Nate Launchbury Document: Sep. 2018

Lamport One-Time Signature

Sequence of Games explanations

```
Definition A_forges :=
  k <-$ OTSKeyGen H n l;
  [prikey, pubkey]<-2 k;
  a1 <-$ A1 n l pubkey;
  [msg, state]<-2 a1;
  sig <-$ OTSSign prikey msg;
  a2 <-$ A2 n l (sig, state);
  [msg', sig']<-2 a2;
  if msg' ?= msg then ret false
  else ret OTSVerify H pubkey msg' sig'.
```

```
Definition A_forges_inv :=
  i <-$ [0..l);
  r <-$ {0,1};
  k <-$ OTSKeyGen H n l;
  [prikey, pubkey]<-2 k;
  a1 <-$ A1 n l pubkey;
  [msg, state]<-2 a1;
  sig <-$ OTSSign prikey msg;
  a2 <-$ A2 n l (sig, state);
  [msg', sig']<-2 a2;
  if negb (msg' ?= msg)
  then ret OTSVerify H pubkey msg' sig'
  else ret false.
```

1. `A_forges_A_forges_inv :`
   `Pr[A_forges] == Pr[A_forges_inv].`

`A_forges` is the inlining and simplification of the `OTSigForge_Advantage` game. `A_forges_inv` is the exact same game except it also samples a random index into the 2xl dimension of the public and private keys, though it does nothing with this sample. It also inverts the condition of the final if-statement while also swapping the branches (to match the Invert game later on). The proof removes the irrelevent samples and then skips over the body of the game before performing a basic computation.

```
Definition Game1 :=
  i <-$ [0..l);
  r <-$ {0,1};
  k <-$ OTSKeyGen H n l;
  [prikey, pubkey]<-2 k;
  a1 <-$ A1 n l pubkey;
  [msg, state]<-2 a1;
  sig <-$ OTSSign prikey msg;
  a1 <-$ A2 n l (sig, state);
  [msg', sig']<-2 a1;
  (* Pr[column] = 1/l *)
  column <- negb (ith_element msg i ?= ith_element msg' i);
  if negb (msg' ?= msg) && column
  then ret OTSVerify H pubkey msg' sig'
  else ret false.
```

2. `A_forges_inv_Game1 :`
   `(1/l) * Pr[A_forges_inv] == Pr[Game1].`

`Game1` adds an additional condition that the queried message and the forged message differ at the insertion column. This condition, along with the condition on the insertion row, is what allows the reduction to work since A2 will be forced to invert y if their signature forgery is successful. The proof is in progress. The basic idea is that, since the messages must differ, there is a 1/l chance that they differ in the i-th position. In reality, they may differ at more than one position so this is really an inequality. The “distribution isomorphism” theorem will be used to transform the column check to a check with the same probability which is independent of the rest of the game, allowing us to use the `indep_and` theorem to discharge the goal.

```
Definition Game2 :=
  i <-$ [0..l);
  r <-$ {0,1};
  k <-$ OTSKeyGen H n l;
  [prikey, pubkey]<-2 k;
  a1 <-$ A1 n l pubkey;
  [msg, state]<-2 a1;
  sig <-$ OTSSign prikey msg;
  a2 <-$ A2 n l (sig, state);
  [msg', sig']<-2 a2;
  column <- negb (ith_element msg i ?= ith_element msg' i);
  (* Pr[row] = 1/2 *)
  row <- negb (ith_element msg i ?= Some r);
  if negb (msg' ?= msg) && column && row
  then ret OTSVerify H pubkey msg' sig'
  else ret false.
```

3. `Game1_Game2` :
`1/2 * Pr[Game1] == Pr[Game2].`

`Game2` adds a check that the sampled inversion position differs from the row of the queried message at that position. Since there are two rows, this condition will be true half the time. We rely on this condition to properly sign the message and also so that the adversary's forgery has a chance of inverting y. The proof uses the "distribution isomorphism" theorem to show that the condition "`negb (ith_element msg i ?= Some r)`" has the same probability as the independent check simply on the boolean "`r`". We can then reformat `Game2` to use the `indep_and` theorem on the isomorphic condition "`r`".

```
Definition Game3 :=
  i <-$ [0..l);
  r <-$ {0,1};
  k <-$ KeyGenInsert i r;
  [prikey, pubkey]<-2 k;
  a1 <-$ A1 n l pubkey;
  [msg, state]<-2 a1;
  sig <-$ OTSSign prikey msg;
  a2 <-$ A2 n l (sig, state);
  [msg', sig']<-2 a2;
  row <- negb (ith_element msg i ?= Some r);
  column <- negb (ith_element msg i ?= ith_element msg' i);
  if negb (msg' ?= msg) && row && column
  then ret OTSVerify H pubkey msg' sig'
  else ret false.
```

```
Definition KeyGenInsert i r :=
  x <-$ {0,1}^n;
  [prikey,pubkey] <-$2 OTSKeyGen H n l;
  newpri <- keyInsert i r x prikey;
  newpub <- keyInsert i r (H n x) pubkey;
  ret (newpri,newpub).
```

```
Definition PrivKeyGenInsert {l : nat} i r n :=
  x <-$ {0,1}^n;
  z <-$ OTSKeyGen_priv n l;
  ret (keyInsert i r x z).
```

4. `Game2_Game3` :
`Pr[Game2] <= Pr[Game3].`

`Game3` is the game where the crux of the reduction occurs. The change from `Game2` is that we sample a uniform bitvector, `x`, and insert `x` into the private key at position `(i,r)` and insert `(H x)` into the public key at the same position. The proof is then straightforward but contingent upon the important lemma that `OTSKeyGen` and `KeyGenInsert` have the same distribution. This is quite believable since `OTSKeyGen` simply samples random bitvectors for the key elements and being sure to insert `(H x)` in the public key preserves the relationship between the keys.

A wary reader might point out that we can hardly celebrate successfully inverting `(H x)` later in the reduction if we sample — and therefore know — what `x` is in the first place. But note that `Game3` is not our constructed adversary, merely a game in our sequence with a useful distribution; it will not be the one playing the Inversion game.

```
Definition B_inverts d :=
  x <-$ {0,1}^n;
  a <-$ OTSKeyGen H n l;
  [prikey,pubkey] <-2 a;
  i <-$ [ 0 .. l);
  r <-$ (m <-$ { 0 , 1 }^ 1; ret Vector.hd m);
  z <-$ A1 n l (keyInsert i r (H n x) pubkey);
  [m, state]<-2 z;
  s <-$ OTSSign (keyInsert i r x prikey) m;
  x0 <-$ A2 n l (s, state); [_, s']<-2 x0;
  a0 <- nth_error s' i;
  a1 <-$
  match a0 with
  | Some x' => ret (x', false)
  | None => ret (d, true)
  end; x' <-$ ret fst a1;
  ret ((H n x' ?= H n x) && (negb (ith_element m i ?= Some r))).
```

```
Definition B (y : Bvector n) :=
  key <-$ @OTSKeyGen H n l;
  [privkey,pubkey] <-2 key;
  i <-$ [0..l);
  r <-$ {0,1};
  invertKey <- keyInsert i r y pubkey;
  [m,state] <-$2 A1 n l invertKey;
  if ith_element m i ?= Some r
  then ret None
  else
    s <-$ @OTSSign n l privkey m;
    x <-$ A2 n l (s,state);
    [m',s'] <-2 x;
    ret (nth_error s' i).
```

```
5. Game3_B : forall d,
       Pr[Game3] <= Pr[B_inverts d].
```

B is our constructed adversary which uses A1,A2 to invert y. B_inverts is a slightly reformated copy of the proof-state after inlining and unfolding the Inverst game with B as input. This proof is the central result of the reduction since it is the link between the SigForge game and the Invert game. It relies on a number of supporting lemmas including a specification for OTSVerify, various relationships between keyInsert and other functions, and the uniformity of OTSKeyGen. B_inverts was one of the first games constructed in the reduction and served as a target point for the sequence.

The final step in the sequence is showing that B_inverts matches the Invert game with B as the adversary. This is relatively straightforward given how B_inverts was constructed but also relies on a lemma that we can competently sign the adversary's query so long as the row condition, present in the return of B_inverts, holds. This follows since the signature will "miss" this inserted mystery element (importantly, we now don't know x such that (H x) = y) whenever the row differs from our insertion point.

All together, we can sequence these proofs together to yield our desired result (omitting certain parameters for brevity):

```
(1/l) * (1/2) * OTSigForge_Advantage <= Invert_Advantage B
```