Analysis of Client-side Security for Long-term Time-stamping Services

Long Meng, Liqun Chen *

Abstract. Time-stamping services produce time-stamp tokens as evidences to prove that digital data existed at given points in time. Time-stamp tokens contain verifiable cryptographic bindings between data and time, which are produced using cryptographic algorithms. In the ANSI, ISO/IEC and IETF standards for time-stamping services, cryptographic algorithms are addressed in two aspects: (i) Client-side hash functions used to hash data into digests for nondisclosure. (ii) Server-side algorithms used to bind the time and digests of data. These algorithms are associated with limited lifespans due to their operational life cycles and increasing computational powers of attackers. After the algorithms are compromised, time-stamp tokens using the algorithms are no longer trusted. The ANSI and ISO/IEC standards provide renewal mechanisms for time-stamp tokens. However, the renewal mechanisms for client-side hash functions are specified ambiguously, that may lead to the failure of implementations. Besides, in existing papers, the security analyses of long-term time-stamping schemes only cover the server-side renewal, and the client-side renewal is missing. In this paper, we analyse the necessity of client-side renewal, and propose a comprehensive long-term time-stamping scheme that addresses both client-side renewal and server-side renewal mechanisms. After that, we formally analyse and evaluate the client-side security of our proposed scheme.

1 Introduction

Digital data is ubiquitous in our modern world. To prove the existence time of digital data, a time-stamping service produces verifiable cryptographic bindings between digital data and time to form time-stamp tokens. Such cryptographic bindings could be digital signatures, hash values, message authentication codes etc. Most of the bindings are generated through cryptographic algorithms. Therefore, time-stamp tokens are valid only when the underlying cryptographic algorithms remain secure.

For time-stamping services specified in the ANSI [1], IETF [2] and ISO/IEC [3–6] standards, the cryptographic algorithms used to generate time-stamp tokens could be categorized into two sides: (i) Client-side hash functions used to hash data into digests for nondisclosure; (ii) Server-side algorithms used to bind digests of a data item and a given point in time. These algorithms are time-restricted due to their limited operational life cycles and the increasing computational power of attackers [7]. For instance, the upcoming quantum computers are considered to break some broadly-used signature algorithms [8] and to increase the speed of attacking hash functions [9]. Once the algorithms are compromised, the corresponding time-stamp tokens are no longer valid.

* *Long Meng and Liqun Chen are with the Department of Computer Science, the University of Surrey (e-mail: lm00810@surrey.ac.uk; liqun.chen@surrey.ac.uk).*

However, for many types of digital data, the validity of time-stamp tokens need to be maintained for a long time. For example, the identity information of citizens should be kept permanently; the health records of people follow their lifetimes; mp3 files produced by musicians may last for decades etc. In these cases, the validity periods of time-stamp tokens need to be longer than any individual cryptographic algorithm's lifetime. For the purpose of this paper, *if a time-stamping service (or scheme) is able to prove the existence of data at given points in time through valid and secure time-stamp tokens in a long period of time, which is not bounded with the lifetimes of underlying cryptographic algorithms, we say it is a Long-Term Time-Stamping (LTTS) service (or scheme).* Clearly, for a long-term time-stamping service, time-stamp tokens should be constantly renewed.

The ANSI [1] and ISO/IEC [3–6] standards provide time-stamp renewal mechanisms for both client-side and server-side algorithms. For server-side renewal, the standards clearly say that a requester sends a time-stamp request with inputting a new server-side algorithm identifier, hash value(s) of a data item, and a previous time-stamp token on this data item to a Time-Stamping Authority (TSA), the TSA then produces a new time-stamp token on the input content using the indicated algorithm.

However, the client-side renewal mechanisms in both standards are specified ambiguously. In the ISO/IEC standard, the renewal of client-side hash functions is not mentioned as a motivation for time-stamp renewal, and how to implement the client-side renewal is not explicitly specified. In the ANSI standard, a list of reasons for time-stamp renewal includes that a requester needs to replace the hash value using a stronger hash function, but when a requester "needs" to replace the hash value is not specified in detail. These ambiguities may cause the failure of client-side renewal implementations and therefore the failure of long-term time-stamping services.

In the existing papers [10] and [11], long-term time-stamping schemes based on signatures and hash functions have been formally analysed respectively (the details will be introduced in Section 2). Nevertheless, the analyses are only related to renewal mechanisms of server-side algorithms, the client-side renewal is not covered. Specifically, in the security model of [10], the client-side renewal is not discussed, and client-side hash functions are treated as random oracles. This security notion does not truly model the case of practical implementations. In [11], the client-side hash functions are not considered.

The motivation of this work is based on the following observation. The security of client-side is as significant as server-side, since the time-stamp tokens are generated on the hash values of data items. If the client-side hash functions are broken and the client-side renewal mechanism is not performed effectively, the time-stamp tokens are no longer valid regardless whether the server-side is secure or not. Even if a client-side renewal mechanism is clearly specified, a formal security analysis of the mechanism is necessary and it does not exist in the literature.

In this paper, we provide following contributions:

- We firstly analyse several possible failures of client-side renewal implementations by complying with the ANSI and ISO/IEC standards, and discuss the importance of a well-specified client-side renewal mechanism.
- We then propose a comprehensive long-term time-stamping scheme that addresses both client-side and server-side renewal mechanisms.

– After that, we formally analyse the client-side security of our proposed long-term time-stamping scheme, and provide a quantified evaluation to the client-side security level.

2 Related works

In 1990 [12], Haber and Stornetta introduced the first concept of digital time-stamping with two techniques: linear linking and random witness. In this paper, they also proposed a solution for time-stamp renewal, in which a time-stamp token could be renewed by time-stamping the token with a new implementation before the old implementation is compromised.

In 1993 [13], Bayer, Haber and Stornetta proposed another time-stamping technique: publish linked trees into a widely visible medium (e.g., newspapers). Besides, they spotted that the renewal idea in the 1990 paper [12] is insufficient to time-stamp a digital certificate alone (without the original data being certified). They proposed a corrected renewal solution: time-stamping a (data, signature) pair or a (data, time-stamp) pair to extend the signature or time-stamp's lifetime.

In further years, the ideas of [12, 13] have been polished and recorded into various standards: The NIST standard specified several signature-based time-stamping applications for proving time evidences of digital signatures [14]; the IETF standard [2] (an update is [15]) specified signature-based time-stamping protocols; the ISO/IEC and ANSI standards cover various time-stamping mechanisms and renewal mechanisms. Notice that the time-stamping services in both the NIST and IETF are not specified in long-term, the ANSI and ISO/IEC standards contain the specifications for long-term time-stamping services.

Apart from the standards, the ideas of [13] have been extended into several long-term integrity schemes [16–22], but the security analyses of such schemes were not given until 2016, Geihs et al. formalized this idea separately into a signature-based long-term integrity scheme [10], and a hash-based long-term time-stamping scheme [11] in 2017. These two schemes are related to the security of two types of server-side algorithms: signature schemes and hash functions, and their renewal mechanisms, but the renewal mechanisms for client-side hash functions are not addressed. In [10], the client-side hash functions are ideally modelled as random oracles, and the renewal of client-side hash functions is not discussed; in [11], the client-side hash functions are not considered in the scheme, time-stamp tokens carry out on actual data items.

Similarly, in Geihs' PhD thesis [23] (includes [10, 11]), the signature-based time-stamping scheme is slightly different with [10]: time-stamp tokens are created on a data item and signature pair, and the consideration of client-side hash functions is removed. For all these analyses [10, 11, 23], the client-side security is guaranteed with ideal assumptions.

Nevertheless, the papers [10, 11, 23] provide substantial frameworks for analysing the security of long-term time-stamping schemes. For example, they presented a new computational framework based on [24], and a global time model based on [25] for modelling the computational power of long-lived adversaries; they created "long-term unforgeability" model for the integrity of signature-based time-stamping [10];

they constructed "long-term extraction" model for the integrity of hash-based time-stamping [11], which is an integration of "extraction-based" time-stamping proposed in [26] and "preimage awareness" hash functions defined in [27].

In addition, the security of hash functions in time-stamping has been explored [28–31], and only in [29] Buldas et al. analysed the security of client-side hash functions. They proposed a new notion named "unpredictability preservation" and argued that this property, rather than collision resistance or second preimage resistance (the definitions are in Section 3.1), is necessary and sufficient for client-side hash functions in secure time-stamping. However, their conclusions are not in the case of long-term time-stamping since the time-stamping renewal is not considered in their works.

In this paper, we create a "long-term integrity" model for our long-term time-stamping scheme including both client-side and server-side renewal (will be introduced in Section 6.3). In this model, we follow the computational framework of long-lived adversaries, and refer to the analysis results of server-side security in [10, 11, 23]. In our analysis, we mainly focus on analysing the security at the client-side.

3 Review the ANSI and ISO/IEC time-stamping services

The ISO/IEC 18014 standard specifies time-stamping services in four parts: the framework in Part 1 [3], mechanisms producing independent tokens in Part 2 [4], mechanisms producing linked tokens in Part 3 [5], and traceability of time sources in Part 4 [6]. The ANSI X9.95 standard [1] specifies both independent and linked tokens, which are similar to the mechanisms specified by the ISO/IEC in [3–5].

In this section, we review some common specifications from the first three parts of the ISO/IEC 18014 [3–5] and the ANSI X9.95 [1] standards, which includes the definition of hash functions, two types of time-stamp tokens and time-stamp transactions between entities.

3.1 Hash functions

A secure hash function [32] maps a string of bits of variable (but usually upper bounded) length to a fixed-length string of bits, satisfying the following three properties:

- *Preimage Resistance*: it is computationally infeasible to find, for a given output, an input which maps to this output.
- *Second Preimage Resistance*: it is computationally infeasible to find a second input which maps to the same output.
- *Collision Resistance*: it is computationally infeasible to find any two distinct inputs which map to the same output.

Note that the hash functions discussed in this paper are compression functions. That means, the collision resistance of a hash function implies preimage resistance [33, 34]. In other words, if a hash function is collision resistant, then it is also preimage resistant; if a hash function is not preimage resistant, then it is not collision resistant.

3.2 Types of time-stamp tokens

There are two types of time-stamp tokens that can be generated by a time-stamping service:

1. *Independent tokens*: An independent time-stamp token can be verified without involving other time-stamp tokens. The protection mechanism used to generate this type of tokens can be digital signatures, message authentication codes (MAC), archives or transient keys [1]. For instance, for signature-based time-stamping, a Time-Stamping Authority (TSA) digitally signs a data item and a time value that results a cryptographic binding between the data and time. The data, time and the corresponding signature together form a time-stamp token.
2. *Linked tokens*: A linked time-stamp token is associated with other time-stamp tokens produced by the same methods. The protection mechanism used to generate this type of tokens can be hash functions and a public repository, therefore a time-stamping service generating this type of tokens is referred to "hash-based time-stamping" or "repository-based time-stamping". In specific, a TSA hashes a data item and a time value together and aggregates the hash output with other data items produced at the same time, (e.g., uses a Merkle Tree [35]). The aggregation result can be linked to other data produced at previous times, (e.g., uses linear chain linking [12]). Eventually, the aggregation or linking result is published at a widely visible media (e.g., newspapers). The data, time record, published information, and group values that are contributed to determine the published result, together form a time-stamp token.

3.3 Time-stamp transactions

There are two time-stamp transactions that are performed between a requester and one or more TSAs, or between a requester and a verifier, respectively:

1. Time-stamp request transaction: A requester sends a *time-stamp request* to a TSA and the TSA returns a *time-stamp response* to the requester.
2. Time-stamp verification transaction: A requester sends a *verification request* to a verifier and the verifier returns a *verification response* to the requester.

The data formats of a time-stamp request and response are shown in Fig. 1. *A time-stamp request* contains a "messageImprint" field, which is comprised of a hash value of a data item and its hash function identifier, an "extensions" field and other information.

More specifically, the "extensions" field contains three types of additional information: ExtHash, ExtMethod and ExtRenewal, which work as follows:

1. ExtHash: In this field, a requester could submit multiple "messageImprint" fields, in which each hash value could be computed from a different hash function so that it prevents the failure of any single hash function.
2. ExtMethod: In this field, a requester could indicate a specific protection mechanism (e.g., a digital signature scheme) to bind the data item and time.

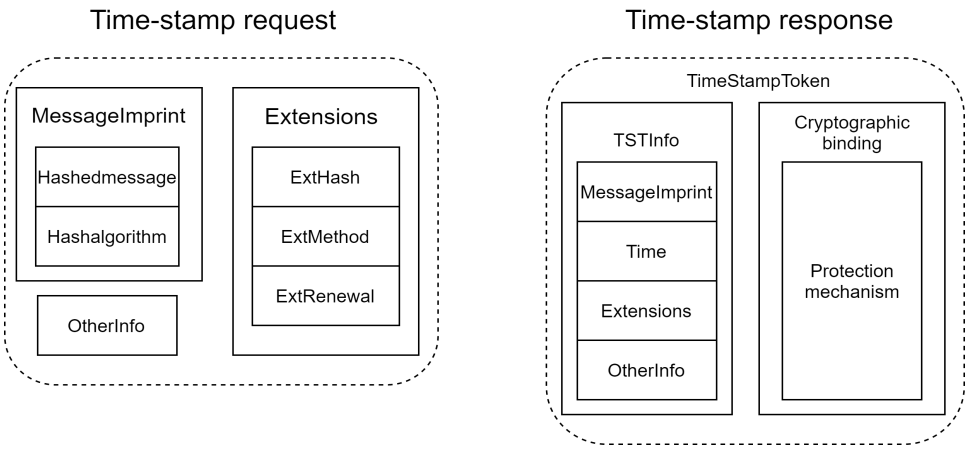

Fig. 1: Data formats of time-stamp request and time-stamp response

3. ExtRenewal: In this field, a requester could submit an existing time-stamp token on the data item in the purpose of extending the validity period of the time-stamp token.

After the TSA receives the request, it adds the current time to the request content to form a "TSTInfo" structure, and produces a cryptographic binding on the TSTInfo by using the indicated protection mechanism or a default one if it is not indicated. The TSTInfo and the cryptographic binding together form a time-stamp token, then the TSA returns a *time-stamp response* with the time-stamp token to the requester.

In order to validate the time-stamp token, the requester could send a *verification request* that contains the time-stamp token to a verifier at time t_v. For a single time-stamp token that has not been renewed, the verifier checks the following:

- The token is syntactically well-formed.
- Every hash value of the data item is correctly computed through the corresponding hash function.
- At least one of the hash functions that is used to generate digests of the data item is collision resistant at t_v.
- The protection mechanism of the time-stamp token is not broken at t_v.
- The cryptographic binding is correctly computed on the data and time.

If all above conditions are held, the time-stamp token is valid at time t_v, so the verifier returns a *verification response* with a "true" result to the requester. Otherwise, the verifier returns a "false" result to the requester.

For a renewed time-stamp token, the verifier checks the validity of each nested time-stamp token at the time it was generated or renewed, and validity of the latest time-stamp token at t_v following the above checking steps. The verifier returns a *verification response* with a "true" result to the requester if all verifications are successful, or a "false" otherwise.

4 Discussions on client-side renewal

In Section 3, we have reviewed some common specifications in the ANSI and ISO/IEC time-stamping services. In this section, we observe that the client-side renewal mechanisms in both standards are not explained thoroughly, which may cause some ambiguities for implementations. To make our discussions clear, we analyse several possible scenarios that the client-side hash functions are not renewed correctly by following the standards and their consequences.

4.1 The ambiguities in the ANSI and ISO/IEC standards

As specified in Section 3.3, a time-stamp token consists of hash value(s) of a data item, a time value and a cryptographic binding. The cryptographic algorithms used in the token include client-side hash functions and server-side algorithms. However, the lifetimes of these cryptographic algorithms are restricted due to the operational life cycles or advanced computational architectures. Once the algorithms are compromised, the time-stamp token becomes invalid and the existence of the data item could not be proved after that. Thus, time-stamp tokens should be constantly renewed to extend their validity periods.

In both the ANSI and ISO/IEC standards, the server-side renewal could be achieved by using the "ExtMethod" and "ExtRenewal" fields as following: when the server-side algorithm in the time-stamp token is close to the end of its lifecycle, or there is strong evidence that it will be compromised in the near future, the requester associates the time-stamp token in the "ExtRenewal" field, and indicates a new server-side algorithm in the "ExtMethod" field. The TSA then maintains these contents in the "TSTInfo" structure, and generates a new time-stamp token on TSTInfo using the indicated algorithm.

For client-side hash functions, as Section 3.3 shows, the ISO/IEC and ANSI standards both introduce the "ExtHash" field that allows multiple hash values of a data item to be submitted in the time-stamp request, but how to renew the client-side hash functions are not introduced clearly. For example, as the quote from the ISO/IEC 18014-1 [3], Section 5.7, Time-stamp renewal:

"Time-stamped data may be time-stamped again at a later time. This process is called time-stamp renewal and may optionally be implemented by the TSA. This may be necessary for example for the following reasons:

- *The mechanism used to bind the time value to the data is near the end of its operational life cycle (e.g., when using a digital signature and the public key certificate is about to expire).*
- *The cryptographic function used to bind the time value to the data is still trusted; however, there is strong evidence that it will become vulnerable in the near future (e.g., when a hash function is close to begin broken by new attacks or available computing power).*
- *The issuing TSA is about to terminate operations as a service provider."*

We can see that the "mechanism used to bind the time value and data" and "cryptographic function used to bind the time value to the data" do not include the

client-side hash functions, which means that the client-side hash functions are not defined as a motivation for time-stamp renewal. Apart from this, there are no other specifications in the ISO/IEC standard about how to renew client-side hash functions.

In the ANSI standard [1], the client-side renewal is briefly addressed in the definition of the "renewal" term, as the quote from the ANSI X9.95 [1], Section 3.29, Renewal:

- *"A renewal is the extension of the validity of an existing time stamp token. Legitimate reasons to renew a TST include: (i) the public key certificate used to verify the TSA digital signature is nearing its expiration date, or (ii) a requestor needs to replace the hash value using a stronger hash algorithm. "*

We can see that "a requestor could replace the hash value using a stronger hash algorithm" is a statement that allows requesters to replace the client-side hash value, but when to replace the hash value, how many hash values should be replaced are not specified. The ambiguities in both standards may mislead the implementers to ignore or improperly operate client-side renewal.

4.2 Possible failed implementations of client-side renewal

Based on the observations in Section 4.1, we further analyse some possible scenarios for implementations that do not effectively renew client-side hash functions. Note that the following three cases are arguably compatible with both the ISO/IEC and ANSI time-stamping standards.

- Case 1: A requester only submits one hash value of a data item without renewal.
- Case 2: A requester submits multiple hash values of a data item without renewal.
- Case 3: A requester replaces hash values using stronger hash functions after all current hash functions are not collision resistant.

Case 1: Let D denote a data item, and the hash value of D is h_0, which is computed through a client-side hash function H_0, i.e., $h_0 = H_0(D)$. The requester sends the pair (h_0, H_0) to a TSA, the TSA generates a time-stamp token TST_0 at time t_0. When the server-side algorithm in TST_0 is nearly compromised, the requester sends (h_0, H_0, TST_0) to a TSA, the TSA produces a new time-stamp token TST_1 on the input at time t_1. Repeat the server-side renewal in a long-term period, the requester eventually has TST_0, ..., TST_n ($n \in \mathcal{N}$).

Assume the collision resistance of H_0 is broken at time t_{b0}. After t_{b0}, the verification condition "at least one client-side hash function is not broken at t_v" is failed. Thus, time-stamp tokens generated after t_{b0} are verified as "false", the time-stamping service could prove the existence time of data item D at most between t_0 and t_{b0}. After t_{b0}, any server-side renewal does not extend the validity of time-stamp tokens any more.

Case 2: Let D denote a data item, and the hash values of D are h_0, ..., h_m, which are computed through client-side hash functions H_0, ..., H_m separately, i.e., $h_0 = H_0(D)$, ..., $h_m = H_m(D)$. The requester sends (h_0, H_0), ..., (h_m, H_m) to a TSA, the TSA generates a time-stamp token TST_0 at time t_0. After that, the server-side renewal is implemented correctly in a long-term period, the requester obtains TST_0, ..., TST_n ($n \in \mathcal{N}$) at the end.

Assume the collision resistance of H_0, ..., H_m are all broken at time t_{bm}. After t_{bm}, the verification condition "at least one client-side hash function is not broken at t_v" is failed. The time-stamp tokens produced after t_{bm} are verified as "false", the time-stamping service could prove the existence time of D at most between t_0 and t_{bm}, not any longer.

Case 3: With the same notation as in Case 2, if the requester replaces one or more hash values in h_0, ..., h_m using stronger hash functions at time $t_1 > t_{bm}$, for the same reason as Case 1 and Case 2, the new time-stamp token generated at t_1 is valid, but the time-stamp tokens generated before t_1 are verified as "false". The time-stamping service only proves the existence of D at t_1 or after, and certainly can not prove its existence at time t_0.

Summary: If a requester does not renew client-side hash functions correctly, the time-stamping service is only able to prove the existence of data items with limited time periods, when at least one of the client-side hash functions in the set is collision resistant. Multiple hash values only extend the lifetime of a single hash function, but the overall lifetime of them is still limited. In other words, a time-stamping service without correct client-side renewal does not satisfy the definition of "long-term" in Section 1. In order to achieve a long-term time-stamping service, the client-side renewal is necessary and should be specified clearly.

5 Proposed long-term time-stamping scheme

In this section, we propose a comprehensive long-term time-stamping scheme that describes how the client-side hash functions and server-side algorithms are used and renewed. Notice that the server-side protection mechanism is not described as a particular one, which could be any of the mechanisms for an independent token or a linked token, as specified in either the ISO/IEC 18014-2 [4], ISO/IEC 18014-3 [5] or the ANSI X9.95 [1].

$n \in \mathcal{N}$	Total number of time-stamp renewal processes
$i \in \{0, n\}$	Index number of time-stamp renewal
D	The data item to be time-stamped
H_0^*, ..., H_n^*	Client-side hash functions' identifiers, each of them could be a set of identifiers.
h_0^*, ..., h_n^*	Hash values computed through hash function H_0^*, ..., H_n^* respectively, each of them could be a set of hash values.
t_0, ..., t_n	Time points of requesting time-stamp renewal
TST_0, ..., TST_n	Time-stamp tokens generated at time t_0, ..., t_n respectively
C_0, ..., C_n	Cryptographic binding in time-stamp token TST_0, ..., TST_n respectively

Table 1: Notation

Our proposed scheme has three functionalities: time-stamp generation, time-stamp renewal and time-stamp verification. Fig. 2 shows the time-stamp generation and renewal together, the notation is listed in Table 1. *For simplicity, we assume that each*

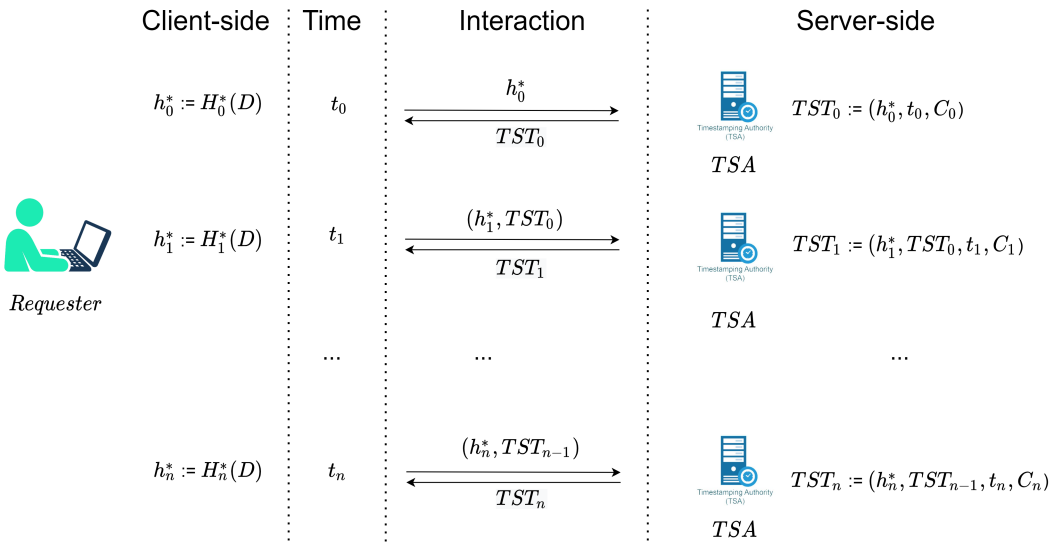

Fig. 2: The proposed long-term time-stamping scheme

hash value also contains its hash identifier, e.g., we denote (h_0^*, H_0^*) *pair as* h_0^**. For every pair* (h_0, H_0) *in* (h_0^*, H_0^*) *satisfying* $h_0 = H_0(D)$*, we denote them as* $h_0^* = H_0^*(D)$. Note that some message formats that are not relevant to security analysis are omitted.

5.1 Time-stamp generation

As the top row in Fig. 2, at time t_0 $(i=0)$: a requester computes one or more hash values of D and sends them to a TSA. i.e., $h_0^* = H_0^*(D)$. The TSA generates a cryptographic binding C_0 on (h_0^*, t_0), and returns the time-stamp token $TST_0 := (h_0^*, t_0, C_0)$ to the requester.

5.2 Time-stamp renewal

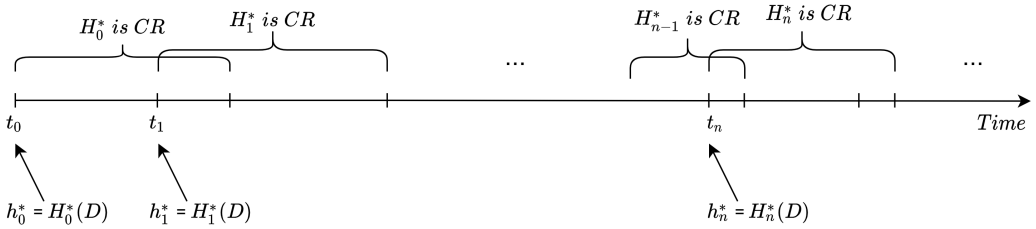

Fig. 3: Timeline of client-side renewal (CR represents "Collision Resistant")

As the second to the last row in Fig. 2, at time t_i ($i \in \{1, n\}$): the requester sends (h_i^*, TST_{i-1}) to a TSA. The TSA produces a new time-stamp token $TST_i := (h_i^*, TST_{i-1}, t_i, C_i)$ to the requester. h_i and C_i are determined with different renewal mechanisms as follows:

1. **Server-side renewal**: When the server-side algorithm is about to be compromised or reach the end of its life cycle, the requester remains the previous hash value(s) of D, i.e., $h_i^* = h_{i-1}^*$, then indicates a stronger server-side algorithm in the time-stamp request. The TSA generates a new cryptographic binding C_i with the indicated server-side algorithm.
2. **Client-side renewal**: When the collision resistance of all client-side hash functions in the latest time-stamp token are about to be broken, and at least one of them is still collision resistant, the requester computes one or more new hash values of D using stronger hash functions, i.e., $h_i^* = H_i^*(D)$, then replaces some of the old hash values with the new ones, or directly adds the new ones into the time-stamp request. The TSA generates a new cryptographic binding C_i using the server-side algorithm used in C_{i-1}. As the timeline shows in Fig. 3, each renewal should happen between the current set of hash functions are all compromised.
3. **Both-side renewal**: A combination of the above two cases: when the security of server-side algorithm and collision resistance of client-side hash functions are all threatened as above scenarios, the requester computes one or more new hash values of D with stronger hash functions, i.e., $h_i^* = H_i^*(D)$, replaces the old hash values with new ones, or adds the new ones into the request, and then indicates a stronger server-side algorithm in the request. The TSA generates a new cryptographic binding C_i with the indicated server-side algorithm.

*Note that the message format of C_i depends on the server-side protection mechanisms and their details are not discussed in this paper. We stress that the scheme is applicable for any type of server-side protection mechanism.

5.3 Time-stamp verification

At the verification time t_v, the verifier receives a time-stamp token TST_i ($i \in \{0, n\}$) and checks the following conditions:

- The time-stamp token is syntactically well-formed.
- The hash values of D through H_0^*, ..., H_i^* match the corresponding hash values in time-stamp tokens, i.e., $h_0^* = H_0^*(D)$, $h_1^* = H_1^*(D)$, ..., $h_i^* = H_i^*(D)$.
- At least one hash function in H_0^* and in H_1^* is collision resistant when H_0^* is renewed, ..., at least one hash function in H_{i-1}^* and in H_i^* is collision resistant when H_{i-1}^* is renewed, at least one hash function in H_i^* is collision resistant at time t_v.
- The server-side algorithm used in C_0 and C_1 are secure at the time the one for C_0 is renewed, ..., the server-side algorithm used in C_{i-1} and C_i are secure at the time the one for C_{i-1} is renewed, the server-side algorithm for C_i is secure at t_v.
- Cryptographic binding C_0, ..., C_i are correctly computed on the corresponding input content.

If all above conditions are satisfied, we say the time-stamp token TST_i is valid at time t_v, and the verifier returns "true" to the requester if the verifications are successful. Otherwise, return a "false" to the requester. The valid time-stamp token TST_i indicates that the data item D existed at the time t_0.

6 Security notions

In this section, we formalize the syntax of a long-term time-stamping scheme, the security assumptions that are required for analysis, and the security properties that a long-term time-stamping scheme should satisfy.

6.1 Syntax of a long-term time-stamping scheme

As defined as follows, a comprehensive long-term time-stamping scheme consists of three algorithms, which are respectively associated with time-stamp generation, time-stamp renewal and time-stamp verification.

Definition 1. *(Long-term time-stamping (LTTS) scheme.) A LTTS scheme is a tuple of the following algorithms* ($TSGen$, $TSRen$, $TSVer$)*:*

- $TSGen(h_0^*) \rightarrow TST_0$*: the algorithm* $TSGen$ *takes as input a set of hash values* h_0^**, outputs a time-stamp token* TST_0*.*
- $TSRen(h_i^*, TST_{i-1}) \rightarrow TST_i$*: the algorithm* $TSRen$ *takes as input a set of hash values* h_i^* *and a previous time-stamp token* TST_{i-1}*, outputs a new time-stamp token* TST_i*.*
- $TSVer(D, TST_i, VD, t_v) \rightarrow b$*: the algorithm* $TSVer$ *takes as input a data item* D*, a time-stamp token* TST_i*, the necessary verification data* VD *(e.g., revocation lists of cryptographic algorithms that can be updated over time), and the verification time* t_v*, outputs* $b=1$ *if the time-stamp token is valid, otherwise outputs* $b=0$*.*

6.2 Security assumptions

In the following models and proofs, we assume that

1. The verifier correctly performs the verification algorithm.
2. TSAs correctly perform the $TSGen$ and $TSRen$ algorithms.
3. The verification data VD is trusted and cannot be tampered.
4. Each cryptographic algorithm is associated with a validity period and provides correct outputs within their validity periods.

6.3 Security models and definitions

A long-term time-stamping (LTTS) scheme should achieve three security properties: correctness, nondisclosure, and long-term integrity. The formal definitions of these properties are given as follows:

Correctness. This property means that assuming every entity is honest, a long-term time-stamping scheme is able to prove existence of data items in a long period of time that is not bounded with the lifetimes of underlying cryptographic algorithms. The formal definition of correctness is given below.

Definition 2. *(Correctness.) Let* $LTTS=(TSGen, TSRen, TSVer)$ *be a long-term time-stamping scheme,* D *be a data item to be time-stamped,* TST_n $(n \in \mathcal{N})$ *is a time-stamp token produced as follows.*

At time t_0*, a requester computes a set of hash values of* D*, i.e.,* $h_0^* = H_0^*(D)$*, then the algorithm* $TSGen$ *takes input* h_0^* *(includes identifiers* H_0^**) and outputs a time-stamp token* TST_0*. Then for* $i=1, ..., n$*, at time* t_i*, the algorithm* $TSRen$ *takes as input a set of hash values* $h_i^*=H_i^*(D)$ *and a time-stamp token* TST_{i-1} *at a point in time when the server-side algorithm and client-side hash functions in* TST_{i-1} *are still secure, and outputs a time-stamp token* TST_i*. In the end, at a point in time* $t_v > t_n$*, the algorithm* $TSVer$ *takes as input the data item* D*, the time-stamp token* TST_n*, the verification data* VD *and verification time* t_v*. Assume at* t_v*, at least one client-side hash function in* H_n^* *is still collision resistant, and the server-side algorithm in* TST_n *is still secure.*

For a long-term time-stamping scheme to be correct, it must satisfy that if a time-stamp token TST_n *is generated for any data item* D *following the above process, the verification algorithm outputs* $TSVer(D, TST_n, VD, t_v)=1$*.*

Nondisclosure. This property means that the data item to be time-stamped is not exposed to any party except for the requester and verifier. Similar to the definition of a long-term time-stamping scheme, if the nondisclosure could be achieved with limited duration that is bounded by the lifetimes of corresponding cryptographic algorithms, we say it is *short-term nondisclosure*, otherwise it is *long-term nondisclosure*. The formal definition of nondisclosure in a long-term time-stamping scheme is as follows.

Definition 3. *(Nondisclosure.) A long-term time-stamping service provides nondisclosure for data items to be time-stamped if it is computationally infeasible for any party except the requester and verifier to reveal the data items.*

Long-term Integrity. The security notion of long-term integrity is based on the concept of "compromising" a time-stamping scheme. In specific, we say an attacker is able to compromise a time-stamping scheme, if it is able to claim that a data object exists at a point in time that actually it does not exist, or to tamper valid time-stamp tokens without being detected. Thus, we say a time-stamping scheme has "long-term integrity" if an attacker is unable to compromise the time-stamping scheme in a long period of time that is not bounded with the lifetimes of underlying cryptographic algorithms.

The long-term integrity model is defined as a game running between a long-lived adversary $\mathcal{A}$, a simulator $\mathcal{B}$ and a set of TSAs. As same in [10, 11, 23], $\mathcal{A}$ is modelled as a set of computing machines that have abilities to develop computational power and computing architectures with time increasing, but also being restricted within each time period. $\mathcal{B}$ has computational resources comparable to $\mathcal{A}$. Besides, $\mathcal{A}$ is able to advance time by calling a global clock oracle $Clock(t)$,

and communicate with TSAs through available queries in different time periods. Based on timely manner, the long-term integrity model could be divided into two stages:

Stage 1 ($t{=}t_0$):

1. Set time and power: $\mathcal{A}$ is able to set current time as t_0 by querying the oracle $Clock(t)$, i.e., $t_{cur} = t_0$, and use computing machine $\mathcal{M}_0$ and computational power $\mathcal{P}_0$.
2. Request time-stamps: The adversary $\mathcal{A}$ is able to select a secure TSA, then send one or more hash values of a data object x to the TSA. The TSA returns a time-stamp token TST_0 to $\mathcal{A}$. i.e., $h_0^* := H_0^*(x)$, $TST_0 \leftarrow h_0^*$. H_0^* and the server-side algorithm used in TST_0 are secure against $(\mathcal{M}_0, \mathcal{P}_0)$.

Stage 2 ($t{=}t_i$, $i{\in}\{1, n\}$):

1. Set time and power: $\mathcal{A}$ is able to set current time as t_i by querying the oracle $Clock(t)$, i.e., $t_{cur}{=}t_i$, and use computing machine $\mathcal{M}_i$ and computational power $\mathcal{P}_i$.
2. Request time-stamp renewal: The adversary $\mathcal{A}$ is able to select a secure TSA, then send one or more hash values of a data object x with a previous time-stamp token TST_{i-1} to the TSA. The TSA returns a new time-stamp token TST_i to $\mathcal{A}$. i.e., $h_i^*{=}H_i^*(x)$, $TST_i{\leftarrow}(h_i^*, TST_{i-1})$. H_i^*,H_{i-1}^* and the server-side algorithm used in TST_i are secure against $(\mathcal{M}_i, \mathcal{P}_i)$.
3. Compromise TSAs: The adversary $\mathcal{A}$ is able to select an expired TSA, and obtain the relevant secret information kept by the TSA (e.g., the private key for signature-based time-stamping).

The winning conditions of the long-lived adversary $\mathcal{A}$ and the simulator $\mathcal{B}$ are defined as:

- $\mathcal{A}$: At any point in time t_v, $\mathcal{A}$ outputs a pair (x', TST), $\mathcal{A}$ wins the game if the pair (x', TST) is not queried from the TSAs, and $TSVer(x', VD, t_v, TST){=}1$.
- $\mathcal{B}$: At any point in time t_v, $\mathcal{B}$ breaks any set of the client-side hash functions, or any of the server-side algorithms within their validity periods.

We denote the probability that $\mathcal{A}$ wins the game as $\mathcal{A}_{LTTS}^{LTI}$. Until time t_v, the sum probability that $\mathcal{B}$ breaks at least one client-side hash function within its validity period is denoted as $\mathcal{B}_{t_v}^{CS}$, and the sum probability that $\mathcal{B}$ breaks at least one server-side algorithm within its validity period is denoted as $\mathcal{B}_{t_v}^{SS}$. Furthermore, we define the $\mathcal{B}_{t_v}^{Cryp}$ as the sum probability of the failure of cryptographic algorithms within their validity periods:

$$\mathcal{B}_{t_v}^{Cryp}{=}\mathcal{B}_{t_v}^{CS} + \mathcal{B}_{t_v}^{SS}.$$

Definition 4. *(Long-term Integrity.) Let $LTTS = (TSGen, TSRen, TSVer)$ be a long-term time-stamping scheme, let $\mathcal{A}$ and $\mathcal{B}$ be a long-lived adversary and a simulator respectively as specified in the game above. we say a LTTS has long-term integrity if there exists a constant c for $\mathcal{B}$ such that for any point in time t_v,*

$$\mathcal{A}_{LTTS}^{LTI} \leq c{\cdot}\mathcal{B}_{t_v}^{Cryp}.$$

7 Security analysis

In terms of the security models and definitions in Section 6.3, we now prove our proposed long-term time-stamping scheme holding each security property.

7.1 Proof of correctness

Theorem 1. *The proposed long-term time-stamping scheme is correct.*

Proof. In our proposed $LTTS=(TSGen, TSRen, TSVer)$, we assume that a data item D has been through the $TSGen$ and $TSRen$ algorithms separately at time t_0 and time t_i for $i \in \{1, n\}$ as the process described in Definition 2, and finally outputs a time-stamp token TST_n $(n \in \mathcal{N})$. At time $t_v > t_n$, the verification algorithms takes input D, TST_n, VD and t_v, the verification result could be analysed for each condition specified in Section 5.3 as follows:

First, the time-stamp token TST_n is generated through $TSGen$ and $TSRen$ legitimately, so every enclosed time-stamp token TST_0, ..., TST_n is syntactically correct.

Second, every set of hash values h_i^* of the data item D are computed through the corresponding set of hash functions H_i^*, and every cryptographic binding is generated by the $TSGen$ and $TSRen$ algorithms, it is clear that the hash values of D through H_0^*, ..., H_i^* match the corresponding hash values in time-stamp tokens, and all cryptographic bindings are correctly computed on the corresponding input contents.

Third, since the algorithm $TSRen$ is implemented every time before client-side hash functions in the latest time-stamp tokens are all broken, and also before the server-side algorithm in the latest cryptographic binding is compromised, the validity or client-side hash functions and server-side algorithms are all guaranteed at their renewal times. With the assumption that at t_v, at least one client-side hash function in TST_n is still collision resistant, and the server-side algorithm in TST_n is still secure, all verification steps are satisfied. Therefore, the verification algorithm outputs $TSVer(D, TST_n, VD, t_v)=1$ and the theorem follows. □

7.2 Proof of nondisclosure

Theorem 2. *The proposed long-term time-stamping scheme is able to provide nondisclosure for data items when all client-side hash functions are preimage resistant.*

Proof. Assume a requester obtains a time-stamp token, which contains a set of hash values of a data item D. These hash values are computed using a set of client-side hash functions, $H_0^*=(H_0, ..., H_m)$, i.e., $h_0=H_0(D)$, ..., $h_m=H_m(D)$. Assume that the preimage resistance of H_0, ..., H_m are compromised at t_{p0}, ..., t_{pm} respectively, and that the hash function H_f is one of H_0, ..., H_m, the preimage resistance of H_f is broken at t_{pf}, with $\{t_{p0}, ..., t_{pm}\}_{min}=t_{pf}$ ($\{...\}_{min}$ denotes the earliest time in the set). Then at time $t_0<t<t_{pf}$, if an attacker is able to find a preimage for any of h_0, ..., h_m with non-negligible possibilities, the preimage resistance of at least one of H_0, ..., H_m is broken within its validity period, which contradicts our assumption. After time t_{pf}, the attacker is able to attack at least the hash function H_f that determine preimages of h_f to D with non-negligible possibilities. Thus, the proposed time-stamping scheme provides short-term nondisclosure in the duration (t_0, t_{pf}). Therefore, the theorem follows. □

7.3 Proof of long-term integrity

Based on the assumptions discussed in Section 6.2, TSAs and the verifier are trusted parties and always perform operations correctly. The integrity of data objects only relies on the security of client-side hash functions and server-side algorithms. In this paper, we do not limit out discussion with a specific server-side protection mechanism. A time-stamp token could be any type as introduced in Section 3.2.

For the security of server-side mechanisms, the existing security analyses from [10] and [11] have proved the security of a signature-based long-term time-stamping scheme as well as a hash-based one. As introduced in Section 2, these two schemes satisfy the long-term integrity property under the condition that client-side security is guaranteed. Thus, their results can be fitted in our analysis. We assume that server-side security is satisfied in our proposed scheme, and focus on the analysis of client-side security. In other words, as defined in Section 6.3, the probability of the adversary breaking the scheme through server-side is reduced to $\mathcal{B}_{t_v}^{SS}$.

Theorem 3. *If the security of server-side is guaranteed, the proposed time-stamping scheme has long-term integrity.*

Proof. That the server-side security is guaranteed means that all the time-stamp tokens $TSTs$ must be created by the corresponding trusted TSAs. The adversary $\mathcal{A}$ can only join the long-term integrity game as defined in Section 6.3 to obtain these tokens. These token are not tampered after their generations.

If $\mathcal{A}$ wins the game, it must output a time-stamp token $TST = (TST_0, ..., TST_n)$ on a data item x', which is distinct to the original x value that was used to request any of $TST_0, TST_1, ..., TST_n$ but somehow to manage letting $TSVer(x', VD, t_v, TST) = 1$. Based on Section 5.3, this equation guarantees that at time t_i for $i \in \{1,\ n\}$, the two corresponding sets of client-side hash functions used by $\mathcal{A}$, denoted by $H_{i-1}^* = (H_{(i-1)1}, H_{(i-1)2}, ..., H_{(i-1)m_{i-1}})$ and $H_i^* = (H_{i1}, H_{i2}, ..., H_{im_i})$, must both contain at least one collision resistant hash function. Besides, each set of hash values $H_i^*(x)$ is a part of token TST_i. Now let us check the following reasoning:

At time t_0, $\mathcal{A}$ submits a set of hash values $H_0^*(x)$ of a data item x to a TSA, the TSA returns a time-stamp token TST_0 on $H_0^*(x)$. Assume that the set of hash functions $H_0^* = (H_{01}, H_{02}, ..., H_{0m_0})$, H_{0j} for $j \in \{1,\ m_0\}$ is collision resistant at t_0.

At time t_1, $\mathcal{A}$ decides to renew the token TST_0 by using another set of client-side hash functions $H_1^* = (H_{11}, H_{12}, ..., H_{1m_1})$. Since at least one of the hash functions in H_0^*, which is still collision resistant at this time, we assume H_{0j} is still collision resistant at t_1, although it may have become weak, and the corresponding hash value $H_{0j}(x)$ is a part of TST_0. Then $\mathcal{A}$ can submit $(H_1^*(x), TST_0)$ for requesting a time-stamp renewal and obtains TST_1 (Case 1) or $\mathcal{A}$ may submit $(H_1^*(x'), TST_0)$ for requesting a time-stamp renewal and obtains TST_1 (Case 2). If Case 2 happens, there must have $H_{0j}(x) = H_{0j}(x')$ with a pair of collisions (x, x'). $\mathcal{B}$ can then obtain this pair. This result is contradict to the assumption that H_{0j} is collision resistant at t_1. If Case 1 happens, let us carry on with our reasoning. The TSA returns a renewed time-stamp token TST_1. We now assume that $H_{1j} \in H_1^*$ for $j \in \{1,\ m_1\}$ is collision resistant at time t_1.

At time t_2, H_{0j} and all other client-side hash functions used at t_0 may have been broken, but we assume that H_{1j} is still collision resistant, and the hash value $H_{1j}(x)$ is

a part of TST_1. Now repeating the previous situation, $\mathcal{A}$ can submit $(H_2^*(x), TST_1)$ for requesting another time-stamp renewal and obtains TST_2 (Case 1) or $\mathcal{A}$ may submit $(H_2^*(x'), TST_1)$ for requesting another time-stamp renewal and obtains TST_2 (Case 2). Again, Case 2 allows $\mathcal{B}$ to obtain a pair of collisions satisfying $H_{1j}(x) = H_{1j}(x')$ and it contradicts the assumption, and Case 1 leads us to continue our reasoning.

Carrying on our argument as before, only Case 1 for each time-stamp renewal is considered. We assume that $H_{(n-1)j}$ for $j \in \{1,\ m_{(n-1)}\}$ is collision resistant at both t_{n-1} and t_n, and the hash value $H_{(n-1)j}(x)$ is a part of TST_{n-1}. If $\mathcal{A}$ finally submits $(H_n^*(x'), TST_{n-1})$ and successfully obtains TST_n, then $\mathcal{B}$ obtains a pair of collisions $(x,\ x')$ satisfying $H_{(n-1)j}(x) = H_{(n-1)j}(x')$.

In summary, based on the above reasoning, as long as $\mathcal{A}$ wins the game, $\mathcal{B}$ can break at least one client-side hash function within its validity period. Therefore, the winning probability of $\mathcal{A}$ through client-side is reduced to the same level of the probability that $\mathcal{B}$ breaks at least one client-side hash function within its validity period. With adding the probability of the failure of server-side algorithms $\mathcal{B}_{t_v}^{SS}$, there exists a constant c such that:

$$\mathcal{A}_{LTTS}^{LTI} \leq c \cdot (\mathcal{B}_{t_v}^{CS} + \mathcal{B}_{t_v}^{SS}).$$

Thus, we have proved Theorem 3. □

8 Evaluations of client-side security level

In this section, we determine the client-side security level $\mathcal{L}_{CS}$ in practical, which represents the probability of a long-lived adversary as defined in Section 6.3 breaks the client-side security of the proposed scheme. In terms of the ISO/IEC and ANSI standards, multiple hash values are allowed in every time-stamp request, and the system is available to set up policies for the number of client-side hash functions in every time-stamp request, and the interval of time-stamp renewal. Therefore, there are two parameters that affect the client-side security level:

1. l^{set}: the security level of a set of client-side hash functions in a time-stamp request, which means the probability that a long-lived adversary as defined in Section 6.3 breaks all collision resistant hash functions in the set within their validity periods.
2. n: the number of sets of client-side hash functions in time-stamp tokens, which means the number of client-side renewal process.

Assume the security level of each set of client hash functions are l_1^{set}, ..., l_n^{set} respectively. The winning probability of the adversary is the aggregated probability of the failure of every set of client-side hash functions:

$$\mathcal{L}_{CS} = \sum_{i=1}^{n} l_i^{set}.$$

We can see that the more sets of client-side hash functions are used in the scheme, the higher probability that the adversary breaks the client-side security of

the proposed scheme. The stronger of each set of client-side hash functions, the lower probability that the adversary breaks the client-side security of the proposed scheme.

Furthermore, the security level of a set of client-side hash functions l^{set} is decided by another two parameters:

1. l: the security level of a specific client-side hash function in the set, which means the probability of a long-lived adversary as defined in Section 6.3 breaks the collision resistance of the specific hash function within its validity period.
2. m: the number of client-side hash functions required in a set.

Assume the security level of each hash function in a set is l_1, ..., l_m respectively. Then the probability of the failure of a whole set of hash functions, is equal to the probability that every hash function in the set fails:

$$l^{set} = \prod_{i=1}^{m} l_m.$$

Based on the bounded computational resources in each time period, a long-lived adversary has not enough resources to break the collision resistance of whole set of hash functions. That means, the computational resources of the adversary is not enough to break at least one of the hash functions in the set. If the adversary owns computational power to break some of the hash functions, then the security level of these hash functions are equal to 1, the l^{set} is only determined by the security level of the other hash functions.

Summary: The evaluation results show that with more times the client-side renewal happens, the probability of the adversary breaks the scheme increases; for multiple hash values submitted in each time-stamp request, the more collision resistant hash functions are required in each time-stamp request, the lower probability of the adversary breaks the scheme.

9 Conclusions

In this paper, we have discussed the importance of client-side renewal: it is not enough for a requester to only use the same set of multiple hash values in an initial time-stamping request as well as a time-stamp renewal request, new hash values computed through stronger hash functions should be used before the failure of current set of hash functions. This argument is straightforward but is not explicitly addressed in the ISO/IEC and ANSI standards. Then we propose a long-term time-stamping scheme with specifications of both client-side and server-side mechanisms. We have proved that our scheme achieves correctness, short-term nondisclosure and long-term integrity properties. Finally, we have provided a quantified evaluation for the client-side security level of our proposed scheme.

We argue that the short-term nondisclosure of our scheme could be accepted, since the integrity could naturally be required for much longer time than nondisclosure. For instance, intellectual-property data is usually protected in secret for a certain period before it is released but its integrity should be maintained in perpetuity.

As the future work, we will implement the proposed scheme in a time-stamping service environment to measure the timing overhead and to determine the network channel affectation. Besides, our research could be carried on covering other renewable applications that require long-term integrity. The renewal mechanisms in time-stamping services may have other application scenarios and such applications and their security analyses should be explored.

Acknowledgements

This work is supported by the European Union's Horizon 2020 research and innovation program under grant agreement No.779391 (FutureTPM) and grant agreement No. 952697 (ASSURED).

References

1. American National Standard Institute (ANSI). ANSI X9.95-2016 – Trusted Timestamp Management and Security, 2016.
2. Carlisle Adams, Pat Cain, Denis Pinkas, and Robert Zuccherato. RFC 3161: Internet X. 509 Public Key Infrastructure Time-Stamp Protocol (TSP), 2001.
3. ISO/IEC 18014-1:2008. Information technology – Security techniques – Time-stamping services – part 1: Framework. Standard, 2008.
4. ISO/IEC 18014-2:2009. Information technology – Security techniques – Time-stamping services – part 2: Mechanisms producing independent tokens. Standard, 2009.
5. ISO/IEC 18014-3:2009. Information technology – Security techniques – Time-stamping services – part 3: Mechanisms producing linked tokens. Standard, 2009.
6. ISO/IEC 18014-4:2015. Information technology – Security techniques – Time-stamping services – part 4: Traceability of time sources. Standard, 2015.
7. Arjen K Lenstra. Key length. Contribution to the handbook of information security. 2004.
8. Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.
9. Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings, 28th Annual ACM Symposium on the Theory of Computing*, pages 212–219, 1996.
10. Matthias Geihs, Denise Demirel, and Johannes Buchmann. A security analysis of techniques for long-term integrity protection. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 449–456. IEEE, 2016.
11. Ahto Buldas, Matthias Geihs, and Johannes Buchmann. Long-term secure time-stamping using preimage-aware hash functions. In *International Conference on Provable Security*, pages 251–260. Springer, 2017.
12. Stuart Haber and W Scott Stornetta. How to time-stamp a digital document. In *Conference on the Theory and Application of Cryptography*, pages 437–455. Springer, 1990.
13. Dave Bayer, Stuart Haber, and W Scott Stornetta. Improving the efficiency and reliability of digital time-stamping. In *Sequences II*, pages 329–334. Springer, 1993.
14. National Institute of Standards and Technology (NIST). Recommendation for Digital Signature Timeliness. Standard, 2009.
15. N. Pope S. Santesson. RFC 5816: Esscertidv2 update for RFC 3161, 2010.
16. D Pinkas, N Pope, and J Ross. CMS Advanced Electronic Signatures (CAdES). *IETF Request for Comments*, 5126, 2008.

17. Martin Centner. XML Advanced Electronic Signatures (XAdES), 2003.
18. Stuart Haber and Pandurang Kamat. A content integrity service for long-term digital archives. In *Archiving Conference*, volume 2006, pages 159–164. Society for Imaging Science and Technology, 2006.
19. T Gondrom, R Brandner, and U Pordesch. Evidence Record Syntax (ERS). *Request For Comments–RFC*, 4998, 2007.
20. A Jerman Blazic, Svetlana Saljic, and Tobias Gondrom. Extensible Markup Language Evidence Record Syntax (XMLERS). Technical report, IETF RFC 6283, http://www. ietf. org/rfc/rfc6283. txt, 2011.
21. Dimitris Lekkas and Dimitris Gritzalis. Cumulative notarization for long-term preservation of digital signatures. *Computers & Security*, 23(5):413–424, 2004.
22. Martin Vigil, Daniel Cabarcas, Johannes Buchmann, and Jingwei Huang. Assessing trust in the long-term protection of documents. In *2013 IEEE Symposium on Computers and Communications (ISCC)*, pages 000185–000191. IEEE, 2013.
23. Matthias Geihs. *Long-Term Protection of Integrity and Confidentiality–Security Foundations and System Constructions*. PhD thesis, Technische Universität, 2018.
24. Ran Canetti, Ling Cheung, Dilsun Kirli Kaynar, Nancy A Lynch, and Olivier Pereira. Modeling computational security in long-lived systems, version 2. *IACR Cryptology ePrint Archive*, 2008:492, 2008.
25. Jörg Schwenk. Modelling time for authenticated key exchange protocols. In *European Symposium on Research in Computer Security*, pages 277–294. Springer, 2014.
26. Ahto Buldas and Sven Laur. Knowledge-binding commitments with applications in time-stamping. In *International Workshop on Public Key Cryptography*, pages 150–165. Springer, 2007.
27. Yevgeniy Dodis, Thomas Ristenpart, and Thomas Shrimpton. Salvaging merkle-damgård for practical applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 371–388. Springer, 2009.
28. Ahto Buldas and Märt Saarepera. On provably secure time-stamping schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 500–514. Springer, 2004.
29. Ahto Buldas and Sven Laur. Do broken hash functions affect the security of time-stamping schemes? In *International Conference on Applied Cryptography and Network Security*, pages 50–65. Springer, 2006.
30. Ahto Buldas and Aivo Jürgenson. Does secure time-stamping imply collision-free hash functions? In *International Conference on Provable Security*, pages 138–150. Springer, 2007.
31. Ahto Buldas and Margus Niitsoo. Can we construct unbounded time-stamping schemes from collision-free hash functions? In *International Conference on Provable Security*, pages 254–267. Springer, 2008.
32. ISO/IEC 10118 (all parts). Information technology – Security techniques – Hash functions. Standard.
33. Lindell Jonathan Katz, Yehuda. *Introduction to modern cryptography*. 2014.
34. Scott A. Vanstone Alfred J. Menezes, Paul C. van Oorschot. *Handbook of applied cryptography*. 1996.
35. Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.