Table of Contents

Basic Definitions

Disk

```
Definition addr := nat.
Axiom addr_eq_dec : forall (a b: addr), {a=b}+{a<>b}.

Axiom value : Type.
Axiom nat_to_value : nat -> value.
Axiom value_to_nat : value -> nat.

Axiom nat_to_value_to_nat:
  forall n, value_to_nat (nat_to_value n) = n.
Axiom value_to_nat_to_value:
  forall v, nat_to_value (value_to_nat v) = v.
Axiom value_dec: forall v v': value, {v=v'}+{v<>v'}.
```

Cryptography

Hashing

```
Axiom hash : Type.
Axiom hash0 : hash.
Axiom hash_dec: forall h1 h2: hash, {h1 = h2}+{h1<>h2}.
Axiom hash_function: hash -> value -> hash.

Definition hashmap := mem hash hash_dec (hash * value).

Fixpoint rolling_hash h vl :=
  match vl with
  | nil => h
  | cons v vl' => rolling_hash (hash_function h v) vl'
  end.

Fixpoint rolling_hash_list h vl :=
  match vl with
  | nil => nil
  | cons v vl' =>
    let h':= hash_function h v in
    cons h' (rolling_hash_list h' vl')
  end.

Fixpoint hash_and_pair h vl :=
  match vl with
  | nil => nil
  | cons v vl' =>
    let h':= hash_function h v in
    cons (h, v) (hash_and_pair h' vl')
  end.
```

Encryption

```
Axiom key: Type.
Axiom key_dec: forall k1 k2: key, {k1 = k2}+{k1<>k2}.
Axiom encrypt: key -> value -> value.
Axiom encrypt_ext: forall k v v', encrypt k v = encrypt k v' -> v = v'.

Definition encryptionmap := mem value value_dec (key * value).
```

Simulation

Definitions

LTS

```
Record LTS :=
  {
    Oracle : Type;
    State : Type;
    Prog : Type -> Type;
    exec : forall T, Oracle -> State -> Prog T -> @Result State T -> Prop;
  }.
```

LTS Refinement Relations

We need these relations because Coq doesn't have a good way to derive LTS's from existing ones by restricting state spaces, transition relations etc.

Refines to Related

This relation allow us to restrict properties to two low level states that refines to two high level states that are related by `related_h`.

Input:

- two input states (`sl1` and `sl2`),
- a refinement relation (`refines_to`), and
- a relation (`related_h`)

Assertions

- there are two other states (`sh1` and `sh2`) such that,
- `sl1` (`sl2`) refines to `sh1` (`sh2`) via `refines_to` relation, and
- `sh1` and `sh2` are related via `related_h`

```
Definition refines_to_related {State_L State_H: Type}
           (refines_to: State_L -> State_H -> Prop)
           (related_h: State_H -> State_H -> Prop)
           (sl1 sl2: State_L)
  : Prop :=
  exists (sh1 sh2: State_H),
    refines_to sl1 sh1 /\
    refines_to sl2 sh2 /\
    related_h sh1 sh2.
```

Refines to Valid

This definition allows us to restrict properties to low level states that refine to a valid high level state.

Input

- an input state (`sl`),
- a refinement relation (`refines_to`), and
- a validity predicate (`valid_state_h`)

Assertions

- for all states `sh`,
- if `sl` refines to `sh` via `refines_to` relation,
- then `sh` is a valid state (satisfies `valid_state_h`)

```
Definition refines_to_valid {State_L State_H: Type}
           (refines_to: State_L -> State_H -> Prop)
           (valid_state_h: State_H -> Prop)
           (sl: State_L)
  : Prop :=
  forall (sh: State_H),
    refines_to sl sh ->
    valid_state_h sh.
```

Compiles to Valid

This definition allows us to restrict properties on low level programs that are a valid compilation of a high level program.

Input

- an input program (`pl`),
- a refinement relation (`refines_to`), and
- a validity predicate (`valid_prog_h`)

Assertions

- there is a program `ph` such that,
- `pl` is compilation of `ph`, and
- `ph` is a valid program (satisfies `valid_prog_h`)

```
Definition compiles_to_valid {Prog_L Prog_H: Type -> Type}
           (valid_prog_h: forall T, Prog_H T -> Prop)
           (compilation_of: forall T, Prog_L T -> Prog_H T -> Prop)
           (T: Type)
           (pl: Prog_L T)
  : Prop :=
  exists (ph: Prog_H T),
    compilation_of T pl ph /\
    valid_prog_h T ph.
```

Transition Preserves Validity

This definition ties notion of validity with restricting transitions by stating that any reachable state from a valid state is also valid.

```
Definition exec_compiled_preserves_validity lts1 lts2
       (compilation_of: forall T, Prog lts1 T -> Prog lts2 T -> Prop) valid_state :=
    forall T (p1: Prog lts1 T) (p2: Prog lts2 T) o s ret,
      compilation_of T p1 p2 ->
      valid_state s ->
      exec lts1 T o s p1 ret ->
      valid_state (extract_state ret).
```

Restrictions on Refinement Relations

Following two properties ensures that your refinement relations has desired properties that allows transferring self simulations between layers

High Oracle Exists

This definition states that

for all low level oracles `o1`

which results in a successful execution of a compiled program `p1` (that is compilation of `p2`)

from a low level state `s1` (that refines to a high level state),

there exists an high level oracle `o2` (that is a refinement of `o1`) for `p2`.

```
Definition high_oracle_exists {low high}
           (refines_to: State low -> State high -> Prop)
           (compilation_of : forall T, Prog low T -> Prog high T -> Prop)
           (oracle_refines_to : forall T, State low -> Prog high T -> Oracle low -> Oracle high -> Prop) :=
  forall T o1 s1 s1' p1 p2,
    (exists sh, refines_to s1 sh) ->
    exec low T o1 s1 p1 s1' ->
    compilation_of T p1 p2 ->
    exists o2, oracle_refines_to T s1 p2 o1 o2.
```

Oracle Refines to Same from Related

This definition states that our oracle refinement is agnostic to low level states that refine to related high level states. This property captures the fact that if two states are related, then they don't change the nondeterminism in different ways during refinement.

```
Definition oracle_refines_to_same_from_related {low high}
           (refines_to: State low -> State high -> Prop)
           (related_states_h: State high -> State high -> Prop)
           (oracle_refines_to: forall T, State low -> Prog high T -> Oracle low -> Oracle high -> Prop) :=
  forall T o oh s1 s2 p2,
    refines_to_related refines_to related_states_h s1 s2 ->
    oracle_refines_to T s1 p2 o oh ->
    oracle_refines_to T s2 p2 o oh.
```

Self Simulation

This is a generalized two-safety property definition. Data confidentiality will be an instance of this. This a little more stronger than a standard simulation because it forces two transitions in two executions to be the same.

```

Record SelfSimulation (lts: LTS)
       (valid_state: State lts -> Prop)
       (valid_prog: forall T, Prog lts T -> Prop)
       (R: State lts -> State lts -> Prop) :=
  {
    self_simulation_correct:
      forall T o p s1 s1' s2,
        valid_state s1 ->
        valid_state s2 ->
        valid_prog T p ->
        (exec lts) T o s1 p s1' ->
        R s1 s2 ->
        exists s2',
          (exec lts) T o s2 p s2' /\
          result_same s1' s2' /\
          R (extract_state s1') (extract_state s2') /\
          (forall def, extract_ret def s1' = extract_ret def s2') /\
          valid_state (extract_state s1') /\
          valid_state (extract_state s2') ;
  }.
```

Strong Bisimulation

This is our refinement notion between two LTS's. It is stronger than a standard bisimulation because it requires transitions to be coupled, instead of just existing a transition in other LTS.

```
Record StrongBisimulation
       (lts1 lts2 : LTS)
       (compilation_of: forall T, Prog lts1 T -> Prog lts2 T -> Prop)
       (refines_to: State lts1 -> State lts2 -> Prop)
       (oracle_refines_to: forall T, State lts1 -> Prog lts2 T -> Oracle lts1 -> Oracle lts2 -> Prop)
  :=
  {
    strong_bisimulation_correct:
      (forall T p1 (p2: Prog lts2 T)) s1 s2 o1 o2,

          refines_to s1 s2 ->
          compilation_of T p1 p2 ->
          oracle_refines_to T s1 p2 o1 o2 ->

          (forall s1',
              (exec lts1) T o1 s1 p1 s1' ->
              exists s2',
                (exec lts2) T o2 s2 p2 s2' /\
                result_same s1' s2' /\
                refines_to (extract_state s1') (extract_state s2') /\
                (forall def, extract_ret def s1' = extract_ret def s2')) /\
          (forall s2',
              (exec lts2) T o2 s2 p2 s2' ->
              exists s1',
                (exec lts1) T o1 s1 p1 s1' /\
                result_same s1' s2' /\
                refines_to (extract_state s1') (extract_state s2') /\
                (forall def, extract_ret def s1' = extract_ret def s2')))
  }.
```

Metatheory

Main Theorem

```
Theorem transfer_high_to_low:
  forall low high

    related_states_h
    refines_to
    compilation_of
    oracle_refines_to

    valid_state_h
    valid_prog_h,

    SelfSimulation
      high
      valid_state_h
      valid_prog_h
      related_states_h ->

    StrongBisimulation
      low
      high
      compilation_of
      refines_to
      oracle_refines_to ->

    high_oracle_exists refines_to compilation_of oracle_refines_to ->

    oracle_refines_to_same_from_related refines_to related_states_h oracle_refines_to ->

    exec_compiled_preserves_validity
    low
    high
    compilation_of
    (refines_to_valid
       refines_to
       valid_state_h) ->

    SelfSimulation
      low
      (refines_to_valid
         refines_to
         valid_state_h)
      (compiles_to_valid
         valid_prog_h
         compilation_of)
      (refines_to_related
         refines_to
         related_states_h).
```

Layers

Disk Layer (Layer 1)

Definitions

```
Inductive token :=
  | Key : key -> token
  | Crash : token
  | Cont : token.

Definition oracle := list token.

Definition state := (((list key * encryptionmap)* hashmap) * disk (value * list value)).

Inductive prog : Type -> Type :=
  | Read : addr -> prog value
  | Write : addr -> value -> prog unit
  | GetKey : list value -> prog key
  | Hash : hash -> value -> prog hash
  | Encrypt : key -> value -> prog value
  | Decrypt : key -> value -> prog value
  | Ret : forall T, T -> prog T
  | Bind : forall T T', prog T -> (T -> prog T') -> prog T'.
```

Operational Semantics

```
Definition consistent (m: mem A AEQ V) a v :=
  m a = None \/ m a = Some v.

Fixpoint consistent_with_upds m al vl :=
  match al, vl with
  | nil, nil => True
  | a::al', v::vl' =>
      consistent m a v /\
      consistent_with_upds (upd m a v) al' vl'
  | _, _ => False
  end.
```

```
Inductive exec : forall T, oracle ->  state -> prog T -> @Result state T -> Prop :=

...

  | ExecHash :
      forall em hm d h v,
        let hv := hash_function h v in
        consistent hm hv (h, v) ->
        exec [Cont] (em, hm, d) (Hash h v) (Finished (em, (upd hm hv (h, v)), d) hv)

  | ExecEncrypt :
      forall kl em hm d k v,
        let ev := encrypt k v in
        consistent em ev (k, v) ->
        exec [Cont] (kl, em, hm, d) (Encrypt k v) (Finished (kl, (upd em ev (k, v)), hm, d) ev)

  | ExecDecrypt :
      forall kl em hm d ev k v,
        ev = encrypt k v ->
        em ev = Some (k, v) ->
        exec [Cont] (kl, em, hm, d) (Decrypt k ev) (Finished (kl, em, hm, d) v)

  | ExecGetKey :
      forall vl kl em hm d k,
        ~In k kl ->
        consistent_with_upds em
             (map (encrypt k) vl) (map (fun v => (k, v)) vl) ->
        exec [Key k] (kl, em, hm, d) (GetKey vl) (Finished ((k::kl), em, hm, d) k).
```

Key Cryptographic Assumptions

No Hash Collisions

This assumption embodied as execution getting stuck if a collision happens during execution. Each hashed value is stored in a map after execution and each input is checked before executing the hash operation.

Justification

It is exponentially unlikely to have a hash collision in a real system.

No Encryption Collisions

This assumption embodied as execution getting stuck if a collision happens during execution. Each key and block pair is stored in a map and each input checked before executing the encryption operation. This is a stronger assumption than the traditional one because it requires no collision for (key, value) pairs instead of two values for the same key.

Why do we need a stronger assumption?

It is required to prevent the following scenario:

There are two equivalents states `st1` and `st2.`

- a transaction is committed,
- header and all but one data block makes to the disk
- crash happens
- non-written data block matches what is on the disk on `st1` only
- recovery commits in `st1` but not in `st2`, leaking confidential information.

Justification

In real execution, it is exponentially unlikely to have a collision even for (key, value) pairs. Also, even in the case that such collision happens, leaked data is practically garbage because it is encrypted.

Generated Key Does Not Cause Collision

This strong assumption (combined with the total correctness requirement) enforces some restrictions for key generation in operational semantics. We need to ensure that generated key will not create an encryption collision. To ensure that, `GenKey` operation takes the blocks that will be encrypted as input.

Justification

In real execution, it is exponentially unlikely to have a collision. Also, even in the case that such collision happens, leaked data is practically garbage because it is encrypted.

One way to circumvent this would be combining `GenKey` and `Encrypt` operation into one operation

`EncryptWithNewKey: list block -> prog (key * list block)`

Which takes blocks to be encrypted and encrypts them with a new key, returning both key and the encrypted blocks. This operations limitation is not a problem for us because every time we are encrypting, we do it with a fresh key anyway.

Components

Log

Structure

```
-------------------------------------------------
|          |       TXN 1       |  TXN 2   |          |
|  Header |  Addr   |  Data   |          |  . . .  |
|          | Blocks | Blocks |          |          |
-------------------------------------------------
```

Header Contents

```
Record txn_record :=
  {
    txn_key : key;
    txn_start : nat; (* Relative to start of the log *)
    addr_count : nat;
    data_count : nat;
  }.

Record header :=
  {
    old_hash : hash;
    old_count : nat;
    old_txn_count: nat;
    cur_hash : hash;
    cur_count : nat;
    txn_records : list txn_record;
  }.
```

Functions

```
Definition commit (addr_l data_l: list value) :=
  hdr <- read_header;
  if (new_count <=? log_length) then
    new_key <- GetKey (addr_l++data_l);
    enc_data <- encrypt_all new_key (addr_l ++ data_l);
    _ <- write_consecutive (log_start + cur_count) enc_data;
    new_hash <- hash_all cur_hash enc_data;
    _ <- write_header new_hdr;
    Ret true
  else
    Ret false.
```

```
Definition apply_log :=
  hdr <- read_header;
  log <- read_consecutive log_start cur_count;
  success <- check_and_flush txns log cur_hash;
  if success then
    Ret true
  else
    success <- check_and_flush old_txns old_log old_hash;
    Ret success.
```

```
Definition check_and_flush txns log hash :=
  log_hash <- hash_all hash0 log;
  if (hash_dec log_hash hash) then
    _ <- flush_txns txns log;
    Ret true
  else
    Ret false.

Definition flush_txns txns log_blocks :=
  _ <- apply_txns txns log_blocks;
  _ <- write_header header0;
  Ret tt.

Fixpoint apply_txns txns log_blocks :=
  match txns with
  | nil =>
    Ret tt
  | txn::txns' =>
    _ <- apply_txn txn log_blocks;
    _ <- apply_txns txns' log_blocks;
    Ret tt
  end.

Definition apply_txn txn log_blocks :=
  plain_blocks <- decrypt_all key txn_blocks;
  _ <- write_batch addr_list data_blocks;
  Ret tt.
```

Block Allocator

Implementation

Definitions

```
Axiom block_size: nat.
Axiom block_size_eq: block_size = 64.

Definition valid_bitlist l :=
    length l = block_size /\ (forall i, In i l -> i < 2).

Record bitlist :=
{
  bits : list nat;
  valid : valid_bitlist bits
}.

Axiom value_to_bits: value -> bitlist.
Axiom bits_to_value: bitlist -> value.
Axiom value_to_bits_to_value :
    forall v, bits_to_value (value_to_bits v) = v.
Axiom bits_to_value_to_bits :
    forall l, value_to_bits (bits_to_value l) = l.
```

Representation Invariant

```
Definition rep (dh: disk value) : @pred addr addr_dec (set value) :=
  (exists bitmap bl,
    let bits := bits (value_to_bits bitmap) in
    0 |-> (bitmap, bl) *
    ptsto_bits dh bits *
    [[ forall i, i >= block_size -> dh i = None ]])%pred.
```

`ptsto_bits dh bits` says that `forall i < length bits, bits[i] = 0 <-> dh (S i) = None`.

Functions

```
Definition read a : prog (option value) :=
  v <- Read 0;
  if a < block_size then
    if bitmap[a] = 1 then
      h <- Read (S a);
      Ret (Some h)
    else
      Ret None
  else
    Ret None.
```

```
Definition write a v' : prog (option unit) :=
  v <- Read 0;
  if a < block_size then
    if bitmap[a] = 1 then
      Write (S a) v'
    else
      Ret None
  else
    Ret None.
```

```
Definition alloc (v': value) : prog (option addr) :=
  v <- Read 0;
  if first_zero bitmap < block_size then
    _ <- Write (S (first_zero bitmap)) v';
    _ <- Write 0 (to_block (updN bitmap (first_zero bitmap) 1));
    Ret (Some index)
  else
    Ret None.
```

```
Definition free a : prog unit :=
  v <- Read 0;
  if a < block_size then
    if bitmap[a] = 1 then
      Write 0 (to_block (updN bitmap a 0);
    else
      Ret tt
  else
    Ret tt.
```

Abstraction (Block Allocator Layer)

Definitions

```
Inductive token:=
  | BlockNum : addr -> token
  | DiskFull : token
  | Cont : token
  | Crash1 : token
  | Crash2 : token
  | CrashAlloc: addr -> token.

Definition state := disk value.

Inductive prog : Type -> Type :=
  | Read : addr -> prog (option value)
  | Write : addr -> value -> prog (option unit)
  | Alloc : value -> prog (option addr)
  | Free : addr -> prog unit
  | Ret : forall T, T -> prog T
  | Bind : forall T T', prog T -> (T -> prog T') -> prog T'.
```

We need two different crash tokens because implementation and abstraction must have the same overall behavior (`Finished -> Finished` and `Crashed -> Crashed`). This prevents us from letting abstraction succeed if implementation crashes after "its commit point" even though resulting states may be the same. Therefore, in case of a crash, we approximately need to know where crash happened so that abstraction can follow accordingly.

Operational Semantics

```
Inductive exec :
    forall T, oracle -> state -> prog T -> @Result state T -> Prop :=

...

  | ExecAllocSucc :
      forall d a v,
        d a = None ->
        exec [BlockNum a] d (Alloc v) (Finished (upd d a v) (Some a))

  | ExecAllocFail :
      forall d v,
        exec [DiskFull] d (Alloc v) (Finished d None)

...

  | ExecAllocCrashBefore :
      forall d v,
        exec [Crash1] d (Alloc v) (Crashed d)

  | ExecAllocCrashAfter :
      forall d a v,
        d a = None ->
        exec [CrashAlloc a] d (Alloc v) (Crashed (upd d a v))

...
```

Refinement

```
Fixpoint oracle_refines_to T
    (d1: State layer1_lts) (p: Layer2.prog T)
    (o1: Oracle layer1_lts) (o2: Layer2.oracle) : Prop :=
  oracle_ok (compile p) o1 d1 /\
    match p with
    | Alloc v =>
      if In Crash o1 then
        forall d1',
          Layer1.exec o1 d1 (compile p) (Crashed d1') ->
          (d1 0 = d1' 0 -> o2 = [Crash1]) /\
          (d1' 0 <> d1' 0 -> o2 = [CrashAlloc first_zero])
      else
        if first_zero bitmap < block_size then
            o2 = [BlockNum (first_zero bitmap)]
        else
```

```
          o2 = [DiskFull]
      end
  | Bind p1 p2 =>
    exists o1' o1'',
    o1 = o1'++o1'' /\
   ((exists d1',
      Layer1.exec o1 d1 (compile p1) (Crashed d1') /\
      oracle_refines_to d1 p1 o1 o2 /\ o1'' = [])
      \/
    (exists d1' r ret,
       Layer1.exec o1' d1 (compile p1) (Finished d1' r) /\
       Layer1.exec o1'' d1' (compile (p2 r)) ret /\
       (exists o2' o2'',
       oracle_refines_to d1 p1 o1' o2' /\
       oracle_refines_to d1' (p2 r) o1'' o2'' /\
       o2 = o2' ++ o2''))
  ...
  end.

Definition refines_to d1 d2 :=
  exists F, (F * rep d2)%pred d1.
```