Verified Multiple-Time Signature Scheme from One-Time Signatures and Timestamping

Denis Firsov
Tallinn University of Technology
Tallinn, Estonia
denis@cs.ioc.ee

Henri Lakk
Guardtime
Tallinn, Estonia
henri.lakk@guardtime.com

Ahto Truu
Guardtime
Tallinn, Estonia
ahto.truu@guardtime.com

2021 IEEE 34th Computer Security Foundations Symposium (CSF) | 978-1-7281-7607-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSF51468.2021.00051

***Abstract*—Buldas, Laanoja, and Truu designed a family of server-assisted digital signature schemes (BLT signatures) built around cryptographic timestamping and forward-resistant tag systems. The original constructions had either expensive key generation phase or stateful client-side computations.**

In this paper, we construct a stateless tag system with efficient key generation from one-time signature schemes. We prove that the proposed tag system is forward-resistant and when combined with cryptographic timestamping, it induces a secure (existentially unforgeable) multiple-time signature scheme. Our constructions are developed and verified using the EasyCrypt framework.

***Index Terms*—digital signatures, EasyCrypt, formalized cryptography, timestamping**

I. Introduction

Buldas, Laanoja, and Truu [1] observed that many practical deployments of digital signatures rely on timestamping services to handle key revocation. By combining key status tracking and timestamping, signatures created before the key revocation can be treated as valid, whereas signatures created afterwards can be considered invalid. This enables non-repudiation of signatures, i.e. the possibility to use signatures as evidence against the signer. Without key revocation a user may (fraudulently) claim that their private key was stolen and someone else may have created signatures in their name.

The observation led to a novel type of digital signature schemes (*BLT scheme* in the following) which is built on the fact that digital signatures must be timestamped. The main goal was to design more efficient signature schemes by intrinsically relying on the cryptographic properties of timestamps. The BLT schemes could be decomposed into two functional components: *cryptographic timestamping* and *forward-resistant tag systems*. The decomposition made it possible to describe the family of BLT signatures in an abstract way and also propose other forward-resistant tag systems and arrive at a rich family of different BLT signature schemes with distinct functional and security properties [2], [3].

The practicality of the original BLT signature was limited by the fact that the initialization phase pre-generated all time-bound keys. For practical deployments, the number of these keys tends to be large, which makes the key generation prohibitively slow on constrained devices such as smart cards [1]. Later, the problem of expensive key generation was solved by introducing stateful computations to the user (signer) side. In practice, it is undesirable to require a user to keep state, as this would introduce security threats, and would make it hard to use the same keypair on different devices of the same user [2]. Also, it is important to note that the security of the prior systems was only analyzed in one-time use setting.

Historically, the first digital signature schemes were "one-time use" which means that a public-private keypair could only be used for signing a single message [4]. Later, Merkle [5], [6] and Goldreich [7] proposed generic ways to turn one-time schemes into many-time schemes. In this paper, we use their ideas to implement a stateless multiple-time tag system from one-time signatures, which we then combine with cryptographic timestamping to obtain an efficient and secure multiple-time BLT signature scheme with stateless client side.

The technical contributions of the paper are:

- We present a construction of multiple-time BLT signature scheme with stateless client side based on a stateless multiple-time tag system and timestamping service (Sect. III-C).
- We formally derive unforgeability of the multiple-time BLT signature (EUF-CMA) from multiple-time forward-resistance of the tag system. (Sect. III-D).
- We propose a computationally efficient construction of a stateless multiple-time tag system from a one-time signature scheme (Sect. IV-A).
- We formally derive multiple-time forward-resistance of the proposed tag system. (Sect. IV-B and Sect. IV-C).

All of our results have been formalized in the EasyCrypt theorem prover and the code is available as the accompanying supplementary material. The presentation in this paper follows the EasyCrypt formalization, but for sake of readability we skip the technical and auxiliary details. The EasyCrypt formalism is introduced by stepping through an example where we define one-time signatures and develop a variation of the hash-based Lamport's signatures (Sect. II-A).

II. Background

In this section, we introduce EasyCrypt by developing a simple one-time digital signature scheme. Also, we describe two standard techniques for turning one-time schemes into multiple-time schemes.

A. EasyCrypt: One-Time One-Bit Signature

EasyCrypt (EC) is a framework for building and verifying security of cryptographic constructions. In this section, we briefly outline some basic concepts behind EC by formally introducing one-time signatures. More information on EC can be found in [8].

We illustrate the proof development process on a simple example regarding one-time signatures and preimage resistance of hash functions. More specifically, our goal is to construct a variation of historically first digital signature based on hash functions [4].

The proof development starts with 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).

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 abstract signing and verification functions with their respective types.

```
type pkeyOTS, skeyOTS, msg, sig.
op otsSig : skeyOTS → msg → sig.
op otsVer : pkeyOTS → msg → sig → bool.
```

We assume a deterministic secret key generation algorithm parameterized by an unpredictable token from a set `r`. Every secret key has a corresponding public key that can be computed from it with the function `sk2pk`. We use this two-step decomposition of keypair generation to simplify some techical work in the following analysis.

```
type r.
op otsKey : r → skeyOTS.
op sk2pk  : skeyOTS → pkeyOTS.
```

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* `AdvOTS` describing the interface of adversaries whose task is to forge a signature:

```
module type AdvOTS(O : OTSOracleT) = {
  proc forge(pk : pkeyOTS) : msg * sig {O.sign}
}.
```

The adversary has access to a one-time signing oracle of type `OTSOracleT` and is allowed to execute the `sign` procedure (specified in curly braces next to the `forge` method).

```
module type OTSOracleT = {
  proc * init(pk : pkeyOTS, sk : skeyOTS) : unit
  proc sign(m : msg) : sig option
  proc fresh(m : msg) : bool
}.
```

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.

The standard one-time signing oracle is implemented as module `OTSO` (see Appendix A). The `sign(m)` procedure allows to sign a single message. The global state of `OTSO` consists of variables indicating whether the oracle has been used and which message was signed.

Next, we formalize the existential unforgeability under chosen message attack (EUF-CMA) as a parameterized module `GameOTS` which is played by an adversary of type `AdvOTS`. The adversary receives a public key and has access to a one-time signing oracle. If the adversary manages to produce a valid signature on a fresh message then it wins the game. Message is fresh if the oracle did not sign it for the adversary. The game is also parameterized by a key generator of type `KeyGenT`.

```
module type KeyGenT = {
 proc keyGen() : pkeyOTS * skeyOTS
}.

module GameOTS(O : OTSOracleT, A : AdvOTS,
    K : KeyGenT) = {
  module A = A(O)
  var s : sig
  var m : msg
  proc main() : bool = {
    var pk, sk, forged, fresh;
    (pk, sk) = K.keyGen();
    O.init(pk, sk);
    (m, s) = A.forge(pk);
    forged = otsVer pk m s;
    fresh  = O.fresh(m);
    return forged ∧ fresh;
  }
}.
```

This concludes the abstract definitions of one-time signatures, and we can proceed to the preimage resistance of hash functions. We let our development be parameterized by an abstract hash function with the respective input and output types, and an unpredictable distribution of its input values:

```
type hash_input, hash_output.

op h : hash_input → hash_output.
op rDistr : hash_input distr.
```

To formalize the preimage resistance we define a parameterized module `GamePRE` which is played by an adversary of type `AdvPRE`.

```
module type AdvPRE = {
  proc invert(v : hash_output) : hash_input
}.
```

The adversary receives an output of the hash function `h` and returns its input. The adversary wins the game if the hash of the value returned by the adversary equals the value given to the adversary as an argument:

```
module GamePRE(A : AdvPRE) = {
  proc main() = {
    var r1, r2;
    r1 =$ rDistr;
    r2 = A.invert(h r1);
    return h r2 = h r1;
  }
}.
```

B. Lamport One-Time Signature Scheme

Now we are ready to implement and prove the properties of the Lamport signature scheme [4] restricted to signing single bits. The idea behind the scheme is that the secret key consists of two unpredictable values `r1` and `r2`. The public key consists of the hashes of these values. Then to sign the bit `true` we output the tuple which consists of values `r1` and `(h r2)`. Symmetrically, the signature of `false` is the tuple `(h r1, r2)`. We start by instantiating the parameters:

```
type hash_input     = bits.
type hash_output    = bits.
type msg            = bool.
type skeyOTS        = bits * bits.
type pkeyOTS        = bits * bits.
type sig            = bits * bits.

op otsSig (sk : skeyOTS, m : msg) : sig =
  m ? (sk.`1, h sk.`2) : (h sk.`1, sk.`2).

op otsVer (p : pkeyOTS, m : msg, s : sig) : bool =
   m ∧ p = (h s.`1, s.`2) ∨
  !m ∧ p = (s.`1, h s.`2).
```

We also need to implement the key generation algorithm. Since unpredictable values are the secret keys themselves then `otsKey` is simply an identity function, and we ignore it here.

```
module LKG : KeyGenT = {
  proc keyGen() = {
    var r1, r2;
    r1 =$ rDistr;
    r2 =$ rDistr;
    return ((h r1, h r2), (r1, r2));
  }
}.
```

Next, we address the security properties of the implemented signature scheme. The goal is to prove that the probability of breaking the Lamport signature is bounded by the probability of breaking the preimage resistance of the hash function.

We start by fixing the adversary `A : AdvOTS`. Then, we split the probability of the adversary winning the OTS game into two cases: adversary forges the signature either for the bit `false` or the bit `true`.

```
lemma prSplit &m :
  Pr[ GameOTS(OTSO, A, LKG).main() @ &m : res ]
= Pr[ GameOTS(OTSO, A, LKG).main() @ &m
      : res ∧ GameOTS.m = false ]
+ Pr[ GameOTS(OTSO, A, LKG).main() @ &m
      : res ∧ GameOTS.m = true ].
```

Due to the symmetry of both cases we only describe the case when `A` forges signature for the bit `false`. We convert the adversary `A` into and adversary that breaks preimage resistance whenever `A` forges the signature for a `false` bit:

```
module OTS2PRE_F(A : AdvOTS) = {
  module A = A(OTSO')
  proc invert(v : hash_output) = {
    var r1, m, s;
    r1 =$ rDistr;
    OTSO.init((h r1, v), (r1, witness));
    (m, s) = A.forge((h r1, v));
    return s.`2;
  }
}.
```

Note that forging the signature for the bit `false` requires finding a preimage for the second component of the public key. Therefore, `OTS2PRE_F` initializes the OTS oracle by using values `h r1` and `pk` as the public key and values `r1` and `witness` as the secret key. Since `witness` is an arbitrary value from the set `bits` then at this stage the signing oracle is only semi-functional—it can correctly sign the bit `true`, but not the bit `false`. The adversary `A` will not notice this, however, as by assumption it forges the signature for the bit `false`, and the forgery can be successful only if oracle is not asked to sign the bit `false`.

Next, we use the program logics of EasyCrypt to derive a logical consequence that if `A` wins `GameOTS` then `OTS2PRE_F(A)` wins `GamePRE`. The probabilistic interpretation of this consequence is that the probability of `A` winning the OTS game is bounded by the probability of `OTS2PRE_F(A)` winning the preimage resistance game.

```
lemma c1 &m :
  Pr[ GameOTS(OTSO, A, LKG).main() @ &m : res ∧
      GameOTS.m = false ]
≤ Pr[ GamePRE(OTS2PRE_F(A)).main() @ &m : res ].
```

If we repeat the exercise from above for the case when `A` forges a signature for the bit `true`, then we end up with the following upper bound on the successful forgery for the Lamport signature scheme:

```
lemma lamportSecurity &m :
  Pr[ GameOTS(OTSO, A, LKG).main() @ &m : res ]
≤ Pr[ GamePRE(OTS2PRE_F(A)).main() @ &m : res ]
+ Pr[ GamePRE(OTS2PRE_T(A)).main() @ &m : res ].
```

We can conclude that Lamport signature is at least as strong as the preimage resistance of the used hash function.

C. Merkle Multiple-Time Signature Scheme

Once one-time digital signature schemes were introduced, it became important to answer whether it is possible to construct multiple-time signatures. Merkle [9] proposed a generic way of turning one-time signature schemes into stateful multiple-time

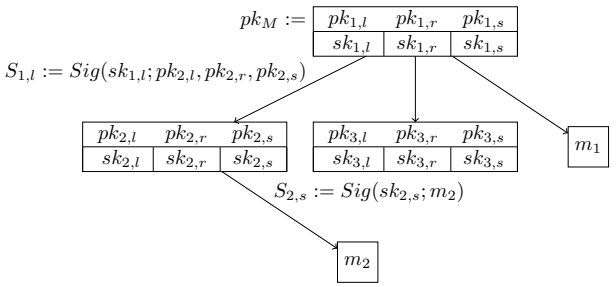

Fig. 1. An example of the Merkle multiple-time signature scheme with two signing keys $sk_{1,s}$ and $sk_{2,s}$ spent on messages m_1 and m_2, respectively, and one unspent signing key $sk_{3,s}$.

schemes. The construction is based on a binary tree, where every node is composed of three one-time signature keypairs, of which two are used to sign the left and right child nodes and one is for signing a message.

To sign a message, the signer picks a fresh node which was not used before. The signature consists of a chain of public-key authentications from the root to the fresh node and the one-time signature on the message itself with the secret key of the fresh node. In this way, the public keys of child nodes are always authenticated by their parents. Therefore, the public key of the resulting multiple-time scheme consists of the one-time public keys associated with the root node and the secret key consists of the one-time secret keys of all the nodes (possibly derived from a single seed by using a pseudorandom function). The state is a counter which is needed to exclude the already used nodes. The signature size in this scheme grows logarithmically with the number of messages signed.

Figure 1 depicts a tree from this signature scheme, where the public key pk_M of the tree is formed by the three public keys of the root node ($pk_{1,l}$, $pk_{1,r}$ and $pk_{1,s}$). Two of the three signing keys ($sk_{1,s}$ and $sk_{2,s}$) have been used to sign messages m_1 and m_2, respectively, and $sk_{3,s}$ is still available for signing a new message. The signature of the message m_2 is the one-time signature $S_{2,s}$ on the message m_2 itself, the public keys of the tree node 2 ($pk_{2,l}$, $pk_{2,r}$, $pk_{2,s}$), and the signature $S_{1,l}$ on the three public keys. In order to verify the signature of the message m_2, one has to verify the one-time signature $S_{2,s}$ on message m_2 with the public key $pk_{2,s}$. The public key $pk_{2,s}$ itself, alongside with $pk_{2,l}$ and $pk_{2,r}$, is signed and the signature $S_{1,l}$ can be verified using the public key $pk_{1,l}$ which is part of pk_M.

D. Goldreich Multiple-Time Signature Scheme

Goldreich [7] proposed a stateless hash-based signature scheme (GSS) using a binary authentication tree of one-time signatures. The scheme is based on a finite binary tree, where every node contains a one-time signature keypair. A message is signed using a leaf node one-time keypair while every non-leaf is used to authenticate (sign) the public keys of both of its children. The public key of this scheme is the public key of the root node and the secret key consists of secret keys of all tree nodes (which again may be generated pseudorandomly).

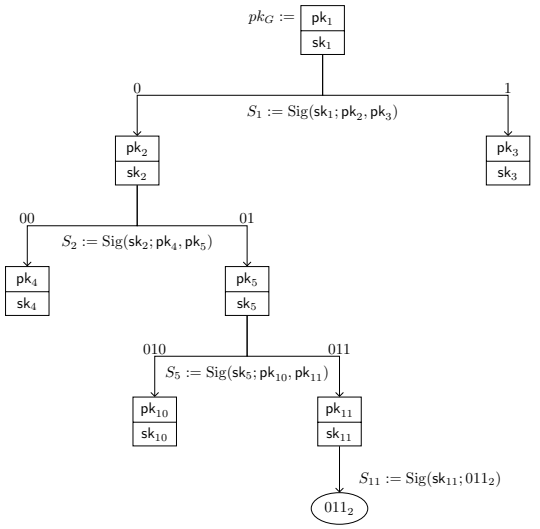

Fig. 2. An example of the Goldreich authentication tree generated while signing the message 011_2.

As the scheme is based on one-time signatures, no leaf node may be used to sign more than one message. A way to achieve this without introducing a state (e.g., a list of used leaf nodes) is to use the tree with a sufficient height so that the probability of using the same leaf twice becomes negligible. A deterministic alternative is to set the tree height equal to the length of the message (or the hash of it) and use the binary representation of the message (or the hash) itself as the leaf index.

Figure 2 illustrates an authentication tree for the GSS capable of signing three-bit messages with the public key $\mathsf{pk}_G = pk_1$. Only the depicted components, that also make up the signature, are needed to create the signature of the binary message 011_2. The resulting signature is composed of S_{11}, pk_{10}, pk_{11}, S_5, pk_4, pk_5 and S_2. This signature can be verified by verifying the signature S_{11} on the message using the public key pk_{11}. The public key pk_{11} and its neighbor pk_{10} can be authenticated by verifying the signature S_5 using the public key pk_5. This verification is continued until the signature S_1 is verified with pk_G.

Unfortunately, this approach results in large signatures which makes the scheme impractical. For example, instantiating the scheme with the efficient Winternitz one-time signatures to sign 256-bit messages results in GSS signatures larger than 1 MB.

III. Multiple-Time BLT Signature Scheme

In this section, we introduce the multiple-time BLT signature scheme which we build around backdating resistant timestamping and a forward-resistant tag system. We also prove the existential unforgeability under chosen message attack (EUF-CMA) in the multiple use setting.

A. Cryptographic Timestamping

Cryptographic timestamping allows users to prove that some data existed at some moment in the past. Haber and Stornetta introduced the first timestamping which does not rely on trusted third parties [10]. They proposed 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 showed how to increase the efficiency by operating in rounds, where the 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 [11].

In this paper, we will use an idealized model of timestamping. This can be seen as an assumption of the timestamping service being a white-box trusted third party. The alternative would be to formally derive universal composability (UC) of a particular timestamping construction and then apply composition theorem to instantiate the timestamping protocol.

We note that universally composable timestamping constructions exist [12], however, their design already relies on digital signatures. This is not a problem for the BLT scheme, since signatures are generated by the timestamping service and its clients only need to verify these signatures to be sure of the origin of the timestamps. This is similar to timestamping the signatures of other schemes. However, in our case the unforgeability of BLT depends on unforgeability of the signatures used by the timestamping service.

Here, we restate that the ultimate goal of the BLT scheme is to provide server-assisted secure multiple-time hash-based signatures with efficient and stateless client-side computations.

We 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 `Repo` describes the interface of timestamping services. A service allows a user to timestamp (`put`) values of type `data` and returns associated timestamps:

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

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

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

```
module Ts : Repo = {
  var i, t : time
  var m : (time, data) fmap

  proc init() = {
    i =$ tdistr;
    t =  i;
    m =  empty;
  }
  proc clock() = {
    return t;
  }
  proc put(d : data) = {
    t = t + 1;
    m = m.[t ← d];
    return t;
  }
  proc check(t : time, d : data) = {
    return m.[t] = Some d;
  }
}.
```

The inner state of module `Ts` consists of the initial time `i` (sampled from `tdistr`), the current time `t`, and the repository (finite map) `m` which associates data items to time values. Note that "time" advances only when a new value is added to the repository, which models a linear ordering over the timestamped values.

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

```
module type AdvTs(TsO : Repo) = {
  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 `Repo`) and is not allowed to directly access global variables of the module `Ts`.

Backdating Resistance: The essential property of any timestamping service is its resistance against backdating: once a datum `d` is timestamped (that is, associated with some time `x` where $x \leq$ `Ts.t`), this association remains intact after arbitrary computations 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). A Hoare triple means that if the precondition is true before the execution of the program, then the postcondition will be true after the program terminates.

In our example, the precondition and the postcondition coincide ($x \leq$ `Ts.t` $\wedge$ `Ts.r.[x] = d`). Also note the program `A(Ts).main` is abstract in our example.

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

B. Forward-Resistant Tag Systems

In this section, we describe the second ingredient of the multiple-time BLT signature scheme: multiple-time forward-resistant tag systems. Such systems consist of tag generation and verification functions, and a probabilistic key generation module.

```
type pkey, skey, tag.
op tagGen : skey → time → tag.
op tagVer : pkey → time → tag → bool.
declare module K : KeyGenT.
```

The keypair of a tag system also has an expiration date (`EXP`) associated with it. The implementer of a tag system is obliged to prove that the tag verification function agrees with the tag generation function for any valid keypair at a time preceding the expiration date (`tagSnd`).

```
op EXP : time.

axiom tagSnd : forall pk sk t,
  hoare [ K.keyGen : true ⟹ res = (pk, sk)
    => t ≤ EXP => tagVer pk t (tagGen sk t) = true
       ].
```

Next, we introduce the multiple-time forward-resistance of tag systems which is the main security property and also illustrates the essential difference between signature schemes and tag systems.

Multiple-Time Forward-Resistance: The main motivation behind forward-resistance is to define a lightweight security property which can be used to derive secure digital signatures when combined with backdating resistant timestamping. A tag system is multiple-time forward resistant (FR) if adversaries cannot generate a valid tag for any time `t` given that they only observed tags for times prior to `t`.

Clearly, any existentially unforgeable multiple-time signature scheme can also be treated as multiple-time forward resistant tag system where the messages are time values. However, observe the absence of the "backward-resistance" requirement in the description above. More precisely, if adversary observed a tag `tg` for time `t` then it is permitted that the tags for times prior to `t` could be derivable from tag `tg`. This makes the notion of the tag system weaker and suggests that there could be constructions of tag systems that are computationally more efficient than signature schemes. This observation is a key point for the entire family of BLT schemes: *the intrinsic combination of computationally efficient tag systems with timestamping suggests efficient constructions of digital signatures*.

To describe FR formally we need to specify the interface of 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 keypair and provides tag generation and logging functionality.

```
module type TagOracleT = {
  proc * init(pk : pkey, sk : skey) : unit
  proc oracleTagGen(t : time)       : tag
  proc oracleLog()                  : time list
}.
```

The module `TagOracle` (see Appendix B) is the standard implementation of the `TagOracleT` interface. The `oracleTagGen(t)` procedure provides the tag generation functionality. The global state of `TagOracle` consists of a public-secret keypair and the variable `log` which keeps the history of user queries.

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 tags associated with some times of their choosing and must produce a valid tag for some later time.

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

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

```
module GameFR(A : AdvFR, T : TagOracleT,
    K : KeyGenT) = {
  module A = A(T)
  var t, tg
  proc main() : bool = {
    var pk, sk, log;
    (pk, sk) = K.keyGen();
    T.init(pk, sk);
    (tg, t) = A.forge(pk);
    log = T.oracleLog();
    return tagVer pk t tg
                ∧ max (0 :: log) < t ≤ EXP;
  }
}.
```

Note that forward resistance of prior constructions was analyzed only in one-time setting. We will introduce the first multiple-time forward-resistant tag system in Sect. IV.

C. Multiple-Time BLT Signature Scheme

In this section, we implement the BLT signature scheme parameterized by a timestamping service and a multiple-time forward resistant tag system. We start by giving an intuitive overview of the main phases.

a) Initialization: Generate a public-private keypair `(pk, sk)` for the tag system.

b) Signing: To sign a message `m`:

1) Query the time `t` of the timestamping service.
2) Use the private key `sk` to generate a tag `tg` for the next time (`t+1`).
3) Bind the tag with the message by timestamping the tuple `(m, tg)`.
4) Output `(tg, t+1)` as the signature on `m`.

c) 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`.

Next, we implement the scheme as the `BLTO` module parameterized by a timestamping service and a tagging oracle. The signing oracle keeps the log of all signed messages and we say that a message is fresh if it is not in the log.

```
module BLTO(R : Repo, T : TagOracleT) = {
  var log : msg list
  proc init(pk : pkey, sk : skey) : unit = {
    R.init();
    T.init(pk, sk);
    log = [];
  }
  proc sign(m : msg) : time * tag = {
    var t, tg;
    t = R.clock();
    tg = T.tag(t+1);
    R.put(m, tg);
    log = m :: log;
    return (t, tg);
  }
  proc fresh(m : msg) : bool = {
    return !(m \in log);
  }
}.
```

D. Security Analysis

The security properties of the BLT signature scheme depend on the kind of access the adversary has to the timestamping service and on the type of values stored in its repository [3].

In this section we analyze the multiple-time BLT scheme in the model where the adversary has read-write access to the timestamping service whose repository contains plain message-tag pairs. In other words, adversary has full access to the timestamping repository.

As usual, we start the analysis by defining the types of oracles and adversaries:

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

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

We formalize the existential unforgeability under the chosen message attack (EUF-CMA) of multiple-time BLT as a parameterized module `GameBLT`. The adversary receives a public key and has access to a signing oracle and a timestamping service. To win the game, the adversary must produce a valid signature on a fresh message. The main difference of this setting from `GameOTS` (Sec. II-A) is that we consider a multiple-time signature scheme and thus the adversary can use the signing oracle multiple times.

```
module GameBLT(A : AdvBLT, R : Repo,
    BLT : BLTOracleT, K : KeyGenT) = {
  module A = A(R, BLT)
  var m, tg, t
  proc main() : bool = {
    var pk, sk, fresh, timestamped;
    (pk, sk) = K.keyGen();
    BLT.init(pk, sk);
    (m, tg, t) = A.forge(pk);
    timestamped = R.check(t, (m, tg));
    fresh = BLT.fresh(m);
    return t ≤ EXP ∧ tagVer pk t tg
                   ∧ fresh ∧ timestamped;
  }
}.
```

Theorem 1: If the backdating resistant timestamping repository holds plain message-tag pairs then the probability of a read-write adversary performing a successful BLT forgery is bounded by the probability of breaking the forward-resistance of a tag system.

Proof: The main challenge in transforming BLT adversary into adversary who breaks forward-resistance is that the adversary might continue using the signing oracle after timestamping a successful BLT forgery and this will produce tags for times later than the time associated with the BLT forgery. This in turn will prevent us from using the forgery tag as evidence of the adversary being able to break the forward resistance. Therefore, we need to switch the signing oracle off immediately after a forgery is produced. The simplest way to achieve this is to implement a wrapper around the timestamping and tagging oracles, so that when the adversary writes a valid tag associated with the actual time, the oracles update their states so that no further requests will be answered and, hence, no new tags will be generated.

```
module TsWrap(Ts : Repo) : Repo = {
  var t : time
  var f : bool
  var pk : pkey
  proc put(d : msg * tag) = {
    var t;
    if (!f) {
      t = Ts.put(d);
      f = f ∨ tagVer pk t d.'2;
    }
    return t;
  }
  (*** clock() and check() deletegate to Ts ***)
}.

module BLTWrap(Ts : Repo, Tag : TagOracleT) = {
  module BLT = BLTO(Ts, Tag);
  proc sign(m : msg) = {
    var t, tg;
    if (!TsWrap.f) {
       (t, tg) = BLT.sign(m);
     }
     return (t, tg);
  }
  (*** fresh() delegates to BLTO ***)
}
```

Note that the adversary might decide to change the output once the oracles stop working. Therefore, we completely ignore the output of the adversary and return the tag which actually triggered the stop flag:

```
module BLT2FR(A : AdvBLT, T : TagOracleT) = {
```

```
  module A = A(TsWrap(Ts), BLTWrap(Ts, T))
  proc forge(pk : pkey) = {
    var r;
    Ts.init();
    TsWrap.pk = pk;
    A.forge(pk);
    r = oget Ts.m.[TsWrap.t];
    return (TsWrap.t, r.'2);
  }
}.
```

We use the `BLT2FR` reduction to prove that if `A` breaks the BLT scheme then `BLT2FR(A)` breaks the forward resistance. The probabilistic interpretation is the following:

```
lemma blt2frUB &m :
  Pr[ GameBLT(A, Ts, BLTO(Ts, TagOracle), K).main()
      @ &m : res ]
≤ Pr[ GameFR(BLT2FR(A), TagOracle, K).main()
      @ &m : res ].
```

■

IV. Multiple-Time Tag System

In previous section we constructed a secure multiple-time BLT signature scheme from timestamping and a multiple-time forward resistant tag system. In this section, we present a multiple-time forward resistant tag system.

A. Construction

The main idea of our construction is to use "authenticated" paths similar to Merkle and Goldreich signatures (see Sec. II-C and Sec. II-D) as tags. More specifically, we index nodes of a binary tree with time values in a top to down, left to right fashion and let every node hold an OTS public-private keypair.

The tag for time `t` is the signature S_t on the public keys of the child nodes of the `t`-th node (verifiable with the public key of the `t`-th node) and an authenticated chain of sibling nodes from the root of the tree to the `t`-th node (Fig. 2). Every node on the path authenticates the next node by signing its public key (along with the public key of its sibling). The public key of the root node is the public key of the tag system and needs no authentication.

Note that having the tag for time `t`, one can easily derive tags for times associated with its parent nodes (just by dropping last nodes in the path). This is not a problem for the forward-resistance of the tag system, as the parents are associated with prior times (i.e., smaller time values).

Since our construction is based on one-time signatures, we assume some well-defined types `skeyOTS`, `pkeyOTS`, `sig` and the associated key generation, signing, and verification algorithms (see Sec. II-A).

The secret key of a tag system consists of all secret keys associated with the nodes of the tree (represented as a function which associates the OTS secret key with the node of a tree).

```
type skey = time → skeyOTS option.
type pkey = pkeyOTS.
```

We implement tags as lists of tuples where each tuple is associated with a node in the tree and holds its label and a one-time signature of that label. A label of a node is a triple of a node index (time value) and the public keys of its children.

```
type tag = (label * sig) list.
type label = time * pkey * pkey.
```

a) Tag Generation: The indices of parents of the `t`-th node are computed by function `binRec`. The function `lbl` constructs a label associated with the `t`-th node which contains the public keys of its children.

```
op binRec (t : time) : time list =
  with i = 1   => 1 :: []
  with i = j+1 => binRec (i 'div' 2) ++ (i :: []).

op lbl (skf : skey, t : time) : label =
  (t, sk2pk (oget (skf 2*t)),
      sk2pk (oget (skf 2*t+1))).
```

Here, `[]` is the empty list, `++` denotes list concatenation, and `::` is the list prepend constructor.

To construct the `t`-th tag we produce an authenticated path from the root of the tree to the `t`-th node where each element of the path holds signed public keys of its children.

```
op tagGenAux (skf : skey, ts : time list) : tag =
  with ts = []       => []
  with ts = t :: ts' => nd :: tagGenAux skf ts'
    where nd = (lb, sg), lb = lbl skf t,
      sg = otsSig (oget (skf t)) (lbl skf t)).

op tagGen (skf : skey, t : time) : tag =
  t ≤ 0 ? [] : tagGenAux skf (binRec t).
```

b) Verification: To verify a tag generated for `t`-th node we check that the path (tag) starts from the root (node with index 1), ends with the node with index `t`, the signature of the message of the first node of a tag verifies with the public key of a tag system, and the label of each node is signed by the previous node in the path.

```
op pathVer (tg : tag) : bool =
  with tg = []          => true
  with tg = x :: []     => true
  with tg = x' :: x :: xs => pathVer (x :: xs) ∧
    otsVer (x.'1.'1 mod 2 = 0 ? x'.'1.'2 : x'.'1.'3)
           x.'1 x.'2.

op tagVer (pk : pkey, t : time, tg : tag) : bool =
  tg != [] ∧
  (head tg).'1.'1 = 1 ∧
  (last tg).'1.'1 = t ∧
  otsVer pk (head tg).'1 (head tg).'2 ∧
  pathVer tg.
```

c) Correctness: If a secret key of a tag system consists of valid OTS secret keys then the tag verification function agrees with the tag generation function.

```
lemma multTagsCorrect skf pk :
   (forall j, j ≤ EXP
         => exists r, Some (otsKey r) = skf j)
=> forall t, t ≤ EXP ∧ pk = sk2pk (oget (skf 1))
=> tagVer pk t (tagGen skf t) = true.
```

The correctness is proved by induction on time.

B. Multiple-Time Forward Resistance

Let us start by analyzing security of the case when we explicitly generate all independent OTS keypairs in the initialization phase.

```
module TagKeys : KeyGenT = {
  proc keyGen() = {
    var skf, t, sk, rs;
    skf = fun _ => None;
    t = 1;
    while (t ≤ 2*EXP + 1) {
      rs  =$ dR;
      sk  = otsKey rs;
      skf = fun x => x = t ? Some sk : skf x;
      t = t + 1;
    }
    return (sk2pk (oget (skf 1)), skf);
  }
}.
```

For the uniformity of the tag generation, we ask for `2*EXP+1` keypairs to produce `EXP` tags.

Theorem 2: The probability of breaking the forward resistance of multiple-time tag system with EXP independent secret keys is bounded by EXP times the probability of breaking the existential unforgeability of the one-time signature scheme.

Proof: We say that `sk` is the `i`-th canonical secret key if `skf` is the secret key of the tag system and `Some sk = skf i`, and that `pk` is the `i`-th canonical public key if `pk = sk2pk sk`. If `pkf` is the table of canonical public keys and `tg` is a successfully forged tag then the function `parseTag` returns a label and a signature from `tg` so that the returned signature verifies with its canonical public key and none of the succeeding signatures in the tag verify with their respective canonical keys.

```
op parseTag (pkf : time → pkey option,
    tg : tag) : label * sig =
  with tg = []      => witness
  with tg = x :: xs => if all (fun e =>
      !otsVer (pkf e.'1.'1) e.'1 e.'2) xs
    then x else parseTag pkf xs.
```

We claim that if adversary wins the FR game by producing a tag `tg` for time `t` then `parseTag pkf tg` finds an OTS necessarily forged by the adversary. To see that, we must analyze two cases.

If `parseTag` returns the last signature from `tg` then this signature was necessarily forged. Indeed, the signature verifies with the `t`-th canonical public key. However, if the adversary was successful in `GameFR`, then a tag with index greater than or equal to `t` was never generated by the oracle.

If the pair (l, s) returned by `parseTag` is not the last in `tg` then it is also necessarily forged. Indeed, by definition, the pair which follows (l, s) verifies with the key stored in the label l, but not with the canonical public key. Therefore, the label l must have been signed by the adversary since the tagging oracle only produces and signs labels with the canonical keys.

Let `A` be an adversary that breaks the forward-resistance of the tag system, by computing a tag `tg` for time `t`. Our goal is to use `A` to construct an adversary that breaks the existential unforgeability of the underlying one-time signature scheme. Recall the setting of the OTS game: the module `GameOTS` generates a public-secret keypair and the adversary is given a public key and a one-time oracle; to win the game, the adversary must forge a signature for a fresh message which verifies with the public key provided by the game.

Let us additionally assume that `A` forges the OTS for the `p`-th node in the tag `tg`. Then, to get a successful OTS adversary, we simply need to simulate the environment of the FR game and supply to `A` a tagging oracle `TagOracle'` which initializes the environment of FR game in the beginning and then behaves exactly as the standard `TagOracle` except that the `p`-th node is associated with the public key provided by the OTS game and the OTS of the `p`-th node is generated by the oracle provided by the game. Since we cannot predict the node for which `A` is going to forge the signature, the best strategy is to sample the position `p` uniformly from 1 to `EXP`.

```
module TagOracle'(O : OTSOracleT) = {
  var pkf, p, pkTag, skTag
  proc init(pkOTS : pkeyOTS) = {
    p =$ uniform 1 EXP;
    (pkTag, skTag) = TagKeys.keyGen();
    pkf = fun t => if t = p then pkOTS
      else sk2pk (oget (skTag t));
  }
  proc tag(t : time) = {
    var tg = [];
    var ys = binRec t;
    while (ys <> []) {
      y = head ys;
      sig = if y = p then O.sign(lbl pkf y)
        else otsSig (oget (skTag y)) (lbl pkf y);
      xs = xs ++ [y];
      ys = behead ys;
    }
    return tg;
  }
}.
```

The module `FR2OTS` transforms an FR adversary into an OTS adversary by initializing `TagOracle'` with the public key provided by the OTS game. The final OTS forgery is extracted by the function `parseTag` from the tag returned by the adversary.

```
module FR2OTS(A : AdvFR, O : OTSOracleT) = {
  module A = A(TagOracle'(O))
  proc forge(pkOTS : pkeyOTS) : msg * sig = {
    var tg, t;
    TagOracle'.init(pkOTS);
    (tg, t) = A.forge(oget (TagOracle'.pkf 1));
    return (parseTag TagOracle'.pkf tg);
  }
}.
```

If `TagKeys` and `OTSKeys` key generators sample individual keypairs from the same distribution then with probability `1/EXP` the choice of node `p` will match the node whose signature will be forged by `A`; therefore, we can conclude that `A` is no more than `EXP` times successful in forging tags than `FR2OTS(A)` is successful in forging one-time signatures.

```
lemma fr2otsUB &m :
  Pr[GameFR(TagOracle, A, TagKeys).main()
```

```
    @ &m : res]
≤ Pr[GameOTS(OTSOracle, FR2OTS(A), OTSKeys).main()
    @ &m : res] * EXP.
```

■

C. Efficient PRF Based Key Generation

In the previous section, we defined the key generator `TagKeys` which used the distribution `rDistr` as the source of randomness which then was used to generate OTS keys. The downside of this approach is that all the keys are produced at the initialization stage which can be prohibitively slow.

To avoid the expensive initialization phase we could use the strategy proposed by Merkle (see Sec. II-C)—to generate each OTS keypair when it is actually needed. This approach would solve the problem of expensive initialization, but would make the tag system stateful. More specifically, to produce a new tag for time `t`, the user would have to generate a fresh OTS keypair and save it for later signing the keys of its children. Otherwise, the user would risk reusing the OTS keypair of `t`'s parent to sign another keypair which will compromise the security of the tag system since an OTS keypair can be used to sign only one message.

In this section, our goal is to implement and prove security of a stateless tag system with cheap initialization. To achieve this, we formalize the standard cryptographic approach based on pseudorandom functions (PRF). Informally, a PRF is a pure function that takes a key (source of randomness) and returns a function indistinguishable from a truly randomly sampled function for any "reasonably" efficient adversary. We formalize this idea in the next few steps.

A PRF module must be initialized and could be executed on the input values:

```
module type PRFOracleT = {
    proc init() : unit
    proc exec (t : time) : r
}.
```

A real cryptographically constructed PRF is a pure function `F`. The first argument of `F` is a key (seed) which acts as a source of true randomness. Then, `F rs` is a function which maps input values to pseudorandom values. The module `RealPRF` generates the PRF seed `rs` at initialization and delegates the requests to `F rs` when executed.

```
op F : r → time → r.

module RealPRF : PRFOracleT = {
  var rs : r
  proc init() : unit = {
    rs =$ rDistr;
  }
  proc exec(t : time) : r = {
    return F rs t;
  }
}.
```

The module `IdealPRF` models an ideal random function using the lazy sampling technique. It keeps a mapping (empty in the beginning) of input values to truly random values which are sampled when user provides a new input:

```
module IdealPRF : PRFOracleT = {
  var m : (time, r) fmap
  proc init() : unit = {
    m = empty;
  }
  proc exec(t : time) : r = {
    if (x \notin m)
      m.[x] =$ rDistr;
    return (oget m.[x]);
  }
}.
```

The PRF adversaries are given a PRF module, and must decide whether they are interacting with the real or the ideal PRF.

```
module type Distinguisher(F : PRFOracleT) = {
  proc distinguish() : bool {F.exec}
}

module IND(PRF : PRFOracleT, D : Distinguisher) = {
  module D = D(PRF)
  proc main() : bool = {
    var b;
    PRF.init();
    b = D.distinguish();
    return b;
  }
}.
```

We say that the PRF `F` underlying `RealPRF` is secure if the following relation holds for some negligible value `ubPRF`:

```
axiom prfSec : forall (D : Distringuisher),
 | Pr[ IND(RealPRF,  D).main() @ &m : res ]
 - Pr[ IND(IdealPRF, D).main() @ &m : res ] |
 ≤ ubPRF.
```

Armed with a secure PRF `F` we define an efficient key generation module `TagKeysEFF`:

```
module TagKeysEFF : KeyGenT = {
  proc keyGen() = {
    var rs;
    rs =$ dR;
    return (sk2pk (otsKey (F rs 1)),
            fun t => Some (otsKey (F rs t)));
  }
}.
```

In this scenario the "true" secret key of the tag system is the seed `rs`. Then the keypair associated with the node (time) `t` is the result of executing the `otsKey` algorithm on the pseudorandom output of `F rs t`.

Theorem 3: The probability of breaking the forward resistance of the multiple-time tag system with the PRF based key generation is bounded by the sum of the indistinguishability of the PRF and `EXP` times the probability of breaking the existential unforgeability of the one-time signature scheme.

Proof: Let us fix `A` as the FR-game adversary. We start by defining an "inefficient" key generation module which is parameterized by a PRF:

```
module TagKeysPRF(F : PRFOracleT) : KeyGenT = {
  proc keyGen() = {
    var skf, t, sk, rs;
    skf = fun _ => None;
    t = 1;
    while (t ≤ 2*EXP+1) {
      rs = F.exec(i);
```

```
      skf = fun x =>
        x = t ? Some (otsKey rs) : skf x;
      t = t + 1;
    }
    return (sk2pk (oget (skf 1)), skf);
  }
}.
```

Next, we use this key generation module to implement a PRF distinguisher where adversary A plays the FR game and the distinguisher returns a boolean flag which indicates whether A was successful.

```
module D(F : PRFOracleT) = {
  module Game = GameFR(TagOracle, A, TagKeysPRF(F))
  proc distinguish() = {
    return Game.main();
  }
}.
```

Since we assumed that F is a secure PRF then the ability of D to distinguish RealPRF from IdealPRF is bounded:

```
| Pr[ IND(RealPRF, D).main() @ &m : res ]
- Pr[ IND(IdealPRF, D).main() @ &m : res ] |
≤ ubPRF.
```

In other words, if F is a secure PRF then A cannot be much more (or less) successful against PRF-generated keys compared to the truly randomly generated keys.

Moreover, from the definition of D it follows that the probability of winning the IND(RealPRF) game by D equals the probability of winning the FR-game with the efficient PRF-based key generation TagKeysEFF by A. Also, the probability of winning the IND(IdealPRF) game by D equals the probability of A winning the FR-game with the inefficient independently sampled keys:

```
Pr[ IND(RealPRF, D).main() @ &m : res ] =
Pr[ GameFR(TagOracle, A, TagKeysEFF).main()
      @ &m : res ].

Pr[ IND(IdealPRF, D).main() @ &m : res ] =
Pr[ GameFR(TagOracle, A, TagKeys).main()
      @ &m : res ].
```

The equations above bound the probability of A winning the FR-game with efficient PRF-based keys:

```
Pr[ GameFR(TagOracle, A, TagKeysEFF).main()
      @ &m : res ] ≤
Pr[ GameFR(TagOracle, A, TagKeys).main()
      @ &m : res ] + ubPRF.
```

After combining the above equation with the upper bound derived in the previous section, we can conclude the proof:

```
Pr[ GameFR(TagOracle, A, TagKeysEFF).main()
      @ &m : res ] ≤
Pr[ GameOTS(TagOracle, FR2OTS(A), OTSKeys).main()
      @ &m : res] * EXP + ubPRF.
```

■

D. Efficiency and Optimizations

To estimate the efficiency of the tag system we assume POSIX time with one second granularity. This means the tree height is $h = 31$ for any practical time frame. The tag size for this scheme is $|\sigma_{tag}| \approx h|\sigma_{auth}|$ where $|\sigma_{auth}|$ is the size required to authenticate the child node public keys (e.g. the one-time signature size $|\sigma_{OTS}|$ in the Goldreich construction). This means the size of the tag can be reduced by either making the tree height smaller or reducing the size of the authentication. The signing process generates $2h + 1$ keypairs and performs h signing operations. The following optimizations are a compromise between signature size and computational effort.

The nodes in the tag system do not have to be OTS keypairs. Instead, the hash values of the public keys can be used to authenticate them. The tag generated by this type of node is the pre-image of the hash value (the public keys of the the child nodes). This is similar to the hyper-tree construction proposed in [13] that can be used to divide the tree into layers with height l, reducing the size of the tree to $|\sigma_{tag}| \approx \lceil h/l \rceil |\sigma_{OTS}| + h$ hash values. This optimization reduces the number of signing operations to h/l but the number of keypairs generated per signing increases to $(h/l)2^l + 1$.

An optimization unique to this construct is to turn the binary tree into a ternary tree as in Sec II-C, but use the hash-sequence based BLT tags [14] with a relatively short length of 2^m in the additional branch. This reduces the tree height by m and adds additional 2^m hash function computations per node in signing, but increases the verification by only up to m hash function calls as only the last node in the authentication chain can use a value from this branch.

Combining these optimzations leads to a tag size $\sigma_{tag} \approx \lceil (h-m)/l \rceil |\sigma_{OTS}| + (h-m)$ hash values. For example, by choosing $h = 30$, $m = 10$, $l = 4$ and using Winternitz-style one-time signatures with a 256-bit hash function, the resulting tag size is about $24kB$ and the number of keypairs generated increases from 61 to 81.

V. Related Work

The most related and also motivational work for our paper are the results by Buldas et al. [1], [2]. They formulated the key insight that the construction of digital signatures could be simplified if a backdating resistant timestamping service is combined with a forward resistant tag system. The original constructions showed promising efficiency in terms of the size of the resulting signatures, but suffered either from slow initialization phase or stateful user-side computations. In this work, we proposed a single solution to both problems and proved its correctness and security in the multiple use setting.

In [3], Buldas et al. used EasyCrypt to formally derive security of one-time BLT signatures from security of one-time forward-resistant tag systems. The authors left open particular constructions of tag systems, but analyzed the security in the context of different types of timestamping repositories. In this work, we use the simplest type of timestamping server, but focus on the security and particular constructions of multiple-time tag systems.

SPHINCS [13] is a family of stateless signature schemes capable of signing an indefinite number of messages. Although

SPHINCS is based on the idea of Goldreich, it produces significantly smaller signatures (e.g. 41 kB for SPHINCS-256). This is achieved by reducing the tree height combined with the use of Merkle trees. To mitigate the risk of reusing a one-time key, the keypairs in the leaf nodes are from a few-time sheme [15] and are selected for use on a pseudorandom schedule. Our work differs in three ways that allows for an overall smaller size for a pair of a signature and a timestamp. First, the timestamp is an intrinsic part of the signature. Second, it chooses the signing key based on the binary representation of the time value which can only grow in time, rather than using a pseudorandom value. For example, the overall tree height can be reduced to 31 for POSIX time values for the practical future. Third, as the ever-increasing time value assures no key can be used after it has expired, we do not rely on few-time signatures that are larger than one-time signatures.

Stateful signature schemes that use index based keypair selection for signing, such as XMSS [16] and the Merkle infinite signatures (Sec. II-C), can be converted to use time for key selection by replacing the local state with the time value from a timestamping service. In such constructions the signatures and timestamps unnecessarily duplicate the "backward-resistance" (Sec. III-B) property. More importantly, this introduces a new issue: the signer must make sure no two messages are signed at the same time; otherwise a signing OTS keypair will be used more than once.

VI. Discussion

We found that in EasyCrypt it is relatively simple to define protocols and interaction among different actors of a protocol (i.e., signing oracles, timestamping service, adversaries). Also, the module-level (sub)typing system guided the implementation and forced us to specify and satisfy definitions precisely.

It was easy to develop a functional implementation of a multiple-time tag construction and then prove the correctness property. Moreover, we could specify the high-level proof strategy and delegate the low-level details to the SMT solvers. This allowed our proofs of correctness to remain intact even when the construction changed.

The most complicated parts of our formalization are statements about probabilities, and equality of distributions. The proofs of these statements are done using (relational) probabilistic Hoare logic and depend on the operational semantics of programs. For these statements the automation support is low, and it is hard to define high-level proof strategies which tolerate (small) changes in the definitions.

VII. Conclusions and Future Work

We have shown that multiple-time forward resistant tag systems induce secure multiple-time time BLT signature schemes. Moreover, we implemented a particular construction of a forward-resistant multiple-time tag system based on one-time signatures.

In the future we plan to investigate more realistic models of timestamping in the context of BLT signatures. For example, in this paper we assume that there is no "lag" in time between the signer and the timestamping service. It would be more realistic to assume that if current time is `t`, then the timestamping query will reach the service by time `t + lag`, where `lag` comes from some distribution.

Another important aspect is whether the timestamping holds a single element associated with every time slot, or a set where the elements may come from different users of the timestamping service. Moreover, we can strengthen the adversarial model by allowing the adversary to "observe" the elements before they reach the timestamping repository.

Finally, the optimizations that reduce the signature size and signing time of the scheme need further analysis. This would allow a more detailed comparison with existing hash-based constructions such as the SPHINCS family (Sec. V).

Acknowledgment

This research was partly supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001). Also, all three authors of this paper are currently employed by Guardtime, a company offering timestamping services.

References

[1] A. Buldas, R. Laanoja, and A. Truu, "A server-assisted hash-based signature scheme," in *NordSec 2017, Proceedings*, ser. LNCS, vol. 10674. Springer, 2017, pp. 3–17.

[2] A. Buldas, D. Firsov, R. Laanoja, H. Lakk, and A. Truu, "A new approach to constructing digital signature schemes," in *IWSEC 2019, Proceedings*, ser. LNCS, vol. 11689. Springer, 2019, pp. 363–373.

[3] A. Buldas, D. Firsov, R. Laanoja, and A. Truu, "Verified security of BLT signature scheme," in *ACM SIGPLAN CPP 2020, Proceedings*. ACM, 2020, pp. 244–257.

[4] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theor.*, vol. 22, no. 6, pp. 644–654, Nov 1976.

[5] R. C. Merkle, "Secrecy, authentication and public key systems," Ph.D. dissertation, Stanford University, 1979.

[6] ——, "Protocols for public key cryptosystems," in *IEEE Symposium on Security and Privacy*, 1980, pp. 122–134.

[7] O. Goldreich, *Foundations of Cryptography*. Cambridge University Press, 2004, vol. 2.

[8] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "EasyCrypt: A tutorial," in *Foundations of Security Analysis and Design VII*. Springer, 2013, pp. 146–166.

[9] R. C. Merkle, "A digital signature based on a conventional encryption function," in *CRYPTO'87, Proceedings*, ser. LNCS, vol. 293. Springer, 1987, pp. 369–378.

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

[11] J. Benaloh and M. de Mare, "Efficient broadcast time-stamping," Clarkson University, Tech. Rep., 1991.

[12] T. Matsuo and S. Matsuo, "On universal composable security of time-stamping protocols," *IWAP*, vol. 2005, pp. 169–181, 2005.

[13] D. J. Bernstein, D. Hopwood, A. Hülsing, T. Lange, R. Niederhagen, L. Papachristodoulou, M. Schneider, P. Schwabe, and Z. Wilcox-O'Hearn, "SPHINCS: Practical stateless hash-based signatures," in *EUROCRYPT 2015, Proceedings, Part I*, ser. LNCS, vol. 9056. Springer, 2015, pp. 368–397.

[14] A. Buldas, R. Laanoja, and A. Truu, "Efficient quantum-immune keyless signatures with identity," Cryptology ePrint Archive, Report 2014/321, 2014, https://eprint.iacr.org/2014/321.

[15] L. Reyzin and N. Reyzin, "Better than BiBa: Short one-time signatures with fast signing and verifying," in *ACISP 2002, Proceedings*, ser. LNCS, vol. 2384. Springer, 2002, pp. 144–153.

[16] J. A. Buchmann, E. Dahmen, and A. Hülsing, "XMSS—A practical forward secure signature scheme based on minimal security assumptions," in *PQCrypto 2011, Proceedings*, ser. LNCS, vol. 7071. Springer, 2011, pp. 117–129.

APPENDIX A
ONE-TIME SIGNATURE ORACLE

```
module OTSO : OTSOracleT = {
  var qs   : msg option
  var used : bool
  var pk : pkeyOTS
  var sk : skeyOTS

  proc init(pk : pkeyOTS, sk : skeyOTS) : unit = {
    OTSOracle.pk = pk;
    OTSOracle.sk = sk;
    qs   = None;
    used = false;
  }

  proc sign(m : msg) : sig option = {
    var r, q;
    if (!used) {
      qs = Some m;
      q = otsSig sk m;
      r = Some q;
    } else {
      r = None;
    }
    used = true;
    return r;
  }

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

APPENDIX B
MULTIPLE-TIME TAGGING ORACLE

```
module TagOracle  : TagOracleT = {

  var lt : Time list
  var pk : pkey
  var sk : skey

  proc init(pk : pkey, sk : skey) = {
    TagOracle.pk = pk;
    TagOracle.sk = sk;
    lt = [];

  }


  proc tag(t : Time) = {
    var tg;
    lt = t :: lt;
    tg = tagGenFun sk t;

    return tg;
  }


  proc getTagLog() :  Time list = {
    return lt;
  }

}.
```