation and key-exchange phase, and τ_p is over all messages in phase p. □

6 Discussion

In this work we introduced a new authenticated time-synchronization protocol called ANTP, designed to securely synchronize the time of a client and server, using public-key infrastructure. Our design is efficient, allowing a server to perform a single public key operation per client, and then use only faster symmetric key operations for each subsequent request from that client. Furthermore, the server need not even store per-client state, instead securely offloading storage of that state to the client.

Our ANTP protocol is accompanied by a thorough provable security analysis showing that it provides secure time synchronization within user-specified accuracy bounds. The analysis is carried out in a new provable security framework. A novel aspect of our new framework, when compared with the long line of work on authentication definitions, is that our framework models an adversary with the ability to control the flow of time, meaning the adversary can initialize different parties' clocks to different times, and even control the rate at which their clocks are advanced. Our new security framework can be used for the analysis of other time-synchronization protocols such as the Network Time Security (NTS) protocol and the Precise-Time Protocol (PTP).

Several interesting open problems in the area of secure time synchronization remain. All existing time synchronization protocols that rely on public keys, including ours, need to initially validate the certificate of the time server, specifically that it is within its validity period. While nonces can be combined with OCSP responses to check freshness, this cannot completely solve the "first-boot" problem. A detailed study of denial of service attacks against secure time synchronization protocols including ANTP would also be worthwhile, giving detailed consideration to both the cost of cryptographic operations in practice and the bandwidth amplification afforded by directing protocol responses to a victim.

References

[1] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993.

[2] F. Bergsma, B. Dowling, F. Kohlar, J. Schwenk, and D. Stebila. Multi-ciphersuite security of the secure shell (SSH) protocol. In *ACM CCS 14*, pages 369–381. ACM Press, 2014.

[3] Certicom Research. Standards for Efficient Cryptography (SECG) — SEC 1: Elliptic Curve Cryptography. `http://www.secg.org/secg_docs.htm`, Sept. 20, 2000. Version 1.0.

[4] Certicom Research. Standards for Efficient Cryptography (SECG) SEC 2: Recommended Elliptic Curve Domain Parameters. Jan. 2007. `http://www.secg.org/secg_docs.htm`.

[5] L. Chen. Recommendation for Key Derivation Using Pseudorandom Functions. *NIST SP 800-108*, Oct. 2009.

[6] B. Dowling, D. Stebila, and G. Zaverucha. ANTP: Authenticated NTP (Implementation Specification). *Microsoft Research Technical Report*, MSR-TR-2015-19, 2015. `http://research.microsoft.com/apps/pubs/?id=240885`.

[7] M. Dworkin. Recommendation for Block Cipher Modes of Operation: Galois/Counter mode (GCM) and GMAC. *NIST SP 800-38D*, Nov. 2007.

[8] B. Haberman and D. Mills. Network Time Protocol Version 4: Autokey Specification. RFC 5906 (Informational), June 2010.

[9] C. Hedrick. Routing Information Protocol. RFC 1058 (Historic), June 1988. Updated by RFCs 1388, 1723.

[10] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), Nov. 2012.

[11] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk. On the security of TLS-DHE in the standard model. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 273–293. Springer, Aug. 2012.

[12] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Aug. 2010.

[13] H. Krawczyk, K. G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 429–448. Springer, Aug. 2013.

[14] J. McCann, S. Deering, and J. Mogul. Path MTU Discovery for IP version 6. RFC 1981 (Draft Standard), Aug. 1996.

[15] D. McGrew, J. Foley, and K. Patterson. Authenticated Encryption with AES-CBC and HMAC-SHA. *IETF Internet Draft*, July 2014. `http://tools.ietf.org/html/draft-mcgrew-aead-aes-cbc-hmac-sha2-05`.

[16] Microsoft Corporation. [MS-W32T]: W32Time Remote Protocol. May 2014. `https://msdn.microsoft.com/en-us/library/cc249627.aspx`.

[17] D. Mills. Network Time Protocol (NTP). RFC 958, Sept. 1985. Obsoleted by RFCs 1059, 1119, 1305.

[18] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305 (Draft Standard), Mar. 1992. Obsoleted by RFC 5905.

[19] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[20] D. L. Mills. On the accuracy and stablility of clocks synchronized by the network time protocol in the internet system. *ACM SIGCOMM Computer Communication Review*, 20(1):65–75, 1989.

[21] T. Mizrahi. Security Requirements of Time Protocols in Packet Switched Networks. RFC 7384 (Informational), Oct. 2014.

[22] J. Mogul and S. Deering. Path MTU discovery. RFC 1191 (Draft Standard), Nov. 1990.

[23] National Institute for Standards and Technology (NIST). The NIST Authenticated NTP Service. `http://www.nist.gov/pml/div688/grp40/auth-ntp.cfm` Accessed February 2015.

[24] K. G. Paterson, T. Ristenpart, and T. Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 372–389. Springer, Dec. 2011.

[25] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), Apr. 2006. Obsoleted by RFC 6347, updated by RFC 5746.

[26] J. Reynolds. Request for Comments Summary Notes: 1100-1199. RFC 1199 (Informational), Dec. 1991.

[27] S. Röttger. Analysis of the NTP Autokey Protocol. *Masters Thesis*, Feb. 2012. `http://zero-entropy.de/autokey_analysis.pdf`.

[28] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 5077 (Proposed Standard), Jan. 2008.

[29] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard), June 2013.

[30] C.-P. Schnorr. Efficient identification and signatures for smart cards. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Aug. 1990.

[31] J. Schwenk. Modelling time, or a step towards reduction-based security proofs for otp and kerberos. *IACR Cryptology ePrint Archive*, 2013:604, 2013.

[32] R. Selvi. Bypassing HTTP Strict Transport Security. *Presented at Black Hat Europe*, 2014. `https://www.blackhat.com/docs/eu-14/materials/eu-14-Selvi-Bypassing-HTTP-Strict-Transport-Security-wp.pdf`.

[33] D. Sibold, S. Röettger, and K. Teichel. Network Time Security. *IETF Internet Draft*, July 2014. `https://tools.ietf.org/html/draft-ietf-ntp-network-time-security-06`.

[34] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). RFC 3610 (Informational), Sept. 2003.

A Network Time Protocol Message

This section details the NTP message format as specified in [18].

```
struct {
uint8 leap_indicator : 2;
uint8 version : 3;
uint8 mode : 3;
uint8 stratum;
int8 pollinterval;
int8 precision;
int32 root_delay;
int32 root_dispersion;
int32 reference_identifier;
int64 reference_timestamp;
int64 originate_timestamp;
int64 recieve_timestamp;
int64 transmit_timestamp;
} NtpMessage
```

where:

- `leap_indicator`: An unsigned two-bit code used to indicate leap seconds or warnings.
- `version`: A unsigned three-bit integer used to indicate supported version of S/NTP.
- `mode`: A unsigned three-bit integer used to indicate mode of operation (client, server, etc.).
- `stratum`: An eight-bit integer used to indicate the stratum level of the local machine.
- `pollinterval`: A signed eight-bit integer used to indicate the maximum interval of time between NTP queries sent by the client, to the nearest power of two.
- `precision`: A signed eight-bit integer *n* used to indicate the resolution of the client local clock to the nearest power of two.
- `root_delay`: A signed 32-bit fixed-point number, used to indicate the total round-trip time from the client local clock to the hardware clock at the primary server.
- `root_dispersion`: A signed 32-bit fixed-point number, used to indicate the nominal error of the local clock relative to the hardware clock at the primary server.
- `reference_identifier`: A 32-bit string used to identify the primary server used as reference.
- `reference_timestamp`: A unsigned 64-bit NTP timestamp in big-endian format, used to indicate the last update of the client local clock.
- `originate_timestamp`: A unsigned 64-bit NTP timestamp in big-endian format, used to indicate the time that `req` was sent according to the client local clock.
- `receive_timestamp`: A unsigned 64-bit NTP timestamp in big-endian format, used to indicate the time the `req` arrived at the server, according to the server local clock.
- `transmit_timestamp`: An unsigned 64-bit NTP timestamp in big-endian format, used to indicate the time the time the message departed the local machine, according to the local clock.

B Authenticated Network Time Protocol Messages

The following section contains excerpts from an implementation specification for ANTP intended for submission to the IETF. Recall that all messages are designed to be sent in the NTP extension fields similarly to the Autokey Protocol [8]. When the `msg_type` of the extension field equals 0x01, 0x02, 0x03, 0x04, or 0x05 the client MUST NOT use the information in the NTP message fields for synchronization. If the `msg_type` of the extension fields equal 0x01 or 0x03 the server MAY process the NTP message normally. When the `namefield` reads 0x06 or 0x05, the client and server MUST process the respective NTP messages as specified in the NTP specification.

This protocol follows DTLS [25] regarding message fragmentation. If the message requires fragmentation, the client divides the message into a series of *N* contiguous data ranges, each at least 56 bytes shorter than the maximum message size (to account for the NTP Packet and the `msg_type`, `Length`, `Offset` and `FragmentLength` field lengths). Each of these *N* data ranges becomes a new message, each attached to an identical NTP packet, and with identical `msg_type` and `Length`. The `Offset` of a message fragment is the number of bytes in previous fragments, and `FragmentLength` is the length of the current message fragment. When any party receives an NTP message with an extension field containing a `msg_type` with value 0x01 (`ClientAssoc`), 0x02 (`ServerAssoc`), 0x03 (`ClientKey`), 0x04 (`ServerKey`), 0x05 (`ClientReq`), 0x06 (`ServerResp`), the party checks if `Length` = `FragmentLength`. If not, the party MUST buffer until it has the entire message, and process as if the message were a single NTP packet attached to a extension field with zeroed `fragment_offset` field and `fragment_length` set to `length`. This fragmentation strategy is applied to each ANTP protocol message, as required. Setting the maximum message length depends on the path MTU between the client and server. Clients can use path MTU discovery [22] [14]. See also Section 4.1.1.1 ”PMTU Discovery” from [25] for information on how path MTU is set in DTLS.

We specify the following structure to describe the `FragmentInfo` structure of all ANTP packets:

```
struct {
uint24 length;
```

```
uint16 offset;
uint16 FragmentLength;
} FragmentInfo
```

where:

- `length`: An unsigned 24-bit integer describing the length of the unfragmented message
- `fragment_offset`: An unsigned 16-bit integer describing the number of bytes contained in previous fragments of the message. When the message requires no fragmentation this value is 0.
- `fragment_length`: An unsigned 16-bit integer describing the length of this fragment on the message. When the message requires no fragmentation, this value is length.

Note that since ANTP allows buffering of messages, it is possible that multiple ANTP messages that require fragmentation may be received by another party interleaved. Since each ANTP message that is fragmented is attached to an identical NTP message, it is trivial to distinguish fragmented ANTP messages via the NTP packet. In order to reduce complexity however, the parties MUST NOT send multiple ANTP messages with identical NTP packets, but instead generate a new NTP message for each message flow.

In a similar way to TLS all values are stored in big-endian format, and the smallest block size is a single byte. We define variable- length vectors by specifying a range of legal lengths and sizes of the elements in the vector as follows:

```
type Name <floor,...,ceiling>
```

where `type` is the type of each element, `floor` is the smallest number of elements in the vector, and `ceiling` the largest. Note that for each vector the number of elements in the vector is prepended to the vector as an unsigned integer, using as many bytes as necessary to express ceiling (the length of the largest possible vector).

We define the following structure to represent a variable-length string of bytes:

```
struct {
uint32 length;
uint8 data<0, ..., 2^32 -1>
} ByteString
```

where:

- `length`: An unsigned 32-bit integer indicating the number of bytes that follow.
- `data`: A sequence of bytes (octets).

Note that for the `ByteString` structure, the data field is not serialized as a vector (with the length prepended), as the length is explicitly given by the first field.

B.1 Negotiation Phase

The negotiation phase begins with the exchange of messages to negotiate the key-exchange, hash algorithms and versions to be used throughout the protocol. In addition, the server sends the certificate necessary to validate the public-key of the server.

B.1.1 Client Association Message

The negotiation phase begins with the client sending the first negotiation message, with the following structure. The description of each field can be found below:

```
struct {
uint8 msg_type = 0x01;
FragmentInfo f;
uint8 client_version;
uint8 client_kdf_algs<0,...,255>;
uint8 client_hash_algs<0,...,255>;
uint8 client_kex_algs<0,...,255>;
uint8 client_mac_algs<0,...,255>;
uint256 nonce;
} ClientNegotiation
```

- `client_version`: An unsigned 8-bit integer indicating the highest supported version of ANTP that the client supports.
- `client_kdf_algs`: An ordered list of unsigned 8-bit integers representing the preferred key-derivation functions supported by the client.
- `client_hash_algs`: An ordered list of unsigned 8-bit integers representing the preferred hash algorithms supported by the client.
- `client_kex_algs`: An ordered list of unsigned 8-bit integers representing the preferred key-exchange algorithms supported by the client.
- `client_mac_algs`: An ordered list of unsigned 8-bit integers representing the preferred MAC schemes supported by the client.
- `nonce`: An unsigned 256-bit integer.
- `msg_type`: A unsigned byte of value $0x01$ indicating the `ClientAssoc` message.

B.1.2 Server Association Message

The negotiation phase continues with the server processing the `ClientAssoc` message and sending the `ServerAssoc` message, with the following structure:

```
struct {
uint8 msg_type = 0x02;
FragmentInfo f;
uint8 server_version;
uint8 server_kdf_algs<0,...,255>;
uint8 server_hash_algs<0,...,255>;
```

```
uint8 server_kex_algs<0,...,255>;
uint8 server_mac_algs<0,...,255>;
ByteString server_cert;
ByteString opaque1
} ServerNegotiation
```

- `server_neg`: A unsigned byte of value 0*x*02 indicating the `ServerAssoc` message.
- `server_version`: An unsigned 8-bit integer indicating the highest supported version of the authentication protocol that the server supports.
- `server_kdf_algs`: An ordered list of unsigned 8-bit integers representing the preferred key-derivation functions supported by the server.
- `server_hash_algs`: An ordered list of unsigned 8-bit integers representing the preferred hash algorithms supported by the server.
- `server_kex_algs`: An ordered list of unsigned 8-bit integers representing the preferred key-exchange algorithms supported by the server.
- `server_mac_algs`: An ordered list of unsigned 8-bit integers representing the preferred MAC schemes supported by the server.
- `server_cert`: The certificate containing the server public-key. Note that the public-key corresponds to the key-exchange algorithm negotiated with the two ordered lists `client_kex_algs` and `server_kex_algs`.
- $\texttt{opaque}_1$ An encrypted value created by the server, opaque to the client.

B.2 The Key-Exchange Phase

The key-exchange phase establishes secret-key material, and implicitly authenticates both the key-exchange and negotiation phases to the client.

B.2.1 Client Key Exchange Message

The key-exchange phase begins with the client sending the `ClientKey` message, with the following structure and description:

```
struct {
uint8 msg_type = 0x03;
FragmentInfo f;
uint8 neg_version;
uint8 neg_kdf;
uint8 neg_hash;
uint8 neg_kex;
uint8 neg_mac;
ByteString opaque1
ByteString kex_mat
} ClientKEX
```

- `msg_type`: A unsigned byte of value 0x03 indicating the ClientKEX message.
- `neg_version`: unsigned 8-bit integer describing the negotiated version of the protocol that the parties will be using.
- `neg_kdf`: An unsigned 8-bit integer describing the negotiated KDF that the protocol will be using.
- `neg_hash`: An unsigned 8-bit integer describing the negotiated hash algorithm that the protocol will be using.
- `neg_kex`: An unsigned 8-bit integer describing the negotiated key- exchange algorithm that the protocol will be using.
- `neg_hash`: An unsigned 8-bit integer describing the negotiated MAC algorithm that the protocol will be using.
- $\texttt{opaque}_1$: The opaque value sent in the `ServerAssoc` message.
- `kex_mat`: The public key exchange material.

B.2.2 Server Key Exchange Message

The server now processes the ClientKEX message to compute the shared secret key. The server then produces a second opaque encryption, this time of the key *k*, and generates a MAC tag authenticating the `ClientKey` and `ServerKey` messages. The structure and description of the `ServerKey` message is as follows:

```
struct {
uint8 msg_type = 0x04;
FragmentInfo f;
ByteString opaque2
ByteString mac_tag
} ServerKEX
```

- `msg_type`: A unsigned byte of value 0x04 indicating the ServerKEX message.
- $\texttt{opaque}_2$: A second encrypted value created by the server, opaque to the client.
- `mac_tag`: The MAC of the concatenated hash value, `ClientKey`, and `ServerKey` messages using the agreed key. The length of the tag is known to both parties based on the negotiated hash function, and clients MUST check that the received `mac_tag` has the correct length.

B.3 Time Synchronization Phase

The Time-Synchronization Phase is for the client to request synchronization from a server that has previously been authenticated and established a shared secret key.

B.3.1 Client Request Message

The Time Synchronization phase begins with the client computing the NTP packet as specified in the SNTP standards, and additionally completing the `ClientReq` extension as structured and described below:

```
struct {
uint8 msg_type = 0x05;
FragmentInfo f;
uint8 neg_kdf;
uint8 neg_hash;
uint8 neg_kex;
uint8 neg_mac;
uint256 nonce;
ByteString opaque2
uint8 AccuracyFlag flag
} ClientRequest
```

- `msg_type`: A unsigned byte of value 0x05 indicating the ClientRequest message.
- `neg_kdf`: An unsigned 8-bit integer describing the negotiated KDF that the protocol will be using.
- `neg_hash`: An unsigned 8-bit integer describing the negotiated hash algorithm that the protocol will be using.
- `neg_kex`: An unsigned 8-bit integer describing the negotiated key- exchange algorithm that the protocol will be using.
- `neg_mac`: An unsigned 8-bit integer describing the negotiated MAC algorithm that the protocol will be using.
- `nonce`: An unsigned 256-bit integer.
- $\texttt{opaque}_2$: The opaque value sent in the ServerKEX message.
- `flag`: An unsigned 8-bit integer describing whether the client requires high accuracy. Legal values are $0x01$ (the flag is set) or $0x00$ (the flag is not set).

B.3.2 Server Response Message

The server processes the client NTP request as standardized, and computes the SNTP response. If the `flag` in the `ClientReq` is $0x01$, the server immediately sends the message without a `ServerResp` extension. Afterwards, the server computes the `ServerResp` fields as described below, and attaches it as an extension to the previously computed NTP packet, sending the message to the client.

```
struct {
uint8 msg_type = 0x06;
FragmentInfo f;
ByteString mac_tag
} ServerResponse
```

$\mathsf{Encrypt}(m_0, m_1)$:	$\mathsf{Decrypt}(C)$:
$C^{(0)} \leftarrow\!\!\$\ \mathsf{AuthEnc}(k, m_0)$	$(m) \leftarrow \mathsf{AuthDec}(k, C)$
$C^{(1)} \leftarrow\!\!\$\ \mathsf{AuthEnc}(k, m_1)$	If $m = \perp_p$, then return $\perp$
If $C^{(0)} = \perp$ or $C^{(1)} = \perp$, return $\perp$	If $b = 0$, then return $\perp$
Return $C^{(b)}$	Return m

Figure 3: Encrypt and Decrypt oracles in the authenticated-encryption security experiment.

- `msg_type`: A unsigned byte of value $0x06$ indicating the `ServerResp` message.
- `mac_tag`: The MAC of the concatenated `ClientReq` and `ServerResp` messages using the derived secret-key. The length of the tag is known to both parties based on the negotiated hash function, and clients MUST check that the received `mac_tag` has the correct length.

C Cryptographic Definitions

C.1 Authenticated-Encryption Scheme

An *authenticated-encryption (AE) scheme* is a pair of algorithms $\mathsf{AE} = (\mathsf{AuthEnc}, \mathsf{AuthDec})$ described in Figure 3. Security of a AE scheme is defined via the following security game played between a challenger $\mathcal{C}$ and a polynomial-time adversary $\mathcal{A}$.

1. The challenger picks $b \leftarrow\!\!\$\ \{0,1\}$ and $k \leftarrow\!\!\$\ \{0,1\}^{\kappa}$.
2. The adversary may adaptively query the encryption oracle Encrypt and decryption oracle Decrypt which respond as shown in Figure 3.
3. The adversary outputs a guess $b' \in \{0,1\}$.

The advantage of $\mathcal{A}$ in breaking the AE scheme AE is $\mathrm{Adv}^{\mathsf{AE}}_{\mathsf{AuthEnc}}(\mathcal{A}) = \left|\Pr(b = b') - \frac{1}{2}\right|$. Note that in our use of an AE scheme, the purpose is to allow the server to regenerate per-client-state in an authenticated way. The length of all inputs to the AE scheme in each phase is public information and thus the length-hiding security property (introduced by Paterson, Ristenpart and Shrimpton [24]) is not necessary .

C.2 Public-Key Encryption Schemes

A *public-key encryption* (PKE) scheme is a tuple of algorithms $\mathsf{PKE} = (\mathsf{PKE.KeyGen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$ where: KeyGen is a probabilistic key generation algorithm taking security parameter (1^k) as input and returning a valid public-key/secret-key pair (pk, sk); Enc is an encryption algorithm taking a message and public-key (m, pk) as input and returning a ciphertext c; and Dec is a deterministic decryption algorithm, taking a ciphertext c as input, and returning plaintext m. IND-CCA security of

$\underline{\mathsf{Encrypt}(m_0, m_1)}$:	$\underline{\mathsf{Decrypt}(c)}$:
If $length.m_0 \neq length.m_1$,	$m \leftarrow \mathsf{Dec}(sk, c)$
then return $\perp$	If $c \in \vec{list}$,
$c^{(0)} \leftarrow\$ \mathsf{PKE.Enc}(pk, m_0)$	then return $\perp$
$c^{(1)} \leftarrow\$ \mathsf{PKE.Enc}(pk, m_1)$	Return m
$\vec{list} \leftarrow c^{(b)}$	
If $C^{(0)} = \perp$ or $C^{(1)} = \perp$,	
return $\perp$	
Return $c^{(b)}$	

Figure 4: Encrypt and Decrypt oracles in the IND-CCA PKE security experiment.

a PKE scheme is defined via the following security game played between a challenger $\mathcal{C}$ and a polynomial-time adversary $\mathcal{A}$.

1. The challenger picks $b \leftarrow\$ \{0,1\}$ and uses $\mathsf{PKE.KeyGen}(1^k)$ to compute (pk, sk) and sends pk to $\mathcal{A}$.
2. The adversary may adaptively query the encryption oracle Encrypt and decryption oracle Decrypt which respond as shown in Figure 3.
3. The adversary outputs a guess $b' \in \{0,1\}$.

The advantage of $\mathcal{A}$ in breaking IND-CCA security of PKE is $\mathrm{Adv}^{\mathsf{PKE}}_{\text{ind-cca}}(\mathcal{A}) = \left|\Pr(b = b') - \frac{1}{2}\right|$.

C.3 Collision-Resistant Hash Functions

A *collision-resistant hash function* is a deterministic algorithm Hash which given a key $k \in \mathcal{K}_{\mathsf{Hash}}$ (with $\log(|\mathcal{K}_{\mathsf{Hash}}|)$ polynomial in κ) and a bit string m outputs a hash value $w = \mathsf{Hash}(k, x)$ in the hash space $\{0,1\}^{\chi}$ (with χ polynomial in κ). We say that the advantage of a polynomial-time adversary $\mathcal{A}$ breaking the collision-resistance of the hash function Hash is $\mathrm{Adv}^{\mathsf{Hash}}_{coll}(\mathcal{A}) = |\Pr(\mathsf{Hash}(in) = \mathsf{Hash}(in'))|$ with $in \neq in'$.

C.4 Message Authentication Codes

A *message authentication code* (MAC) scheme is a pair of algorithms $\mathsf{MAC} = (\mathsf{MAC.KeyGen}, \mathsf{MAC.Tag})$ where: MAC.KeyGen is a probabilistic key generation algorithm taking input security parameter 1^{λ} and returning a random key k in the keyspace $\mathcal{K}$ of MAC and MAC.Tag is a deterministic algorithm that takes as input a secret key k and a arbitrary message m and returns a MAC tag τ. Security is formulated via the following game that is played between a challenger $\mathcal{C}$ and a probabilistic polynomial-time adversary $\mathcal{A}$.

1. The challenger samples $k \leftarrow\$ \mathcal{K}$.
2. The adversary may adaptively query the challenger; for each query value m_i, the challenger replies with $\tau_i = \mathsf{Tag}(k, m_i)$.
3. The adversary outputs a pair of values (m^*, τ^*) such that $m^* \notin \{m_0, ..., m_i\}$.

The adversary $\mathcal{A}$ wins the game if $\mathsf{Tag}(k, m^*) = \tau^*$, producing a MAC forgery. We define the advantage of $\mathcal{A}$ in breaking the unforgeability security property of a MAC scheme MAC under chosen-message attack is $\mathrm{Adv}^{\mathsf{MAC}}_{euf\text{-}cma}(\mathcal{A}) = \Pr(\mathsf{Tag}(k, m^*) = \tau^*)$.

C.5 Key-Derivation Function

A *key-derivation function* (KDF) is a deterministic algorithm KDF, which takes input: a source of randomness σ; optional salt s; optional context c; and output length L, will output a bit string k of length L. Security of a KDF is formulated via the following security game (we follow the KDF assumption as defined by Krawczyk [12] with simplified notation), played between a challenger $\mathcal{C}$ and a polynomial-time adversary $\mathcal{A}$.

1. $\mathcal{C}$ queries a source of key material algorithm Σ to produce (σ, α), where σ is random sample and α is auxiliary information about the distribution of σ
2. $\mathcal{C}$ chooses a random salt value s from salt distribution defined by KDF, if necessary.
3. $\mathcal{A}$ is given (α, s).
4. $\mathcal{A}$ can now arbitrarily query a KDF oracle KDF with input (c_i, L_i), and receives output $k = \mathsf{KDF}(\sigma, s, c_i, L_i)$
5. $\mathcal{A}$ at some point queries a Test oracle with input (c, L) such that $c \notin \{c_1, ..., c_i\}$.
6. $\mathcal{C}$ samples a random bit $b \in \{0,1\}$, and computes $k_0 = \mathsf{KDF}(\sigma, s, c, L)$ and $k_1 \leftarrow\$ \{0,1\}^L$.
7. $\mathcal{C}$ returns k_b to $\mathcal{A}$.
8. $\mathcal{A}$ can again arbitrarily query a KDF oracle KDF with input (c_i, L_i)such that $c \notin \{c_1, ..., c_i\}$, and receives output $k = \mathsf{KDF}(\sigma, s, c_i, L_i)$
9. $\mathcal{A}$ outputs a bit b'.

The $\mathcal{A}$ wins the game if $b' = b$. The advantage of $\mathcal{A}$ in breaking a key-derivation function KDF is $\mathrm{Adv}^{\mathsf{KDF}}_{kdf}(\mathcal{A}) = \left|\Pr(b = b') - \frac{1}{2}\right|$.

D Cryptographic Algorithms

The following section contains excerpts from an implementation specification for ANTP intended for submission to the IETF.

D.1 Authenticated-Encryption Scheme

This section discusses the functions AuthEnc and AuthDec functions described in Section 3. No interoperability is required from the authenticated encryption algorithm, as the value is entirely opaque to the client. It is critical for security, as a malicious party who can decrypt client Alice's $\texttt{opaque}_2$ value may masquerade as

the server and create valid ServerResp messages for Alice. Therefore, the server MUST use one of the following algorithms:

- AES-GCM as specified [7].
- AES-CCM as specified [34].
- AES-CBC combined with HMAC-SHA, as specified [15]

Note that AES-GCM requires an explicit counter that is never reused, and thus the server MUST generate a nonce for each new opaque value to be encrypted, and attach the nonce to the end of the encrypted opaque value. The security level (determined by the key length and algorithm choice) SHOULD meet or exceed the security level of the negotiated hash function. An additional consideration is key-lifetime. Each authenticated encryption algorithm has a maximum amount of data that can be encrypted with a key. To avoid exceeding this limit, servers SHOULD generate a new authenticated encryption key every two months (or sooner, depending on the amount of NTP traffic) and update the key. This also serves to force clients to renegotiate a new key, as the $\texttt{opaque}_2$ value will no longer decrypt correctly, forcing the server to reject time-synchronization. Additionally, a server SHOULD generate a new key whenever a new certificate is used.

Servers may also choose to gradually "phase in" use of a new key, since when the $\texttt{opaque}_2$ value in the ClientReq message is rejected, the client will restart the key exchange phase. If a large number of clients simultaneously begin key exchange the computational costs may overwhelm the server. Servers may include a small amount of metadata (like a key identifier) as part of the $\texttt{opaque}_2$ value to allow them to identify which key was used to create the opaque value. This allows the server to continue using the first key, while migrating clients over to the new key. After a fixed period of time (such as a day) the server should delete the first key.

D.2 Hash Algorithms

Following the direction set by Network Time Security [33], it is required that all parties MUST support SHA-256, MAY support SHA-384, SHOULD support SHA-512 and MUST NOT support SHA-1, MD5 or "weaker" hash algorithms. We recommend HMAC as a MAC scheme when computing mac_tag, which is HMAC-H, where H is the negotiated hash function. Note that the length of the mac_tag field is dependent on the size of the output of the hash algorithm negotiated. We use the assumption that the HMAC constructing utilizing SHA-256, SHA-384 and SHA512 form a PRF. Each Hash function represented in the client_hash_algs and server_hash_algs fields is assigned an unsigned 8-bit ID for negotiation:

- 0x00: SHA-256
- 0x01: SHA-384
- 0x02: SHA-512

D.3 Key-Exchange Algorithms

We note that in our protocol, the key-exchange algorithms are required to provide both authentication and confidentiality of the secret key material without the server using a signature algorithm. The RSA-OAEP option is an encryption-based key transport, i.e., the client chooses a random key and encrypts it with the server's public key. The ECDH (elliptic-curve Diffie-Hellman) option has the client generate an ephemeral public key, and the server's long term public key is used for key agreement. The parameters (kex_params) in the case of RSA-OAEP kex_params is the modulus size are forced by the server certificate. The client may abort if the server parameters are deemed too weak.

For interoperability, implementations MUST support RSA-OAEP with modulus size ≥ 2048 and MUST not support smaller modulus sizes. ECDH MUST be supported with the named curve secp256r1 [3]. Other curves MAY be supported, but they must provide 128-bits of security or above. Servers MUST implement the point validation steps before operating on received points (as specified for ECDH [4]).

Each key-exchange algorithm represented in the ANTP message fields is assigned an unsigned 8-bit ID for negotiation:

- 0x00: RSA-OAEP
- 0x01: ECDH

Our description of ANTP uses public key encryption of a premaster secret, which maps directly to RSA-OAEP. For ECDH, the premaster secret is the shared secret computed from the client secret and the static server public key (on the client side) or the server secret and the client ephemeral public key (on the server side).

D.4 Key Derivation Function

The ANTP protocol uses a Key-Derivation Function in the key-exchange phase to derive a shared secret-key. The KDF in ANTP is the *KDF in Counter Mode* defined in NIST SP 800-108 [5, §5.1], implemented with HMAC and the negotiated hash function. We have used the notation: $\mathsf{KDF}(Z)$ where Z is the shared secret, and omitted (for simplicity) other the inputs *context*, *label* and L where *context* and *labels* are additional input strings, and L is the output length of the KDF. Our detailed implementation specification includes these fields. Each Key-Derivation function represented in the ANTP message fields is assigned an unsigned 8-bit ID for negotiation:

- 0x00: SP800-108 KDF in Counter Mode