A Universally Composable Treatment of Network Time*

Ran Canetti† BU & TAU
Kyle Hogan‡ MIT
Aanchal Malhotra BU
Mayank Varia BU

December 29, 2017

Abstract

The security of almost any real-world distributed system today depends on the participants having some "reasonably accurate" sense of current real time. Indeed, to name one example, the very authenticity of practically any communication on the Internet today hinges on the ability of the parties to accurately detect revocation of certificates, or expiration of passwords or shared keys.

However, as recent attacks show, the standard protocols for determining time are subvertible, resulting in widespread security loss. Worse yet, we do not have security notions for network time protocols that (a) can be rigorously asserted and (b) rigorously guarantee security of applications that require a sense of real time.

We propose such notions, within the universally composable (UC) security framework. That is, we formulate ideal functionalities that capture a number of prevalent forms of time measurement within existing systems. We show how they can be realized by real-world protocols, and how they can be used to assert security of time-reliant applications — specifically, certificates with revocation and expiration times. This allows for relatively clear and modular treatment of the use of time in security-sensitive systems.

Our modeling and analysis are done within the existing UC framework, in spite of its asynchronous, event-driven nature. This allows incorporating the use of real time within the existing body of analytical work done in this framework. In particular it allows for rigorous incorporation of real time within cryptographic tools and primitives.

1 Introduction

Most existing large-scale networks, and in particular the global Internet, are predominantly asynchronous and do not require the participants to be "synchronized" with other entities in any way or have a global sense of time. In fact, this non-reliance on a common notion of time can be seen as one of the reasons for the success of the TCP/IP design.

However, as it turns out, several important mechanisms that are central to the usability of networks as a platform for communication and distributed computation do indeed require parties to have some global, common sense of real-time. Interestingly, the need for a global sense of time does not arise from the desire to provide synchronous communication, quality of service, or other "sophisticated" networking primitives. Rather, awareness to real time is often coupled with the safe use of cryptography to thwart attacks against the network.

One prevalent use of real time is in revoking, and limiting the duration of certificates for public keys. Indeed, verifying the validity of the public key of one's peer for communication is a crucial step in setting up authenticated communication, which in turn is the basis for practically any security-aware interaction on the Internet today. Setting time limit to the validity of certificates, and furthermore revoking certificates when necessary, is a crucial component in making Public-Key Infrastructure (PKI) a valid, usable basis for secure communication. Such ability, in turn, hinges on having good sense of current real time. Furthermore, not only mainframe servers need to have such ability – even low end clients need it, in fact arguably even more so than servers. Indeed, without a good sense of current time, a client cannot verify whether a certificate is valid, or whether a given certificate revocation list is the up-to-date one.

*Supported by National Science Foundation Grant #1414119 for the MACS Frontier project (bu.edu/macs). This is an extended version of [1].
†Member of CPIIS. Supported in addition by ISF grant 1532/14.
‡Work done while at Boston University.

Other uses of real time to improve security include various forms of timestamping for contracts and timing transactions in public ledgers.

It may appear that measuring real time is a relatively easy task; indeed, most computing platforms today, even low-end ones, are equipped with a built-in clock. Still, synchronizing and adjusting these clocks, and in particular reaching agreement on time in a large, asynchronous network like the Internet turns out to be non-trivial. In particular, NTP, the current IETF standard protocol for computers on the Internet to determine time [2], is rather complex. It assumes a hierarchical system of "time servers," where lower-stratum servers are assumed to have a more accurate notion of time, and higher-stratum servers determine time by querying several lower-stratum ones and performing some complex aggregation of the responses. The protocol has mechanisms for protecting from errors introduced by network delays, but is built on complete trust in the queried time servers, as well as in the authenticity of the communication. Indeed, NTP has been demonstrated to be easily subvertible, resulting in massive loss of security [3–6].

Several variants of NTP such as sntpd [7], ptpd [8], chronyd [9], OpenNTPd [10], ntimed [11], and Roughtime [12] have been proposed. These protocols offer varying degree of clock accuracy, correctness, precision and security guarantees. They have different packet semantics and a different mechanism on how the querying client chooses to update its local time, if at all, after interacting with one or potentially many time servers.

When coming to assess these proposals, it becomes evident that we don't currently have a good measure to test these proposals against. Indeed, while great many analytical works propose ways to model time (either real, global, or relative) within network protocols, and even within security protocols, we do not have a way to rigorously capture the security guarantees from a network time protocol that provably suffice for security-sensitive applications that require an agreed-upon time measurement— for instance for guaranteeing the validity of certificates in a way that, in turn, will guarantee authenticated and secure communication. (See Section 1.4 for a brief account of related work on the modeling of time.)

1.1 Our Contributions

We provide a modular, composable formalism of the security requirements from network-time protocols — or, more generally from protocols that provide a reading of real time with the assistance of other nodes over an asynchronous network. Specifically, we propose formal abstractions of secure network time, and show that:

- Our abstractions of network-time suffice for securely incorporating expiration times in certificates, as well as freshness guarantees for public certificate lists, in a way that guarantees PKI-based secure communication *even in face of an adversary who tries to subvert the measurement of time and at the same time corrupts revoked and expired certificates.*
- Our abstractions are realizable by simple protocols that mimic the behavior of authenticated NTP.

We use the Universally Composable (UC) security framework as a basis for our formalism. Indeed, the UC framework provides a general mechanism for specifying security properties of cryptographic protocols in a way that facilitates composing protocols together, and in particular guarantees that composition of secure components results in overall security of the composed protocol. Furthermore, the UC framework is geared towards analyzing the security of cryptographic protocols, which facilitates incorporating the results in this work with existing analytical results for cryptographic protocols.

Specifically, we build upon an existing analytical work by Canetti et al. that asserts, within the UC framework, the security of authentication and key exchange protocols that are based on global public-key infrastructure (PKI) [13]. We incorporate our analysis of timing consensus achieved via network time into the UC analysis of a global PKI. The combined analysis extends the security guarantees provided by [13] to the case of revocable and expirable certificates.

Our methodology of incorporating network time into existing UC protocols and functionalities is quite generic. Hence, our work paves the way toward instantiating time consensus and reaping its security benefits within other UC formalisms in a seamless fashion.

Technical and Conceptual Challenges. A priori it appears that the UC framework might be unsuitable for representing real time. Indeed, the framework is centered around modeling completely asynchronous, event-driven systems. Furthermore, in the UC formalism the basic computational elements (Turing machine instances) are activated one by

one, and the order of execution and activation of components is under total adversarial control. This is done with good reason, namely in order to provide security even against adversaries that have full control over the network; however, this structure appears to be incompatible with the modeling of real time that advances "at the same rate" within all components of a physically spread-out system. (It should be noted that this asynchronous, event-driven formalism that gives the adversary total control over the scheduling of events is not unique to the UC framework. Indeed, it is the common methodology for modeling and analyzing cryptographic protocols in general — for the same reason outlined above.)

Our first contribution is thus to propose a construct that represents global time even within such a system. The construct is simple: It is a trusted entity (formalized as a global ideal functionality) that keeps a counter. This counter is incremented adversarially by the environment, but is guaranteed to never decrease. All entities in the system have access to this counter (or, rather, some perturbed version of it, as described below) — which they treat as Time. Indeed, this adversarially incremented counter does not in any way approximate the passing of real physical time. Still, we argue that from the point of view of capturing the validity of mechanisms that use time in order to provide some security guarantees, this simple gadget is good enough. Said otherwise, any security property that is expressible and asserted within our formal framework would be preserved even when implemented in a real system that has access to real physical time.

Another set of challenges has to do with the modeling of the "imperfections" that one encounters when using the currently available mechanisms for measuring time. We consider two main methods for measuring time, each with its own imperfections: One method is measuring one's own local physical clock. This method provides fast response and relatively accurate measurement of time elapsed between events that occur at the same location; however, the response may be arbitrarily "shifted" relative to actual real time. The second method is asking one (or more) other entities in the network ("time servers") for their current time reading. This method can potentially provide reading of real time, but is susceptible to measurement errors due to network delays, spoofing attacks, and faulty servers. It may also be slow in providing a response. Indeed, a good network time protocol is one that combined these two methods in a "secure way" in order to provide a reliable reading of real time. Our goal is to capture that property.

1.2 Our Formalism in a Nutshell

We provide a brief overview of our formalism. See Figure 1 for the relationships between these primitives.

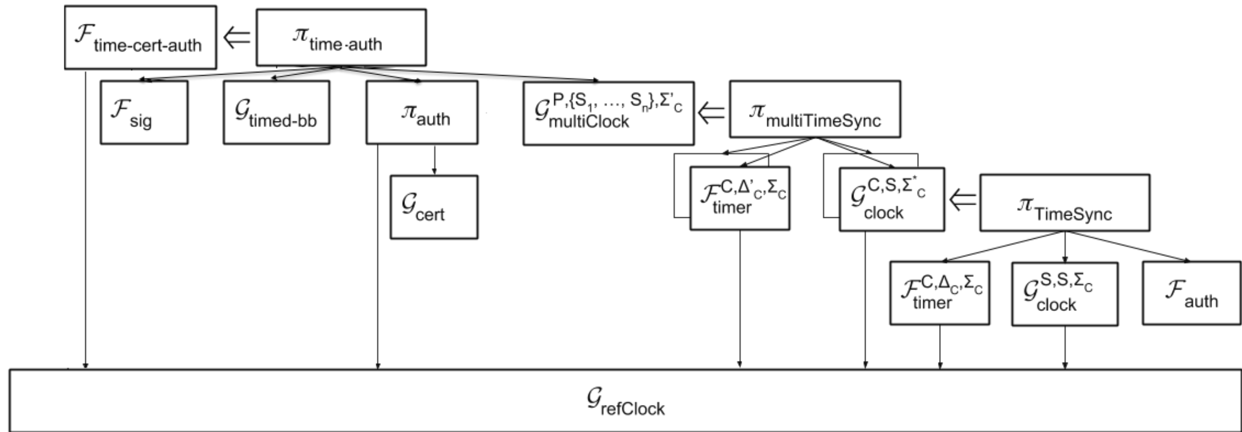


Fig. 1: Overview of our formalism, from the exact, approximate, and relative time functionalities to ideal certification with limited-time certificates. Double arrows mean "UC realizes," and single arrows mean "uses as a subroutine."

The GUC framework. Writing a specification within the Global Universal Composability (GUC) framework amounts to writing a program for an *ideal functionality* $\mathcal{F}$ that captures the expected behavior of the analyzed system π. Here $\mathcal{F}$ captures both the expected functionality and the expected security properties. Formally, system π is said to GUC-realize $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that no external environment $\mathcal{E}$ can tell whether it is interacting with $\mathcal{A}$ and π or with $\mathcal{S}$ and $\mathcal{F}$. Here $\mathcal{E}$ plays the role of a calling protocol that provides inputs to π (or $\mathcal{F}$) and obtains the outputs of π (or $\mathcal{F}$), whereas $\mathcal{A}$ controls the communication between the parties running π. The communication between $\mathcal{S}$ and $\mathcal{F}$ captures the "security imperfections" that $\mathcal{F}$ allows.

The main added feature of the GUC framework beyond the original UC framework is that it allows incorporating in the model of execution "global functionalities" that represent trusted services that exist in the system regardless of the analyzed protocol. That is, the global functionalities exist both in the ideal model for functionality $\mathcal{F}$ and in the model for executing π. This modeling allows to better capture long-term services such as public-key infrastructure (as done in [13]), or network time — as done here.

Exact and Approximate Clocks. Our first basic construct is a global ideal functionality $\mathcal{G}_{\text{refClock}}$ that provides an exact clock. Formally, it provides a non-decreasing counter that the environment can increment at will. In a sense, the clock's idealistic time serves as a reference or benchmark to which everyone aspires, even though none of the parties directly interacts with $\mathcal{G}_{\text{refClock}}$ itself.

Instead, parties only interact with $\mathcal{G}_{\text{refClock}}$ indirectly through a timer functionality and a network clock functionality that provide *approximate* relative and absolute notions of time, respectively. The timer functionality $\mathcal{F}_{\text{timer}}$ captures a cheap but low-latency device that can only provide measurements locally without delay, and the measurements "drift" significantly.

The network clock $\mathcal{G}_{\text{clock}}$ provides information more globally, to all parties in the system; however it may not respond to queries right away (or ever!).This functionality captures the behavior expected from a single client-server execution of the network time protocol, since the adversary controls the delays of packets transmitted over the asynchronous network. On the plus side, $\mathcal{G}_{\text{clock}}$ guarantees that timing measurements are approximately accurate (up to some bound) at the moment that they are eventually given.

Realizing $\mathcal{G}_{\text{clock}}$. We provide two network protocols that realize the network clock functionality. The first protocol π_{timeSync} involves a single query-response exchange between a client C with has access to a local timer (i.e., an instance of $\mathcal{F}_{\text{timer}}$) and a server S that has access to her own clock. This protocol allows the client to "bootstrap" the server's clock into one of her own, as long as the server is honest (i.e., uncorrupted).

We also capture a generalization of $\mathcal{G}_{\text{clock}}$ whose accuracy depends on multiple servers in such a way that it is robust to the corruption of a few servers. This generalization, denoted $\mathcal{G}_{\text{multiClock}}$, allows a client to request time from multiple servers (each with their own $\mathcal{G}_{\text{clock}}$) and then to select time as a function of all responses obtained before its $\mathcal{F}_{\text{timer}}$ times out (e.g., by picking the median response). By selecting the time in this way, the client obtains resilience against network corruptions. Even if many of the sessions are compromised, the client's timing measurement approximates the reference time as long as the majority of the servers whose $\mathcal{G}_{\text{clock}}$ boxes responded quickly to a query were uncorrupted, akin to the "sleepy model" of consensus [14, 15].

PKI with Expiration and Revocation. The final piece in our formalism is a time-aware variant of public-key infrastructure and signature verification. The starting point of our formalism is the certified signature verification functionality $\mathcal{G}_{\text{cert}}$ of [13] that utilizes infinite-duration keys. However, since the guarantee provided by $\mathcal{G}_{\text{cert}}$ has no time limit, it follows that $\mathcal{G}_{\text{cert}}$ cannot be realized in a system where signature keys get compromised after some time has elapsed.

In this work, we extend $\mathcal{G}_{\text{cert}}$ by allowing each signer to provide an expiration time t^*; furthermore, the signer can update this time in order to emulate revocation. Our extended functionality guarantees that if the global time at the time of verification is larger than t^* plus some "fudge factor" that accounts for the inaccuracies in time measurement, then the verification necessarily fails. This "fudge factor" is of crucial importance: it determines the length of time for which certificate authorities must respond to CRL or OCSP queries about certificates after they expire.

Functionality $\mathcal{F}_{\text{auth}}$

1. Upon receiving $(\mathsf{Send}, \mathsf{sid}, B, m)$ from party A, send $(\mathsf{Sent}, \mathsf{sid}, A, B, m)$ to $\mathcal{A}$.
2. Upon receiving $(\mathsf{Send}, \mathsf{sid}, B', m')$ from the $\mathcal{A}$, do: If A is corrupted then output $(\mathsf{Sent}, \mathsf{sid}, A, m')$ to party B'. Else, output $(\mathsf{Sent}, \mathsf{sid}, A, m)$ to party B. Halt.

Figure 2: The authenticated communication functionality, $\mathcal{F}_{\text{auth}}$. Reproduced from [16].

1.3 Additional Discussion

Incorporating Time in Existing UC Modeling, a General Paradigm. Our method for incorporating the time constraints in $\mathcal{G}_{\text{cert}}$ and in the protocol that realizes it minimal and general: To obtain time-aware certification, we only add a simple, self-contained time check to the existing code of to $\mathcal{G}_{\text{cert}}$. Additionally, we observe that the security of timing-agnostic protocols is unaffected by the presence of time-sensitive protocols.

Putting together these two observations, we obtain a general methodology for adding time-sensitive protocols and functionalities to the existing UC framework and its corpus of secure functionalities in a seamless way.

Time is Global. We model time as a global construct. That is, the time-related functionalities $\mathcal{G}_{\text{refClock}}$, $\mathcal{G}_{\text{clock}}$, and $\mathcal{G}_{\text{multiClock}}$ are global: they are accessible by anyone in the system. In particular, they always exist both in the "ideal" and in the "real" system. This modeling simplifies the composition of protocols that use these joint functionalities and provide a closer modeling of reality. We choose to model $\mathcal{F}_{\text{timer}}$ as a local functionality since it represents a service that is available only locally to a party. However this functionality too can in principle be modeled as a global functionality. (Such modeling might indeed be useful for analyzing systems where several protocols that use time have access to the same local physical clock.)

Implementing Authenticated Communication. Our network time protocols rely upon authenticity of communication between C and S. When specifying the protocols, we assume the existence of an ideal authenticated communication functionality $\mathcal{F}_{\text{auth}}$ [16] as specified in Fig. 2, and we use the modularity of the framework to remain agnostic about $\mathcal{F}_{\text{auth}}$'s underlying implementation.

We can instantiate $\mathcal{F}_{\text{auth}}$ using a PKI-based authenticated communication, as in, e.g., [13]. But we intend to use time to bolster the PKI! Ergo, we must avoid circularity in our arguments.

One way to do so is to assume that time servers have certificates that do not expire, or else where revocation is done out-of-band. Alternatively can instantiate $\mathcal{F}_{\text{auth}}$ as NTP does: have the client and server use an out-of-band key exchange mechanism to perform symmetric key authentication. Specifically, the maintainer of most stratum 1 NTP servers, NIST, shares keys with its clients over U.S. mail or fax machines [17]. This method completely circumvents the reliance on PKI for realizing $\mathcal{F}_{\text{auth}}$. Potentially, there are other out-of-band mechanisms for key exchange such as biometric human identification.

1.4 Related Work

The ability for parties to obtain a notion of time is an integral part of distributed computations [18]. These computations often require that timing measurements satisfy specific properties depending on the nature of the computation. The most basic of these is that time be monotonically increasing to allow for a consistent and correct ordering of events in, e.g., a time stamping protocol [19–21]. However, many protocols such as those of [22] and [23] require stronger guarantees, namely that time is both synchronized between parties and advances in a relatively uniform and expected pace.

A number of formalisms have been proposed over the years for incorporating time (both absolute and relative) in the security analysis of protocols. Some of these formalisms, like ours, are based on the UC framework [23–27]. We briefly review them.

UC Analyses of Time. The modeling of time by Kalai et al. [23] is perhaps the closest to the one in this work. There too, time is modeled as an additional "counter" that is available to all machines, and is incremented by the adversary on each machine, individually, subject to some global constraints. However, there it is assumed that all parties have ideal access to the global time (and furthermore that communication delays are within known time bounds.) In fact our work can be viewed as a more detailed and "faithful" modeling of the propagation of time in real-life networks, in a way justifying the more abstract modeling of [23]. In other words, if one assumes that a majority of the instances of $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ used by the parties are uncorrupted, [23] can be used as an additional application for our modeling. Furthermore, the impossibility result in [23] implies that one cannot realize $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ or $\mathcal{G}_{\text{refClock}}$ with appropriate parameters in the plain model.

Katz et al. [26] and Canetti [27] provide, within the UC framework, ideal functionalities that give abstractions that mimic synchronous communication among participants. However, these works do not provide ways for realizing these abstractions from existing mechanisms like network time. Furthermore, these abstractions do not suffice to capture the prevalent use of limited-time certificates.

Backes et al. [24] provide an alternative formalism of time based on the UC framework that differs from the present formalism in a number of crucial ways. First, [24] significantly modifies the existing UC framework, thus making its formalism incompatible with the body of work in the existing framework. Second, the [24] modeling assumes that machines have specific and fixed relative speeds and where time passage is directly proportional to the number of computational steps. In contrast, in our modeling time is not necessarily tied to other computational aspects of the system. Third, [24] analyzes only standard, time-unaware protocols; the modeling of time is used only to bound the success of attacks on the protocol. In contrast, we model protocols where time is crucially used by the protocol itself.

Vajda [25] provides a number of high-level proposals for general modeling of real-time within the UC framework. However this work does not address network time protocols or the cryptographic applications treated here.

Security-Aware Network Time Protocols. Several frameworks [5] [28] [29] [30] aim to define and analyze the security requirements of time synchronization protocols. RFC 7384 [29] provides guidelines for important security features of PTP and NTP as they relate to possible attacks. Itkin and Wool [30] build on this with new attack vectors and suggested mitigations for PTP, but they do not provide proofs for their mitigations nor any accuracy guarantees for the time protocols themselves.

Dowling et al. [28] extend NTP to include lightweight authentication for servers and provide game based proofs for its accuracy relative to the time at the server. They do not provide any accuracy guarantees relative to a global notion of time, and thus they fail to provide the global synchronization that is necessary for time sensitive crypto such as PKI.

Malhotra et al. [5] focus on several security concerns when deploying NTP in practice, at the expense of full coverage. They study concrete security bounds for NTP against off/on-path attacks in the standalone model. By contrast, the security guarantee in our work addresses composable security and additionally covers adversaries who have full control over the network and may use it to drop packets sent by honest parties. Both of these properties are crucial towards our work in Section 6 of realizing time-sensitive crypto primitives like the PKI.

1.5 Organization

The paper is organized as follows. Section 2 gives an overview of the Network Time Protocol. In Section 3, we introduce three new ideal functionalities: $\mathcal{G}_{\text{refClock}}$, $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$, and $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$. Section 4 formalizes and proves the security of a protocol that permits a single server to share its view of time with a single client. Section 5 generalizes this basic protocol in two ways for improved resilience: a client takes timing measurements from multiple servers, and the network topology is dispersed to reduce resource and network resource congestion. Finally, Section 6 integrates our time consensus protocol with time-sensitive applications such as PKI.

2 Preliminaries

This section summarizes (separately!) the universally composable security framework and the network time protocol.

2.1 Universally Composable Security

We provide a brief overview of the UC framework. See [27] and [31] for more details. (This overview is taken almost verbatim from [13].)

We focus on the notion of protocol *emulation*, wherein the objective of a protocol π is to imitate another protocol ϕ. In this work, the entities and protocols we consider are polynomial-time bounded Interactive Turing Machines (ITMs), in the sense detailed in [27].

Systems of ITMs. To capture the mechanics of computation and communication among entities, the UC framework employs an extension of the ITM model. A computer program (such as for a protocol, or perhaps program of the adversary) is modeled in the form of an ITM. An execution experiment consists of a system of ITMs which are instantiated and executed, with multiple instances possibly sharing the same ITM code. A particular executing ITM instance running in the network is referred to as an ITI. Individual ITIs are parameterized by the program code of the ITM they instantiate, a party ID (pid) and a session ID (sid). We require that each ITI can be uniquely identified by the identity pair id = (pid,sid), irrespective of the code it may be running. All ITIs running with the same code and session ID are said to be a part of the same protocol session, and the party IDs are used to distinguish among the various ITIs participating in a particular protocol session.

The Basic UC Framework. At a very high level, the intuition behind security in the basic UC framework is that any adversary $\mathcal{A}$ attacking a protocol π should learn no more information than could have been obtained via the use of a simulator $\mathcal{S}$ attacking protocol ϕ. Furthermore, we would like this guarantee to hold even if ϕ were to be used as a subroutine in arbitrary other protocols that may be running concurrently in the networked environment and after we substitute π for ϕ in all the instances where it is invoked. This requirement is captured by a challenge to distinguish between actual attacks on protocol ϕ and simulated attacks on protocol π. In the model, attacks are executed by an environment $\mathcal{E}$ that also controls the inputs and outputs to the parties running the challenge protocol. The environment $\mathcal{E}$ is *constrained* to execute only a single instance of the challenge protocol. In addition, the environment $\mathcal{E}$ is allowed to interact freely with the attacker (without knowing whether it is $\mathcal{A}$ or $\mathcal{S}$). At the end of the experiment, the environment $\mathcal{S}$ is tasked with distinguishing between adversarial attacks perpetrated by $\mathcal{A}$ on the challenge protocol π, and attack simulations conducted by $\mathcal{S}$ with protocol ϕ acting as the challenge protocol instead. If no environment can successfully distinguish these two possible scenarios, then protocol π is said to *UC-emulate* the protocol ϕ.

Balanced environments. In order to keep the notion of protocol emulation from being unnecessarily restrictive, we consider only environments where the amount of resources given to the adversary (namely, the length of the adversary's input) is at least some fixed polynomial fraction of the amount of resources given to all protocols in the system. From now on, we only consider environments that are balanced.

Definition 1 (UC-emulation)**.** Let π and ϕ be multi-party protocols. We say that π UC-emulates ϕ if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that for any (constrained) environment $\mathcal{E}$, we have:

$$\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{E}} \approx \mathrm{EXEC}_{\phi,\mathcal{S},\mathcal{E}}$$

Defining protocol execution this way is sufficient to capture the entire range of network activity that is observable by the challenge protocol but may be under adversarial control.

Furthermore, the UC framework admits a very strong composition theorem, which guarantees that arbitrary instances of ϕ that may be running in the network can be safely substituted with any protocol π that UC-emulates it. That is, given protocols ρ, π and ϕ, such that ρ uses subroutine calls to ϕ, and protocol π UC-emulates ϕ, let $\rho^{\phi\rightarrow\pi}$ be the protocol which is identical to ρ except that each subroutine call to ϕ is replaced by a subroutine call to π. We then have:

Theorem 1 (UC-Composition)**.** Let ρ,π and ϕ be protocols such that ρ makes subroutine calls to ϕ. If π UC-emulates ϕ and both π and ϕ are subroutine-respecting, then protocol $\rho^{\phi\rightarrow\pi}$ UC-emulates protocol ρ.

The Generalized UC Framework. As mentioned above, the environment $\mathcal{E}$ in the basic UC experiment is unable to invoke protocols that share state in any way with the challenge protocol. In contrast, in many scenarios we would like to be able to analyze challenge protocols that share information with other network protocol sessions. For example, protocols may share information via a global setup such as a public Common Reference String (CRS) or a standard Public Key Infrastructure (PKI). To overcome this limitation and allow analyzing such protocols in a modular way, [31] propose the Generalized UC (GUC) framework. The GUC challenge experiment is similar to the basic UC experiment, only with an *unconstrained* environment. In particular, now $\mathcal{E}$ is allowed to invoke and interact with arbitrary protocols, and even multiple sessions of the challenge protocol. Some of the protocol sessions invoked by $\mathcal{E}$ may even share state information with challenge protocol sessions, and indeed, those protocol sessions might provide $\mathcal{E}$ with information related to the challenge protocol instances that it would have been unable to obtain otherwise. To distinguish this from the basic UC experiment, we denote the output of an unconstrained environment $\mathcal{E}$, running with an adversary $\mathcal{A}$ and a challenge protocol π in the GUC protocol execution experiment, by $\mathrm{GEXEC}_{\pi,\mathcal{A},\mathcal{E}}$. GUC emulation is defined analogously to the definition of basic UC emulation outlined above:

Definition 2 (GUC-emulation)**.** Let π and ϕ be multi-party protocols. We say that π GUC-emulates ϕ if for any adversary $\mathcal{A}$ there exists an adversary $\mathcal{S}$ such that for any (unconstrained) environment $\mathcal{E}$, we have:

$$\mathrm{GEXEC}_{\pi,\mathcal{A},\mathcal{E}} \approx \mathrm{GEXEC}_{\phi,\mathcal{S},\mathcal{E}}.$$

The UC theorem directly extends to the GUC model.

2.2 The Network Time Protocol (NTP)

As the name suggests, NTP permits several computers on a network to share information about the time.

In this work, we focus on NTP's most popular method of operation: a hierarchical client-server fashion in which a client queries a server who has (ostensibly) higher fidelity timing information than the client. A client can use multiple invocations of NTP's fundamental query-response protocol (either with the same server or with multiple servers) to gather several timing measurements, which it then uses to set or update its own notion of time.

Query-Response Protocol. We first describe NTP's two-round timing exchange protocol over IPv4. Four timing measurements are relevant during the execution of this protocol:

T_1 *Origin timestamp.* Client's system time at the moment that the client sends the query.

T_2 *Receive timestamp.* Server's system time at the moment that the server receives the query.

T_3 *Transmit timestamp.* Server's system time at the moment that the server sends the response.

T_4 *Destination timestamp.* Client's system time at the moment that the client receives the response.

The client's query packet includes measurement T_1. The server's response packet repeats T_1 and appends T_2 and T_3. The client locally computes T_4 upon receipt of the response packet.

Setting, or Updating, the Client's Time. The client makes two assumptions when analyzing the timestamps.

1. Her clock and the server's clock move in relative synchrony while the NTP session is live (even if they have different absolute notions of time).

2. The network delay is symmetric. That is, the query packet's client $\to$ server latency equals the response packet's server $\to$ client latency.

Deviations from these assumptions do lead to small but bounded error in the client's eventual measurement of time; we will return to this issue later.

If assumption 1 is accurate, then the round-trip network *delay* δ during the exchange equals:

$$\delta = (T_4 - T_1) - (T_3 - T_2) \tag{1}$$

If assumption 2 is accurate, then the absolute gap between the server and client clock is $T_2 - (T_1 + \frac{\delta}{2})$ for the client query, and $T_3 - (T_4 - \frac{\delta}{2})$ for the server response. Averaging these two quantities gives us the absolute *offset* between the client and server clocks:

$$\theta = \tfrac{1}{2}\left((T_2 - T_1) + (T_3 - T_4)\right) \quad (2)$$

While talking to multiple servers, the client chooses a single server to which it synchronizes its local clock. This decision is made adaptively by a set of selection, cluster, combine and clock discipline algorithms. For the purpose of this paper, we assume that the client will make an update to its clock if median θ is less than a certain threshold. Client/server packets are not authenticated by default, but a Message Authentication Code (MAC) can optionally be appended to the packet [2, Sec. 13]. In this work, we restrict our attention to authenticated NTP.

3 Modeling Absolute and Relative Time

In this section, we introduce three ideal functionalities that aid a client C or server S to learn the time. The first two provide exact and approximate *absolute* notions of time, whereas the third functionality approximates the *relative* passage of time. These functionalities are depicted formally in Figs. 3 through 5. We stress that the functionalities only respond to the methods explicitly stated in the figures; when given a message that cannot be parsed into one of the provided forms, they simply hand the execution back to the caller without providing any output.

The functionalities themselves are referred to as $\mathcal{G}^{\text{paramaters}}_{\text{functionality}}$ for global functionalities and as $\mathcal{F}^{\text{paramaters}}_{\text{functionality}}$ for local (i.e., non-global) functionalities. Protocols are specified in the format $\pi^{\text{paramaters}}_{\text{functionality}}$. The protocols and functionalities may use subroutines, which we indicate in the text and sometimes denote using square brackets.

3.1 The Reference Clock Functionality $\mathcal{G}_{\text{refClock}}$

We begin by introducing a simple global functionality $\mathcal{G}_{\text{refClock}}$ that provides a universal *reference clock*. When queried, it provides an abstract notion of time represented as an integer G. It is monotonic, and only the environment may increment it; we stress that the simulator $\mathcal{S}$ *cannot* forge the reference time.

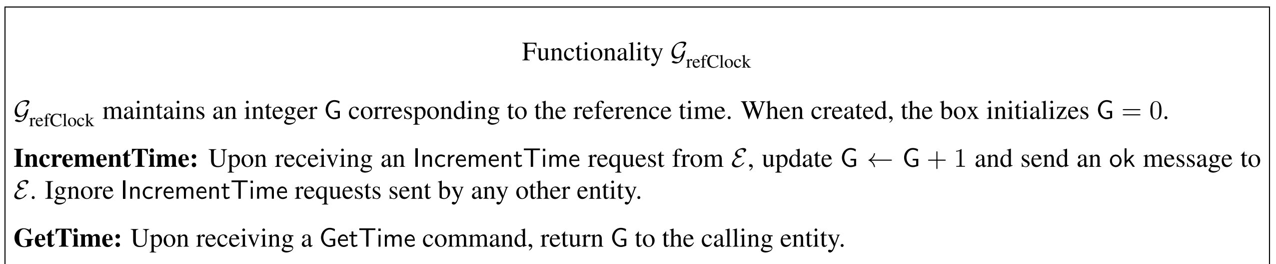
In this work we use subscripts to denote the relative order in which requests are made to the reference clock. Hence, if $x > y$ then $\mathsf{G}_x \geq \mathsf{G}_y$.

Figure 3 formally codifies $\mathcal{G}_{\text{refClock}}$. It functions similarly to Vadja's ideal notion of time [25], with one crucial exception: we do not intend for any honest party to access $\mathcal{G}_{\text{refClock}}$ directly. Instead, in this work $\mathcal{G}_{\text{refClock}}$ exclusively functions as a subroutine for the remaining two functionalities.

3.2 Delayed Approximate Clock Functionality $\mathcal{G}^{P,S,\Sigma}_{\text{clock}}$

In Figure 4, we construct the global functionality $\mathcal{G}^{P,S,\Sigma}_{\text{clock}}$ that provides *delayed, approximate time*. This functionality has three major distinctions from $\mathcal{G}_{\text{refClock}}$.

First, the clock communicates with a single party P, who we refer to as its owner, and the accuracy of the clock can be influenced by (potentially but not necessarily different) party S. This degree of freedom allows us to use the



Functionality $\mathcal{G}_{\text{refClock}}$

$\mathcal{G}_{\text{refClock}}$ maintains an integer G corresponding to the reference time. When created, the box initializes $\mathsf{G} = 0$.

IncrementTime: Upon receiving an $\mathsf{IncrementTime}$ request from $\mathcal{E}$, update $\mathsf{G} \leftarrow \mathsf{G} + 1$ and send an ok message to $\mathcal{E}$. Ignore $\mathsf{IncrementTime}$ requests sent by any other entity.

GetTime: Upon receiving a $\mathsf{GetTime}$ command, return G to the calling entity.

Figure 3: Global Ideal functionality representing reference time $\mathcal{G}_{\text{refClock}}$. It is expected that honest parties do not talk to this functionality directly.

Functionality $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ [$\mathcal{G}_{\text{refClock}}$]

A clock functionality identified by a session id $\mathsf{sid}_{\mathsf{clock}} = (\mathsf{sid}'_{\mathsf{clock}}, P, S)$ that denotes its owner P as well as a (potentially but not necessarily different) party S whose honesty influences the accuracy of the clock. It is also parameterized by the maximum allowable shift Σ from the reference time. It operates as follows.

Corrupt: Upon receiving a $\mathsf{Corrupt}$ message, record that S is now corrupted.

GetTime: Upon receiving input $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{clock}})$ from party P', ignore this request if $P' \neq P$, otherwise:

1. Send $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{clock}})$ to the adversary $\mathcal{A}$. Wait for a response of the form $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{clock}}, \sigma)$ from $\mathcal{A}$.
2. If $\sigma == \bot$, output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, \bot)$ to P.
3. Else send $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$ to receive G. Next, compute $T_P = \mathsf{G} + \sigma$. Then do the following:
   - If $|\sigma| > \Sigma$ and no $\mathsf{Corrupt}$ record exists, then reset $T_P = \bot$.
   - Output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, T_P)$ to P.

Figure 4: Ideal functionality $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ that provides delayed, approximately accurate time measurements to its owner P. The functionality doesn't provide any guarantee on when a timing measurement will be delivered. It only guarantees that at the instant the measurement is given, its value is approximately correct.

Functionality $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ [$\mathcal{G}_{\text{refClock}}$]

A timer functionality identified by a session id $\mathsf{sid}_{\mathsf{timer}} = (\mathsf{sid}'_{\mathsf{timer}}, C)$ that denotes its owner C. It is also parameterized by the maximum allowable delay Δ_C, and the maximum allowable shift Σ_C. It operates as follows.

SetShift: Upon receiving a command $(\mathsf{SetShift}, \mathsf{sid}_{\mathsf{timer}}, M)$ from $\mathcal{E}$, record M as the code of a Turing machine (replacing any previously-stored code) and send an $(\mathsf{ok}, \mathsf{sid}_{\mathsf{timer}})$ message to $\mathcal{E}$.

Start: Upon receiving input $(\mathsf{Start}, \mathsf{sid}_{\mathsf{timer}})$ from a party P: if $P \neq C$ or if a Start command was previously received then ignore this request. Otherwise:

1. Send $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$. Denote its response as G'. Record the tuple $(\mathsf{G}', \mathsf{sid}_{\mathsf{timer}})$.
2. Send an $(\mathsf{ok}, \mathsf{sid}_{\mathsf{timer}})$ message to C.

TimeElapsed: Upon receiving input $\mathsf{TimeElapsed}$ from a party P: if no previous Start command was issued or if $P \neq C$ then ignore this request. Otherwise:

1. Send $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$. Denote its response as G. Also, retrieve the previously-recorded G'.
2. Run $M(\mathsf{sid}_{\mathsf{timer}}, \mathsf{G}', \mathsf{G})$ and denote its output as σ_C. (Also, maintain M's state for future calls.)
3. Compute $\delta = \mathsf{G} - \mathsf{G}' + \sigma_C$. If $\delta \leq \Delta_C$, output $(\delta, \mathsf{sid}_{\mathsf{timer}})$ to C. Else output $(\bot, \mathsf{sid}_{\mathsf{timer}})$ to C.

Figure 5: Ideal functionality $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ that returns to its owner C the approximate relative time elapsed between the Start and $\mathsf{TimeElapsed}$ commands. While the environment $\mathcal{E}$ may influence the timer's accuracy, it must do so 'out of band': once C requests $\mathsf{TimeElapsed}$, it learns the answer instantaneously. Additionally, the adversary doesn't directly interact with $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ at all.

clock functionality in this work as an abstraction of two very different situations: (1) a physical clock that is actually under the control of its owner, such as an atomic clock owned by a stratum 1 NTP server, and (2) an ideal service akin to that expected from the Network Time Protocol itself, which permits a client to operate "as if" she owned a clock herself, modulo the unavoidable imperfections (cf. Theorems 2-3).

Second, the clock is inaccurate, in the sense that its belief about the time $T = \mathsf{G} + \sigma$ is somewhat shifted away from the reference time. Still, the clock is guaranteed to *approximate* the reference time up to some maximum shift value $|\sigma| \leq \Sigma$.

Third, the clock is *not* instantaneous. Instead, it only returns a time measurement after some adversarially-controlled *delay* δ (which may be infinite). The approximate correctness guarantee from above holds at the moment that the time is *eventually* returned.

$\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ is used both as a goal, or *benchmark*, in the sense that everyone strives to attain it (with the best parameters possible), and at the same time it is used as a service for other protocols in order to achieve other tasks (or even another instance of $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ but with better parameters or another server S).

3.3 Approximate Timer Functionality $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$

In Figure 5, we construct the $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ functionality. Like local clocks, a timer functionality has a single owner C and its measurements only guarantee approximate correctness. However, $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ differs from the prior two functionalities in three ways.

First, it doesn't provide an absolute notion of time; instead, it provides the *relative* difference in time between a starting and ending point.

Second, the timer has a short lifetime: after a maximum delay Δ_C, it will "time out" and only output $\perp$. This limitation ensures that $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ cannot be used as a substitute for long-term time measurements of the type provided by $\mathcal{G}_{\text{refClock}}$ and $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$.

Third, it is only used locally within a specific instance of a time synchronization protocol. By contrast, $\mathcal{G}_{\text{refClock}}$ and $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ are Global UC functionalities.

4 Single Server Time Sync

In this section, we formally specify a simplified version of the way that a client C uses the Network Time Protocol (NTP) to query a single server S for its belief about the time. We show that this protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ allows C to operate as if she had a delayed approximate clock of her own. Formally, we prove that $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ GUC-realizes a clock $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ owned by the client.

As depicted in Fig. 7, $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ internally uses the functionalities specified in Section 3. The server has access to its own instance of $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ that approximates the reference time, whereas the client can only measure the relative passage of time via $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$. Additionally, S and C communicate using $\mathcal{F}_{\text{auth}}$.

We fully specify the protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ in Figure 6. The protocol is natural: C sends a time request to S and uses $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ to measure the time elapsed until the response arrives. S responds with (roughly) the times at which she receives the client query (T_2) and sends the response packet (T_3) to allow the client to distinguish network transmission time from server processing time. The client uses its local timer to determine how long the server took to respond as well as to calculate the average network delay in an NTP-like manner; however, the client times out if the server responds after too long a delay (measured on the client's local timer).

For simplicity of exposition, we chose to "hardwire" the identity of the server in the code of both the ideal functionality and the protocol. However this is not essential: our results continue to hold in an alternative model where the identity of the server (or servers, in Section 5) is given to the client as a part of the input.

The remainder of this section contains a formal theorem and proof about the accuracy of $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$.

Theorem 2 (Single server UC security)**.** Given any parameters that satisfy $\Sigma_C^* \geq \frac{1}{2} \cdot \Delta_C + \Sigma_C + \Sigma_S$, it holds that the single server approximate time synchronization protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ GUC-realizes $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$.

Protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ [$\mathcal{F}_{\text{auth}}$, $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$]

GetTime: Begins when the caller (e.g., $\mathcal{E}$) sends the input $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{ts}})$ to C, where $\mathsf{sid}_{\mathsf{ts}} = (\mathsf{sid}'_{\mathsf{ts}}, C, S)$. Ends when C responds back to the caller with $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{ts}}, T_c)$.

$\mathcal{E}$ Client C | $\mathcal{F}_{\text{auth}}$ Server S | $\mathcal{A}$

$\longrightarrow$ 1: Record the tuple $(\mathsf{sid}_{\mathsf{ts}}, \mathsf{sid}_{\mathsf{timer}})$

2: Send Start to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, receive $(\mathsf{ok}, \mathsf{sid}_{\mathsf{timer}})$

3: Send $(\mathsf{query}, \mathsf{sid}_{\mathsf{ts}})$ to S

$\longrightarrow$ 4: Record the tuple $(\mathsf{sid}_{\mathsf{ts}}, \mathsf{sid}_{\mathsf{clock}})$

5: Send $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{clock}})$ to $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ $\longleftrightarrow$

6: Get $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, T_2)$

7: Append T_2 to the record $(\mathsf{sid}_{\mathsf{ts}}, \mathsf{sid}_{\mathsf{clock}})$

8: Send $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{ts}})$ to $\mathcal{A}$ $\longrightarrow$

9: Get $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{ts}})$ from $\mathcal{A}$ $\longleftarrow$

10: Send $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{clock}})$ to $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ again $\longleftrightarrow$

11: Get $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, T_3)$

12: Retrieve $(\mathsf{sid}_{\mathsf{ts}}, \mathsf{sid}_{\mathsf{clock}}, T_2)$

$\longleftarrow$ 13: Send $(\mathsf{response}, \mathsf{sid}_{\mathsf{ts}}, T_2, T_3)$ to C

14: Retrieve $(\mathsf{sid}_{\mathsf{ts}}, \mathsf{sid}_{\mathsf{timer}})$, abort if no tuple exists

15: Send $\mathsf{TimeElapsed}$ to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, receive $(\delta, \mathsf{sid}_{\mathsf{timer}})$

16: If $\delta == \perp$, then set $T_C = \perp$

17: Else, set $T_C = T_3 + \frac{1}{2} \cdot (\delta - T_3 + T_2)$

18: Delete record $(\mathsf{sid}_{\mathsf{ts}}, \mathsf{sid}_{\mathsf{timer}})$

$\longleftarrow$ 19: Output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{ts}}, T_C)$ to $\mathcal{E}$

Continue: Begins when $\mathcal{A}$ sends a $\mathsf{Continue}$ message to C, so that C can determine whether an ongoing session of $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ has timed out. If so, C ends the session.

$\mathcal{A}$ Client C

$\longrightarrow$ 1: Retrieve $(\mathsf{sid}_{\mathsf{ts}}, \mathsf{sid}_{\mathsf{timer}})$, abort if no tuple exists

2: Send $\mathsf{TimeElapsed}$ to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, receive $(\delta, \mathsf{sid}_{\mathsf{timer}})$ in response

$\longleftarrow$ 3: If $\delta == \perp$, then delete record $(\mathsf{sid}_{\mathsf{ts}}, \mathsf{sid}_{\mathsf{timer}})$ and output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{ts}}, \perp)$ to $\mathcal{E}$

$\longleftarrow$ 4: If $\delta \neq \perp$, then end activation

Figure 6: Time Synchronization Protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$. The two participants C and S communicate through $\mathcal{F}_{\text{auth}}$, and they have access to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ and $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$, respectively. Note that S does not have any inputs or outputs.

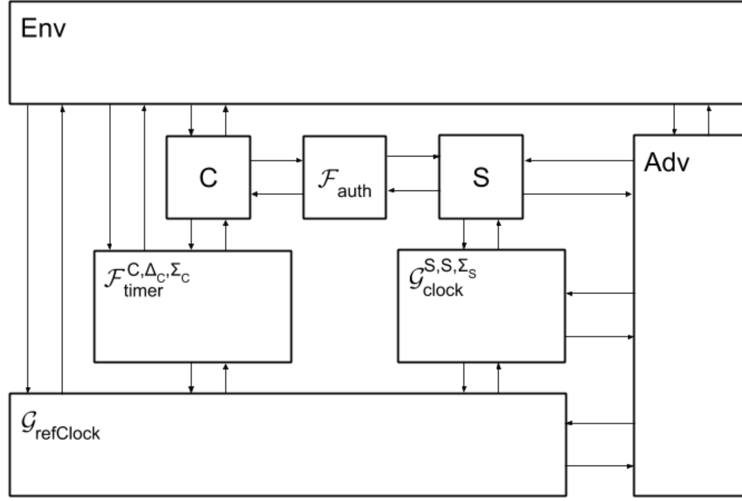


Fig. 7: Interactions between participants and functionalities during an execution of the single server time synchronization protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$.

We emphasize that the *shift* of the client's purported time depends on the *delay* that the client waits for the timing information. Hence, C's insistence upon a maximum delay Δ_C isn't merely a matter of convenience: it affects the accuracy of her notion of time as well.

Our proof has two components. First, in Section 4.1 we design a simulator $\mathcal{S}$ that successfully emulates the execution of any real-world adversary $\mathcal{A}$ from the environment $\mathcal{E}$'s point of view. Then, in Section 4.2 we analyze the time bound that it achieves.

4.1 Designing the Simulator

Fig. 8 depicts the high-level interaction between components in the ideal world. As usual, the simulator $\mathcal{S}$ runs an emulated copy of the real world protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ inside its head while also playing the role of $\mathcal{E}$ inside this simulation.

In more detail, $\mathcal{S}$ internally emulates the execution of $\mathcal{A}$, $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$, $\mathcal{F}_{\text{auth}}$ and each of the involved parties. In addition, $\mathcal{S}$ externally instantiates the global $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ that's called by the protocol (unless it already exists). $\mathcal{S}$ also relays the messages sent from the emulated parties to $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ and $\mathcal{G}_{\text{refClock}}$, and from $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ and $\mathcal{G}_{\text{refClock}}$ to the emulated parties.

While conducting this simulation, $\mathcal{S}$ monitors the traffic of its emulated $\mathcal{A}$ and $\mathcal{E}$, along with any messages that $\mathcal{S}$ directly receives from $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. The messages that $\mathcal{S}$ views/receives causes it to make changes in the ideal world or the emulated world.

Simulating $\mathsf{GetTime}$: When $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ sends a message of the form $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{clock}})$, then $\mathcal{S}$ instantiates C with the environmentally-provided message $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{ts}})$.

Simulating $\mathsf{Continue}$: When $\mathcal{S}$ observes $\mathcal{A}$ sending a $\mathsf{Continue}$ message to the emulated C, it waits to see which of the two possible outcomes occur at C. If C simply ends its activation, then a timeout event has not yet occurred and $\mathcal{S}$ does nothing. If instead C outputs $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, \perp)$ then a timeout event has occurred; in this case, $\mathcal{S}$ sends a $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{clock}}, \sigma)$ message to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ to cause the dummy C to produce the same output in the ideal world.

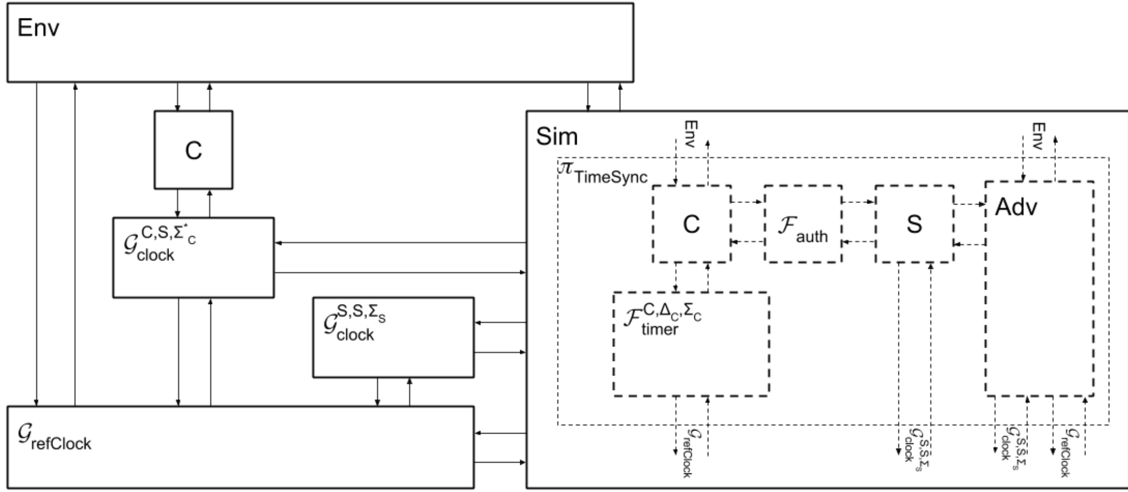


Fig. 8: Interactions between participants in the ideal world execution of single server time sync, including the emulation of the real world inside of the simulator.

Relaying messages to $\mathcal{E}$: When the emulated $\mathcal{A}$ sends a message to the emulated $\mathcal{E}$, $\mathcal{S}$ relays it to the real $\mathcal{E}$. Conversely, $\mathcal{S}$ forwards messages sent by the real $\mathcal{E}$ to the emulated $\mathcal{A}$.

Corrupting the server: When $\mathcal{A}$ sends a $\mathsf{Corrupt}$ message in the emulated world, then $\mathcal{S}$ sends a $\mathsf{Corrupt}$ message to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$.

Completing the simulation: When the emulated C sends to $\mathcal{E}$ its output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{ts}}, T_C)$, then $\mathcal{S}$ knows both *when* and *what* to send back to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. If $T_C == \bot$, then $\mathcal{S}$ simply sends $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{clock}}, \bot)$ to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. Otherwise: $\mathcal{S}$ send $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$ to retrieve the current reference time T. Then, $\mathcal{S}$ computes the shift $\sigma_{\mathsf{Sim}} = (T_C - T)$ and sends $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{clock}}, \sigma_{\mathsf{Sim}})$ to $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$.

4.2 Analyzing the Accuracy of the Simulator

It is straightforward to verify that the simulator's Wake responses perfectly emulate those in the real world: its simulation of $\mathsf{Continue}$ ensures that time-out actions occur identically in the real and ideal worlds, and otherwise its calculation of σ_{Sim} within the Wake message agrees with the message sent by $\mathcal{A}$.

Therefore, it only remains to show that the answer $\mathcal{S}$ returns can meet the approximate correctness bound required by $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. If S is corrupted or if C times out then there is no bound to meet. Ergo, in the rest of this section we assume that S is uncorrupted and also that $\delta < \Delta_C$ in response to all queries C makes to $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ so that no timeout occurs.

In the emulated protocol $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$, the output time T_C is computed by the client as $T_C = T_3 + \frac{1}{2}(\delta - T_3 + T_2)$. T_2 and T_3 are the times returned from the server and are equal to $\mathsf{G}_2 + \sigma_2$ and $\mathsf{G}_3 + \sigma_3$ respectively. δ is returned from $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ to the client and is computed as $\mathsf{G}_4 + \sigma_4 - \mathsf{G}_1 - \sigma_1$. So, for the emulated client, $T_C = \mathsf{G}_3 + \sigma_3 + \frac{1}{2}(\mathsf{G}_4 + \sigma_4 - \mathsf{G}_1 - \sigma_1 - \mathsf{G}_3 - \sigma_3 + \mathsf{G}_2 + \sigma_2)$.

In the ideal world, the client interacting with $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ outputs the time $\mathsf{G}_4 + \sigma_{\mathsf{Sim}}$, where the simulator provides σ_{Sim} to account for discrepancy between the emulated client's output and the output of the client interacting with the

$\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$. Combining the two equations yields:

$$\sigma_{\text{Sim}} = \frac{\mathsf{G}_2 - \mathsf{G}_1 + \mathsf{G}_3 - \mathsf{G}_4}{2} + \frac{\sigma_2 - \sigma_1 + \sigma_3 + \sigma_4}{2}.$$

The simulator must be able to correct for the maximum possible value of σ_{Sim}. It is straightforward that $|\sigma_{\text{Sim}}|$ is maximized when the following criteria hold.

1. The server's clock is maximally shifted: σ_3 and σ_2 both equal Σ_S.
2. The client's timer shifts to the maximum extent permissible between the Start and TimeElapsed queries: $\sigma_4 = \Sigma_C$ and $\sigma_1 = -\Sigma_C$.
3. The network latency is maximally asymmetric: $\mathcal{E}$ maximizes $(\mathsf{G}_2 - \mathsf{G}_1 + \mathsf{G}_3 - \mathsf{G}_4)$ by incrementing the reference time a large amount between G_1 and G_2 and not at all between G_3 and G_4, or vice versa.

We desire an upper bound on the network asymmetry described in item 3. For the client to avoid timing out, it must be the case that the total time elapsed obey the constraint that

$$(\mathsf{G}_4 + \sigma_4) - (\mathsf{G}_1 + \sigma_1) \leq \Delta_C.$$

It follows that $\mathsf{G}_4 - \mathsf{G}_1 \leq \Delta_C + 2 \cdot \Sigma_C$. Also, since the four times are monotonic, $0 \leq \mathsf{G}_4 - \mathsf{G}_3 \leq \mathsf{G}_4 - \mathsf{G}_1$ and $0 \leq \mathsf{G}_2 - \mathsf{G}_1 \leq \mathsf{G}_4 - \mathsf{G}_1$. Therefore: $|\mathsf{G}_2 - \mathsf{G}_1 + \mathsf{G}_3 - \mathsf{G}_4| \leq \Delta_C + 2 \cdot \Sigma_C$.

Combining the three bounds above yields

$$|\sigma_{\text{Sim}}| \leq \frac{\Delta_C}{2} + \Sigma_C + \Sigma_S.$$

Hence, it suffices for the Σ_C^* for the $\mathcal{G}_{\text{clock}}^{C,S,\Sigma_C^*}$ to be $\frac{1}{2}\Delta_C + \Sigma_C$ larger than the shift in the server-owned clock $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ in order for the simulator to be able to simulate correctly, proving Theorem 2.

5 More Robust Network Time

In this section, we provide a more robust method to acquire time over the Internet. It is better representative of the way NTP operates: each client queries multiple servers to increase its resilience to compromise, and different clients query different servers to remove network and resource bottlenecks.

Section 5.1 considers the case of a single client accessing multiple servers. Then, Section 5.2 considers the multi-stratum case in which each server in a stratum receives its notion of time not from a local clock, but instead by acting also as a client and querying several servers in the stratum below. We use the composition theorem to provide modular and relatively simple analysis of these these rather intricate interactions.

5.1 Multiple Server Time Sync

At a high level, the new $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ protocol involves a client who queries n different servers for the time. Once all timing measurements have been collected or time-out, then the client outputs the *median* of all non-$\perp$ timing measurements.

The full protocol to aggregate times from multiple servers is shown in Fig. 9. This protocol requires the client to keep an extra timer per server in order to calculate and remember the freshness of responses.

Producing a real multi-server protocol should involve the composition of $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ protocols with each server. Thanks to the UC composition theorem, it suffices to analyze a simpler protocol in which the $\pi_{\text{timeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ with each server S_i is replaced with its corresponding ideal functionality $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma_C^*}$. We make a few remarks about this use of composition:

Protocol $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$ [$\mathcal{G}^{C,S_1,\Sigma^*_C}_{\text{clock}}, \ldots, \mathcal{G}^{C,S_n,\Sigma^*_C}_{\text{clock}}$ and $2n$ instances of $\mathcal{F}^{C,\Delta_C,\Sigma_C}_{\text{timer}}$]

GetTime: Begins when the caller sends to C the input $(\mathsf{GetTime}, \mathsf{sid_{mts}})$. In response, C provisions a timing measurement array $\mathcal{T}_{\mathsf{sid_{mts}}}$ of length n with all values initialized to a special '?' symbol. C also allocates sufficient storage space to record the session ids of all timers and clocks with which it interacts. Finally, C invokes QueryClock.

QueryClock: Begins when invoked by GetTime or Continue. Invariant: at least one clock has not been invoked.

1. C identifies a previously-unqueried $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$.
2. C sends a Start command to the ith timer and waits for an ok response.
3. C sends the command (GetTime, $\mathsf{sid}_{\mathsf{clock}\,i}$) to $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$.

ResponseReceived: Begins when C receives a response (TimeReceived, $\mathsf{sid}_{\mathsf{clock}\,i}$, T_i) from a clock $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$:

1. C records $\mathcal{T}_{\mathsf{sid_{mts}}}[i] \leftarrow T_i$.
2. C sends a TimeElapsed message to the ith timer. If it returns $\perp$, then C updates $\mathcal{T}_{\mathsf{sid_{mts}}}[i] \leftarrow \perp$.
3. If $\mathcal{T}_{\mathsf{sid_{mts}}}[i] \neq \perp$, then C sends a Start command to the $(n+i)^{\text{th}}$ timer and waits for an ok response.
4. Invoke the Continue routine.

Continue: Begins when $\mathcal{A}$ sends to C the Continue command or when ResponseReceived ends.

1. If C has not yet queried each $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$, then C begins the QueryClock protocol as stated above.
2. Else, C begins the CheckTimeout protocol as stated below.

CheckTimeout: Begins when invoked by Continue. Invariant: each $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$ has already received a GetTime query.

1. For all i: C sends a TimeElapsed message to the ith timer if $\mathcal{T}_{\mathsf{sid_{mts}}}[i] ==$? and to the $(n+i)^{\text{th}}$ timer otherwise. If the timer returns a $\perp$ response, then C updates $\mathcal{T}_{\mathsf{sid_{mts}}}[i] \leftarrow \perp$.
2. If none of the records in $\mathcal{T}_{\mathsf{sid_{mts}}}$ equal '?', then invoke Finalize. Else, end the current activation.

Finalize: Begins when invoked by CheckTimeout. Note that $\mathcal{A}$ never gets control during Finalize.

1. For all i such that $\mathcal{T}_{\mathsf{sid_{mts}}}[i] \neq \perp$, send a TimeElapsed message to the $(n+i)^{\text{th}}$ timer and wait for a response of the form $(\delta, \mathsf{sid}_{\mathsf{timer}\,n+i})$.
   - If $\delta == \perp$, then update $\mathcal{T}_{\mathsf{sid_{mts}}}[i] \leftarrow \perp$.
   - Otherwise, update $\mathcal{T}_{\mathsf{sid_{mts}}}[i] \leftarrow \mathcal{T}_{\mathsf{sid_{mts}}}[i] + \delta$.
2. C sets T_C to be the median of the non-$\perp$ values within $\mathcal{T}_{\mathsf{sid_{mts}}}$. If all values equal $\perp$, then C sets T_C to $\perp$.
3. Output $(\mathsf{TimeReceived}, \mathsf{sid_{mts}}, T_C)$ to the caller.

Figure 9: Time Synchronization Protocol $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$ in between a client C with access to unused $\mathcal{F}^{C,\Delta_C,\Sigma_C}_{\text{timer}}$ functionalities and a set of n $\mathcal{G}^{C,S_i,\Sigma^*_C}_{\text{clock}}$ functionalities each parameterized by a server S_i from the set $\{S_1, ..., S_n\}$.

Functionality $\mathcal{G}^{P,\mathcal{S},\Sigma}_{\text{multiClock}}$ [$\mathcal{G}_{\text{refClock}}$]

This ideal functionality is identified by a session id $\mathsf{sid}_{\mathsf{mclock}} = (\mathsf{sid}'_{\mathsf{mclock}}, P, \{S_1, ..., S_n\})$ where $\mathsf{sid}'_{\mathsf{mclock}} = \mathsf{sid}_{\mathsf{clock}1} \ldots \mathsf{sid}_{\mathsf{clock}n}$ that denotes the clock's owner P as well as a set of parties $\mathcal{S} = \{S_1, ..., S_n\}$ whose honesty influences the accuracy of the clock. It is also parameterized by the maximum allowable shift Σ from the reference time. It operates as follows.

Corrupt: Upon receiving a message $(\mathsf{Corrupt}, S)$, if $S \in \{S_1, \ldots, S_n\}$ then record S_i as corrupted.

GetTime: Upon receiving input $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{mclock}})$ from party P', ignore this request if $P' \neq P$, otherwise:

1. Send $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{mclock}})$ to the adversary. Wait for a response from the adversary, of the form $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{mclock}}, \sigma, L)$ where L is the list of servers that are deemed to provide non-$\perp$ timing measurements.
2. If $\sigma == \perp$, output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}}, \perp)$ to P.
3. Else send $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$. Denote its response as T.
   (a) If the majority of servers in L are corrupted or $|\sigma| \leq \Sigma$, output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{mclock}}, T + \sigma)$ to P.
   (b) Else output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{mclock}}, \perp)$ to P.

Figure 10: Ideal functionality $\mathcal{G}^{P,\mathcal{S},\Sigma}_{\text{multiClock}}$ that outputs a time influenced by the corruption status of servers in $\mathcal{S}$. Note that the $\mathcal{G}^{P,S,\Sigma}_{\text{clock}}$ functionality in Fig. 4 is a special case of this one with a singleton set $\mathcal{S} = \{S\}$.

- There are implicitly two uses of $\mathcal{G}_{\text{clock}}$ here: the server's clock $\mathcal{G}^{S,S,\Sigma_S}_{\text{clock}}$ in the real protocol and the entire ideal functionality $\mathcal{G}^{C,S,\Sigma^*_C}_{\text{clock}}$. We stress the lack of circularity here, as shown in Fig. 1: the first clock is a subroutine of $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{timeSync}}$ whereas the second clock is an ideal abstraction of it.
- The protocol $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$ contains Σ_S as a parameter but its specification in Fig. 9 never mentions Σ_S explicitly. Instead, the only impact of Σ_S is its influence over Σ^*_C, as shown in Theorem 2.

In comparison to the single-server case, this protocol offers one drawback and one benefit. The extra timer adds the price of $2 \cdot \Sigma_C$ additional shift to the time computed by the client. On the plus side, the multiple server protocol can guarantee approximate correctness even if some servers are corrupted, due to the following two observations. First, uncorrupted measurements must be close to the reference time G. Second, if the majority of servers are uncorrupted, then the median time must be bounded on both sides by uncorrupted samples.

These observations yield an approximate correctness guarantee that is quite robust! We do *not* require that all or even most timing measurements reach the client; on the contrary, the adversary might corrupt, drop, or delay almost all requests. Additionally, the adversary may corrupt parties adaptively and may choose their responses conditioned upon the timing measurements of the honest parties. We simply require the following constraint: of the servers whose interactions result in the client receiving a timing measurement (i.e., anything but $\perp$), a majority of those servers are uncorrupted.[1]

The ideal functionality $\mathcal{G}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}_{\text{multiClock}}$ specified in Figure 10 formally captures this accuracy constraint.

Theorem 3 (Multiple server UC security). Given parameters that satisfy $\Sigma'_C \geq 2.5 \cdot \Delta_C + 3 \cdot \Sigma_C + \Sigma_S$, the multiple server approximate time synchronization protocol $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$ GUC-realizes $\mathcal{G}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}_{\text{multiClock}}$.

Proof. As before, the simulator $\mathcal{S}$ internally runs an emulated copy of the real world protocol $\pi^{\Delta_C,\Sigma_C,\Sigma_S}_{\text{multiTimeSync}}$, where $\mathcal{S}$ plays the role of $\mathcal{E}$ inside this simulation. In particular, $\mathcal{S}$ internally emulates the execution of adversary $\mathcal{A}$, all n $\mathcal{F}^{C,\Delta_C,\Sigma_C}_{\text{timer}}$ functionalities, all the instances of $\mathcal{F}_{\text{auth}}$, and each of the involved parties. In addition, $\mathcal{S}$ relays the

[1] This threshold constraint corresponds to the strong "sleepy model of consensus" of Pass and Shi [14] and Micali [15].

messages sent from the emulated parties to $\mathcal{G}_{\text{refClock}}$ and $\mathcal{G}_{\text{clock}}^{C,S,\Sigma^*_C}$, and from $\mathcal{G}_{\text{refClock}}$ and $\mathcal{G}_{\text{clock}}^{C,S,\Sigma^*_C}$ to the emulated parties.

While conducting this simulation, $\mathcal{S}$ monitors the traffic of its emulated $\mathcal{A}$ and $\mathcal{E}$, along with any messages that $\mathcal{S}$ directly receives from $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$. The messages that $\mathcal{S}$ views/receives causes it to make changes in the ideal world or the emulated world.

Simulating $\mathsf{GetTime}$**:** When $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ sends a message of the form $(\mathsf{Sleep}, \mathsf{sid}_{\mathsf{mclock}})$, then $\mathcal{S}$ instantiates C with the environmentally-provided message $(\mathsf{GetTime}, \mathsf{sid}_{\mathsf{mts}})$.

Simulating $\mathsf{Continue}$**:** When $\mathcal{S}$ observes $\mathcal{A}$ sending a $\mathsf{Continue}$ message to the emulated C, then $\mathcal{S}$ sends a $\mathsf{Continue}$ message to C. (Upon receiving this message, the emulated C might invoke $\mathsf{QueryClock}$ or $\mathsf{CheckTimeout}$.)

Maintaining Records: Upon receiving a message of the form $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{clock}i}, T_i)$ from $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma^*_C}$, if $T_i == \bot$, initialize an empty list L if there does not exist one. Append i to list L. Subsequently, if a $\mathsf{CheckTimeout}$ procedure ever returns a $\bot$ when querying the ith timer, then remove i from L.

Relaying messages to $\mathcal{E}$**:** When the emulated $\mathcal{A}$ sends a message m to the emulated $\mathcal{E}$, then $\mathcal{S}$ relays m to the real environment. Conversely, $\mathcal{S}$ forwards messages sent by the real $\mathcal{E}$ to the emulated $\mathcal{A}$.

Corrupting a server: When $\mathcal{A}$ sends a $(\mathsf{Corrupt}, S_i)$ message in the emulated world, then $\mathcal{S}$ sends a $(\mathsf{Corrupt}, S_i)$ message to $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$.

Completing the simulation: When the emulated C sends to $\mathcal{E}$ its output $(\mathsf{TimeReceived}, \mathsf{sid}_{\mathsf{mts}}, T_C)$, then $\mathcal{S}$ knows both *when* and *what* to send back to $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$. If $T_C == \bot$, then $\mathcal{S}$ sends $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{mclock}}, \bot, \bot)$ to $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$. Otherwise: $\mathcal{S}$ sends $\mathsf{GetTime}$ to $\mathcal{G}_{\text{refClock}}$ to retrieve the current reference time T and also retrieves the list L. Then, $\mathcal{S}$ computes the shift $\sigma_{sim} = (T_C - T)$ and sends $(\mathsf{Wake}, \mathsf{sid}_{\mathsf{mclock}}, \sigma_{sim}, L)$ to $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$.

In the ideal model (namely in the execution of $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ with $\mathcal{S}$), $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ receives from $\mathcal{S}$ an offset σ'_{Sim} and a list of the servers whose $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$ boxes output $\bot$ in the simulated $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$. It is also informed when a server is corrupted.

From this information $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ computes T_C as $\mathsf{G}_x + \sigma'_{\text{Sim}}$ where, if fewer than half of the servers whose $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_S}$ boxes did not output $\bot$ are corrupted, $\sigma'_{\text{Sim}} \leq \Sigma'_C$, the maximum offset that a $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ box will allow.

The value output by $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ is selected as the median of the values returned from its $\mathcal{G}_{\text{clock}}^{C,S,\Sigma^*_C}$ boxes plus a δ corresponding to the time passed since the response was received. The value returned from $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma^*_C}$ is of the form $T_i = \mathsf{G}_i + \sigma_i$ where $|\sigma_i| \leq \Sigma^*_{C_{S_i}}$ the max offset for the $\mathcal{G}_{\text{clock}}^{C,S,\Sigma^*_C}$ owned by server S_i if S_i is uncorrupted. The client in $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ will then add a δ to this corresponding to the elapsed time since receiving the response T_i from the ith $\mathcal{G}_{\text{clock}}^{C,S,\Sigma^*_C}$. The delay δ is computed like before as $\mathsf{G}_2 + \sigma_2 - \mathsf{G}_1 - \sigma_1$ and is at most Δ'_C. Finally, it must be the case that $\Delta'_C > \Delta_C$ or else valid responses could timeout while C is waiting for all $\mathcal{G}_{\text{clock}}^{C,S_i,\Sigma^*_C}$ to respond.

In this case $\mathsf{G}_2 = \mathsf{G}_x$ as the $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$ is queried for a δ when all the responses are received in the simulation and there is not a chance for the environment to update the reference time between this point and when it is obtained by $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$. Additionally, $\mathsf{G}_1 = \mathsf{G}_i$ as the timer is started once the response is received. Therefore, $\mathsf{G}_x - \mathsf{G}_i \leq \Delta'_C + 2 * \Sigma_C$ where Σ_C is, as before, the max allowable shift for a C's $\mathcal{F}_{\text{timer}}^{C,\Delta_C,\Sigma_C}$.

The T_C output by $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ is $\mathsf{G}_i + \sigma_i + \delta_i$ so in order to properly simulate $\mathcal{S}$ must be able to input a σ_{Sim} that will make $\mathsf{G}_x + \sigma'_{\text{Sim}}$ equal to $\mathsf{G}_i + \sigma_i + \delta$. G_x is at most $\Delta'_C + 2 \cdot \Sigma_C$ greater than G_i, σ_i is at most $\Sigma^*_{C_{S_i}}$, and $\delta \leq \Delta'_C$. Finally, recall from Section 4.2 that $\Sigma^*_{C_{S_i}} \leq \frac{\Delta_C}{2} + \Sigma_C + \Sigma_{S_i}$.

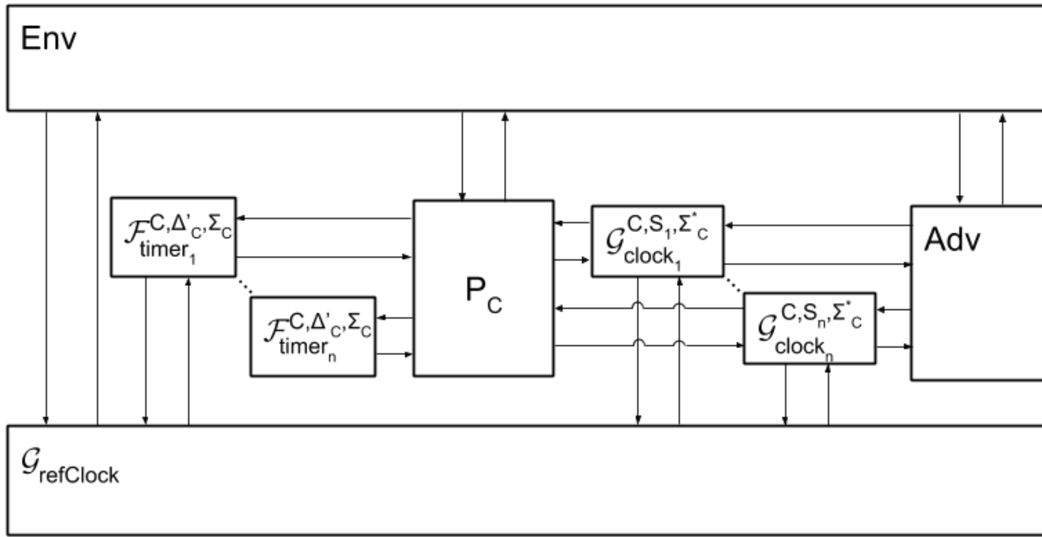


Fig. 11: Interactions between participants in the real world execution of multi-server time sync

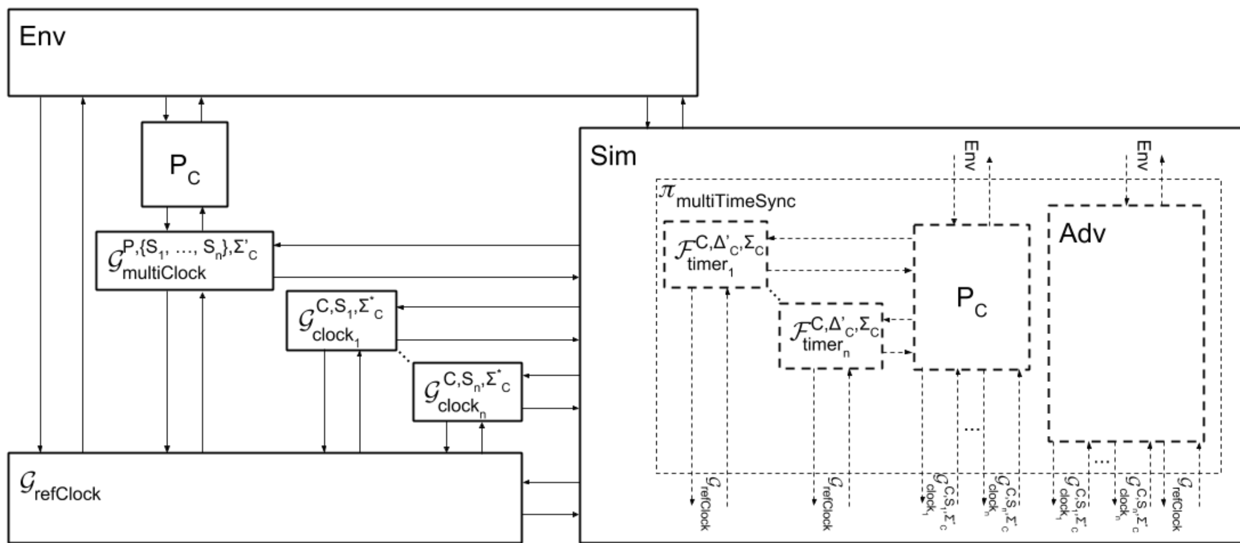


Fig. 12: Interactions between participants in the ideal world execution of multi-server time sync, including the emulation of the real world inside of the simulator.

Combining the above bounds yields

$$|\sigma'_{\text{Sim}}| \leq 2.5 \cdot \Delta_C + 3 \cdot \Sigma_C + \Sigma_S.$$

Therefore, it suffices for the Σ'_C for the $\mathcal{G}_{\text{multiClock}}^{P,\{S_1,\ldots,S_n\},\Sigma'_C}$ to be $2(\Delta'_C + \Sigma_C)$ larger than the Σ^*_C for the simulated $\mathcal{G}_{\text{clock}}^{C,S,\Sigma^*_C}$ in order for the simulator to be able to simulate correctly, proving Theorem 3. □

5.2 Multiple Strata Network Time

In this section, we add another feature of the network time protocol: a hierarchical structure to distribute the network load required to propagate network time. Participants in NTP are stratified, with stratum-0 servers possessing their own source of time $\mathcal{G}_{\text{clock}}^{S,S,\Sigma_0}$ and all other participants serving as both clients *and* servers. We restrict our attention to the case in which NTP servers communicate within the following 'rigid topology.'

- We insist that each individual machine be statically pegged to a single stratum forever. We impose this restriction so that we may compose invocations of $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$. In the more realistic scenario where machines can change strata, the UC composition theorem breaks down since a feedback loop may occur where a client provides an input into its own time measurement.
- We require that machines in stratum j only take timing measurements from servers located in stratum $j-1$ and thus only provide time to stratum $j+1$ clients. We impose this restriction merely to simplify our calculations in Theorem 4; the UC composition theorem would enable more complicated analyses if so desired.
- We use the following parameters: Stratum 0 servers are within shift Σ_0 of $\mathcal{G}_{\text{refClock}}$. All machines in higher strata have timers with maximum delay Δ^* and maximum shift Σ^*.

Additionally, we make two observations that generalize the work we have already provided. First $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ continues to be well-defined if it receives multi-server ideal functionalities $\mathcal{G}_{\text{multiClock}}^{P,\mathcal{S},\Sigma}$ as subroutines rather than single-server functionalities. Additionally, if all servers within all of the $\mathcal{G}_{\text{multiClock}}^{P,\mathcal{S},\Sigma}$ used by the client have the same Σ_S bound, then the statement and proof of Theorem 3 continue to hold in this setting.

Second, we may further generalize $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ by permitting $\mathcal{S}$ to be a set of sets and by modifying the consistency rule in step 3a to state that the adversary only has free reign to alter the time if the set of servers $\mathcal{S}$ is 'bad,' as defined below. Furthermore, Theorem 3 continues to provide bounds on consensus and accuracy in this case as well.

Definition 3. During an execution of $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$, we denote a server as *bad* if it is corrupted. Additionally, a set of parties is deemed to be *bad* if a majority of elements are corrupted. Note that these elements may either be parties or sets themselves; in the latter case, the notion of corruptedness is defined recursively.

Here, the majority vote is only taken over elements that respond to the client's request for timing measurements within its maximum allowable delay Δ (which we stress that the adversary has the capacity to control). The fact that non-responsive servers do not factor either positively or negatively into the badness of a set of parties is consistent with the sleepy model of consensus (cf. footnote 1).

These two observations and the UC composition theorem allow us to bound the worst-case error when timing measurements percolate down multiple strata.

Theorem 4. Consider several machines who conduct multiple server timing measurements following the network topology specified above. Then, a client C at stratum j who executes $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ using the set of servers $\mathcal{S}$ will receive a time whose inaccuracy is bounded by $\Sigma_j \geq 2.5j \cdot \Delta^* + 3j \cdot \Sigma^* + \Sigma_0$ as long as the set $\mathcal{S}$ is not bad.

Proof. The proof of this theorem is straightforward. First, we apply the UC theorem to replace all instances of the $\pi_{\text{multiTimeSync}}^{\Delta_C,\Sigma_C,\Sigma_S}$ protocol (for C and for all of the timeservers who get their measurements from lower strata as well) with instances of the ideal functionality $\mathcal{G}_{\text{multiClock}}^{P,\mathcal{S},\Sigma}$. We remark that our definition of good and bad timeservers matches precisely with the (modified) constraint for timing consensus in step 3a of $\mathcal{G}_{\text{multiClock}}^{P,\mathcal{S},\Sigma}$. Ergo, Theorem 3 upper bounds the inaccuracy of all good servers at stratum i as $\Sigma_i \geq 2.5\Delta^* + 3\Sigma^* + \Sigma_{i-1}$. Summing these inequalities for all $i \in \{1, 2, \ldots, j\}$ yields the desired result. □

We stress that Theorem 4 provides a worst case bound. By contrast, in the remainder of this section we mathematically analyze and computationally simulate average case error propagation over multiple strata by timeservers with *accidental* rather than adversarial timing inaccuracies. Suppose that these accidental network asymmetries and server clock imprecisions contribute to delays δ and shifts σ (respectively) that are randomly distributed (e.g., using a uniform or Gaussian distribution). In this average case setting, the central limit theorem provides much more stringent bounds on error propagation.

- Within a single stratum, the fact that each client invokes multiple servers means that their individual shift errors are very likely to interfere destructively. Network jitter effects do cause a noticeable delay, however.
- The effect of network asymmetry at a particular stratum i upon the shift (as found in Theorems 2 and 3) is also reduced significantly when progressing down the strata.

The net result of the average case analysis is that the expected shift at a high stratum is influenced mostly by the magnitude of the network asymmetry at that stratum only. Hence, a high stratum timeserver with low latency network connections may actually possess *better* timing measurements than a low stratum timeserver in a high latency environment, even if the latter contributes toward the timing measurements of the former.

6 Using Approximate Time in UC Protocols

In this section, we explore the ramifications of injecting approximate time into existing, time-agnostic GUC protocols and functionalities. Although our principal interest is in the expiration and revocation of PKI certificates, much of our analysis applies generically to any protocol whose security depends in part on the approximate accuracy of time.

6.1 Adding Time to Existing Protocols & Functionalities

We begin by considering generically the influence of time upon existing UC protocols and functionalities that have previously been proved secure in the usual untimed, asynchronous setting. The following straightforward theorem states that UC security continues to hold for all untimed protocols:

Theorem 5. Let π be a protocol that GUC-realizes functionality $\mathcal{G}$ in an untimed setting. Then, π continues to GUC-realize $\mathcal{G}$ even in the presence of exact or approximate time functionalities like $\mathcal{G}_{\text{refClock}}$ or $\mathcal{G}_{\text{clock}}^{P,S,\Sigma}$.

Proof. This theorem follows immediately from the UC security guarantee in the presence of global functionalities. Since no environment can distinguish π from $\mathcal{G}$, in particular this condition must hold for environments that either keep track of, or have access to, a time functionality such as G. □

More interestingly, we can automatically *add* a time dependency on top of protocols that previously lacked an understanding of time. In this section, we focus upon protocols and functionalities of the following type.

Definition 4 (Binary decider)**.** We say that a protocol π_{bin} is a *binary decider* if the following constraints hold:

- Only one party P receives output. We denote the collection of inputs by $\vec{x}$ and the output as (b, y).
- The value b is a single bit. (By contrast, y and the elements of $\vec{x}$ are strings of arbitrary length.)

Intuitively, binary deciders provide P with a putative output and a *verification bit* that determines whether P chooses to accept the answer. Protocols of this form include bit commitments, zero-knowledge proofs, and (of particular interest to us) signature and certificate verification checks.

Given any binary decider protocol π_{bin}, Fig. 13 constructs a new protocol $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ that operates identically to π_{bin} except that it subjects the verification bit to a new constraint that rejects responses when P's time is past a threshold t^*. For simplicity, in this section we assume that P's clock $\mathcal{G}_{\text{clock}}^{P,P,\Sigma}$ is corrupted only if P is, so the only relevant parameter is the maximum clock shift Σ. Note that t^* and Σ are explicitly provided to the adversary; $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ makes no attempt to hide them.

Next, we demonstrate a canonical method to transform ideal functionalities analogously. Given any $\mathcal{G}_{\text{bin}}$, Fig. 14 constructs a new ideal functionality $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ that executes $\mathcal{G}_{\text{bin}}$ as a subroutine and also determines the reference time G. It is more restrained than before: the adversary can only change an otherwise-valid response if G is close to the threshold t^*.

Next, we show that these two transformations produce identical outcomes. Intuitively, the adversary can control the output bit of $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ only when her clock skew capability can be used to affect $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$'s output.

Theorem 6. Suppose that the binary decider π_{bin} GUC-realizes $\mathcal{G}_{\text{bin}}$ and that π_{bin}'s output party P has access to a clock $\mathcal{G}_{\text{clock}}^{P,P,\Sigma}$. Then, $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ GUC-realizes $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$.

Proof. Let $\mathcal{D}_\pi$ denote the dummy adversary against π_{bin} and $\mathsf{Sim}_{\mathcal{G}}$ denote its corresponding simulator. Additionally, let $\mathcal{D}_{\hat{\pi}}$ denote the dummy adversary against $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$. Our objective is to design a simulator $\mathsf{Sim}_{\hat{\mathcal{G}}}$ that connects with $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ and produces a view indistinguishable from $\mathcal{D}_{\hat{\pi}}$.

$\mathsf{Sim}_{\hat{\mathcal{G}}}$ emulates the real world interaction with $\mathcal{D}_{\hat{\pi}}$ in its head, and it behaves as follows during each step of the execution of $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ in the ideal world.

1. $\mathsf{Sim}_{\hat{\mathcal{G}}}$ sends (t^*, Σ) to $\mathcal{E}$ and relays $\mathcal{E}$'s response to $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$, just as $\mathcal{D}_{\hat{\pi}}$ does with $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$.
2. During the execution of $\mathcal{G}_{\text{bin}}$, the simulator $\mathsf{Sim}_{\hat{\mathcal{G}}}$ simply acts as $\mathsf{Sim}_{\mathcal{G}}$ would.
3. $\mathsf{Sim}_{\hat{\mathcal{G}}}$ observes the shift σ that $\mathcal{D}_{\hat{\pi}}$ applies to P's clock and then sends $b' = b \wedge [t \stackrel{?}{\leq} t^*]$ to $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$.

Note that $\mathsf{Sim}_{\hat{\mathcal{G}}}$ has nothing to do during steps 4-5.

The messages sent to the environment during steps 1-2 are clearly identical to those of $\mathcal{D}_{\hat{\pi}}$. The only other message received by $\mathcal{E}$ is the output (b', y).

Ergo, to prove simulation, it suffices to show that the output values b' are identical in $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ and $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$. This follows from the fact that the adversary's ability to shift P's clock is bounded such that:

$$[t \stackrel{?}{\leq} t^*] = \begin{cases} 1, \text{ if } \mathsf{G} < t^* - \Sigma, \\ 0, \text{ if } \mathsf{G} > t^* + \Sigma, \\ \text{controlled by } \mathsf{Sim}_{\hat{\mathcal{G}}}, \text{ otherwise.} \end{cases}$$

Hence, $\mathsf{Sim}_{\hat{\mathcal{G}}}$'s inability to influence b' in the first two cases of step 4 is irrelevant because $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$'s output must equal b and 0, respectively. In the third case, $\mathsf{Sim}_{\hat{\mathcal{G}}}$ chooses b' just as $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ does during step 4, so the simulation is perfect. □

6.2 Application to Public Key Infrastructure

In this section, we augment Canetti, Shahaf, and Vald's GUC analysis of signature-based authentication [13] to enable revocation and expiration. The goal of [13] is to provide, within the UC framework, modeling and analysis of the process of (1) generating signing and verification keys for a digital signature scheme, (2) certifying the verification key, and (3) using these keys to authenticate and verify messages by way of signing them on the sending end and verifying the verification key and the signature on the receiving end. An important innovation within [13] is to provide adequate treatment to the fact that the certified verification keys are universally available and may be used to authenticate several messages within many different protocols. The main components of their modeling are:

- A public bulletin-board $\mathcal{G}_{\text{bb}}$ where parties can publicly associate values with their identities. The bulletin board is globally available and guarantees authenticity (a party can only associate values with her own true ID).
- A certification functionality $\mathcal{G}_{\text{cert}}$ that provides its owner with a public key, that it then posts globally using $\mathcal{G}_{\text{bb}}$. When the owner provides a message m to be signed, $\mathcal{G}_{\text{cert}}$ returns an idealized signature string S; later on, when asked by anyone, $\mathcal{G}_{\text{cert}}$ correctly verifies valid message-signature pairs.

Protocol $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$ [π_{bin}, $\mathcal{G}_{\text{clock}}^{P,P,\Sigma}$]

When instantiated with inputs $\vec{x}$ where party P's input has the form $x_P = (x'_P, t^*, \Sigma)$, do the following:

1. P sends (t^*, Σ) to $\mathcal{A}$ and waits for an ok response.
2. The parties execute the subroutine protocol π_{bin} on inputs $\vec{x}'$, where $x'_{P'} = x_{P'}$ for all $P' \neq P$. Eventually, P produces output of the form (b, y).
3. Before submitting this output, P queries her clock (with max shift Σ) for the current time t. This request invokes $\mathcal{A}$ to provide a shift σ.
4. Compute $b' = b \wedge [t \overset{?}{\leq} t^*]$.
5. P outputs (b', y) to the caller.

Figure 13: Time-conditional protocol $\hat{\pi}_{\text{bin}}^{\Sigma,t^*}$. It connects to two subroutines: a binary decider π_{bin} and P's clock.

Functionality $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ [$\mathcal{G}_{\text{bin}}$, $\mathcal{G}_{\text{refClock}}$]

When instantiated with inputs $\vec{x}$ where party P's input has the form $x_P = (x'_P, t^*, \Sigma)$, do the following:

1. Send (t^*, Σ) to $\mathcal{A}$. Wait for an ok response.
2. Send $\vec{x}'$ (as defined in Fig. 13) to subroutine $\mathcal{G}_{\text{bin}}$. Eventually, receive a response of the form (b, y).
3. Query the adversary for a value b'.
4. Obtain the reference time G from $\mathcal{G}_{\text{refClock}}$. Update the value of b' as follows:
   - If $\mathsf{G} < t^* - \Sigma$, then set $b' = b$.
   - If $\mathsf{G} > t^* + \Sigma$, then set $b' = 0$.
   - If $b = 0$, set $b' = 0$.
5. Output (b', y) to P.

Figure 14: The time-conditional $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$ functionality with two subroutines: $\mathcal{G}_{\text{refClock}}$ and an untimed binary decider $\mathcal{G}_{\text{bin}}$. The max shift Σ of P's clock affects the behavior of $\hat{\mathcal{G}}_{\text{bin}}^{\Sigma,t^*}$, even though they never communicate.

Functionality $\mathcal{G}_{\text{timed-bb}}$

Report: Upon receiving from party P a message of the form $(\mathsf{Register}, P, \mathsf{serial}, v, t)$, send the message to the adversary and wait for an ok response. If this is the first request involving (P, serial) then record the tuple $(P, \mathsf{serial}, v, t)$. Otherwise, ignore the message.

Retrieve: Upon receiving from some party P_i or the adversary a message $(\mathsf{Retrieve}, P_j, \mathsf{serial})$, retrieve the record r containing (P_j, serial) and return $(\mathsf{Retrieve}, r)$. If no such record exists, return $(\mathsf{Retrieve}, \perp)$.

ChangeExpiration: Upon receiving from party P a message of the form $(\mathsf{ChangeExpiration}, P, \mathsf{serial}, t')$, retrieve the record of the form $(P, \mathsf{serial}, v, t)$. If $t' < t$, then replace t with t' in this record. Otherwise (including if the record does not exist), do nothing.

Figure 15: A public bulletin board that augments [13, Fig. 3] to incorporate an expiration time.

- An existentially unforgeable signature scheme $\mathcal{F}_{\text{sig}}$ that is used to realize $\mathcal{G}_{\text{cert}}$.
- An authenticated message transmission functionality $\mathcal{F}_{\text{cert-auth}}$ that properly models the non-deniability of an authenticated message, i.e., that it is possible for third parties to verify whether a given message was indeed sent and signed by the sender. (This stands in contrast with the "standard" authenticated message transmission functionality in Fig. 2, which only allows the specified receiver to tell whether a message is authentic.)
- A protocol π_{auth} for realizing $\mathcal{F}_{\text{cert-auth}}$ using $\mathcal{G}_{\text{cert}}$, by way of signing the message by the sender and later verifying the signature on the receiving end, against $\mathcal{G}_{\text{cert}}$ of the sender. An important aspect of this analysis is that $\mathcal{G}_{\text{bb}}$ is global and exists regardless of the specific instance of $\mathcal{G}_{\text{cert}}$. Furthermore, while $\mathcal{G}_{\text{cert}}$ is specific for a single "party" (i.e., long-term entity), it is global in the sense that it exists regardless of any single message-authentication instance.

We remark that, while the analysis of [13] only considers the setting in which each party registers a single certificate, one can verify that their modeling and proofs continue to hold when each pid is replaced with a $(\mathsf{pid}, \mathsf{serial})$ pair, where serial denotes a unique identifier of a certificate issued by a particular CA [32, §4.1.2.2]. As a result, the same analysis applies when participants can request the creation of multiple certificates, which is essential when certificates expire.

Adding time awareness and certificate revocation. To capture expiration and revocation requests, we extend Canetti et al.'s public bulletin board $\mathcal{G}_{\text{bb}}$ into a time-aware bulletin board $\mathcal{G}_{\text{timed-bb}}$ that supports expiration and revocation. Our extension augments the Report method to record the expiration time and adds a new third method called ChangeExpiration to support revocations. Figure 15 shows the details.

Then, we may apply Theorem 6 to "lift" the certification functionality $\mathcal{G}_{\text{cert}}$ and the authentication functionality $\mathcal{F}_{\text{cert-auth}}$ described in [13] to their respective time-dependent versions. In the lifted $\hat{\mathcal{G}}_{\text{cert}}^{\Sigma,t^*}$, the recipient determines the appropriate threshold t^* to use by querying $\mathcal{G}_{\text{timed-bb}}$ with the sender's credentials. Finally, the timed version of the real authentication protocol $\pi_{\text{time-auth}} \triangleq \hat{\pi}_{\text{auth}}^{\Sigma,t^*}$ GUC-realizes the timed ideal functionality $\mathcal{F}_{\text{time-cert-auth}} \triangleq \hat{\mathcal{F}}_{\text{cert-auth}}^{\Sigma,t^*}$ based upon Theorem 6 and [13, Claim 4.4].

This protocol $\pi_{\text{time-auth}}$ combines all of the components designed so far to provide time-based non-deniable authentication, as shown in Fig. 1. It requires a clock, which we know how to instantiate from Sections 4-5. Additionally, it uses $\mathcal{G}_{\text{cert}}$ as a subroutine just as its untimed counterpart did.

By imbuing this subroutine with a notion of time itself, $\hat{\mathcal{G}}_{\text{cert}}^{\Sigma,t^*}$ can interface with our timed bulletin board $\mathcal{G}_{\text{timed-bb}}$ to attest that (1) the signature is valid, just as before and (2) the certificate hasn't yet expired, using the new expiration time t contained within the record returned by $\mathcal{G}_{\text{timed-bb}}$'s Retrieve command. Furthermore, in case of key compromise, the signer can request that her certificate be revoked.

Impact upon use of the PKI today. An important lesson learned from this modeling is that real-life certificate revocation lists and online certificate status requests must continue to answer requests about revoked or expired certificates during the interval $[t^*, t^* + \Sigma]$ because clients may not be able to adjudicate them correctly on their own before this time. Here, Σ denotes the maximum shift expected by Theorems 2 and 3 for all clients on the Internet. After this interval, the adversary cannot convince any clients of the validity of a revoked or expired certificate via network manipulation, so the CA may forget about its existence.

Acknowledgments

We thank Oxana Poburinnaya for her many contributions to this project, and the anonymous CSF reviewers for their helpful insights. This material is based upon work supported by the National Science Foundation under Grant No. 1414119. The first author is also supported by ISF grant 1523/14 and is a member of the Check Point Institute for Information Security.

References

[1] R. Canetti, K. Hogan, A. Malhotra, and M. Varia, "A universally composable treatment of network time," in *IEEE 30th Computer Security Foundations Symposium, CSF 2017, Santa Barbara, California*. IEEE Computer Society, 2017, pp. 360–375.

[2] D. Mills, J. Martin, J. Burbank, and W. Kasch, *RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification*. Internet Engineering Task Force (IETF), 2010, http://tools.ietf.org/html/rfc5905.

[3] A. Malhotra, I. E. Cohen, E. Brakke, and S. Goldberg, "Attacking the network time protocol," in *Network and Distributed System Security Symposium*, 2016. [Online]. Available: http://www.internetsociety.org/sites/default/files/blogs-media/attacking-network-time-protocol.pdf

[4] A. Malhotra and S. Goldberg, "Attacking NTP's authenticated broadcast mode," *Computer Communication Review*, vol. 46, no. 2, pp. 12–17, 2016. [Online]. Available: http://doi.acm.org/10.1145/2935634.2935637

[5] A. Malhotra, M. V. Gundy, M. Varia, H. Kennedy, J. Gardner, and S. Goldberg, "The security of NTP's datagram protocol," in *Financial Cryptography and Data Security - 21st International Conference, FC, Malta*, 2017. [Online]. Available: http://eprint.iacr.org/2016/1006

[6] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir, "Taming the 800 pound gorilla: The rise and decline of NTP DDoS attacks," in *Internet Measurement Conference*, 2014, pp. 435–448. [Online]. Available: http://doi.acm.org/10.1145/2663716.2663717

[7] D. Mills, *RFC 4330: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. Internet Engineering Task Force (IETF), 2006, https://tools.ietf.org/html/rfc4330.

[8] https://sourceforge.net/projects/ptpd2/.

[9] https://chrony.tuxfamily.org/.

[10] http://www.openntpd.org/.

[11] http://nwtime.org/projects/ntimed/.

[12] https://roughtime.googlesource.com/roughtime.

[13] R. Canetti, D. Shahaf, and M. Vald, "Universally composable authentication and key-exchange with global PKI," in *Public-Key Cryptography, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9615. Springer, 2016, pp. 265–296. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-49387-8_11

[14] R. Pass and E. Shi, "The sleepy model of consensus," Cryptology ePrint Archive, Report 2016/918, 2016, http://eprint.iacr.org/2016/918.

[15] S. Micali, "ALGORAND: the efficient and democratic ledger," *CoRR*, vol. abs/1607.01341, 2016. [Online]. Available: http://arxiv.org/abs/1607.01341

[16] R. Canetti, "Universally composable signature, certification, and authentication," in *IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2004, p. 219. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/CSFW.2004.24

[17] "https://www.nist.gov/pml/time-and-frequency-division/time-services/nist-authenticated-ntp-service."

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[19] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *J. Cryptology*, vol. 3, no. 2, pp. 99–111, 1991.

[20] T. Matsuo and S. Matsuo, "On universal composable security of time-stamping protocols," in *Conference on Applied Public Key Infrastructure: 4th International Workshop, IWAP*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2005, pp. 169–181. [Online]. Available: http://dl.acm.org/citation.cfm?id=1564104.1564121

[21] A. Buldas, P. Laud, M. Saarepera, and J. Willemson, *Universally Composable Time-Stamping Schemes with Audit*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 359–373. [Online]. Available: http://dx.doi.org/10.1007/11556992_26

[22] O. Goldreich, *Concurrent Zero-Knowledge with Timing, Revisited.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 27–87. [Online]. Available: http://dx.doi.org/10.1007/11685654_2

[23] Y. T. Kalai, Y. Lindell, and M. Prabhakaran, "Concurrent general composition of secure protocols in the timing model," in *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, ser. STOC '05. New York, NY, USA: ACM, 2005, pp. 644–653. [Online]. Available: http://doi.acm.org/10.1145/1060590.1060687

[24] M. Backes, P. Manoharan, and E. Mohammadi, "Tuc: Time-sensitive and modular analysis of anonymous communication," in *IEEE Computer Security Foundations Symposium*, July 2014, pp. 383–397.

[25] I. Vajda, "On the analysis of time-aware protocols in universal composability framework," *International Journal of Information Security*, vol. 15, no. 4, pp. 403–412, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10207-015-0300-2

[26] J. Katz, U. Maurer, B. Tackmann, and V. Zikas, *Universally Composable Synchronous Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 477–498. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36594-2_27

[27] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," Cryptology ePrint Archive, Report 2000/067 (2013 version), 2013, https://eprint.iacr.org/2000/067/20130717:020004.

[28] B. Dowling, D. Stebila, and G. Zaverucha, "Authenticated network time synchronization," in *USENIX Security Symposium*. Austin, TX: USENIX Association, Aug. 2016, pp. 823–840. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/dowling

[29] T. Mizrahi, *RFC 7384 (Informational): Security Requirements of Time Protocols in Packet Switched Networks*. Internet Engineering Task Force (IETF), 2012, http://tools.ietf.org/html/rfc7384.

[30] E. Itkin and A. Wool, "A security analysis and revised security extension for the precision time protocol," *CoRR*, vol. abs/1603.00707, 2016. [Online]. Available: http://arxiv.org/abs/1603.00707

[31] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *Theory of Cryptography*, 2007, pp. 61–85.

[32] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), Internet Engineering Task Force, May 2008, updated by RFC 6818. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt