Verified Security of BLT Signature Scheme

Ahto Buldas
Tallinn University of Technology
Tallinn, Estonia

Denis Firsov
Tallinn University of Technology
Tallinn, Estonia

Risto Laanoja
Guardtime AS
Tallinn, Estonia

Ahto Truu
Guardtime AS
Tallinn, Estonia

Abstract

The majority of real-world applications of digital signatures use timestamping to ensure non-repudiation in face of possible key revocations. This observation led Buldas, Laanoja, and Truu [Buldas et al. 2017] to a server-assisted digital signature scheme built around cryptographic timestamping.

In this paper, we report on the machine-checked proofs of existential unforgeability under the chosen-message attack (EUF-CMA) of some variations of BLT digital signature scheme. The proofs are developed and verified using the EasyCrypt framework, which provides interactive theorem proving supported by the state-of-the-art SMT solvers.

Keywords digital signatures, EasyCrypt, formalized cryptography, timestamping

1 Introduction

A digital signature is a cryptographic construction for verifying the authenticity of a digital message. A digital signature scheme comes with a key generation algorithm which produces public-private key pairs. The private keys are used to generate signatures and must be kept in secret by the signers. The public keys must be openly available for those who need to validate the signatures.

Historically, the first digital signature schemes were "one-time use" which means that a public-private key pair could only be used for signing a single message [Diffie and Hellman 1976]. Later, Merkle described a generic way to turn one-time schemes into many-time schemes [Merkle 1979, 1980]. In this paper, we focus on one-time schemes.

An important property of signatures is non-repudiation, i.e. the possibility to use the signature as evidence against the signer. This induces the real-world problem of key revocation. Without such capability a user may (fraudulently) claim that their private key was stolen and someone else may have created signatures in their name. With revocation tracking, signatures created before key revocation event can be treated as valid, whereas signatures created afterwards can be considered invalid. Usually this is implemented using cryptographic timestamping and certificate status distribution services. Regardless of the implementation details, this cannot be done without online services, which means that most practical deployments of digital signatures are actually server-supported and rely on the timestamping services (cryptographic or trusted).

Based on this observation Buldas, Laanoja, and Truu [Buldas et al. 2017] proposed a new type of digital signature scheme (*BLT scheme* in the following) which builds on the fact that the valid signatures are always timestamped. Their original idea was to combine one-time time-bound keys with a timestamping service. The legitimate use of a key associated with a particular time is then proven by timestamping the message-key pair at that time.

Since keys are time-bound, the integrity of the signed messages does not depend on the long-term secrecy of private keys. Existential unforgeability against adaptive chosen-message attacks was proven in the random oracle model.

Back-dating new pairs (new messages with already used keys) would allow signatures to be forged. Therefore, to avoid key-based cryptography and trusted third parties, the hash-then-publish timestamping [Buldas and Saarepera 2004; Haber and Stornetta 1991] and the so-called keyless timestamping [Buldas et al. 2013] were employed, so that timestamps become irrefutable proofs of time.

The practicality of the original BLT scheme was limited by the fact that pre-generated keys had to be used at their designated time-slots only. For practical deployments, the number of time-bound keys tends to be large, which makes the key generation prohibitively slow on constrained devices such as smart cards.

Later, the BLT scheme was generalized and decomposed into two functional components: *cryptographic timestamping* and *forward-resistant tag systems*. This made it possible to propose other forward-resistant tag systems and arrive at a rich family of different BLT signature schemes with distinct functional and security properties [Buldas et al. 2019].

In this work, we use the EasyCrypt theorem prover to formalize cryptographic timestamping, forward-resistant tag systems, and analyze the security properties of the BLT signature scheme. We do the following technical contributions:

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

- Specify and implement the ideal model of universally composable cryptographic timestamping service (Section 3).
- Give a formal definition of bounded and unbounded tag systems with their security and correctness properties (Section 4).
- Implement the BLT signature scheme parameterized by a timestamping service and a tag system (Section 5).
- Analyze the unforgeability of the BLT scheme under different types of timestamping repositories (Section 6).

All of our results have been formalized in EasyCrypt theorem prover and the code is available as the accompanying supplementary material.

2 EasyCrypt

EasyCrypt (EC) is a framework for building and verifying the security of cryptographic constructions. In this section, we briefly outline some basic concepts behind EC by stepping through an example. More information on EC can be found in [Barthe et al. 2013].

We illustrate the proof development process on a simple example regarding collision and second-preimage resistance of hash functions. More specifically, our goal is to formally prove the basic fact that collision resistance (CR) implies second-preimage resistance (SPR).

The proof development starts by formally specifying the computational context which usually includes datatypes and operators, where types denote non-empty sets of values and operators are typed functions on these sets. EC provides basic built-in types such as `bool`, `int`, `real`, etc. The standard library includes formalizations of lists, arrays, finite sets, maps, distributions, etc. EC also allows users to implement their own datatypes and functions (including inductive datatypes and functions defined by pattern matching). Later we will make use of an "option" type which represents encapsulation of an optional value. More specifically, a value of type `X option` is either empty (`None`), or it contains a value `x` of type `X` (`Some x`). Function `oget` extracts the value from the `Some` constructor. If the argument is `None` then an arbitrary witness of the target type is returned (as mentioned above, all types in EC are inhabited).

```
type 'a option = [ None | Some of 'a ].

op oget ['a] (o : 'a option) : 'a =
 with o = None   => witness
 with o = Some x => x.
```

The types and operators without definitions are abstract and can be seen as parameters to the rest of the development. In our example we declare an abstract hash function `H` with its input and output types. Moreover, our example is parameterized by a distribution (`his`) of input values as required for SPR:

```
type hash_input.
type hash_output.

op H   : hash_input → hash_output.
op his : hash_input distr.
axiom uhis : is_uniform his.
```

Parameters can additionally be restricted by axioms that can be later discharged during particular instantiation.

The computational hardness assumptions are implemented as probabilistic programs (also called *games*) parameterized by oracles and adversaries. An adversary is modeled as unspecified code with specified interface. We define a *module type* `AdvSPR` describing the interface of adversaries whose task is to break the second-preimage resistance of `H`:

```
module type AdvSPR = {
  proc adv(x : hash_input) : hash_input
}.
```

Modules are stateful "objects" consisting of global variables and procedures. Global variables are visible outside the modules and define their state at any given time. A procedure consists of local variables, assignments, probabilistic assignments (denoted by the infix operator `<$`), and calls to other procedures.

Next, we formalize the second-preimage resistance as a parameterized module `GameSPR` which is played by an adversary of type `AdvSPR`. The adversary receives an input for the hash function `H` and returns another input. The adversary wins the game if these two input values are different, but yield the same output from the hash function `H`:

```
module GameSPR(A : AdvSPR) = {
  proc main() : bool = {
    var x, x' : hash_input;

    x  <$ his;
    x' = A.adv(x);

    return H x = H x' ∧ x ≠ x';
  }
}.
```

Thus, the `main` procedure returns a Boolean value which indicates the outcome of the game.

In case of collision resistance the adversary needs to produce any pair of inputs which gives a collision:

```
module type AdvCR = {
  proc adv() : hash_input * hash_input
}.

module GameCR(A : AdvCR) = {
  proc main() : bool = {
    var x, x';

    (x, x') = A.adv();
```

```
    return H x = H x' ∧ x ≠ x';
  }
}.
```

We want to prove that if H is CR then this implies that H is also SPR. As it is common in cryptographic proofs, instead of proving the implication directly, we prove the contrapositive of it. Namely, let us assume that there exists an adversary A that is successful in breaking the SPR of H, then we show that A can be efficiently transformed into adversary T(A) that can break the collision resistance of H. This will contradict our initial assumption that H is CR and therefore H must be SPR.

To transform SPR adversary into CR adversary we act as follows: sample an input from his, feed the value to the SPR adversary, get a second preimage, and return the two inputs as a collision for H:

```
module T(A : AdvSPR) : AdvCR = {
  proc adv() : hash_input * hash_input = {
    var x, x';
    x  <$ his;
    x' = A.adv(x);
    return (x, x');
  }
}.
```

It should be intuitively clear that whenever A wins GameSPR, T(A) wins GameCR. We use the Probabilistic Relational Hoare Logic (PRHL [Barthe et al. 2012]) to formally establish this intuitive "equivalence".

```
lemma sprpr : forall (A <: AdvSPR),
  equiv [ GameSPR(A).main ~ GameCR(T(A)).main :
   ={glob A} ⟹ res{1} => res{2} ].
```

The lemma sprpr claims the equivalence of the procedures GameSPR(A).main and GameCR(T(A)).main with respect to precondition ={glob A} and postcondition res{1} => res{2} (symbol ~ separates programs; symbol ⟹ separates precondition from postcondition; symbol => denotes ordinary logical implication). The special variable res{1} is a Boolean value which refers to the result of running (left) the procedure GameSPR(A).main on memory &1 (as for memory &2 and procedure GameCR(T(A)).main on the right).

The statement of lemma sprpr means that if for any memories &1 and &2 the global variables of adversary A are equal (precondition) then the output sub-distributions obtained by executing the first procedure on memory &1 and second procedure on &2 satisfy the postcondition. In other words, if A wins the SPR-game then T(A) wins the CR-game as well.

The importance of PRHL equivalences is that they imply the usual statements about probabilities of events. More specifically, sprpr lemma allows us to conclude that for any initial memory &m and adversary A, the probability of A breaking the second-preimage resistance is upper-bounded by the probability of T(A) breaking the collision resistance:

```
lemma SPR_CR : forall &m (A <: AdvSPR),
  Pr[ GameSPR(A).main() @ &m : res ] ≤
  Pr[ GameCR(T(A)).main() @ &m : res ].
```

The res above is an abbreviation for res = true. We can conclude that the second-preimage resistance is stronger than the collision resistance.

The rest of the necessary EC background will be introduced on demand.

3 Cryptographic Timestamping

Intuitively, cryptographic timestamping generates proofs that data existed before a particular time. The proof can be a statement that data (or its hash) existed at a given time, cryptographically signed by a trusted third party. Such statements are useful for data archiving, supporting non-repudiable digital signatures, etc.

Haber and Stornetta [Haber and Stornetta 1991] made the first steps toward trustless timestamping by proposing a scheme where each timestamp would include some information from the immediately preceding one and a reference to the immediately succeeding one. Benaloh and de Mare [Benaloh and de Mare 1991] proposed to increase the efficiency of hash-linked timestamping by operating in rounds, where messages to be timestamped within one round would be combined into a hierarchical structure from which a compact proof of participation could be extracted for each message.

Formally speaking, the timestamping does not provide proofs of wall-clock time, but it rather allows us to prove that data existed at some moment in the past, and also allows us to establish a linear order on the timestamped data.

3.1 Ideal Model of Timestamping

We will use an idealized model of timestamping. This can be seen as an assumption of the timestamping service being a trusted third party or, alternatively, as relying on universal composability of a particular timestamping construction. Roughly speaking, the universal composability is a strong property of cryptographic primitives which states that a "real" cryptographic construction can be replaced with the "ideal" version of it and no efficient adversary will spot the difference in any extensional context. Universally composable timestamping constructions do exist [Buldas et al. 2005; Matsuo and Matsuo 2005].

We take advantage of the typed setting of EC and let the timestamping service be parameterized with the type of values stored in the timestamping repository. Another parameter is the distribution of initial times tdistr (time values are positive integers):

```
type time = int.
type data.

op    tdistr : time distr.
```

```
axiom tpos   : forall t, t ∈ tdistr => t > 0.
```

The module type TS describes interface of timestamping services. A service allows a user to timestamp (put) values of type data and returns associated timestamps:

```
module type TS = {
  proc init()                    : unit
  proc clock()                   : time
  proc put(d : data)             : time
  proc check(t : time, d : data) : bool
}.
```

The procedure clock shall return the current "time" of a service. The procedure check(t, d) returns true iff the value d is associated with the time t.

Next, we introduce module Ts of type TS which implements the standard timestamping functionality:

```
module Ts : TS = {
  var i : time
  var t : time
  var r : (time, data) fmap

  proc init() = {
    i <$ tdistr;
    t =  i;
    r =  empty;
  }

  proc clock() = {
    return t;
  }

  proc put(d : data) = {
    t = t + 1;
    r = r.[t ← d];
    return t;
  }

  proc check(t : time, d : data) = {
    return r.[t] = Some d;
  }
}.
```

The inner state of object Ts consists of the initial time i (sampled from tdistr), the current time t, and the repository (finite map) r which associates data items to time values. Although "time" advances in this implementation only when the new value is being added, it accurately models a linear ordering over the timestamped values. Notice that in Ts.check r.[t] has the option type.

3.2 Properties

To address the properties of the timestamping service we need to define a type of adversaries that can access it:

```
module type AdvTs(TsO : TS) = {
  proc main() : unit {TsO.check TsO.put}
}.
```

As mentioned before, the timestamping service is initialized by the init method. To forbid adversaries to re-initialize the service (and break the invariants) we only allow them to invoke the check and put methods. In EC, this is done by listing the allowed procedures in the module type definition.

Let us fix an adversary A for the rest of this section:

```
declare module A : AdvTs {Ts}.
```

To be able to prove properties of an *abstract* procedure A.main with respect to the *specific* module Ts, we require that global variables of A and Ts must be mutually inaccessible. In EC, this is done by listing "disjoint" modules in curly braces after the module type. This has the effect that the adversary A must use the interface of Ts (specified by type TS) and therefore is not allowed to directly update global variables of the module Ts.

Backdating Resistance The essential property of any timestamping service is its resistance against backdating. In other words, if we assume that a data d is associated with a time x (timestamped in the past, i.e., x ≤ Ts.t) then it remains true after running an arbitrary computation by the adversary A.

In EC, one can use Hoare logic (HL) to prove specific properties of procedures. In HL, properties are expressed as pre- and postconditions of programs (Hoare triples).

In our example, the precondition and the postcondition coincide (i ≤ Ts.t ∧ Ts.r.[x] = d). A Hoare triple means that if the precondition is true before the execution of the program, then the postcondition will be true if the program terminates. Note that in our example the program A(Ts).main is abstract.

```
lemma immutableTs : forall x d,
  hoare [ A(Ts).main :
    i ≤ Ts.t ∧ Ts.r.[x] = d ⟹
    i ≤ Ts.t ∧ Ts.r.[x] = d ].
```

The statement is proved by analyzing the implementation details of methods Ts.put and Ts.check, i.e., those that are accessible by A.

Soundness Clearly, if Ts would ignore its inputs and never store any data it would be trivially backdating resistant, but not useful. So, we also address soundness: storing a value d at a time x results in d being associated with the time x+1.

We used regular HL to prove immutability of Ts. However, regular HL ensures the postcondition only if the program terminates. Therefore, regular HL is not suitable to prove the correct operation of Ts.put method (it would leave the possibility that Ts.put is non-terminating). In EC, probabilistic Hoare logic (PHL) analyzes the probability that the program terminates and the postcondition is true.

```
lemma soundTs : forall x d,
  phoare [ Ts.put : arg = d ∧ Ts.t = x ⟹
```

```
    Ts.r.[x+1] = Some d ] = 1%r.
```

The variable `arg` in the precondition refers to the argument passed to the procedure `put`; `1%r` stands for the `real` number one.

4 One-Time Tag System

In this section we will describe the second ingredient of the BLT signature scheme: forward-resistant tag systems [Buldas et al. 2019]. Such systems consist of a probabilistic key generation algorithm (modeled by a distribution of key pairs), tag generation, and tag verification functions. Similarly to signature schemes, a tag system is one-time if each private key is supposed to be used only once.

```
type pkey, skey, tag.

op keyGen : (pkey * skey) distr.
op tagGen : skey → time → tag.
op tagVer : pkey → time → tag → bool.
```

If the tag verification function always agrees with the tag generation function then we say that tag system is *unbounded*:

```
axiom tagSndU : forall pk sk, (pk, sk) ∈ keyGen
  => tagVer pk t (tagGen sk t) = true.
```

In the original formulation [Buldas et al. 2019], the tag system was also parameterized by a time value `kpe` holding the expiration date of the key pairs. In this case we say that the tag system is *bounded*, and it must satisfy the following property:

```
op kpe : time.

axiom tagSndB : forall pk sk t, (pk, sk) ∈ keyGen
  => tagVer pk t (tagGen sk t) = true ⇔ t ≤ kpe.
```

The proofs and constructions in this paper are valid for both kinds of tag systems.

4.1 Properties

The definition of a tag system is somewhat similar to that of a signature scheme. The main difference is in the associated security properties. In their previous work, Buldas et al. designed efficient forward-resistant tag systems. They also proved that forward-resistance is sufficient to imply the security of a simple version of the BLT scheme (Section 6.6). In this work, we make a further step and analyze security of a more complicated variation of the BLT scheme (Section 6.7), where it turns out that forward-resistance alone is not sufficient, and we need to require additional security properties from a tag system.

To describe properties of one-time tag systems we need to specify one-time tagging oracles. Note that `tagGen` and `tagVer` are pure functions which cannot affect the state of the variables of modules. To control and monitor the access of an adversary to the tag generation function we parameterize adversaries with a stateful module (oracle) which stores a key pair and provides one-time tag generation functionality.

```
module type TagOracleT = {
  proc init(pk : pkey, sk : skey) : unit
  proc genTag(t : time)           : tag option
  proc verTag(tg : tag, t : time) : bool
  proc usedTime()                 : time
}.
```

The module `TagOracle` implements the `TagOracleT` functionality (see Appendix A). The `genTag(t)` procedure provides one-time tag generation functionality: on the first run the method delivers a value `Some (tagGen sk t)`, whereas afterwards it returns value `None`. The global state of `TagOracle` consists of a key pair, variable `used` telling if the `genTag` was already invoked, and `usedTime` which stores the time argument of the first call to the `genTag` procedure.

Forward-Resistance One-time tag system is forward resistant (FR) if adversaries cannot generate a valid tag for any time `t` given that they observed a tag for time `t'` and `t' < t`. This is the main motivating property of tag systems which also illustrates the essential difference between signature schemes and tag systems.

Recall that a secure (existentially unforgeable) one-time signature scheme must ensure that if an adversary has seen a signature for a message `m'` then it must not be able to produce a valid signature for any message `m` so that `m ≠ m'`.

From this perspective, in terms of a timeline, an FR tag system is only required to provide "half" of the security of a signature scheme. Buldas et al. observed that relaxing the security requirements allows building of efficient hash-based tag systems which later could be combined with cryptographic timestamping to give more efficient digital signature schemes [Buldas et al. 2019].

Let us formalize the concept of forward-resistance in terms of cryptographic games. As explained above, the adversaries are allowed to ask the tagging oracle for a tag associated with a time of their choice and must produce a valid tag for some later time.

```
module type AdvFR(TagO : TagOracleT) = {
  proc forge(pk : pkey) : tag * time {TagO.genTag}
}.
```

We say that a tag system is forward-resistant if the probability of winning `GameFR` by any efficient adversary of type `AdvFR` is small.

```
module GameFR(TagO : TagOracleT, A : AdvFR) = {
  module A = A(TagO)

  proc main() : bool = {
    var pk, sk, tg, t, t', forged;

    (pk, sk) <$ keyGen;
```

```
    TagO.init(pk, sk);
    (tg, t) = A.forge(pk);

    forged = TagO.verTag(tg, t);
    t'     = TagO.usedTime();

    return forged ∧ t' < t;
  }
}.
```

Tag-then-Hash Unpredictability The tag-then-hash unpredictability (THU) claims that, given a public key, any efficient adversary cannot produce a pair of time `t` and the hash of a tag for `t`.

```
module type AdvTHU = {
  proc forge(pk : pkey) : hash_output * time
}.

module GameTHU(A : AdvTHU) = {
  proc main() : bool = {
    var pk, sk, y, t;

    (pk, sk) <$ keyGen;
    (y, t) = A.forge(pk);

    return H (tagGen sk t) = y;
  }
}.
```

THU ensures that there are no "special" relations between the tag system and the hash function (see Section 4.2).

Phantom-Freeness A tag system is phantom-free if it is difficult to construct a valid tag for time `t` which is not equal to the canonical tag created by the tag generation function.

```
module type AdvPF(TagO : TagOracleT) = {
  proc forge(pk : pkey) : tag * time {TagO.genTag}
}.

module GamePF(TagO : TagOracleT, A : AdvPF) = {
  module A = A(TagO)

  proc main() : bool = {
    var pk, sk, tg, t, forged;

    (pk, sk) <$ keyGen;
    TagO.init(pk, sk);

    (tg, t) = A.forge(pk);
    forged = TagO.verTag(tg, t);

    return forged ∧ tg ≠ tagGen sk t;
  }
}.
```

4.2 Construction

Since *tag-then-hash unpredictability* and *phantom-freeness* are novel properties, we give an example of a toy n-bounded tag system which satisfies these two properties as well as forward-resistance:

- The private key `sk` is a list of $(z_1, \ldots, z_n)$ of n unpredictable values.
- The public key `pk` is the list $(h(h(z_1)), \ldots, h(h(z_n)))$, where h is one-way function.
- The tagging function `tagGen sk t` outputs the t-th component of the private key.
- The verification function `tagVer pk z t` verifies if $h(h(z))$ results in the t-th component of the public key.

Assuming that h is one-way, the tag system is forward-resistant, tag-then-hash unpredictable, and phantom-free. For the reader, it is instructive to think which property forces us to apply h twice in each component of the public key.

5 One-Time BLT Signature Scheme

In this section, we implement the BLT one-time signature scheme parameterized by a timestamping service and a tag system. We start by giving an intuitive overview of the version of BLT scheme from [Buldas et al. 2019].

Initialization Choose a forward-resistant tag system and a timestamping service. Generate a valid one-time public-private key pair `(pk, sk)` for the tag system.

Signing To sign a message `m`:

1. Query the time `t` of the timestamping service.
2. Use the private key to generate a tag `tg` for the next time `(t+1)`.
3. Bind the tag with the message by timestamping `(m, tg)`.
4. Output `(tg, t+1)` as a signature of `m`.
5. Dispose of the private key to avoid its accidental second-time use.

Verification To verify `m` against signature `(tg, t)`:

1. Verify `tg` against the time `t` and the public key `pk`.
2. Verify that the timestamping service contains a binding of `tg` and `m` at time `t`.

Security (EUF-CMA) Intuitively, the security of the described scheme is based on backdating resistance of the timestamping service and forward-resistance of the tag system. Let us assume that an adversary obtained a signature `(tg, t)` for a message `m`. One possible signature forging strategy is to obtain a valid tag-time pair `(tg', t')` where `t' < t` (this might be possible since the tag system is not required to be backward-resistant). However, to produce a valid signature the adversary must then backdate the tuple `(tg', m')` to the time `t'` and we assume it to be impossible. An alternative strategy is to timestamp a pair `(tg', m')`

at time t' so that t' > t, but this would require breaking forward-resistance of the tag system.

Notice that in the above description we timestamp a plain message-tag pair. This might be undesirable if the message is confidential or too large, or we do not want to expose the tag to the timestamping service. To this end, we provide an abstract function bind which converts message-tag pairs into bindings which could be timestamped later.

```
op bind : message * tag → data
```

The particular implementation of the bind function plays a crucial role in security proofs.

Next, we implement the main signature functionality as the BLTScheme module parameterized by a binding function, a timestamping service, and a tagging oracle. The module is stateless and has procedures for signature generation and verification.

```
module BLTScheme(TsO : TS, TagO : TagOracleT) = {
  proc sign(m : message) : tag * time = {
    var t, tg;

    t  = TsO.clock();
    tg = TagO.genTag(t+1);
    TsO.put(bind(m, oget tg));

    return (oget tg, t+1);
  }

  proc verify(m : message, tg : tag, t : time) :
      bool = {
    var valid, timestamped;

    valid       = TagO.verTag(tg, t);
    timestamped = TsO.check(t, bind(m, tg));

    return valid ∧ timestamped;
  }
}.
```

Correctness The signature scheme is functionally correct if given a valid key pair, signing a message with the private key produces a signature which verifies with the public key. Since BLT signature generation affects the global state, we express its correctness as a game which returns a verification result:

```
module BLTCorrect = {
  module BLT = BLTScheme(Ts, TagOracle)
  proc main(m : message) = {
    var pk, sk, t, tg, r;
    (pk, sk) <$ keyGen;
    Ts.init();
    TagOracle.init(pk, sk);
    (tg, t) = BLT.sign(m);
    r = BLT.verify(m, tg, t);
    return r;
  }
}.
```

For our correctness proof we use the previously defined timestamping service Ts and the tagging oracle TagOracle.

```
lemma bltCorrect :
  phoare [ BLTCorrect.main : true ⟹ res ] = 1%r.
```

In case of bounded tag systems we also must assume that any initial time sampled from tdistr is smaller than the key expiration date.

6 Security Analysis

The security of the BLT signature scheme strongly depends on how tags are bound to messages (plain or hashed) and whether adversaries have only read or also write access to the timestamping repository. In this section, we analyze four cases which arise from these possibilities.

6.1 One-Time Signing Oracles

We implement a wrapper around the stateless BLTScheme module which logs whether it was used, what message was signed, and which signature was produced.

```
module type BLTOracleT = {
  proc init(pk : pkey, sk : skey) : unit
  proc sign(m : message) : (tag * time) option
  proc verify(m : message, tg : tag, t : time) :
      bool
  proc fresh(m : message) : bool
}.
```

Signing Oracle The BLTOracle module (Appendix B) is a straightforward instantiation of the BLTOracleT interface. The global state of the oracle consists of variables qs, qt, and used. The variables qs and qt log the argument (message) and the time of the first execution of sign procedure, respectively. The variable used logs whether sign has already been invoked. The procedure init initializes the global variables, the timestamping service (Ts), and the tagging oracle (TagOracle). On the first run, sign(m) procedure returns the value Some (tg, t) where (tg, t) is a signature of m with t being the time of Ts and tg is the corresponding tag. On the following executions sign returns value None. The procedure fresh(m) checks whether the message m was previously signed by the oracle.

Dummy Oracle In some cases we might have an adversary who is successful in an attack without ever using the signing oracle. However, to run this adversary we still need to provide a module which fulfills the BLTOracleT interface. Therefore, we implement the BLTDummy module which always returns None when asked to sign a message (Appendix B).

6.2 Existential Unforgeability

It is common to assume that adversaries can trick a user into creating a valid signature of a message of their choice. Therefore we say that a digital signature is secure if adversaries who can use the real signer as "an oracle" cannot efficiently forge signatures for any messages whose signature were not obtained from the real signer. This property of digital signatures is known as existential unforgeability against chosen-message attacks (EUF-CMA).

We formalize these ideas below. The BLT adversary is parameterized by a timestamping service and a BLT oracle. The goal of an adversary is to produce a valid signature for a "fresh" message.

```
module type AdvBLT(T : TS, O : BLTOracleT) = {
  proc forge(pk : pkey) : message * tag * time
}.
```

The existential unforgeability game starts by generating a key pair and initializing the BLT oracle. Then the adversary generates a message-tag-time triple. If the tag-time pair represents a valid signature of the message and the message is fresh (i.e., was not previously signed by the BLT oracle) then the adversary wins the game.

```
module GameBLT(BLTO : BLTOracleT, A : AdvBLT) = {
  module A = A(Ts, BLTO)

  var tg : tag
  var m  : message
  var t  : time

  proc main() : bool = {
    var pk, sk, forged, fresh;

    (pk, sk) <$ keyGen;
    BLTO.init(pk, sk);

    (m, tg, t) = A.forge(pk);
    forged = BLTO.verify(m, tg, t);
    fresh  = BLTO.fresh(m);

    return forged ∧ fresh;
  }
}.
```

Our goal is to prove that for any efficient adversary the probability of winning the `GameBLT` is small. We do this by proving that this probability is upper-bounded by a sum of probabilities which are assumed to be small. In other words, we show that if `A` can win `GameBLT` then we can efficiently transform `A` into winning adversaries against security assumptions of tag systems and hash functions.

Clearly, no digital signature scheme is secure against *computationally unbounded* adversary: given a verification key, an unbounded adversary can try all possible signing keys to find one which gives a signature that passes verification. When we say "efficiently", we mean in time polynomial in the size of the key pair. EC does not provide any tools to verify computational complexity of modules. Therefore, an EC user must manually verify that implemented adversaries and their transformations remain efficient in this sense.

6.3 Illegal Reductions

Assume that `A` is a successful BLT adversary and we want to reduce it to the adversary against forward-resistance of a tag system. Here is a possible reduction:

```
module LameAdv(A : AdvBLT, T : TagOracleT) = {
  proc forge(pk : pkey) = {
    var t, tg;
    t  = Ts.clock();
    tg = tagGen TagOracle.sk t;
    return (tg, t);
  }
}.
```

While it is easy to prove that `LameAdv(A)` always wins the forward-resistance game (`GameFR`), this is clearly cheating, because the reduction `LameAdv` directly accesses the global variable of the tagging oracle which holds the secret key. In EC, one can check the validity of an adversary reduction by analyzing the accessed global variables:

```
print glob LameAdv(A).
```

The above command will indicate that, among others, the variable `TagOracle.sk` is accessed and therefore we consider the reduction `LameAdv` to be illegal.

Alternatively, we can ask EC to statically check whether `LameAdv(A)` belongs to the set `AdvFR{TagOracle}` which contains all FR-adversaries disjoint form `TagOracle`.

6.4 Plain Data, Read-Only Access

Let us first look at the simplest case when the timestamping repository contains plain data and adversary has no write access to the timestamping service (note that `T.put` is not listed in the allowed methods):

```
op bind: message * tag → message * tag = fun x, x.

module type AdvBLT(T : TS, O : BLTOracleT) = {
  proc forge(pk : pkey) : message * tag * time
                          {T.check O.sign}
}.
```

Let us fix a BLT-adversary for the rest of this section:

```
declare module A : AdvBLT
                   {Ts, TagOracle, BLTOracle}.
```

Theorem 6.1. *If the timestamping repository stores plain message-tag pairs then the probability of a read-only adversary performing a successful BLT forgery is zero.*

The most important step in the proof is establishing the following pre- and postcondition for the `A.forge` procedure:

```
Ts.r = empty ∧ TagOracle.used = BLTOracle.used ⟹
 (size Ts.r) ≤ 1 ∧
  forall i y, Ts.r.[i] = y ∧ y ≠ None =>
   exists m, Some m = BLTOracle.qs ∧
    y = Some (m, tagGen TagOracle.sk BLTOracle.qt)
```

The first conjunct of the postcondition claims that the size of the repository after running `A.forge` will be at most one. Indeed, since the adversary cannot directly write to the repository then the only entry can be created by using the one-time signing oracle. The second conjunct claims that if the repository contains a value then it will be `(m, tagGen sk qt)` where `m` is the message signed by signing oracle at time `qt`.

If `A` wins `GameBLT` then the repository has a message-tag pair `(m', tg')` and `m'` is fresh (`Some m' ≠ BLTOracle.qs`) which contradicts the established postcondition and hence:

```
lemma sec : forall &m,
 Pr[ GameBLT(BLTOracle, A).main() @ &m: res ]=0%r.
```

6.5 Hashed Data, Read-Only Access

Next, we look at the case when binding is a hash of a message-tag pair and adversaries are read-only.

```
op bind: message * tag → hash_output = fun x, H x.
```

Theorem 6.2. *If the timestamping repository holds hashes of message-tag pairs then the probability of a read-only adversary performing a successful BLT forgery is upper-bounded by collision resistance of the hash function.*

Similarly to the previous case, we establish that if the repository contains anything at all after `A.forge` then it is a hash of a message-tag pair used in the signing oracle:

```
Ts.r = empty ∧ TagOracle.used = BLTOracle.used ⟹
 (size Ts.r) ≤ 1 ∧
  forall i y, Ts.r.[i] = y ∧ y ≠ None =>
   exists m, Some m = BLTOracle.qs ∧
    y = Some (H (m,
     tagGen TagOracle.sk BLTOracle.qt))
```

Next, we implement a transformation of a BLT adversary into an adversary for collision resistance (see Section 2).

```
module T(A:AdvBLT) = {
  module G = GameBLT(BLTOracle, A)

  var p1, p2 : message * tag
  var b : bool

  proc adv() : (message*tag) * (message*tag) = {
    b = G.main();
    p1 = (oget BLTOracle.qs, tagGen TagOracle.sk
         BLTOracle.qt);
    p2 = (GameBLT.m, GameBLT.tg);
    return (p1, p2);
  }
}.
```

The CR-adversary `T(A)` runs the BLT-game with `A` (stores the outcome in the global variable `b`) and returns a tuple of message-tag pairs as a collision for `H`. The first pair consists of the message signed by the BLT oracle (`qs`) and the respective tag. The second pair consists of the message and the tag which `A` returned in the BLT-game. Note that our transformation directly accesses the secret key of a `TagOracle` module. This is a legal move, because `T(A)` is an adversary for collision resistance (not for forward-resistance as in Section 6.3) which does not require secrecy of any keys.

Next, we show that if `A` wins the BLT-game (incorporated in the `T` transformation) then `T(A)` wins the CR-game:

```
lemma reductionCR : phoare [ GameCR(T(A)).main :
  true ⟹ T.b => res ] = 1%r.
```

If `A` wins the BLT-game, it produces a triple `(m', tg', t')` such that `T.r.[t'] = Some (H (m', tg'))` and `m'` is fresh (`m ≠ m'`, where `m` is the message signed by BLT-oracle, i.e., `BLTOracle.qs = Some m`). We also proved that any value stored in the timestamping repository must be equal to `Some (H (m, tagGen sk qt))`. Hence, we constructed a collision since tuples `(m, tg)` and `(m', tagGen sk qr)` are different, but their hashes are equal.

We now prove that running the BLT-game with `A` in the `T` module produces the same outcome as simply running the BLT-game:

```
lemma rel1: equiv [ GameBLT(BLTOracle, A).main ~
  GameCR(T(A)).main : ={glob A, glob BLTOracle} ⟹
    res{1} = T.b{2} ].
```

The next step is to combine the results of the two previous lemmas into the equivalence which states that the positive outcome of `A` in the BLT-game implies positive outcome of `T(A)` in the CR-game:

```
lemma rel2: equiv [ GameBLT(BLTOracle, A).main ~
  GameCR(T(A)).main : ={glob A, glob BLTOracle} ⟹
    res{1} => res{2} ].
```

The equivalence `rel2` implies that CR is an upper bound for EUF of the BLT scheme:

```
lemma sec : forall &m,
  Pr[ GameBLT(BLTOracle, A).main() @ &m : res ] ≤
  Pr[ GameCR(T(A)).main() @ &m : res ].
```

6.6 Plain Data, Read-Write Access

In this section, we analyze the BLT scheme when adversaries have read-write access to the timestamping repository which contains plain message-tag pairs.

```
op bind: message * tag → message * tag = fun x, x.
```

```
module type AdvBLT(T : TS, O : BLTOracleT) = {
  proc forge(pk : pkey) : message * tag * time
    {T.check T.put O.sign}
}.
```

Theorem 6.3. *If the timestamping repository holds plain message-tag pairs then then probability of a read-write adversary performing a successful BLT forgery is upper-bounded by probability of breaking forward-resistance of the tag system.*

After running the BLT-game, the variable `BLTOracle.qt` holds the time value when the adversary used the BLT-oracle to sign a message (if `qt` is zero then the oracle was not used). Similarly, the variable `GameBLT.t` stores the time value which is associated with the forged signature computed by the adversary. We split the probability of winning the BLT game into three cases based on the relative order of these time values.

In the first case, the adversary `A` first uses the signing oracle and later timestamps a forged message-tag pair. More formally, we are interested in finding an upper bound for the following probability:

```
Pr[ GameBLT(BLTOracle, A).main() @ &m :
  res ∧ BLTOracle.qt < GameBLT.t ]
```

We note that after using the signing oracle, the adversary observes a tag for time `BLTOracle.qt` and then successfully produces a tag for the later time `GameBLT.t`. In this scenario, `A` manages to break forward-resistance of the tag system. To turn `A` into an FR-adversary, we must not forget to manually initialize the `used` variable of the BLT-oracle.

```
module D(A : AdvBLT, O : TagOracleT) = {
  module A = A(Ts, BLTOracle)

  proc forge(pk:pkey) = {
    var tg, m, t;
    Ts.init();
    BLTOracle.used = false;
    (m, tg, t) = A.forge(pk);
    return (tg, t);
  }
}.
```

If we analyze the BLT-game with `A` and the FR-game with `D(A)` then we see that the same computations are performed, but the FR-game ignores the message and its freshness. Thus, if `A` wins the BLT-game then `D(A)` wins the FR-game:

```
lemma c1 : equiv [ GameBLT(BLTOracle, A).main ~
  GameFR(TagOracle, D(A)).main : ={glob A} ⟹
  res{1} ∧ BLTOracle.qt{1} < GameBLT.t{1} =>
    res{2} ].
```

The equivalence `c1` allows us to conclude that the probability of the first case is upper-bounded by the forward-resistance.

```
lemma case1 : forall &m,
  Pr[ GameBLT(BLTOracle, A).main() @ &m :
    res ∧ BLTOracle.qt < GameBLT.t ] ≤
  Pr[ GameFR(TagOracle, D(A)).main() @ &m : res ].
```

In the second case, we analyze the probability of forgery being associated with the time when the signing oracle was invoked (`GameBLT.t = BLTOracle.qt`). Since the timestamping repository contains *plain* message-tag pairs, this case cannot occur. To show it, we prove that when `A` wins then BLT-game, then the time associated with the forgery is always different from the time of using the oracle.

```
lemma c2 : phoare [ GameBLT(BLTOracle, A).main :
  true ⟹ res => GameBLT.t ≠ BLTOracle.qt] = 1%r.
```

Let us sketch the proof. If a forged signature is valid (`res` is true) then `GameBLT.t ≠ 0`. If the oracle was not used then `BLTOracle.qt = 0` and we are done. On the other hand, if the BLT-oracle was used then `Ts.r.[BLTOracle.qt]` contains a "non-fresh" message signed by the oracle. This fact, combined with the premise that `A` was successful implies that `GameBLT.t` must differ from `BLTOracle.qt`.

We use the De Morgan's law to convert the PHL statement `c2` to the probability statement for this case:

```
lemma case2 : forall &m,
  Pr[ GameBLT(BLTOracle, A).main() @ &m :
    res ∧ GameBLT.t = BLTOracle.qt ] = 0%r.
```

In the third case, the adversary wins by using the signing oracle after timestamping the forged message-tag pair (`GameBLT.t < BLTOracle.qt`). Intuitively, it is clear that if adversary managed to timestamp a forgery *before* using the signing oracle then he must also be successful in performing a forgery with the "dummy" oracle. One might be tempted to prove the following equivalence:

```
lemma c3' : equiv [ GameBLT(BLTDummy, A).main ~
  GameBLT(BLTOracle, A).main : ={glob A} ⟹
    res{2} ∧ BLTOracle.used{2} ∧
      GameBLT.t{2} < BLTOracle.qt{2} => res{1} ].
```

Unfortunately, it does not work since the adversary might change his mind about revealing the forgery after discovering that the access to the real oracle was not granted. Therefore, we need to implement a wrapper around `A`:

```
module C(A : AdvBLT, T : TS, O : BLTOracleT) = {
  module A = A(Ts, O)

  proc forge(pk:pkey) = {
    var z, t;
    A.forge(pk);
    (t,z) = filter (fun (a:int) (b:message * tag)
        => tagVer pk a b.2) Ts.r;
    return (z.1, z.2, t);
  }
```

```
}.
```

The wrapper C runs A and then scans the entries of the timestamping repository to find an entry which contains a valid tag. C(A) is guaranteed to find at least one valid tag since we assumed that A wins the BLT-game and that the relevant entries are filled before accessing the BLT-oracle:

```
lemma c3 : equiv [ GameBLT(BLTDummy, C(A)).main ~
  GameBLT(BLTOracle, A).main : ={glob A} ==>
    res{2} /\
      GameBLT.t{2} < BLTOracle.qt{2} => res{1} ].
```

Another useful fact is that if an adversary A wins the BLT-game with BLTDummy then N(A) wins the forward-resistance game, where N transforms A into the FR-adversary by running A and returning the tag-time tuple computed by A.

```
lemma d2f : forall (A <: AdvBLT{Ts, TagOracle}),
 equiv [ GameBLT(BLTDummy, A).main ~
  GameFR(TagOracle, N(A)).main :
    ={glob A} ==> res{1} => res{2} ].
```

By combining the equivalences c3 and d2f, we arrive at the conclusion that this case is also bounded by forward-resistance:

```
lemma case3 : forall &m,
  Pr[ GameBLT(BLTOracle, A).main() @ &m :
    res /\ GameBLT.t < BLTOracle.qt ] <=
  Pr[ GameFR(TagOracle, N(C(A))).main() @ &m :
    res ].
```

To sum up, the described three cases cover all of the possibilities to win the BLT-game and we have shown that in each of these cases the probability is upper-bounded by forward-resistance of the tag system.

6.7 Hashed Data, Read-Write Access

Finally, we analyze the BLT scheme in combination with read-write adversaries and the hashed values in the repository.

```
op bind : message * tag -> hash_output *
     hash_output = fun x, (HM x.1, HT x.2).
```

Note that in this case the bind function returns a hash of a message paired with the hash of a tag. Our analysis shows that binding a message-tag pair into one hash value (as in Section 6.5) leads to a non-trivial *non-malleability* requirements on the hash function. In the definition of bind, one might think of HM and HT as the same hash function, but differently typed.

Theorem 6.4. *If the timestamping repository holds tuples of hashes computed from message-tag pairs then probability of a read-write adversary performing a successful BLT forgery is bounded by a sum of probabilities of breaking the forward-resistance, phantom-freeness, collision resistance, and tag-then-hash unpredictability scaled by the number of entries in the repository.*

We analyze the same cases as in the previous section. The security proof for the first case (forgery done after using the oracle) remains exactly the same. Like the proof in Section 6.5, the second case (the forgery time and the time of using the signing oracle are the same) is bounded by collision resistance of HM.

This leaves us with the last possibility when the adversary uses the signing oracle after timestamping the hashes of a message-tag pair (GameBLT.t < BLTOracle.qt). In this case, the actions of the adversary can be split into three phases:

1. Filling the timestamping repository with entries.
2. Signing a message with the BLT-oracle.
3. Computing a tag-message-time triple which comprises a forgery.

Notice how the new information accumulates through the phases. In the first phase, the adversary fills the repository based only on the public key of a tag system. In the second phase, the adversary chooses the message and the signing time based on the public key and the entries in the repository. In the final phase, the adversary also learns the signature from the second phase and since the tag system is not backward-resistant it does not prohibit deducing tags for the previous time values.

Recall that after running the BLT-game, the variables GameBLT.t and GameBLT.tg store the components of the BLT signature returned by the adversary. If GameBLT.tg is not equal to tag sk GameBLT.t then adversary managed to construct a phantom tag. Since in the phantom-freeness game the adversary can access the tagging oracle, the conversion from a BLT-adversary to a PF-adversary is trivial:

```
module F(A : AdvBLT, O : TagOracleT) : AdvPF = {
  module A = A(Ts, BLTOracle)

  proc forge(pk : pkey) = {
    var tg, m, t;
    BLTOracle.used = false;
    Ts.init();
    (m, tg, t) = A.forge(pk);
    return (tg, t);
  }
}.
```

If A manages to construct a non-canonical tag, then F(A) wins the phantom-freeness game:

```
lemma tgc : equiv [ GameBLT(BLTOracle, A).main ~
  GamePF(TagOracle, F(A)).main :
  ={glob A, glob TagOracle, glob BLTOracle,
    glob Ts} ==>
(tagGen TagOracle.sk GameBLT.t <> GameBLT.tg){1} =>
  res{2} ].
```

```
lemma case3_1 : forall &m,
  Pr[ GameBLT(BLTOracle, A).main() @ &m :
    res ∧ GameBLT.t < BLTOracle.qt ∧
    tagGen TagOracle.sk GameBLT.t ≠ GameBLT.tg ] ≤
  Pr[ GamePF(TagOracle, F(A)).main() @ &m : res ].
```

In the last case, when `GameBLT.tg = tag sk GameBLT.t`, we notice that in the first phase (filling the repository) the adversary managed to find a hash of a tag, i.e., to break tag-then-hash unpredictability (THU). The important aspect is that the THU-adversary has no access to the tagging oracle, which fits nicely with the fact that the adversary "commits" to the repository entries before using the signing oracle. At the same time, without the real signing oracle in the second and third phases, the adversary can arbitrarily change his behavior. Therefore, our strategy is to run the adversary with the dummy oracle and then uniformly choose one of the entries from the repository:

```
module H(A:AdvBLT) = {
  module A = A(Ts, BLTDummy)

  var guess : int

  proc forge(pk:pkey) = {
    Ts.init();
    A.forge(pk);
    guess <$ [1 .. kpe];
    return (oget Ts.r.[guess].2, guess);
   }
}.
```

It might be instructive to think why we cannot search the repository for the right entry similarly to what we did in the third case of Section 6.6.

In the reduction above we use the `kpe` constant which denotes the expiration date of a bounded tag system. In the unbounded case, `kpe` constant can be replaced by an upper bound on the number of entries that an efficient (i.e., polynomial-time) adversary can possibly produce.

First, we prove that if `A` wins the BLT-game and the guess coincides with the forgery time, then `H(A)` also wins the THU-game:

```
lemma htueq' : equiv [ GameTHU(H(A)).main ~
  GameBLT(BLTOracle, A).main :
  ={glob A, glob BLTOracle} ⟹
  res{2} ∧ GameBLT.t{2} = H.guess{1} => res{1} ].
```

Unfortunately, by the rules of PRHL the equivalence `htueq'` cannot be reinterpreted into probabilities of events. This is due to the fact that the postcondition compares values from the different runs (memories). This problem can be fixed by coding a small wrapper around `A`, which chooses the value from the same interval as `H(A)`:

```
module W(A : AdvBLT, T : TS, O : BLTOracleT) = {
  var guess : int

  proc forge(pk : pkey) = {
    var r;
    r = A.forge(pk);
    guess <$ [1 .. kpe].
    return r;
  }
}.
```

Now we can prove a similar equivalence which only compares values from the same memory and can therefore be transformed into a statement about probabilities:

```
lemma htueq : equiv [ GameTHU(H(A)).main ~
  GameBLT(BLTOracle, W(A)).main :
  ={glob A, glob BLTOracle} ⟹
  res{2} ∧ GameBLT.t{2} < BLTOracle.qt{2} ∧
    tagGen TagOracle.sk GameBLT.t = GameBLT.tg ∧
    GameBLT.t{2} = W(A).guess{2} => res{1} ].
```

This trick works out because `H(A)` and `W(A)` are sampling the guess "simultaneously". This allows us to identify the guess value in the first game with the guess in the second game.

```
lemma case3_2 : forall &m,
  Pr[ GameBLT(BLTOracle, W(A)).main() @ &m :
    res ∧ GameBLT.t < BLTOracle.qt ∧
    tagGen TagOracle.sk GameBLT.t = GameBLT.tg ∧
    GameBLT.t = W(A).guess ] ≤
  Pr[ GameTHU(H(A)).main() @ &m : res ].
```

Finally, we observe that the value `guess` is sampled independently and uniformly and the wrapper `W(A)` does not play any role in the outcome of the BLT-game (`A` wins iff `W(A)` wins), which leads us to the following upper bound:

```
lemma case3 : forall &m,
  Pr[ GameBLT(BLTOracle, A).main() @ &m :
    res ∧ GameBLT.t < BLTOracle.qt ] ≤
  Pr[ GameTHU(H(A)).main() @ &m : res ] * kpe +
  Pr[ GamePF(TagOracle, F(A)).main() @ &m : res ].
```

By putting all the cases together we conclude that the existential unforgeability of the BLT scheme depends on forward-resistance, tag-then-hash unpredictability, phantom-freeness, collision resistance, and the key expiration date (or the size of the repository in case of unbounded tag system).

7 Conclusions and Future Work

We have presented a formalization of the BLT signature scheme. Along our way we defined an ideal model of universally composable timestamping and formally specified tag systems and their security properties. We have shown that the existential unforgeability of the BLT scheme depends on

the kind of message-tag binding used and whether we assume that adversaries have write access to the timestamping repository.

In the future, it would be interesting to look at more involved, but also practically more relevant types of timestamping repositories. For example, some real-world timestamping services store multiple entries per round, combined into a Merkle tree.

Another interesting direction is to investigate implementions of efficient and provably secure tag systems based only on the standard properties of hash functions.

References

Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. Easycrypt: A tutorial. In *Foundations of security analysis and design vii*. Springer, 146–166.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2012. Probabilistic relational Hoare logics for computer-aided security proofs. In *International Conference on Mathematics of Program Construction*. Springer, 1–6.

Josh Benaloh and Michael de Mare. 1991. *Efficient Broadcast Time-Stamping*. Technical Report. Clarkson University.

Ahto Buldas, Denis Firsov, Risto Laanoja, Henri Lakk, and Ahto Truu. 2019. A New Approach to Constructing Digital Signature Schemes. In *Advances in Information and Computer Security*, Nuttapong Attrapadung and Takeshi Yagi (Eds.). Springer International Publishing, Cham, 363–373.

Ahto Buldas, Andres Kroonmaa, and Risto Laanoja. 2013. Keyless Signatures' Infrastructure: How to Build Global Distributed Hash-Trees. In *NordSec 2013, Proceedings (LNCS)*, Vol. 8208. Springer, 313–320.

Ahto Buldas, Risto Laanoja, and Ahto Truu. 2017. A Server-Assisted Hash-Based Signature Scheme. In *NordSec 2017, Proceedings (LNCS)*, Vol. 10674. Springer, 3–17.

Ahto Buldas, Peeter Laud, Märt Saarepera, and Jan Willemson. 2005. Universally composable time-stamping schemes with audit. In *International Conference on Information Security*. Springer, 359–373.

Ahto Buldas and Märt Saarepera. 2004. On Provably Secure Time-Stamping Schemes. In *ASIACRYPT 2004, Proceedings (LNCS)*, Vol. 3329. Springer, 500–514.

Whitfield Diffie and Martin E. Hellman. 1976. New Directions in Cryptography. *IEEE Trans. Inf. Theor.* 22, 6 (Nov 1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

Stuart Haber and W. Scott Stornetta. 1991. How to Time-Stamp a Digital Document. *Journal of Cryptology* 3, 2 (1991), 99–111.

Toshihiko Matsuo and Shin'ichiro Matsuo. 2005. On Universal Composable Security of Time-Stamping Protocols. *IWAP* 2005 (2005), 169–181.

Ralph C. Merkle. 1979. *Secrecy, Authentication and Public Key Systems*. Ph.D. Dissertation. Stanford University.

Ralph C. Merkle. 1980. Protocols for Public Key Cryptosystems. In *IEEE Symposium on Security and Privacy*. 122–134.

A Tagging Oracle

```
module TagOracle : TagOracleT = {

  var usedFlag : bool
  var usedTime  : int
  var pk        : pkey
  var sk        : skey

  proc init(pk : pkey, sk : skey) : unit = {
    TagOracle.pk = pk;
    TagOracle.sk = sk;
    usedFlag     = false;
    usedTime      = 0;
  }

  proc genTag(t : int) : tag option = {
    var r = None;
    if(!usedFlag){
      usedTime = t;
      r = Some (tagGen sk usedTime);
    }
    usedFlag = true;

    return r;
  }

  proc verTag(tg : tag, t : int) : bool = {
    return tagVer pk t tg;
  }

  proc usedTime() : int = {
    return usedTime;
  }
}.
```

B BLT Oracles

```
module BLTOracle : BLTOracleT = {
  module BLT = BLTScheme(Ts, TagOracle)

  var qs   : message option
  var qt   : int
  var used : bool

  proc init(pk : pkey, sk : skey) : unit = {
    TagOracle.init(pk, sk);
    Ts.init();
    qs   = None;
    used = false;
  }

  proc sign(m : message) : (tag * int) option = {
    var r, q;
    if (!used) {
      qs = Some m;
      (q, qt) = BLT.sign(m);
      r = Some (q, qt);
    } else {
      r = None;
    }
    used = true;
    return r;
  }

  proc verify(m : message, tg : tag, t : int):bool
       = {
      var b : bool;
      b = BLT.verify(m, tg, t);
```

```
      return b;
  }

  proc fresh(m : message) : bool = {
    return qs <> Some m;
  }
}.

module BLTDummy : BLTOracleT = {
  module BLT = BLTScheme(Ts, TagOracle)

  proc init(pk : pkey, sk : skey) : unit = {
    TagOracle.init(pk, sk);
    Ts.init();
  }

  proc sign(m : message) : (tag * int) option = {
     return None;
  }

  proc verify(m : message, tg : tag, t : int) :
      bool = {
      var b : bool;
      b = BLT.verify(m, tg, t);
      return b;
  }

  proc fresh(m : message) : bool = {
    return true;
  }
}.
```