Authenticated Network Time Synchronization

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

Douglas Stebila
Queensland University of Technology
McMaster University
stebilad@mcmaster.ca

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 present an authenticated version of NTP, called *ANTP*, to protect against desynchronization attacks. To make ANTP suitable for large-scale deployments, it is designed to minimize server-side public-key operations by infrequently performing a key exchange using public key cryptography, then relying solely on symmetric cryptography for subsequent time synchronization requests; moreover, it does so without requiring server-side per-connection state. Additionally, ANTP ensures that authentication does not degrade accuracy of time synchronization. We measured the performance of ANTP by implementing it in OpenNTPD using OpenSSL. Compared to plain NTP, ANTP's symmetric crypto reduces the server throughput (connections/second) for time synchronization requests by a factor of only 1.6. We analyzed the security of ANTP using a novel provable security framework that involves adversary control of time, and show that ANTP achieves secure time synchronization under standard cryptographic assumptions; our framework may also be used to analyze other candidates for securing NTP.

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 [14] 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 [7] 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. A related mechanism, *HTTP Public Key Pinning* [3] also relies on accurate client time for security.

For clients who set their clocks using NTP, these security mechanisms (and others) 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 by an upstream network device and modified to present a consistently incorrect time to a victim. Such an attack on HSTS was demonstrated by Selvi [30], who provided a tool to advance the clock of victims in order to expire HSTS policies. Malhotra et al. [10] presents a variety of attacks that rely on NTP being unauthenticated, further emphasizing the need for authenticated time synchronization. (Confidentiality, however, is not a requirement for time synchronization, since all time synchronization is public. Similarly, client-to-server authentication is not a goal.)

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 [21]. Establishing pre-shared symmetric keys with billions of client PCs and other NTP-synchronizing devices seems impractical. NTPv4 introduced a public-key authentication mechanism called Autokey which has not seen widespread adoption; and unfortunately, 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 [32], which we discuss in §**??**.

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 every nine hours (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, but cumulative small-scale time changes can build in the long-term to large-scale time inaccuracies. Such a technique is demonstrated by Teichel *et al.* when attacking time-synchronization as secured by TESLA-like protocols [34]. In Windows domains (a network of computers, often in an enterprise), the domain controller provides the time with an authenticated variant of NTPv3 [13].

1.1 Contributions

We present the ANTP protocol for authenticated network time synchronization, along with results on its performance and security. ANTP protocol messages are transported in the extension fields of NTP messages. ANTP allows a server to authenticate itself to a client using public key certificates and public key exchange, and provides cryptographic assurance using symmetric cryptography that no modification of the packets has occurred in transit. Like other authenticated time synchronization protocols using public keys [32] we assume an out-of-band method for certificate validation exists, as certificate validation requires an accurate clock. We follow the direction set by the IETF Informational document "Security Requirements of Time Protocols in Packet-Switched Networks" (RFC 7384) [19] to determine what cryptographic, computational, and storage properties ANTP should achieve.

ANTP has three phases. In the *negotiation phase*, the client and server agree on which cryptographic algorithms to use; this phase would be carried out quite infrequently, on the order of monthly or less. In the *key exchange phase*, the client and server use public key cryptography to establish a symmetric key that the server will use to authenticate later time synchronization responses; this phases would also be carried out infrequently, say monthly. In the *time synchronization phase*, the client sends a time synchronization request, and the server replies with an NTP response that is symmetrically authenticated using the key established in the key exchange phase; this may be done frequently, perhaps daily or more often. Notably, the server need not keep per-client state: the server offloads any such state to the client by encrypting and authenticating it under a long-term symmetric key, and the client sends that ciphertext back to the server with each subsequent request.

The time synchronization phase of ANTP can be run in a "no-cryptographic-latency" mode: here, the server sends two response packets, the first being the unauthenticated NTP packet, and the second being the same NTP packet (with unchanged timestamps) along with the ANTP extensions providing authentication. The client measures the roundtrip time based on the unauthenticated response, but does not update its clock until authenticating the response. In this way, no time synchronization inaccuracy is added by the time required to compute the authentication tag over the outgoing timestamp. Since the latency of ANTP's time synchronization phase is nearly as fast as unauthenticated simple NTP time synchronization (only 21 microseconds slower at 50% load in our implementation as reported below), we make this mode optional

Phase	Throughput	Latency at 50% load	Latency at 90% load
ANTP – Negotiation – RSA	58 240	186 ± 26	202 ± 43
ANTP – Negotiation – ECDH	146 808	172 ± 35	233 ± 133
ANTP – Key Exchange – RSA	1 754	891 ± 121	997 ± 346
ANTP – Key Exchange – ECDH	13 210	197 ± 56	344 ± 142
ANTP – Time Synchronization	175 644	168 ± 35	230 ± 158
ANTP – All 3 phases – RSA	–	2 255 ± 587	2 646 ± 345
ANTP – All 3 phases – ECDH	–	1 325 ± 499	2 252 ± 1 172
NTP	291 926	147 ± 34	181 ± 136

Table 1: Performance results for each phase of ANTP (top), a complete 3-phase execution of ANTP (middle), and NTP (bottom). **Throughput**: mean completed phases per second. **Latency**: mean and standard deviation of the latency in microseconds of server responses at either 50% or 90% server load. All are computed over 5 trials, top and bottom over 100 seconds each; see Section 4.2 for details.

since it may be sufficiently accurate for general use.

ANTP performance. Performance constraints on time synchronization protocols are driven by the fact that time servers are heavily loaded, and must provide responses promptly. ANTP's design allows it to achieve high performance while maintaining high security. The frequently performed time synchronization phase uses only symmetric cryptography, making it only slightly more expensive than simple NTP time synchronization. Since the session key established in the key exchange phases is reused across many time synchronization phases, expensive public key operations are amortized, and can be separately load-balanced. And, as noted above, ANTP offloads state to clients, leaving the server stateless.

We implemented ANTP in OpenNTPD's [36] implementation of NTP, using OpenSSL [37] for cryptographic computations. Table 1 reports the performance of our implementation, compared with unauthenticated simple NTP. ANTP does decrease throughput and increase latency, but the impact is quite reasonable. On a single core of a server, ANTP can support 175k authenticated time synchronization phase connections per second, a factor of 1.6 fewer than the 291k unauthenticated simple NTP connections per second. Latency for time synchronization (over a 1 gigabit per second local area network) at 50% load increases from 147 microseconds for unauthenticated simple NTP to 168 microseconds for ANTP's time synchronization phase. The other two phases, negotiation and key exchange, will be performed far less frequently on average by clients. Throughput of negotiation phases is bandwidth-, not CPU-, limited. For exchange, we implemented methods: 2048-bit RSA key transport and static-ephemeral elliptic curve Diffie–Hellman key exchange using the NIST P-256 curve; as expected, both of these are substantially more expensive than time synchronization phases, but are also performed far less frequently. Details of our implementation and testing methodology, as well as more results, appear in Section 4.

Protocol	Auth. type	Security	Asymptotic analysis	Round trips
NTPv0–v2	—	—	—	1
NTPv3 sym. key	sym. key	no proof	1 hash	1
NTPv4 Autokey	pub. key	flaws (App. **??**)	$\frac{2}{n}$ pub. key, $\frac{1}{n}+1$ sym. key	4
NTS [32]	pub. key	ProVerif proof [33]	$\frac{3}{n}$ pub. key, $\frac{2}{n}+2$ sym. key	4
ANTP (Fig. 2)	pub. key (Sec. 6)	proof	$\frac{1}{n}$ pub. key, $\frac{6}{n}+2$ sym. key	3

Table 2: 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 compares well with other authentication methods for NTP, as seen in Table 2. ANTP uses fewer amortized public key operations compared to NTPv4 Autokey and NTS and has fewer rounds. NTPv3 using symmetric key operations is more lightweight, but is highly restricted in that it only supports symmetric authentication via pre-established symmetric keys, making it unsuitable for deployment with billions of devices.

Because ANTP is designed-for-purpose, it is also more efficient than applying general purpose security protocols to NTP. For example, one might consider simply applying TLS or DTLS to NTP packets to obtain authentication. Unfortunately, this results in substantial overhead compared to ANTP. For an indicative comparison, we measured the performance of Apache httpd [35] serving single small pages using OpenSSL on the same server as our ANTP results in Table 1. (Since OpenNTPD is single-threaded, we divided the number of connections per second supported by Apache/OpenSSL by the number of cores to provide a fair comparison.) For 2048-bit

RSA key transport, Apache/OpenSSL could serve 633 connections/second/core, just over one-third of the ANTP RSA key exchange phases; and for ECDHE/ECDSA key exchange, Apache/OpenSSL could serve only 1156 connections/second/core, less than a tenth of ANTP ECDH key exchange phases.

ANTP security. ANTP's design is supported by a thorough analysis of its cryptographic security using the provable security paradigm. To do so, we extend existing frameworks for key exchange and secure channels [2, 8] to develop a novel framework that handles protocols where *time* plays a central role. The adversary in our security analysis is a network attacker capable of deleting, reordering, editing, and creating messages between parties. Since our model is about time synchronization, parties in our model have local clocks, and 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.

We then show that ANTP achieves secure time synchronization as defined by our model, under standard assumptions on the security of the cryptographic primitives (key encapsulation mechanism, hash function, authenticated encryption, message authentication code, and key derivation function) used to construct the protocol.

2 Network Time Protocols

Here we review the two most commonly deployed time synchronization protocols, NTP and SNTP, as well as a recent proposal called Network Time Security [32].

2.1 The Network Time Protocol

The *Network Time Protocol (NTP)* was developed by Mills in 1985 [14], and revised in 1988, 1989, 1992 and 2010 (NTPv1 [6], NTPv2 [15], NTPv3 [16] and NTPv4 [17] respectively). NTP is designed to synchronize the clocks of machines directly connected to hardware clocks (known as *primary servers*) to 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 the attacker can only influence some minority of the queried servers.

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 OS X 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 behaviours are simpler. Note that SNTP and NTP client request messages are the same.

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

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.

SNTP has three distinct stages: (1) the creation and transmission of `req` by the client; (2) the processing of `req` by the server, and transmission of `resp`; and (3) the processing of `resp` and clock update by the client. An abstraction of the protocol behaviour can be found in Figure 1, including the client's clock update procedure. 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, omitting packet content details.

1. The client creates an SNTP `req` packet, sets `transmit_timestamp` (t_1) to $\mathsf{Now}()$ and sends the message.
2. The server creates an SNTP `resp` packet with all fields identical to the received `req`, but signalling Server mode. The server sets `originate_timestamp` to the value `transmit_timestamp` from `req`. The server also sets `receive_timestamp` (t_2) to $\mathsf{Now}()$ immediately after receipt of `req`, and sets `transmit_timestamp` (t_3) to $\mathsf{Now}()$ immediately before sending the message to the client.
3. Upon receiving `resp`, the client notes

the current time (and saves it as t_4). If `resp.originate_timestamp` is not equal to `req.transmit_timestamp`, the client aborts the protocol run. The client calculates the total round-trip time RTT and the local clock offset *offset* as in Figure 1.

(The rest of the fields in the NTP packets are 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.)

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 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 $\mathit{offset}_{actual} = t_3 + \theta_3 - t_4$. The approximated offset is computed as $\mathit{offset} = \frac{1}{2}(t_2 + t_3 - t_1 - t_4)$. When $\theta_1 = \theta_3$, then $\mathit{offset} = t_3 + \tilde{\theta_3} - t_4$ and $\mathit{offset} = \mathit{offset}_{actual}$. 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 $\frac{1}{2}|(\theta_1 - \theta_3)| \leq RTT$.

The error that a passive adversary with the ability to delay packets can 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.

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 specification of NTPv3 describes the checksum as the encryption of the NTP packet with DES, but notes that other algorithms could be negotiated. The distribution of keys and negotiation of algorithms was considered outside the scope of NTP.

NTPv4 Autokey public key authentication. 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. Autokey is designed to work over the top of authenticated NTPv3. Autokey uses MD5 and a variety of Schnorr-like [28] identification schemes to prevent malicious attacks, but as an analysis of Autokey by Röttger shows [25], 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 [18], 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) [32] is an IETF Internet-Draft that uses public-key infrastructure in order to secure time synchronization protocols such as NTP and the Precision Time Protocol (PTP) [1]. However, NTS is costly in terms of server-side public-key operations, is a four round-trip protocol, requires clients to manage public/private key pairs and digital certificates, and does not have an equivalent to ANTP's no-cryptographic-latency feature.

NTS has inherited many design choices from the Autokey protocol, in particular protocol flow, and key derivation strategy using secret server seeds. 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 $\mathit{cookie} = \mathsf{HMAC}(\mathit{server\_seed}, \mathsf{Hash}(\text{client public-key certificate}))$, 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, in the association (or negotiation) phase NTS requires the server digitally sign the `server_assoc` message, which includes the client's selection of hash and key encapsulation algorithms as well as a client nonce. The server must compute costly public-key operations over these values for each association phase. As a result, a NTS server requires three public-key operations per client to establish a shared secret *cookie*.

NTS is a work-in-progress and a future revision may be updated to address some of these issues. We previously discovered a flaw in the association phase which would allow MITM adversaries to perform negotiation downgrade attacks (draft version `-06`) and communicated our findings to the authors. This has since been fixed and we reviewed draft version `-12` for this paper.

3 Authenticated NTP

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 for the client.
- 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.

3.1 Protocol Description

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 for the authenticated encryption algorithm. This is used to encrypt and authenticate offloaded server state between phases.
- *Negotiation Phase*: The client and server communicate supported algorithms; the server sends its certificate and state C_1, an authenticated encryption (using s) of the hash of the message flow. The value C_1 will be later used to authenticate negotiation.
- *Key Exchange Phase*: The client uses a key encapsulation mechanism (KEM) based on the server's public key from its certificate to establish a shared key with the server. The client sends the KEM ciphertext and encrypted state C_1 to the server. The server derives the shared key k, then encrypts it (using s) to compute C_2. The server replies with a MAC (for key confirmation) and offloaded state C_2 (for use in the next phase).
- *Time Synchronization Phase*: The client sends a time synchronization request and includes offloaded server state C_2. The server recovers k from C_2 and uses it to derive a fresh key to authenticate the response, which the client verifies. The client can also request "no-cryptographic-latency" time synchronization, where the server will immediately reply without authentication, and then send a second message with authentication.

3.2 Design Rationale and Discussion

Of the security properties discussed in RFC 7384 [19], 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 [19] are 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. While storage costs are generally not an issue, synchronizing state between multiple servers implementing a high-volume network endpoint like `time.windows.com` is still expensive and complicated to deploy. For reliability and performance these servers are often in multiple data centers, spread across multiple geographic regions. In ANTP the server regenerates per-client state as needed. Our construction uses *authenticated encryption (AE)* in a similar manner to TLS Session Tickets [26] for session resumption, 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 mode. 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 au-

Client		**Server**
supported algorithms $\vec{alg}_C$		supported algorithms $\vec{alg}_S$ long-term secret s certificate $cert_S$ for the KEM keypair (pk_S, sk_S)
	Negotiation phase	
$\alpha \leftarrow \texttt{in-progress}$		
$n_c \leftarrow\!\!\$\, \{0,1\}^{256}$		
$m_1 \leftarrow \vec{alg}_C \| n_c$	$\xrightarrow{m_1}$	$(\mathsf{KDF}, \mathsf{Hash}, \mathsf{KEM}, \mathsf{MAC}) \leftarrow \text{negotiate}(\vec{alg}_C, \vec{alg}_S)$
		$h \leftarrow \mathsf{Hash}(m_1 \| \vec{alg}_S \| cert_S)$
		$C_1 \leftarrow \mathsf{AuthEnc}_s(01 \| h \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{KEM} \| \mathsf{MAC})$
Verify $cert_S$	$\xleftarrow{m_2}$	$m_2 \leftarrow \vec{alg}_S \| cert_S \| C_1$
$pk_S \leftarrow \text{parse}(cert)$		
	Key exchange phase	
$(\mathsf{KDF}, \mathsf{Hash}, \mathsf{KEM}, \mathsf{MAC}) \leftarrow \text{negotiate}(\vec{alg}_C, \vec{alg}_S)$		
$h \leftarrow \mathsf{Hash}(m_1 \| \vec{alg}_S \| cert_S)$		
$(e, pms) \leftarrow \mathsf{KEM.Encap}(pk_s)$		
$m_3 \leftarrow C_1 \| e$	$\xrightarrow{m_3}$	$b \| h \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{KEM} \| \mathsf{MAC} \leftarrow \mathsf{AuthDec}_s(C_1)$
		If $b \neq 01$, then $\alpha \leftarrow \texttt{reject}$ and abort
		$pms \leftarrow \mathsf{KEM.Decap}(sk_S, e)$
$k \leftarrow \mathsf{KDF}(pms)$		$k \leftarrow \mathsf{KDF}(pms)$
		$C_2 \leftarrow \mathsf{AuthEnc}_s(02 \| k \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{KEM} \| \mathsf{MAC})$
		$\tau_1 \leftarrow \mathsf{MAC}(k, h \| m_3 \| C_2)$
Verify $\tau_1 = \mathsf{MAC}(k, h \| m_3 \| C_2)$	$\xleftarrow{m_4}$	$m_4 \leftarrow C_2 \| \tau_1$
If verify fails, then $\alpha \leftarrow \texttt{reject}$ and abort		
	Time synchronization phase $p = 1, \ldots, n$	
$\alpha \leftarrow \texttt{in-progress}$		
$n_{c_2} \leftarrow\!\!\$\, \{0,1\}^{256}$		
$t_1 \leftarrow \mathsf{Now}()$		
$m_5 \leftarrow t_1 \| n_{c_2} \| C_2$	$\xrightarrow{m_5}$	$t_2 \leftarrow \mathsf{Now}()$
		$b \| k \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{KEM} \| \mathsf{MAC} \leftarrow \mathsf{AuthDec}_s(s, C_2)$
		If $b \neq 02$, then $\alpha \leftarrow \texttt{reject}$ or abort
		$t_3 \leftarrow \mathsf{Now}()$
	$\left[\xleftarrow{m_6^*}\right]$	$m_6^* \leftarrow t_1 \| t_2 \| t_3$
		$\tau_2 \leftarrow \mathsf{MAC}(k, m_5 \| t_1 \| t_2 \| t_3)$
$t_4 \leftarrow \mathsf{Now}()$	$\xleftarrow{m_6}$	$m_6 \leftarrow t_1 \| t_2 \| t_3 \| \tau_2$
$RTT \leftarrow (t_4 - t_1) - (t_3 - t_2)$		
If $RTT > E$, then $\alpha \leftarrow \texttt{reject}$ and abort		
Verify $\tau_2 = \mathsf{MAC}(k, m_5 \| t_1 \| t_2 \| t_3)$		
If verify fails, then $\alpha \leftarrow \texttt{reject}$ and abort		
$offset = \frac{1}{2}(t_3 + t_2 - t_1 - t_4)$		
$time_p \leftarrow \mathsf{Now}() + offset$		
$\alpha \leftarrow \texttt{accept}_p$		
If $p = n$, then terminate		

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 $\mathsf{KEM.Decap}$ or AuthDec fails for any ANTP server, the server simply stops processing the message, aborts, and allows the client to time-out. If certificate validation fails, the client aborts the protocol run.

thenticated protocol is also bound by the total round-trip time of the time synchronization phase.

Since cryptographic computations over the synchronization messages adds asymmetrically to propagation time, it introduces error in the approximation of propagation time $\tilde{\theta}_3$, so authentication operations degrade the accuracy of the `transmit_timestamp` in the `resp`. As noted above, ANTP includes a "no-cryptographic-latency" mode to reduce 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` to set its clock, but only after verifying authentication with the ANTP `ServerResp`, aborting if authentication fails, if either message wasn't received, or if messages were received in incorrect order. Here, cryptographic processing time does not introduce asymmetric propagation time. (The TESLA broadcast authentication protocol of Perrig et al. [23] delays authentication as well, to improve efficiency rather than accuracy as in ANTP.)

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 a *key encapsulation mechanism* for establishing the shared secret key. We allow either static-ephemeral elliptic curve Diffie-Hellman key exchange or key transport using RSA public key encryption. While one might ordinarily avoid use of RSA or static-ephemeral DH for key exchange since they do not provide forward secrecy, this is not a concern for ANTP since we do not need confidentiality as the contents of the messages (time synchronization data) are public.

Key freshness and reuse. ANTP allows multiple time synchronization phases for each session using the same shared secret key k but with a new nonce in each Time Synchronization Phase to prevent replay attacks and ensure uniqueness of the protocol flow. This reuse can continue until either the client restarts the negotiation phase or the server rotates public keys or authenticated encryption keys.

Denial of service attacks. Against a man-in-the-middle, some types of denial-of-service (DoS) attacks are unavoidable, as the adversary may always drop messages.

Amplification attacks can be of concern. 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. Based on the experimental results in Table 1, the negotiation and time synchronization phases have less than a 1:2 ration of attacker work to server work. As for the key exchange phase, the server performs a public key operation while a malicious client may not. However, a server under attack can temporarily stop responding to key exchange requests while still responding to time synchronization requests, and since most honest clients will perform key exchange infrequently, their service will not be denied.

Another amplification can be caused by the no-cryptographic-latency feature, since two response packets are sent for each request. This mode can be turned off during attack, the server indicating with a flag that it does not (currently) support this feature.

Finally, in the negotiation phase the server's response is also considerably larger than the client request (because it includes a certificate), 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 this must be done once, and we assume that the client has some out-of-band method for establishing the trustworthiness of public-keys, perhaps using OCSP [27] with nonces to ensure freshness of responses, by the user manually setting the time for first certificate validation, or shipping a trusted certificate with the operating system. 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.

ANTP to NTP downgrade. ANTP servers are also NTP servers, since ANTP is implemented as an NTP extension. This eases deployment; older clients can continue using NTP, while newer clients can use ANTP. Note that a network adversary can drop the ANTP extension from the request, and the server will respond with NTP (having interpreted the request as NTP). For this reason, clients that send an ANTP request must only update their clock based on a valid ANTP response, and ignore NTP responses. For similar reasons, clients are not recommended to implement a fall back from ANTP to NTP.

4 Implementation and Performance

Here we describe our instantiation of ANTP in terms of cryptographic primitives used as well as its implementation and performance testing.

4.1 Instantiation and Implementation

We instantiate ANTP using the following cryptographic algorithms. We use AES128-GCM as the symmetric encryption algorithm for the server to encrypt and decrypt state, SHA-256 as the hash algorithm, and HMAC-SHA256 and HKDF-SHA256 as the MAC and key derivation functions respectively. We support two key encapsulation mechanisms, RSA key transport and static-ephemeral elliptic curve Diffie-Hellman:

- RSA key transport: In KeyGen, the public key and secret key are a 2048-bit RSA key pair. Encap is defined by selecting a key $k \leftarrow\!\$\, \{0,1\}^{128}$ and encrypting k using the RSA public key with RSA-PKCS#1.5 encryption; Decap performs decryption with the corresponding RSA secret key.
- Static-ephemeral elliptic curve Diffie–Hellman: Let P be the generator (base point) of the NIST-P256 elliptic curve group of prime order q. In KeyGen, the secret key is $sk \leftarrow\!\$\, \mathbb{Z}_q$ and the public key is $pk = sk \cdot P$, where $\cdot$ denotes scalar-point multiplication. In Encap, select $r \leftarrow\!\$\, \mathbb{Z}_q$ and compute $c \leftarrow r \cdot P$ and $k \leftarrow \mathsf{KDF}(c \| \mathsf{X}(r \cdot pk))$. In Decap, compute $\mathsf{KDF}(c \| \mathsf{X}(sk \cdot c))$. KDF is HMAC-SHA256 and $\mathsf{X}(Q)$ gives the x-coordinate of elliptic curve point Q. This is the ECIES-KEM [31] which is IND-CCA secure under the elliptic curve discrete logarithm assumption in the random oracle model [4].

We implemented ANTP by extending OpenNTPD version 1.92 [36]. Our implementation relies on OpenSSL version 1.0.2f [37] for its cryptographic components; notably, this version included a recent high-speed assembly implementation of the NIST P-256 curve.

4.2 Performance

Methodology. We collected performance measurements for each of the negotiation, key exchange, and time synchronization phases. We wanted to know the maximum number of connections per second that could be supported in each phase, as well as the latency a client would experience for a typical server. For comparison we also collected performance measurements for unauthenticated NTP time synchronization phases.

Our experiments were carried out between two machines acting as clients, and a single server machine running ANTP. The server had an Intel Core i7-4770 CPU running at 3.40GHz with 15.6 GiB of RAM; we used two similar client machines, which in our experiments were always sufficient to saturate the server. The clients and server were connected over an isolated 1 gigabit local area network. The server was running Linux Mint 17.2 with no other software installed.

It is important to note that OpenNTPD is not multi-threaded, so the OpenNTPD server process runs on a single core, regardless of the number of cores on the machine. As the key exchange phase is CPU bound, in a threaded server implementation we expect key exchange phase throughput to increase linearly with the number of CPU cores until bandwidth is saturated.

For testing throughput (connections/second), we used our own multi-threaded UDP flooding benchmarking tool that sends static packets and collects the number of responses, the average latency of those responses, and the number of dropped packets. We tuned the number of queries per second to ensure that the server's (single) core had around 95% utilization, and that more client packets were sent than being processed, but not so many more that performance became degraded (i.e., the server dropped less than 1% of packets being received per second).

For testing individual phase latency, we again used our UDP benchmarking tool, this time measuring latency of a subset of connections while maintaining a particular background ANTP load at the server (either 50% or 90% of supported throughput), to measure the latency a client would experience at an unloaded or loaded server.

For testing total protocol runtime, we instrumented the OpenNTPD client to report the runtime of a single complete (all three phases) ANTP synchronization, again with background ANTP load as above.

Results – individual phases.

Table 1 shows the results of each phase. Results reported are the average of 5 trials. For throughput and individual phase latency, each trial was run for 100 seconds. For throughput, Table 1 reports the number of response packets received at the client machine.

Negotiation phases. The lower throughput of RSA and ECDH negotiation messages (compared to NTP) is due to larger message size of ANTP messages, as network bandwidth was saturated for this measurement. Latency for ECDH negotiation at 90% load is higher compared to RSA negotiation at 90% load; at that load level, a much larger number of ECDH packets are being sent than RSA packets, so CPU load in the ECDH is higher even though they have the same bandwidth consumption, leading to higher latency for ECDH negotiation.

Key exchange phases. As expected, server key exchange throughput is higher when ECC is used for public key operations compared to RSA. This difference is explained by the relative costs of the underlying cryptographic operations: using OpenSSL's `speed` command for benchmarking individual crypto operations, the runtime of ECC NIST P-256 point multiplication is 8.62× faster than RSA 2048 private key operations, whereas we observe a 7.54× improvement in throughput for ANTP's ECDH key exchange over ANTP's RSA key exchange. Latency for RSA key exchange is approximately 2.9 times that of ECDH key exchange at 90% load.

Time synchronization phases. While ANTP time synchronization phases are more computationally intensive than unauthenticated NTP, throughput is reduced by only a factor of approximately 1.6. Since this phase is CPU bound, we expect a multi-threaded server implementation to increase ANTP throughput. Latency increase for ANTP at 50% load is only about 14% and at 90% load is about 27%.

Results and extrapolation – all 3 phases. Since each client makes a full 3-phase time synchronization (negotiation, followed by key exchange, followed by time synchronization) relatively infrequently, it does not make sense to measure server throughput for complete 3-phase time synchronizations. We did measure latency of a 3-phase time synchronization to note the performance that a client would perceive on its initial synchronization. As expected, the total runtime of a client exceeds the sum of the latencies from each individual phase due to the client performing its own cryptographic operations.

It is interesting to note that latency slows as the server approaches load capacity. Future work on OpenNTPD and other NTP servers could include optimizations to reduce latency and improve time synchronization accuracy under increasing load.

We can extrapolate from the individual phase results the client pool that ANTP could feasibly support running on the same hardware. For example, Windows by default polls time servers every 9 hours [12]. Assuming this is true for all clients (and that the clients synchronize uniformly across the period) 175,644 time synchronization requests per second would correspond to a pool of 5,755,502,592 clients.

ANTP clients would choose how often to restart the negotiation phase and we recommend doing so periodically to ensure the attack window from exposure of the symmetric key is limited. If keys are re-exchanged monthly, this is a ratio of 1:1:1440 for expected negotiation, key exchange, and time synchronization messages, which increases to 1:1:8640 if clients re-exchanged every 6 months. From these or other expected ratios, one could extrapolate the expected performance impact of using ANTP over NTP.

5 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 Precision Time Protocol. It builds on both the Bellare–Rogaway model [2] for *authenticated key exchange* and the Jager et al. framework for *authenticated and confidential channel establishment* [8]. Neither of those models however includes time. Schwenk [29] recently proposed a framework for modelling time in provable security analysis of protocols such as Kerberos: time is a global parameter and each party may query a *time oracle* to receive the time from the global time counter.

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.

5.1 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. In addition, we denote with T and T_c the full transcript and server-session maintained client transcript T_c.

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

- $\rho \in \{\texttt{client}, \texttt{server}\}$: the role of the party.
- $id \in \{1, ..., n_p\}$: the identity of the party.
- $pid \in \{1, \ldots, n_p\}$: the believed identity of the partner.
- $\alpha \in \{\texttt{accept}, \texttt{reject}, \texttt{in-progress}\}$: the session status.
- $k \in \{0,1\}^{128}$: the session key.
- $T_c \in \{\{0,1\}^*, \emptyset\}$: if $\rho = \texttt{server}$, the transcript of client messages, otherwise $T_c = \emptyset$.
- $T \in \{0,1\}^*$: the transcript of messages sent and received.
- $time \in \mathbb{N}$: 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 $\pi_i^s.\mathit{time} = t$. 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, modelling the party reading its current time from memory; immediately before the ℓth such call to the $\mathsf{Now}()$ procedure, the session's $\pi_i^s.\mathit{time}$ 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 $\pi_i^s.\mathit{time}$ 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.

5.2 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 $|\pi_i^s.\mathit{time} - \pi_j^t.\mathit{time}| > \delta$, (where π_j^t is the partner of the session π_i^s such that $\pi_j^t.\mathit{id} = \pi_i^s.\mathit{pid}$, and δ is an accuracy margin) or cause a session π_i^s to accept a protocol run without having a matching 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.

5.2.1 Matching Conversations and Authentication

Authentication is defined similarly to the approach of Bellare and Rogaway [2], by use of matching conversations. We use the variant of matching conversations employed by Jager *et al.* [8], and modify the definition to reflect client authentication of stateless servers.

Definition 1 (Matching Conversations)**.** *We say a session* π_i^s matches *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* T *and* T', *we say that* T *is a* prefix *of* T' *if* $|T| \neq 0$ *and* T' *is identical to* T *for the first* $|T|$ *messages in* T'. *Two transcripts* T *and* T' prefix-match *if* T *is a prefix of* T', *or* T' *is a prefix of* T.

Prefix-matching prevents an adversary from trivially winning the game by dropping the last protocol message after a session has accepted. Note that since our focus is clients authenticating stateless servers,

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

- $\pi_i^s.\alpha = \texttt{accept}$;
- $\pi_i^s.\rho = \texttt{client}$;
- *no* $\mathsf{Reveal}(i,s)$ *or* $\mathsf{Reveal}(j,t)$ *queries were issued before* $\pi_i^s.\alpha \leftarrow \texttt{accept}$ *and* $\pi_j^t.T$ *prefix-matches* $\pi_i^s.T$;
- *no* $\mathsf{Reveal}(i,s')$ *queries were issued before* $\pi_i^s.\alpha \leftarrow \texttt{accept}$ *and* $\pi_i^{s'}.T_c = \pi_i^s.T_c$
- *no* $\mathsf{Corrupt}(j)$ *query was ever issued before* $\pi_i^s.\alpha \leftarrow \texttt{accept}$, *where* $j = \pi_i^s.\mathit{pid}$;

but there exists no session π_j^t *such that* π_i^s *matches* π_j^t.

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

The first Reveal condition prevents $\mathcal{A}$ from trivially winning the game by accessing the session key of the Test session. Similarly the Corrupt condition prevents $\mathcal{A}$ from trivially winning by decrypting the premaster secret with the session peer's public-key. the possibility exists for an adversary to trivially win the game by replaying client messages to a second session and querying the second session with Reveal. Disallowing Reveal queries in general is clearly too restrictive, so we prevent this in the second Reveal condition by disallowing Reveal queries to server sessions sharing client contributions.

5.2.2 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*

- $\pi_i^s.\alpha = \texttt{accept}$*;*
- $\pi_i^s.\rho = \texttt{client}$*;*
- *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 $|\pi_i^s.time - \pi_j^t.time| \leq \delta$.

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:*

- $\pi_i^s.\alpha = \texttt{accept}$*;*
- $\pi_i^s.\rho = \texttt{client}$
- *$\mathcal{A}$ made no* $\mathsf{Corrupt}(j)$ *query before $\pi_i^s.\alpha \leftarrow \texttt{accept}$;*
- *$\mathcal{A}$ made no* $\mathsf{Reveal}(i,s)$ *or* $\mathsf{Reveal}(j,t)$ *query before $\pi_i^s.\alpha \leftarrow \texttt{accept}$ and π_j^t matches 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* $\mathsf{Tick}(i,s,\sum_{i=1}^{\Delta}\Delta'_\ell)$*, 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*
- $|\pi_i^s.time - \pi_j^t.time| > \delta$.

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

5.3 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.

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.

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 $\texttt{in-progress}$.

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 matches 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 $\mathrm{Adv}_{\mathcal{T},\delta}^{\text{multi-time}}(\mathcal{A})$.

6 Security of ANTP

In this section we present our correctness and security theorems on ANTP.

6.1 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 send 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. □

6.2 Security

Security of a single 3-phase execution of ANTP in the sense of Definition 4 is given by Theorem 2 below. Security of multiple phases in the sense of Definition 5 follows with a straightforward adaptation; details appear in Appendix B.

Intuitively, the bound on the possible error that an $\mathcal{A}$ can introduce without altering packets is as 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 $E \geq RTT$).

Theorem 2 (Security of ANTP)**.** *Fix $E \in \mathbb{N}$ and let λ be the length of the nonces in m_1 and m_5 ($\lambda = 256$). Assuming the key encapsulation mechanism* KEM *(with keyspace* $\mathsf{KEM}.\mathcal{K}$*) 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*

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

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 $\mathrm{Adv}^{\mathsf{AuthEnc}}_{\mathsf{AE}}(\mathcal{A})$, $\mathrm{Adv}^{\mathrm{ind\text{-}cca}}_{\mathsf{KEM}}(\mathcal{A})$, $\mathrm{Adv}^{\mathrm{coll}}_{\mathsf{Hash}}(\mathcal{A})$, $\mathrm{Adv}^{\mathrm{euf\text{-}cma}}_{\mathsf{MAC}}(\mathcal{A})$, and $\mathrm{Adv}^{\mathrm{kdf}}_{\mathsf{KDF}}(\mathcal{A})$ are given in Appendix A.

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 session matching) 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 $\mathrm{Adv}^{\mathrm{time}}_{\mathsf{ANTP}_E,E}(\mathcal{A}) \le \mathrm{Adv}^{\mathrm{auth}}_{\mathsf{ANTP}_E}(\mathcal{A})$.

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 abort the simulation if any nonce is used in two different sessions by client instances. There are at most $2n_sn_p$ nonces used by client instances, each λ bits. The probability that a collision occurs among these values is $(2n_sn_p)^2/2^\lambda$, so: $\Pr(break_0) \le \Pr(break_1) + \frac{n_s^2n_p^2}{2^{\lambda-2}}$.

Game 2. Here, we guess the first client session to accept maliciously, aborting if incorrect. We select randomly from two indices $(i,s) \leftarrow\$ \{1,\ldots,n_p\} \times \{1,\ldots,n_s\}$ and abort if π_i^s is not the first session to accept maliciously. Now the challenger responds to $\mathsf{Reveal}(i,s)$ queries (if $\pi_i^s.\alpha = \mathtt{accept}$) by aborting the game, as it follows that the guessed session cannot accept maliciously. There are at most n_pn_s client sessions, and we guess the first session to accept maliciously with probability at least $1/n_pn_s$, so $\Pr(break_1) \le n_pn_s \Pr(break_2)$.

Game 3. Here we guess the partner session to π_i^s, by selecting from two indices $(j,t) \leftarrow\$ \{1,\ldots,n_p\} \times \{1,\ldots,n_s\}$ and abort if π_j^t is not the partner session to π_i^s.Now, the challenger answers $\mathsf{Corrupt}(j)$ and $\mathsf{Reveal}(j,t)$ queries before $\pi_i^s.\alpha \leftarrow \mathtt{accept}$ by aborting the game, as it follows that the guessed session cannot accept maliciously. There are at most n_pn_s server sessions, and we guess the partner of the first session to accept maliciously with probability at least $1/n_pn_s$, so $\Pr(break_2) \le n_pn_s \Pr(break_3)$.

Game 4. Here we abort if a hash collision occurs, by computing all hash values honestly and aborting if there exists two evaluations $(in, \mathsf{Hash}(in)), (\hat{in}, \mathsf{Hash}(\hat{in}))$ such that $in \ne \hat{in}$ but $\mathsf{Hash}(in) = \mathsf{Hash}(\hat{in})$. The simulator interacts with a Hash-collision challenger, outputting the collision if found. Thus: $\Pr(break_3) \le \Pr(break_4) + \mathrm{Adv}^{\mathrm{coll}}_{\mathsf{Hash}}(\mathcal{B}_3^{\mathcal{A}})$.

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 simulates Game 4 identically, except to interact with an AE challenger in the following way: When P_j needs to run $\mathsf{AuthEnc}$ or $\mathsf{AuthDec}$, $\mathcal{B}_4^{\mathcal{A}}$ uses its oracles to compute the required value. 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 $\mathsf{AuthDec}$ oracle is not $\perp$, this corresponds to a ciphertext forgery, and thus: $\Pr(break_4) \le \Pr(break_5) + \mathrm{Adv}^{\mathsf{AE}}_{\mathsf{AuthEnc}}(\mathcal{B}_4^{\mathcal{A}})$.

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 $pms' \leftarrow\$ \mathsf{KEM}.\mathcal{K}$, rather than the pms that was encapsulated using $\mathsf{KEM.Encap}$ 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 KEM encrypted values via plaintext, thus breaking IND-CCA security of the key encapsulation mechanism.

We construct a simulator $\mathcal{B}_5^{\mathcal{A}}$ that interacts with a KEM 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 KEM ciphertext e for the session π_i^s by obtaining a challenge (e, pms) from its KEM challenger.

- $\mathcal{B}_5^{\mathcal{A}}$ 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 Decap 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 Corrupt(j) query because of Game 3.

When the random bit b sampled by the KEM ind-cca challenger is 0, pms is truly the decapsulation of the ciphertext e, in which case $\mathcal{B}_5^{\mathcal{A}}$ perfectly simulates of Game 5. When $b = 1$, pms is random and independent of e, in which case $\mathcal{B}_5^{\mathcal{A}}$ perfectly simulates Game 6. Observe that $\mathcal{B}_5^{\mathcal{A}}$ never asks the challenge ciphertext e to its decapsulation oracle.

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

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 $\mathsf{KEM}.\mathcal{K}$. 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; and 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 $\Pr(break_6) \leq \Pr(break_7) + \mathrm{Adv}^{\text{kdf}}_{\mathsf{KDF}}(\mathcal{B}_6^{\mathcal{A}})$.

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 $z = k \| \mathsf{KDF} \| \mathsf{Hash} \| \mathsf{KEM} \| \mathsf{MAC}$ and submits (z, z') to its 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 AuthEnc or 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 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 $\pi_{i^*}^{s^*}$ 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 $\pi_i^s.\alpha \leftarrow \texttt{accept}$, $\mathcal{B}_8^{\mathcal{A}}$ checks P_j to see if there is a matching session. Since by Game 1 all protocol flows are unique (by unique nonces), if P_j has no matching session 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 a probability negligibly close to 0 of forcing a client session $\pi_{i^*}^{s^*}$ to accept maliciously in Game 9. We briefly summarize the changes in games.

1. Nonces no longer collide for honest parties. Each transcript $\pi_i^s.T$ will have unique honest matching session π_j^t.
2. Guess target session; $\mathcal{C}$ aborts if Reveal(i,s) query asked.
3. Guess partner session; $\mathcal{C}$ aborts if Corrupt(j) or Reveal(j,t) query asked.
4. Hash values no longer collide for honest parties.

Note h is now unique for each negotiation phase, via Game 1.

5. C_1 is not forged in session π_j^t.
6. Replace premaster secret pms in target session π_i^s with a random value, rather than key encapsulated in KEM ciphertext e. Note k is unique and computed via shared secret data. Thus
7. Replace k with uniformly random data of same length when computing τ. Thus verification of τ in Time Synchronization and Key Exchange phases is done via a uniformly random key, independent of the protocol run.
8. C_2 is not forged in session π_j^t and contains random data.
9. MAC tags in session π_i^s are not forged.

π_i^s is a target session where: no $\mathsf{Reveal}(i,s)$ or $\mathsf{Reveal}(j,t)$ queries were issued before $\pi_i^s.\alpha \leftarrow \texttt{accept}$; no $\mathsf{Corrupt}(j)$ query was ever issued before $\pi_i^s.\alpha \leftarrow \texttt{accept}$, 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. □

7 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 Precision 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.

Acknowledgements

We thank Gleb Sechenov at QUT for assistance in setting up the network for the experiments. B.D. and D.S. supported by Australian Research Council (ARC) Discovery Project grant DP130104304. Part of this work performed while B.D. was an intern at Microsoft Research.

References

[1] IEEE Std 1588 for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems Networked Measurement and Control Systems. Technical report, IEEE Instrumentation and Measurement Society, 2008.

[2] 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.

[3] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), Apr. 2015.

[4] D. Galindo, S. Martin, and J. L. Villar. Evaluating elliptic curve based KEMs in the light of pairings. Cryptology ePrint Archive, Report 2004/084, 2004. `http://eprint.iacr.org/2004/084`.

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

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

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

[8] 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, Heidelberg, Aug. 2012.

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

[10] A. Malhotra, I. E. Cohen, E. Brakke, , and S. Goldberg. Attacking the Network Time Protocol. In *NDSS 201616*. Internet Society, February 2016.

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

[12] Microsoft Corporation. Windows Time Service Tools and Settings. Microsoft Developer Network, May 2012. `https://msdn.microsoft.com/de-de/library/cc773263%28v=ws.10%29.aspx#w2k3tr_times_tools_uhlp`.

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

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

[15] D. Mills. Network Time Protocol (version 2) specification and implementation. RFC 1119 (INTERNET STANDARD), Sept. 1989. Obsoleted by RFC 1305.

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

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

[18] 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.

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

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

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

[22] 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, Heidelberg, Dec. 2011.

[23] A. Perrig, R. Canetti, J. Tygar, and D. Song. The TESLA broadcast authentication protocol. *RSA CryptoBytes*, 5(Summer), 2002.

[24] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347 (Proposed Standard), Apr. 2006. Obsoleted by RFC 6347, updated by RFCs 5746, 7507.

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

[26] 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.

[27] 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.

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

[29] J. Schwenk. Modelling time for authenticated key exchange protocols. In M. Kutylowski and J. Vaidya, editors, *ESORICS 2014, Part II*, volume 8713 of *LNCS*, pages 277–294. Springer, Heidelberg, Sept. 2014.

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

[31] V. Shoup. ISO/IEC 18033-2:2006: Information technology – security techniques – encryption algorithms – part 2: Asymmetric ciphers. Technical

report, 2006. See also http://shoup.net/iso/std6.pdf.

[32] D. Sibold, S. Röettger, and K. Teichel. Network Time Security. IETF Internet-Draft, January 2016. https://tools.ietf.org/html/draft-ietf-ntp-network-time-security-12.

[33] K. Teichel, D. Sibold, and S. Milius. First Results of a Formal Analysis of the Network Time Security Specification. In *Security Standardisation Research*, pages 218–245. Springer, 2015.

[34] K. Teichel, D. Sibold, and S. Milius. An Attack Possibility on Time Synchronization Protocols Secured with TESLA-Like Mechanisms, 2016. https://www8.cs.fau.de/staff/milius/AttackPossibilityTimeSyncTESLA.pdf.

[35] The Apache Software Foundation. Apache httpd version 2.4.18, December 2015. https://httpd.apache.org/.

[36] The OpenBSD Project. OpenNTPD version 5.7p4, March 2015. http://www.openntpd.org/.

[37] The OpenSSL Project. OpenSSL version 1.0.2f, January 2016. https://www.openssl.org/.

A Cryptographic Building Blocks

A.1 Key Encapsulation Mechanism

Definition 6 (Key encapsulation mechanism)**.** *A* key encapsulation mechanism (KEM) *for a keyspace $\mathcal{K}$ is a tuple of algorithms* $\mathsf{KEM} = (\mathsf{KeyGen}, \mathsf{Encap}, \mathsf{Decap})$*:*

- $\mathsf{KeyGen}() \xrightarrow{\$} (pk, sk)$*: A probabilistic key generation algorithm that outputs a public key pk and a secret key sk.*
- $\mathsf{Encap}(pk) \xrightarrow{\$} (c, k)$*: A probabilistic key encapsulation algorithm that takes as input a public key pk and outputs a ciphertext c and a (session) key $k \in \mathcal{K}$.*
- $\mathsf{Decap}(sk, c) \rightarrow k$*: A deterministic key decapsulation algorithm that takes as input a secret key sk and ciphertext c and outputs a (session) key $k \in \mathcal{K}$ (or a distinguished error symbol $\perp$).*

A key encapsulation mechanism is correct *if*

$$\Pr\left[k = k' : \begin{matrix} (pk, sk) \leftarrow\!\!\$\, \mathsf{KeyGen}(); \\ (c, k) \leftarrow\!\!\$\, \mathsf{Encap}(pk); \\ k' \leftarrow \mathsf{Decap}(sk, c) \end{matrix}\right] = 1$$

The ind-cca *security experiment for adversary $\mathcal{A}$ against scheme* KEM *is given in Figure 3, and $\mathcal{A}$'s advantage in breaking the* ind-cca *property for* KEM *is defined as*

$$\mathrm{Adv}^{\text{ind-cca}}_{\mathsf{KEM}}(\mathcal{A}) = \left|2\Pr\left[\mathrm{Exp}^{\text{ind-cca}}_{\mathsf{KEM}}(\mathcal{A}) = 1\right] - 1\right| \ .$$

```
$\mathrm{Exp}^{\text{ind-cca}}_{\mathsf{KEM}}(\mathcal{A})$:
$(pk, sk) \leftarrow\!\!\$\, \mathsf{KeyGen}()$
$st \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{Decap}(sk,\cdot)}(pk)$
$(c^*, k_0) \leftarrow\!\!\$\, \mathsf{Encap}(pk)$
$k_1 \leftarrow\!\!\$\, \mathcal{K}$
$b \leftarrow\!\!\$\, \{0,1\}$
$b' \leftarrow\!\!\$\, \mathcal{A}^{\mathsf{Decap}(sk,\cdot\neq c)}(st, c^*, k_b)$
return $b = b'$
```

Figure 3: ind-cca security experiment for key encapsulation mechanism KEM.

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

```
$\mathsf{Decrypt}(C)$:
$m \leftarrow \mathsf{AuthDec}(k, C)$
If $m = \perp_p$, then return $\perp$
If $b = 0$, then return $\perp$
If $b = 1$, then return $m$
```

Figure 4: Encrypt and Decrypt oracles in the authenticated encryption security experiment.

A.2 Authenticated Encryption Scheme

Definition 7 (Authenticated encryption scheme)**.** *An* authenticated encryption (AE) scheme *is a pair of algorithms* AE= (AuthEnc, AuthDec) *described in Figure 4. 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 4.*
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 [22]) is not necessary .*

A.3 Collision-Resistant Hash Functions

Definition 8 (Collision-resistant hash function)**.** *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* $in \neq in'$.

A.4 Message Authentication Codes

Definition 9 (Message authentication code)**.** *A* message authentication code *(MAC) scheme is a pair of algorithms* $\mathsf{MAC} = (\mathsf{MAC.KeyGen},\ \mathsf{MAC.Tag})$ *where:* $\mathsf{MAC.KeyGen}$ *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* $\mathsf{MAC.Tag}$ *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* $\mathrm{Adv}^{\mathsf{MAC}}_{euf\text{-}cma}(\mathcal{A}) = \Pr(\mathsf{Tag}(k, m^*) = \tau^*)$*.*

A.5 Key Derivation Function

Definition 10 (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 [9] 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* $\mathrm{Adv}^{\mathsf{KDF}}_{kdf}(\mathcal{A}) = \left|\Pr(b = b') - \frac{1}{2}\right|$*.*

B ANTP Multi-Phase Security

Multi-phase security of ANTP_E can be established in a similar way to single-phase security as in Appendix **??**, 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*

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

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 matches 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. □

C Network Time Protocol Message

This section details the NTP message format as specified in [16].

```
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 message departed the local machine, according to the local clock.

D Authenticated Network Time Protocol Messages

Recall that all messages are designed to be sent in the NTP extension fields similarly to the Autokey Protocol [5]. 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 [24] 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 [20] [11]. See also Section 4.1.1.1 ”PMTU Discovery” from [24] 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.

D.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 digital certificate necessary to validate the public-key of the server. The server includes an opaque value $\texttt{opaque}_1$, which is the authenticated-encrypted value of the hash value (for authenticating the negotiation phase), the negotiated algorithms and a flag value to distinguish $\texttt{opaque}_1$ from later $\texttt{opaque}_2$ values.

D.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
```

- msg_type: A unsigned byte of value $0x01$ indicating the ClientAssoc message.
- 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.

D.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 $0x02$ 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.

D.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.

D.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 ClientKey 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.

D.2.2 Server Key Exchange Message

The server now processes the ClientKey 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 ServerKey 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.

D.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.

D.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 ClientReq 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).

D.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
```

- 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.