NEXt generation Techno-social Legal Encryption Access and Privacy nextleap.eu

Grant No. 688722. Project started 2016-01-01. Duration 36 months.

DELIVERABLE D4.1

FORMAL MODELLING OF FEDERATED IDENTITY

Natalia Kulatova (INRIA), Harry Halpin (INRIA)

Beneficiaries:	**INRIA** (lead), IMDEA, UCL
Workpackage:	**D4.1 Formal Modelling of Federated Identity**
Description:	Formal definitions of decentralization, privacy, security, and anonymity properties will be developed and proven using automatic proof-proving on the federated identity protocol. Results will be validated against open-source code.

Version:	1.0
Nature:	Report (R)
Dissemination level:	Public (PU)
Date:	2016-08-31

Project co-funded by the European Commission within the Horizon 2020 Programme.

Contents

1 Introduction

In this deliverable, we outline formal models of decentralization, privacy, security, and anonymity for *ClaimChain*, a federated identity system for decentralized key management conceived in D2.2 and completed in D4.2. We present our formal verification techniques using F*, a programming language developed by Inria, and then demonstrate how each of the primary components of ClaimChain can be developed using F*: Skiplists, Merkle Trees, and Verifiable Random Functions (VRFs). Each component is built using F* in order to prove their various properties, and then extracted to CompCert C, a formally verified subset of C. Therefore, code samples in F* are also followed by C code samples. All code has been released as open-source code.[1]

2 Formal Definitions

Note that these definitions are simplified definitions of the formal definitions presented in D4.2. The formal proofs done in D4.2 consist of proving properties via reductions of Claimchain properties to the properties of the underlying primitives manually. These proofs are more sophisticated than can be handled by F*, and so we will only define the properties that F* can prove, i.e. security properties that relate to integrity, the secrecy of key material, and indistinguishability. More complex notions of decentralization and anonymity require changes to the fundamental language F* itself in order to be proven.

2.1 Security

Claimchains provide *integrity* of all claims via the use of a blockchain mechanism based on hash functions. Therefore, for a given block B with one or more claims, any modification of the claim B to B' will alter the hash function H such that $H(B) \neq H(B')$. A ClaimChain is implemented as a list of hashes functions over blocks $H_1 ... H_n$ for $n \geq 1$ where each $H_n(B_n \mid H_{n-1})$ for all n.

2.2 Privacy

Claimchains provide *privacy* by allowing claims to be encrypted. This leads to *claim indistinguishability*, namely that for any two blocks B_1 and B_2 and encryption function E, $E(B_1)$ is computationally indistinguishable from $E(B_2)$.

2.3 Anonymity

Generally it is unknown how to formalize anonymity in a manner that can be proven using automatic proof-proving frameworks such as F*. The informal notion of anonymity is that of *unlinkability*, namely that two or more items of interest cannot be linked together [4]. In terms of the ClaimChain architecture, it should be noted that claims are encrypted to readers. Therefore, one definition of anonymity is *reader unlinkability*, namely that for any given encrypted claim, it cannot be determined which reader the claim was encrypted for by an outside observer (i.e. anyone except the owner of the ClaimChain that encrypted a claim for one or more readers). Thus, for any blocks B encryption function E that requires use of public key K, given two public keys K_1 and K_2 for any two readers, $E(B, K_1)$ is computationally indistinguishable from $E(B, K_2)$.

[1] https://github.com/nextleap-project/verified-claimchain

2.4 Decentralization

As explored in D2.1, decentralization means that there is no single trusted authority in the use of the protocol. In the context of the ClaimChain architecture, this means for any claim, it may be verified using one or more ClaimChains and that, unlike the replicated state of traditional blockchain architectures like Bitcoin or Ethereum, ClaimChains may have different states. Therefore, any given claim C may be recorded in any given block B_C and that for any two ClaimChains n and m given by their respective head imprint hash function using hash function H, it may be the case that $H_n(B_C) \neq H_m(B_C)$.

3 Formal Verification using F*

In general, with formal verification we need to prove the correctness of implemented code with regard to a *specification*. In the F* programming language,the specification is given as F* code itself, and then we can automatically prove the properties that need to be proved

F* (pronounced “F star”) is a typed functional programming language designed for program verification, and a complete tutorial is available online.[2] Although many other languages exist for formal proof-proving, such as the proof assistant Coq,[3], F* was chosen due to several reasons. Languages like Coq and F* are used for protocol and program verification, so they do not create compiled code that can be executed by default. However, OCaml code from F* can executed, allowing verified code to actually be run. Yet, very platforms support OCaml natively. However, OCaml code created by F* can then be translated using Kremlin[4] to C and provably preserve the semantics of the original verification. Therefore, from a subset of F* called Low*, we can translate to CompCert’s C. The final C program is very efficient in terms of performance and preserves all the security guarantees. Therefore, F* can be extracted to C such that the code in C will have the same proofs of correctness and as the source code, allowing the verified cryptographic code to be ran efficiently across multiple platforms and linked to existing software projects, such as those in WP5. As the current version of ClaimChain in D2.2 and D4.2 is built using Python, which has difficulty running on mobile platforms, this could help the adoption of ClaimChain by mobile clients such as K-9 Mail or Signal on Android. F* has already been used in several successful applications for protocol verification, in particular the verified reference implementation of the TLS protocol: miTLS. Code samples and references to F* and C are given using this `code` font if given inline in text.

In order to understand F*, it is important to note that the formal verification is typically done in the type system via the use of *refinement types* in F*. For example, the type `a:nat{a < 10}` is a refinement of the type `nat` (natural numbers), which means that for this given type, all numbers must be less than 10. The subset of the natural numbers is decreased by the predicate. Also, F* allows *effects*, but keeps track of different effects during the computation. The most commonly used effect is `Tot`. This effect is guaranteed (provided the computer has enough resources) to evaluate to a result of the type given by the function without having an effect, and therefore without entering an infinite loop, reading or writing the program’s state, throwing exceptions, performing input or output, and any other source of error. Of course, there are many different types of effects that allow more variation. In particular, there are:

- `Dv`: the effect of a computation that may diverge;
- `ST`: the effect of a computation that may diverge, read, write or allocate new references in the heap;
- `Exn`: the effect of a computation that may diverge or raise an exception.

In general, the verification for the correctness of implementation of data structures and protocols are proofs of correctness of implementation, where the correctness of implementation guarantees the efficiency of various operations. The above-mentioned effect types show that the implementation is behaving as specified, and so

[2] https://fstar-lang.org/
[3] https://coq.inria.fr/
[4] https://github.com/FStarLang/kremlin

there are no unexpected effects, such as memory leaks and out of bound pointers, that can then in turn cause unintended effects, including errors in operation and security flaws.

Furthermore, for cryptographic protocols we want to prove formally-defined security properties. For security protocols, proofs of security of a protocol are done using *reductions* to the underlying primitives. Thus, for a simple data structure and protocol involving preserving integrity using the blockchain, the security of the blockchain can be proven to be equivalent to the security of the underlying hash function (as given in the definition of security in Section 2.1.) Therefore, we assume the underlying primitive is secure and has the correct properties, and can then verify using proofs that the security properties of the protocol can be reduced to those of the primitives employed.

For F*, the formal specification is given by the F* code itself. Therefore, for the data structures underlying ClaimChain, we have present the F* code of the specification composed of a data structure and functions on that data structure. For didactic purposes, for various algorithms we may present the underlying F* code that implements functions if the particular function may have multiple instantiations, in order to show which particular function with what optimization and efficiency properties was implemented. As most programmers will use the C bindings rather than the F* code, so sample C bindings have been attached to the F* code.[5]

4 Sets and Lists

In terms of specification, the mathematical structure is a *set*, where a set is defined as zero or more elements s such that $s \in S$. The set contains the following operations in a terms of a API given an element e:

- Inserting of a new element: *insert(S,e)*
- Removing an existing element: *remove(S,e)*
- Searching for an existing element: *search(S,e)*

A set may also have an index and so be a *list* via an ordering function (f). The ordering function can be given by an ordered set of integers I where for $i \in I$ then $(i-1) < i < i+1$. This means the number of operations is increased so that we can in addition to set operations, add:

- Search for an element by index: *search(S,i)*
- Removing an existing element found by index: *remove(S,i)*
- Splitting the list at index: *split(S,i)*

In terms of implementations, sets and lists may be implemented in many different ways with effect on both the efficiency and security properties of the final implementation. For example, a set may be implemented by a simple list, or for efficiency it may be implemented by a binary tree. However, we are going to implement lists as *skiplists*, given in Section 5. Sets for proof of inclusion are implemented via *Merkle trees*, given in Section 6.

5 Skiplist

A skiplist is a data structure for an ordered list that is used to efficiently store and retrieve data with greater efficiency than a linked list and with less expensive re-ordering than a balanced tree. [5]. A skiplist does this by having an index that can “skip” ahead rather than linearly iterate through the list.

[5]Note that the C bindings may change. The latest version of both the F* code and the C code is available at `https://github.com/nextleap-project/verified-claimchain`

5.1 Formal Definition

A skiplist is an ordered list S of cardinality n with two sets of indices I and J. The first index I is of length n and J is a secondary index of length $m < n$.

5.2 Skiplist Specification

Our specification of a skiplist in F* consists of two elements: A structure to store the data, and then a second structure that is an array of indexes that can store the indexes on different levels (this is contrast to a list that has only a single array of indices).

```
type skipList
(a:eqtype) (f:(a->a->Tot bool)) =
        | Mk:
    values: seq(a){sorted f values}->
    indexes : seq(non_empty_list nat)
                {Seq.length values = Seq.length indexes} -> skipList a f
```

Equivalent code in C is given below, although note that the ordering constraint is not explicitly stated as C does not support these kinds of constraints. However, in compiled CompCert C code, the constraints are enforced via explicit checks on bounds.

```
typedef uint8_t bytes;
typedel uint8_t a;
struct skipList
{
        a* values;
        int* indexes;
}
```

In F* (and unlike the simpler C code), the sequence of values is explicitly specified to be sorted (i.e. for a given index of each element with the index that is more than current one satisfies the predicate f, where f is defined in terms of an ordering function for a given type of data). This is required so that the list can be searched effectively. Second, in order to preserve memory safety, all the indexes are also stored in a sequence that has the same length as the sequence of values. For each value there exists at least one element it it references, which is by default the next element for a linked list. However, in a skip-list a single element may be referenced by more than one index.

Correctness of our implementation is shown via proving lemmas, which are statements that carry useful properties about the program that are used in proofs of functional correctness or security. The lemma below shows that a skiplist can be used as usual linked list (with `counter_global` used to reference the indexes in skiplist). The data structure is organized such way that there exists an index to the following element in the sequence.

```
lemma_linked_list : sl:skipList a f {Sl.length sl > 0} ->
counter_global:nat{counter_global < (Sl.length sl -1)} ->
        Lemma(ensures (last_element_indexed sl counter_global =
        counter_global +1))
```

In order to prove memory correctness, we prove the lemma that shows that all the indexes used are less than than the length of the data structure. It means that none of the indexes are going to go “out of bounds” of the data structure. Note that `counter_global` in used for the referencing of the index in general, while `counter_local` is used to reference the concrete index of a particular point in the array of indexes):

```
lemma_indexLessThanSize: sl: skipList a f{length sl> 0}  ->
        Lemma(ensures(forall (counter_global: nat {counter_global < length sl})
        (counter_local : nat {counter_local < List.length
                (getIndex sl counter_global)}).
                (fun (x: nat) -> x < (length sl))
        (List.index(getIndex sl counter_global) counter_local)))
```

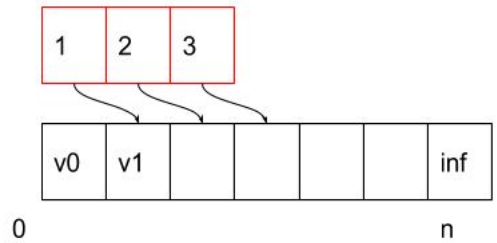
Next, we want to prove that the list of values is in order, namely that all the indices that can have a concrete value have references that are strictly more than the index of the value. It means that all the elements the value is referencing will be strictly more than the value itself.

```
lemma_valuesMoreThanIndex: sl: skipList a f {length sl> 1} ->
        Lemma(ensures
                (forall (counter_global: nat {counter_global < length sl})
                (counter_local : nat
                        {counter_local <List.length
                                (getIndex sl counter_global)}).
                (fun (x: nat) -> x > counter_global)
        (List.index(getIndex sl counter_global) counter_local)))
```

5.3 Complexity of Operations over Skiplists

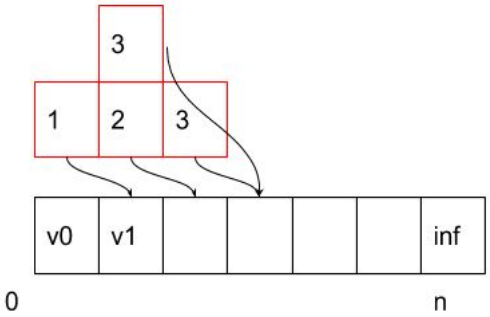
The complexity of linked list is $O(n)$, while the average complexity of search/insert/delete algorithms for skiplist is $O(logn)$ (where n is the number of elements stored in skiplist. Note that the worse-case complexity of a skip list is the same as a linked list and in order to guarantee $O(logn)$ complexity one would have to use a more complex and expensive data structure such as a balanced tree. However, probabilistically the chance of a skiplist having worse case behavior is small, so the average case complexity of $O(logn)$ should hold for queries, and a skiplist does not have the overhead of recreating the list everytime a new item is added.

For an example of how this average case complexity is achieved, let's start with a simple linked list where each list has just one link to the next element. Yet unlike a normal linked list, a skiplist has a separate array of indexes that can point to any other index in the list. So in a linked list, the first layer is used to store the next value and in this case the skiplist could be represented as a usual linked list.




```
type skipList1level(a:eqtype) (f:cmp a) =
| Mk:  values: seq(a){sorted f values}->
        indexes : seq nat{forall (y: nat{y < length indexes).
        indexes[y] = indexes[y]+1 } -> skipList a f
```

To find an element a, one traverses the list of all elements until finding a. As the next step, we assume that the index list now has a reference to the index not only on the next element, but also to the element two positions ahead:




```
type skipList(a:eqtype) (f:cmp a) =
| Mk:  values: seq(a){sorted f values}->
        indexes : seq (non_empty_list nat)
        {forall (y: nat {y < length indexes}).
        indexes[y][0] = indexes[y+1][0]+1 /\
        indexes[y][1] = indexes[y+1][1]+2 // for each second element
} -> skipList a f
```

Therefore, this makes the search twice as quick. While going through the element with index $(y,0) = 1$, one checks whether the element's value at indices $y,1$ is more than the element a. If it's the case, the search will be repeated starting from the element $(y,1) = 3$, thus skipping the second element. In the same manner, if each fourth node has links to four nodes ahead, the search will be done 4 times faster. To generalize,for $i = 0...log(n)$ $1/2^i$ of the nodes will have a link to next 2^i elements, the search efficiency will be $O(log(n))$. Another assumption is that the structure stays balanced, as otherwise there is a loss of performance during the insert and delete procedure because it's needed to rebuild the indices. However, the use of randomized insertion can create a randomized structure with performance that is close to ideal [3]. This is done via a probability p such that the probability of the newly inserted node being on level 2 of the index is equals to $1/2$, to have 3 levels equals to $1/4$ etc.

FStar code:

```
assume val flipcoin : unit -> Tot(r: nat{r = 0 \/ r = 1})

val random : max: nat -> counter : nat{counter <=max} -> result:nat  ->
            Tot(nat)(decreases (max - counter))

let rec random max counter result =
    if counter = max then result else
        let flip = flipcoin () in
            if flip = 0 then result else
            let result = result + 1 in
            random max (counter + 1) result
```

5.4 API

5.4.1 Insert

If the list already exists, the *Insert* is called, otherwise the list is creation helper function is called. This creation function consists on the generation of the list with one element given the value of infinity so it will always be the largest element in a list and does not have a reference to any element. This is necessary as in case of an element doesn't have an element that can be referenced, the reference will go to the last element.

```
val create :  value_max : a -> elements_number: nat {elements_number > 0}
->Tot(sl: skipList a f {length sl = 1})
```

Insert is divided into searching for the location for the new element to be inserted and the insertion itself. The abstract procedure of insertion is as follows:

FStar Code:

```
let insert value sl max_level =
    let place = searchPlace sl value in
    let level = generate_level place max_level in
    let values = change_values value sl place in
    let indexes = change_indexes sl place level in
    Mk values indexes
```

C Function:

```
skipList insert (a value, skipList* sl, int index, int max_level)
```

The procedure *searchPlace* returns the place for the element *value* to put and ensures *value[place]* should be strictly more than *value[place+1]* should be strictly less than *value*.

FStar Code:

```
val searchPlace:  sl: skipList a f{Sl.length sl > 0} -> value: a->
Tot(place: nat{(
        (place = 0 /\ f value (Seq.index (getValues sl) 0))
        \/ (place < (Sl.length sl -1) /\
                f value (Seq.index (getValues sl) (place+1)) /\
                f (Seq.index (getValues sl) place) value ))
})
```

The *generate_level* procedure returns a number of elements that new value will have reference to, and *change_values* procedure takes the existing sequence of values and returns the sequence with the new element put on the place calculated by *searchPlace* procedure, and then *change_indexes* procedure finally takes the existing sequence of indexes for each value and return the sequence with index list for the new element put on the place calculated by *searchPlace* routine. It thus guarantees that the element was inserted to the right position and that the new structure satisfies the skiplist specification.

5.4.2 Search

The search procedure is implemented as a routine in order to find an element in existing skiplist. To find element v, the search procedure starts with index 0 and then takes the largest index element and compares whether it is larger than v. If it's the case, then the search routine starts again with an element referenced by the index. Then, the next largest index is checked. The algorithm continues either until the end of the skiplist, or until the element v in the list is found with $values[place]$ having the element.

FStar Code:

```
val search: sl: skipList a f {Sl.length sl > 1}-> element : a ->
Tot(option(place: nat { place < Sl.length sl /\ element = getValue sl place}))
```

C Function:

```
nat* search (skiplist* sl, a element);
```

Additionally, the search routine provides the following interfaces:

FStar Code:

```
val nextElement: sl: skipList a f {Sl.length sl > 1}->
        element: a -> Tot(option (a))
val nextElement: sl: skipList a f {Sl.length sl > 1}->
        index: nat -> Tot(option (a))
val exist: sl: skipList a f{Sl.length sl >1} ->
        element : a -> Tot(bool)
```

C Functions:

```
a* nextElement(skipList* sl, a element);
a* nextElementIndex(skipList* sl, nat index);
bool exist(skipList* sl, a element);
```

The search routine used in insert and remove procedures.

5.4.3 Split

The *split* procedure splits a skiplist into two different skiplists. Having only part of the skiplist could be useful, such as when reasoning only about part of skiplist such as when searching for only up to a certain element. This could be useful for ClaimChain when in the case of key compromise, one could search until which the chain was trusted and to discard the rest.

FStar Code:

```
let split sl place max =
        let fst_v, fst_i = add_infinity fst_v fst_i max in
        let fst_i = reg_indexes fst_i place in
        let snd_i = right_part_reg_indexes place sl [place + 1] 0 in
        (Sl.Mk fst_v fst_i; Sl.Mk snd_v snd_i)
```

C Function:

```
skipList* split (skipList* sl, nat place, a max)
```

The $addInfinity$ is a procedure (the first line) that takes a splitted sequences of values and indexes and returns the sequence with a tail added. The $regIndexes$ (2) and $rightPartRegIndexes$ (3) procedures take the index lists of both sequences and regenerate them to be able to index the newly changed value sequences. As a result, we ensure the new structures satisfy all the properties of skiplist. The remove procedure uses the place of the element as a divisor:

FStar Code:

```
val split: sl : skipList a f {Sl.length sl > 0} ->
        place: nat {place > 0 /\ place < Sl.length sl -1} ->
        Tot(skipList a f * skipList a f)
```

C Function:

```
skipList* split (skipList* sl, nat place)
```

It could be preceded by a search routine. This function will provide an interface to split using the particular value of skiplist:

FStar Code:

```
val split: sl : skipList a f {Sl.length sl > 0} ->
        element: a  ->
        Tot(skipList a f * skipList a f)
```

C Function:

```
skipList* split(skipList * sl, a element);
```

5.4.4 Remove

Remove procedure is similar to *split*, with the small difference that the first element of the one of sequences will be removed and the skiplist are then merged without that member.

FStar Code:

```
let remove sl place =
    let values_new = rebuildValues values place in
    let indexes_new = rebuildIndexes sl place in
    Sl.Mk values_new indexes_new
```

C Function:

```
void remove(skipList* sl, nat place)
```

In detail, *rebuildValues* splits the skiplist at *place*, and then the indices are rebuilt without the removed element by *rebuildIndexes*. The remove procedure is not used in ClaimChain due to the fact that ClaimChain is append-only, but it is a useful procedure to allow a complete verified implementation of skiplist that can be used by other projects.

If *remove* is given an element rather than an index, it can remove the element as well.

FStar Code:

```
val remove:
        sl: skipList a f {Sl.length sl > 1} ->
        element : a  ->
        ML(r: skipList a f {Sl.length sl = Sl.length r + 1})
```

C listing:

```
void remove(skipList* sl, a element);
```

5.5 Skiplists with cryptographic hash pointers

A cryptographic hash function is a hash function which takes an input and returns a fixed-size alphanumeric string where it is computationally difficult to find two messages for which the same hashes of which will be identical. In terms of skiplist, the hash point in the index will consist of the hash of the data at the value. This mechanism gives the security guarantees of integrity of the data, so that a third-party can check that there was no data was changed. It will lead to a small change of memory consumption for the data structure by adding the

hash of the values, and for every procedure it will be needed to regenerate a hash.

```
type cryptoSkipList
(a:eqtype) (f:(a->a->Tot bool)) (hash:(a -> hash))  =
| Mk:   values: seq(a){sorted f values}->
        indexes : seq(non_empty_list nat)
                {Seq.length values = Seq.length indexes}->
        hashes: seq (hash)
                {Seq.length hashed = Seq.length values /\
                {forall (y:nat{length values < y).
        Seq.index y hash = hash (Seq.index y values) +
        hash (Seq.index y                    indexes)}
        -> skipList a f
```

6 Merkle Tree

Merkle trees are useful implementation of sets, allowing both efficient data storage with integrity guarantees of the data, with a structure that makes it easier to prove inclusion of an element in the set.

6.1 Formal Definition

They were defined by Merkle as way to do one-time Lamport Signatures without requiring an entirely new public key each time [1]. A Merkle Tree is an cryptographically verified implementation of a map M. For a given value a, the map M returns another value b such that $M(a) = b$. Since it is cryptographically verified, a Merkle Tree maintains the property that for a given a there can be only one b and that a proof p can be given show that $M(a) = b$. For Merkle Trees, the proof p is given by the integrity property of a hash function over a path in the tree.

6.2 Specification

Merkle trees are based on a tree where each leaf of the tree has a value and each root contains a concatenation of hashes of two leaves.

FStar Code:

```
open HashFunction

type hash = seq nat
type data = seq nat
type merkleTree: level:nat -> h: hash -> Type  =
| MLeaf: element : data -> merkleTree 0 (hashFunc element)
| MNode: #level: nat -> #h1: hash -> #h2 : hash ->
        lnode: merkleTree level h1 ->
        rnode: option(merkleTree level h2) ->
        merkleTree (level+1) (hashConcat h1 h2)
```

C Code:

```
struct MerkleTree
{
    nat level;
    bytes hash;
    MerkleTree* lnode;
    MerkleTree* rnode;
}
```

6.3 API

6.3.1 Proof of existence

To be able to prove the inclusion of an element in the Merkle Tree, a path is provided as a proof. The path consists of a list of bypes that contain the path, include at every bit the directions of whether left or right should be chosen after each root. This path can then be used to get the value of the element according to path:

FStar Code:

```
val get_elt: #h:hash -> path:path -> tree:mtree (len path) h -> Tot data
             (decreases path)
let rec get_elt #h path tree =
  match path with
    | [] -> L?.data tree
    | bit::path' ->
      if bit then
        get_elt #(N?.h1 tree) path' (N?.left tree)
      else
        get_elt #(N?.h2 tree) path' (N?.right tree)
```

C Function:

```
bytes get_elt(bytes hash, bytes path, MerkleTree tree);
```

A *verifier* can use the path to verify for a given Merkle Tree that the data exists by computing the root hash of the tree given the path, without having to recompute the values of the entire tree but only those needed to compute the hash of the path. First a *prover* generates the path to the value in the Merkle Tree.

FStar Code:

```
val prover: #h:hash ->
            path:path ->
            tree:mtree (len path) h ->
            Tot (p:proof{lenp p = len path})
            (decreases path)
let rec prover #h path tree =
  match path with
    | [] -> Mk_proof (L?.data tree) []

    | bit::path' ->
      let N #dc #hl #hr left right  = tree in
      if bit then
        let p = prover path' left in
        Mk_proof (p_data p) (hr::(p_stream p))
```

```
      else
        let p = prover path' right in
        Mk_proof (p_data p) (hl::(p_stream p))
```

Then a *verifier* verifies the path provided by the prover. The following code creates a verified that matches the path (the *p_stream*, or "proof stream") by checking the hash for each bit.

FStar Code:

```
val verifier: path:path -> p:proof{lenp p = len path} -> Tot hash
let rec verifier path p =
  match path with
    | [] -> gen_hash (p_data p)

    | bit::path' ->
      match p_stream p with
        | hd::_ ->
          let h' = verifier path' (p_tail p) in
          if bit then
            gen_hash (Concat h' hd)
          else
            gen_hash (Concat hd h')
```

This is then verified to be correct in the following code, where the path provided can then verified by the lemma that shows that for each the correctness of the proof of inclusion in the Merkle Tree depends recursively on the values in the rest of the tree.

```
val correctness: #h:hash ->
                 path:path ->
                 tree:mtree (len path) h ->
                 p:proof{p = prover path tree} ->
                 Lemma (requires True) (ensures (verifier path p = h))
                 (decreases path)
let rec correctness #h path tree p =
  match path with
    | [] -> ()
    | bit::path' ->
      if bit then
        correctness #(N?.h1 tree) path' (N?.left tree) (p_tail p)
      else
        correctness #(N?.h2 tree) path' (N?.right tree) (p_tail p)
```

The last lemma shows that the only way a verifier can be tricked into accepting proof for of an non-existent element is if there is a hash collision, and therefore show a reduction of the security of a Merkle Tree to that of the underlying function. If the hash function used is resistance to collision attacks and is therefore second-preimage resistant, the Merkle Tree should be secure.

```
type hash_collision =
    cexists (fun n -> cexists (fun (s1:mstring n) -> cexists (fun (s2:mstring n) ->
            u:unit{gen_hash s1 = gen_hash s2 /\ not (s1 = s2)})))
```

```
val security: #h:hash ->
              path:path ->
              tree:mtree (len path) h ->
              p:proof{lenp p = len path /\ verifier path p = h /\
                      not (get_elt path tree = p_data p)} ->
              Tot hash_collision
              (decreases path)
let rec security #h path tree p =
  match path with
    | [] -> ExIntro data_size (ExIntro (p_data p) (ExIntro (L?.data tree) ()))

    | bit::path' ->
      let N #dc #h1 #h2 left right = tree in
      let h' = verifier path' (p_tail p) in
      let hd = Cons?.hd (p_stream p) in
      if bit then
        if h' = h1 then
          security path' left (p_tail p)
        else
          ExIntro (hash_size + hash_size)
      (ExIntro (Concat h1 h2) (ExIntro (Concat h' hd) ()))
      else
        if h' = h2 then
          security path' right (p_tail p)
        else
          ExIntro (hash_size + hash_size)
      (ExIntro (Concat h1 h2) (ExIntro (Concat hd h') ()))
```

6.3.2 Proof of inclusion

As a proof of inclusion, we are planning to provide two different paths for the trees. Each path will correspond to the existing tree. First path describes the way to reach the existing tree in the new one, the second path shows the way to reach the added tree. Presence of both in merkle tree proves the correctness of addition.

7 Verifiable Random Functions (VRFs)

A Verifiable Random Function (VRF) is the public-key version of pseudorandom function, such as a keyed cryptographic hash as used in MACs. Normally a keyed cryptographic hash takes as its argument a private key and a message, and from these produces a cryptographic hash value that can be verified only with a secret key. However, a Verifiable Random Function requires an asymmetric keypair, where the hash is computed with the secret key and an input and the hash can publicly verified via the use of a public key. If someone does not possess the public key, they cannot verify the hash. This is useful when one wants to achieve indistinguishability between hashed messages, but not have the verification of the hashed message depend on the possession of the cleartext messages. In summary, only the person who has the private key could generate a hash, while all owners of the corresponding public key could verify its correctness.

7.1 Definition

A VRF was defined as a function f that, in combination with a public-private keypair K consisting of private key sk and public key pk, such that for any value x applied to the function $y = f(x, sk)$, the public key pk can prove that $y = f(x, sk)$. The VRF construction was defined by Micali et al. [2]

7.2 Specification

The VRF specification is that of Goldberg et al. currently in standards track in the IETF, although some steps are in flux (such as the order of hash functions) and therefore minor changes may be expected throughout the lifetime of the NEXTLEAP project until the IETF specification finalizes.[6] A VRF does not maintain a data structure, so it is specified entirely in terms of its API as given in Section 7.4.

7.3 Helper Functions

However, we did have to make a number of choices about our concrete implementation of the specification, as the specification specifies a number of options, including the RSA full domain hash. In particular, we used the P-256 (NIST) curve but in the future we will explore if we can use the Ed25519 curve, as that curve has been formally verified itself in F* [**?**].

For a hash function, we use SHA256. As our verified VRF is based on an elliptic curve, it uses the following operations over the curve:

- EC point multiplication
- EC point addition
- Check whether the specified point belongs to the curve

The following helper functions are used to make a casting between data types. They are implemented according to the specification described in Standards for Efficient Cryptography Group (SECG).[7] The point convertion is done using point compression.

7.3.1 OS2ECP

The *OS2ECP* procedure takes as an input a bytes representation of a EC point and returns the corresponding point:

FStar Code:

```
val _OS2ECP : bytes -> Tot(serialized_point)
```

C Function:

```
Point* _OS2ECP (bytes b);
```

[6] https://datatracker.ietf.org/doc/draft-goldbe-vrf/
[7] http://www.secg.org/sec1-v2.pdf

7.3.2 ECP2OS

The *ECP2OS* function does the inverse, namely taking a EC point and returns the corresponding bytes representation of the point:

FStar Code:

```
val _ECP2OS : gamma: serialized_point -> Tot(r: bytes)
```

C Function:

```
bytes _ECP2OS (Point* gamma);
```

7.3.3 I2OSP

. This procedure of that takes in an integer and returns the corresponding byte representation. It also has an inverse *OS2IP*.

FStar Code:

```
val _I2OSP: value1: int -> n: int{n > 0} -> Tot(r: bytes{Seq.length r = n})
```

C Functions:

```
bytes _I2OSP (int value1, int n);
int _OS2IP(bytes s);
```

7.4 API

A VRF consists of three functions over two roles. The first role is the *verifier* that attempts to prove that the *owner* indeed possesses a secret key. The owner produces a hash that is keyed by their secret key. The verifier has a public key for a hash. The verifier receives a *proof* from the owner of the secret key. The verifier then takes the proof and calculates the hash. If the hash computed by verifier is equal to the hash publicly created by the owner from the input, then the owner does indeed possess the secret key. These are given by the following three functions:

- *ECVRF-prove*: The proof generation function
- *ECVRF-proof2hash*: Returns the corresponding hash from a proof
- *ECVRF-verify*: Given a public key, proof, and hash and returns whether or not a proof is valid.

7.4.1 ECVRF-prove

$ECVRF-prove$ takes input to be hashed and an asymmetric keypair and returns the proof that is used to verify the correctness of computed hash. First, the input is mapped to a point on the elliptic curve and converted to a hash. Then there is a nonce generated less than *q-1*. This is used with the secret key material to calculate a proof. A γ is calculated by taking the hash to the exponent of the secret key. Finally, a hash is calculated based on the secret key and the rest of the parameters as well as a value s based on the nonce and hash. The last line concatenates the γ, the hash based on the secret key, and s.

FStar Code:

```
let ECVRF_prove input privateKey =
        let h = ECVRF_hash_to_curve(input, g^x) in
        let gamma = h^x in
        let k = random ( 0 (q-1) ) in
        let c = ECVRF_hash_points (g, h, privateKey, gamma, g^k, h^k) in
        let s = k - c * x mod q in
        let pi = ECP2OS(gamma) || I2OSP(c,n) || I2OSP(s,2n) in pi
```

C Function:

```
bytes ECVRF_prove (bytes input, bytes privateKey)
```

Lemmas have been created to verify the security properties of the VRF. For example, restrictions are added to the length of the input to meet the restrictions of SHA256 hash function.

```
val _ECVRF_prove:
        input: bytes {Seq.length input < pow2 61 - (op_Multiply 2 n) - 5 } ->
        public_key: serialized_point -> private_key : bytes ->
        generator : serialized_point->
        Tot(proof: option bytes {Some?proof ==> Seq.length
        (Some?.v proof) = (op_Multiply 5 n) + 1})
```

7.4.2 ECVRF-proof2hash

$ECVRF - proof2hash$ takes a proof as an input and returns the corresponding hash.

FStar Code:

```
val _ECVRF_proof2hash: pi: bytes{Seq.length pi = op_Multiply 5 n + 1} ->
Tot(hash: bytes)
```

C Function:

```
bytes _ECVRF_proof2hash(bytes pi);
```

7.4.3 ECVRF-verify

The proof verification function ($ECVRF - verify$) takes public key, proof and input and returns the result whether the proof is valid or not.

FStar Code:

```
let ECVRF_verify publicKey proof input =
        let gamma, c, s = ECVRF_decode_proof pi in
        if not isValidPoint gamma then return false else
        let u = (g^x)^c * g^s in
        h = ECVRF_hash_to_curve alpha, g^x  in
        let v = gamma^c * h^s
        let c_ = ECVRF_hash_points g, h, g^x, gamma, u, v  in
        c==c_
```

C listing:

```
bool ECVRF_verify(bytes publicKey, bytes proof, bytes input)
```

The first step is to transform the provided proof back into its components. First, it is verified that γ is a correct point of the curve. Then the input is converted into a point of the curve. Then γ, s, and the hash are used to create v. Finally, v is transformed into a hash c and it is checked to see if that hash is indeed the x-co-ordinate of γ. If so, then the VRF is valid.

8 ClaimChain

8.1 Introduction

These components presented above can be composed to build the entire Claimchain system, as discussed in D2.2 and further refined in D4.2. These components are described in modules. Each module provides the full functionality and are self-contained to allow future re-use.

8.2 Map

The data structures of a *map* is a list of key-value pairs, where each key retrieves a particular value. In a Claimchain, the $claimLabel$ is a key that maps to the values given in by a $claimBody$.

FStar Code:

```
type kv (a: eqtype) (b:eqtype) =
  |MkKV : key: a -> value : b -> kv a b

type map (a: eqtype) (b:eqtype) =
        |MapCons: list (kv a b) -> map a b
        |MapList: list a -> list b -> map a b
```

The API for key-value maps allows the check of presence of the element in the map, getting all the keys/values, getting the value by the key using the following functions:

FStar Code:

```
val containsKey: #a: eqtype -> #b: eqtype -> map a b -> bool
val containsValue: #a: eqtype -> #b: eqtype -> map a b -> bool
val put: #a: eqtype -> #b: eqtype -> map a b -> a -> b -> map a b
val keySet: #a: eqtype -> #b: eqtype -> map a b -> list a
val values: #a: eqtype -> #b: eqtype -> map a b -> list b
```

C Function:

```
bool containsKey (map m);
bool containsValue(map m);
void put(map m, a key, b value);
a* keySet (map m);
b* values (map m);
```

8.3 Metadata

8.3.1 Keys

A user is able to store keys used for cryptographic operations in a ClaimChain,as in order to have the security properties listed earlier in Section , these keys are required for auditing claims and establishing a shared key with their ClaimChain readers. The user has the following types of keys that each have a different *source*: Signature, VRF and Diffie-Hellman keys. Note that a user can store more than one keys, and the key is labeled by the source of their type.

FStar Code:

```
type keyEnt =
        |InitKeyEnt : source: string -> key: bytes -> keyEnt
        |PkSig: key : bytes -> keyEnt
        |PkVRF : key : bytes -> keyEnt
        |PkDH : key : bytes -> keyEnt
```

In a ClaimChain, we assume that the user has all the keys needed for block generation, i.e. the key-presence predicate satisfies the below lemma:

FStar Code:

```
type cryptoKeyEnt =
| CryptoKeyEnt : keys: list keyEnt
    {
        (existsb isKeyEntPkSig keys) /\
        (existsb isKeyEntPkDH keys) /\
        (existsb isKeyEntPkVRF keys) /\
        length keys > 0
    } -> cryptoKeyEnt
```

8.3.2 Identifiers

A block of claims consists of claims as well as metadata about those claims. It could include some data about the user used as an *identifier*, like names and e-mail addresses. identifiers such as e-mail addresses. An identifier could be present as a pair of a source and identifier:

FStar Code:

```
type identifier =
    |InitIndent : source: string -> identifier : string -> identifier
```

Also metadata has keys that are used for block generation. The data structure for keys was discussed in the previous subsection. Finally, all the metadata is hashed. The hash is also stored as a part of metadata.

FStar Code:

```
type metadata =
    |InitMetadata:
        screenName: option string ->
        realName: option string ->
        identifiers: option (list indentifier) ->
```

```
        keys: cryptoKeyEnt ->
        hashMetadata: bytes
        {hashMetadata = hash screenName realName indentifiers keys} ->
        metadata
```

8.4 Claim

A claim is a main data block in a ClaimChain. It provides information about the keys of the owner and the keys of other users. There are several different types of claims: *KeyBindingClaim*, that is used to bind a key, *ClaimChainState* claim, a claim about the particular state of the ClaimChain of another user at some particular moment in time, and a *Revocation* claim needed to declare that keys have been revoked. All the claims consist of a label, the data and a signature. The label is used to reference the claim. The precise type of data inside the claim depends on its type.

FStar Code:

```
|KeyBindingClaim:
        label: string ->
        meta: metadata ->
        key: bytes ->
        signature : bytes  -> claim
```

An example of a claim is: *Claim: label: 'Isaac Newton', metadata : 'realName': 'Isaac Newton', key: 'keyExample', signature : sign (label || meta || key)..* The signature is created using the owner's private key and is used for authenticity of claims. We assume that only the owner could generate a claim with their signature. The claim metadata has a public signing key to check whether the claim was created by the owner.

8.5 Capabilities

Capability lists are used to provide an ability to read the claims to specific (allowed) readers.

8.5.1 claimEncoding

A claim encoding function is used to encrypt a claim so that it can only be read by readers with the required capability. This function (equivalent to Algorithm 1 in D4.2) takes as input the private key of the owner, a nonce, and the claim to be encrypted.

FStar Code:

```
let claimEncoding privateKeyOwnerVRF nonce claim =
   let claimLabel = getClaimLabel claim in
   let claimBody = getClaimBody claim in
   let ncl = concat nonce claimLabel in
   let k, proof = vrf privateKeyOwnerVRF ncl in
   let l = h1 k in
   let ke = h2 k in
   let c = enc ke (concat proof claimBody) in
  MkKV k (l, c)
```

C Function:

```
kv claimEncoding (bytes privateKeyOwnerVRF, bytes nonce, claim claim)
```

First, one gets the label and the claim body. Then one produces the VRF hash and the corresponding proof and produces the lookup key (l). Then one creates the key (ke) that will be used in the next line to encrypt the proof and the claim body. The result of the function is the tuple of hash and key-value of a lookup key and ciphered claim (ke).

8.5.2 decodeClaim

A claim decoding function *decodeClaim* decrypts an encrypted claim (equivalent to Algorithm 5 in D4.2). It is used by the reader to decipher a claim, and so is the inverse of *claimEncoding*. It takes the public key of the owner, nonce, claim label, VRF value k and an encrypted claim. The VRF value k is needed for the decoding of capabilities.

FStar listing:

```
let decodeClaim publicKeyOwnerVRF nonce claimLabel k cipheredClaim =
  let ke = h2 k in
  let proofClaim = dec ke cipheredClaim in
  let vrfPr = vrfProof k proof (concat nonce claimLabel) in
    if vrfPr = true then Some claim else None
```

C Function:

```
claim decodeClaim(bytes publicKeyOwnerVRF, bytes nonce,
string claimLabel, bytes k, bytes cipheredClaim)
```

First, one gets the encryption key and decrypts the encrypted claim. The next step checks that the proof actually corresponds to the claim. If it is the case, the claim body is returned, otherwise the operation was not successful.

8.5.3 encodeCapability

This function is used to encode the capabilities entries by encrypting the VRF value k with a shared secret between the owner and the reader (equivalent to Algorithm 2 in D4.2). The function takes private Diffie-Hellman key of the owner, public reader key, claim label and VRF value k.

FStar Code:

```
let encodeCapability privateKeyOwnerDH publicKeyReaderDH nonce claimLabel k =
    let s = sharedSecret privateKeyOwnerDH publicKeyReaderDH in
    let body = concat nonce s claimLabel in
    let la = h3 body in
    let key = h4 body in
    let pa = enc key k in
    la, pa
```

C Function:

```
void encodeCapability(bytes *la, bytes* pa, bytes privateKeyOwnerDH, bytes publicKeyReaderDH
```

First, we generate the shared secret between the owner and the user, and then encrypt the claim with the shared secret, returning the encoded capability for claim given by *claimLabel* for the reader's public key.

8.5.4 decodeCapability

The capability decoding procedure decrypt the capability of a user so that they can read a claim (equivalent to Algorithm 4 in D4.2). To decrypt the capability, a shared secret is generated between the user and the owner, as well as the decryption key that can then decrypt the VRF value.

FStar Code:

```
let decodeCapability privateKeyReaderDH ownerPublicKeyDH nonce
        claimLabel capabilityCiphered =
        let s = sharedSecret privateKeyReaderDH ownerPublicKeyDH in
        let body = concat nonce s claimLabel in
        let key = h4 body in
        let k = dec key capabilityCiphered in
        let l = h1 k in (k, l)
```

C Function:

```
void decodeCapability(bytes* k, bytes l, bytes privateKeyReaderDH,
bytes ownerPublicKeyDH, bytes nonce, string claimLabel, bytes capabilityCiphered)
```

First, the function computes the shared secret between the reader and the ClaimChain owner. The result of the function is decoded VRF value and lookup key for the claim used to decrypt the encrypted capability.

8.5.5 computeCapabilityLookupKey

This last method *computeCapabilityLookupKey* is used for computing the capability lookup key (equivalent to the Algorithm 3 in D4.2). The key is computed using the shared secret between the reader and the owner.

FStar Code:

```
let computeCapabilityLookupKey privateKeyOwnerDH
publicKeyReaderDH nonce claimLabel =
    let s = sharedSecret privateKeyOwnerDH publicKeyReaderDH in
    let body = concat nonce s claimLabel in
    let la = h3 body in la
```

C Function:

```
void computeCapabilityLookupKey(bytes* la, bytes privateKeyOwnerDH, bytes publicKeyReaderDH,
```

First, the function computes the shared secret between the reader and the claimchain owner. Then the lookup key is computed and returned from the function.

8.6 ClaimChain Module

The ClaimChain module is used to provide high level interfaces for block construction and claim retrieval. The main functionality of the module is to provide high-level operations to allow end-users to make use of a ClaimChain, while hiding the details of capabilities and the use of encryption.

Each block consists on the following information:

- *nonce*: a random number per block that is used only once per chain, which is needed for encryption and decryption.

- *time*: the timestamp that records when the block was generated
- *meta*: the metadata of the block, which may contains optional data that may vary per type of block
- *hashMT*: the header of the Merkle Tree with all the claims and capabilities of the given ClaimChain
- *hashPrevious*: the hash of the previous block
- hash: the hash of the currently created block
- *signature*: Signature of the block as given by owner's signing key.

The security property of integrity is given by the use of hashes in ClaimChain. In particular, *hashPrevious* is used to link the previous block. For audit purposes and consistency of the entire ClaimChain is provided by *hashMT*. The security property is given by reduction to the security property of a strong hash function, such that no adversary is able to inject the block in between the block list without being detected. The use of a *signature* of a block maintains the property of authenticity, as it can prove that each new block was created or authorized by the owner of the ClaimChain.

FStar Code:

```
type claimChainBlock =
        |InitClaimChain :
        nonce: bytes ->
        t: time ->
        meta : metadata ->
        hashMT: bytes ->
        hashPrevious:bytes ->
        hash: bytes ->
        signature :bytes ->
        claimChainBlock
```

8.6.1 generateBlockGeneral

The *generateBlockGeneral* function by the owner to create a new block on a ClaimChain with a new set of claims and capabilities (Equivalent to Section 3.4.3 'Constructing a new block' in D4.2).

FStar Code:

```
private val generateBlockGeneral:
        privateKeyDH: key ->
        privateKeyVRF: key ->
        listClaims: map string claim ->
        accessControl: map (publicKeyReader: key) (labels: list string) ->
        meta: metadata ->
        reference: option(list claimChainBlock) -> ML claimChainBlock
```

C Function:

```
ClaimChainBlock generateBlockGeneral (bytes privateKeyDH, bytes privateKeyVRF,
map listClaims, map accessControl, claimChainBlock* reference);
```

Note that *listClaims* is used to store all the claims the user wants to put to the block. Also, *accessControl* is used to store the map between readers (given by the public key of the reader) and the labels of claims that a reader is allowed to read. References are needed to access to the previous blocks.

8.6.2 cipherClaims

The *cipherClaims* function consists in encrypting all the claims. It does this via applying the function *claimEncoding* to all the claims that the user wants to put onto the block. The owner provides the nonce and his private VRF keys in order to encrypt the claims. The map between claim labels and encrypted claims is returned as a result of the function.

FStar Code:

```
val cipherClaims: cls:  map string claim -> nonce: bytes ->
  privateKeyVRF: bytes -> ML (map string (kv (bytes) (tuple2 bytes bytes)))
```

C Function:

```
map cipherClaims(map cls, bytes nonce, bytes privateKeyVRF);
```

8.6.3 oneUserEncoding

The function *oneUserEncoding* is a helper function needed to encrypt all the claim labels for a particular user. The function requires the access control matrix, the private Diffie-Hellman key, and nonce as well as the rows of claims. The procedure takes all the rows one-by-one and maps the required claim labels to their encrypted claims, followed by applying *encodeCapability* to create the capability encoding.

FStar Code:

```
let rec oneUserEncoding privateKeyDH row claims nonce =
        let labels = row.values in
        match labels with |hd::tl ->
        let claim = get claims hd in
        encodeCapability privateKey row.key nonce hd k
```

C Function:

```
bytes oneUserEncoding(bytes privateKeyDH, row* row, claims* claims, bytes nonce)
```

8.6.4 allUserEncoding

The function *allUserEncoding* applies *oneUserEncoding* for all the users in access matrix.

FStar Code:

```
val allUserEncoding:
        accessControl: map (publicKeyReader: key) (labels: list string)->
        privateKeyDH: key ->
        claims: (map string (kv (bytes) (tuple2 bytes bytes))) ->
        nonce : bytes ->
        oneUserEncoding (list (tuple2 (la: bytes) (pa : bytes)))
```

C Function:

```
bytes allUserEncoding (map accessControl, bytes privateKeyDH, map claims, bytes nonce);
```

These functions create the list of encodings for capabilities and claims. These capabilities and claims are then put to the Merkle Tree for the entire ClaimChain. First, the set containing the capabilities and claims is generated, and this set will be used as a set of leaves for Merkle Tree. Second. the set of leaves is used to generate the hash-root of the tree, and this hash (*hashMT*) is put to the claim chain block, along with the current time and the nonce that was used in all cryptographic operations. Third, the hash of this previous block is also stored in the block. Finally, the block is signed and put to the list of blocks of the ClaimChain.

8.6.5 claimRetrieval

The *claimRetrieval* function takes as input the reader's private keys and public keys of the owner of blockchain, the hash of the ClaimChain, and claim label to retrieve. The result of the function is a decrypted claim body. For the block retrieval we provide the following method:

FStar Code:

```
val claimRetrieval: privateKeyReaderDH: key ->
        publicKeyReaderDH: key ->
        publicKeyOwnerDH:key ->
        publicKeyOwnerVRF : key ->
    tree_hash: bytes ->
        nonce: bytes -> claimLabel: string -> Tot (option bytes)
```

C Function:

```
bytes claimRetrieval(bytes privateKeyReaderDH, bytes publicKeyReaderDH,
bytes publicKeyOwnerDH, bytes publicKeyOwnerVRF,
bytes tree_hash, bytes nonce, string claimLabel);
```

First, the function computes the lookup key using the *computeCapabilityLookupKey*, with the lookup key being used to query the Merkle Tree in order to retrieve the encrypted capability. The capability is then decrypted using the *decodeCapability* to get the tuple of VRF value k and the key needed to retrieve the claim. The claim is then decoded using *decodeClaim* function, with the final result being the value of the claim (*claimBody*) given as plaintext.

FStar Code:

```
  let claimRetrieval keys tree_hash nonce claimLabel =
  let lookUpKey = computeCapabilityLookupKey privateKeyReaderDH
              publicKeyReaderDH nonce claimLabel in
  let capabilityCiphered = queryMerkleTree tree_hash lookUpKey in
  let (k, l) = decodeCapability privateKeyReaderDH
        publicKeyOwnerDH nonce claimLabel capabilityCiphered in
  let c = queryMerkleTree tree_hash l  in
  let claimBody = decodeClaim publicKeyOwnerVRF
        nonce claimLabel k c in claimBody
```

C Function:

```
void claimRetrieval (claim* claim, bytes* keys, bytes tree_hash,
bytes nonce, string claimLabel)
```

9 Conclusion

In this deliverable, we have produced F* code for all of Claimchain as specified initially in D4.1 and finalized in D4.2. This code allows the automatic proof-proving of the security properties defined in this deliverable. This code will serve as a foundation for a generic ClaimChain library that can be integrated into open source code given by Workpackage 5.

References

[1] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[2] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

[3] J Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 367–375. Society for Industrial and Applied Mathematics, 1992.

[4] Andreas Pfitzmann and Marit Köhntopp. Anonymity, unobservability, and pseudonymity—a proposal for terminology. In *Designing privacy enhancing technologies*, pages 1–9. Springer, 2001.

[5] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.