Formal Idealizations of Cryptographic Hashing

Bill White[*]

January 18, 2015

Abstract

We describe a method for representing idealized cryptographic hashing functions using the module system of the proof assistant Coq. The technique allows one to reason about a hashing function as if it is injective. The module system is used to hide the representation and prevent one from computing preimages by analyzing the hash. The technique is imperfect, as it is likely preimages can be computed using a naive search. We also define a precedence relation on hash values.

1 Introduction

Cryptographic hash functions are designed to have certain properties. Among these properties are that it be infeasible to find two inputs which have the same hash and that it be infeasible to find a preimage of a given hash.

A particular such function used in the Bitcoin cryptocurrency [7] is SHA256 [8]. SHA256 hashes an item of data (sequence of bits) into a 256 bit number. This means that there are 2^{256} possible hash values. Simply using the pigeon hole principle, we know that there must exist two distinct inputs of 257 bits which hash to the same value. That is, SHA256 is clearly not injective. However, the fact that two such inputs can (apparently) not be found in an effective manner implies we can reason about SHA256 *as if* it were injective.

We show how the module system for Coq [5] can be used to give an idealized representation of cryptographic hashes. The technique allows one to assume hash functions have certain desired properties and then prove theorems using these properties. For example, hashing two different items is guaranteed to give different hashes. On the other hand, one cannot analyze hashes in any obvious way to compute a preimage. The technique is also imperfect as there are ways to compute preimages using a naive search, and this drawback is also discussed.

A few results are proven. For example, we prove that if we hash a pair of hash values, then the result is neither of the components of the pair. We also define a precedence relation on hash values and prove it is a strict partial

[*]Bitmessage: BM-2cWvQoqzQPJS8CvuRi7DZ2iCud4m9FB1HA
Email: billwhite@bitmessage.ch BTC: 12pbhpqEg7cjaCLLcvdhJhBWGUQWkRK3zS

```
Module Type BoolHashValsType.

  Parameter hashval : Type.
  Parameter hashbool : bool -> hashval.

  Axiom hashboolinj : forall m n, hashbool m = hashbool n -> m = n.

  Parameter hashval_eq_dec : forall (h1 h2:hashval),
                              { h1 = h2 } + { h1 <> h2 }.

  Axiom hashval_ind : forall p:hashval -> Prop,
                          p (hashbool false) ->
                          p (hashbool true) ->
                          forall h, p h.

End BoolHashValsType.
```

Figure 1: Interface for boolean hashes

ordering. The precedence relation could be used to prove that one cannot create a cryptocurrency transaction which uses a hash of one of its own outputs as an input.

Unlike cryptographic hash functions used in practice, the collection of hash values is infinite. Informally, we can think of 2^{256} as being a finite approximation of infinity and SHA256 as being an injection from the natural numbers into 2^{256}.

2 Hashing Booleans

We begin by consider the simple example in which we have two booleans and we want to be able to hash them. In Coq the booleans are given as the inductive type `bool` of booleans with constructors `false` and `true`. Suppose these are the only two items we will be interested in hashing.

We need to have a type `hashval` of hash values and a function `hashbool` mapping each boolean to a hash value. We want `hashbool` to be injective so that `false` and `true` are guaranteed to have different hashes. As a further criteria, we require that there is a way to compare hashes and decide if they are equal. Finally, since we are only interested in these two particular hash values, we assume there are no others. We can guarantee this by requiring a kind of induction principle: if a property p holds for the hash of `false` and for the hash of `true`, then it holds for all hash values. We show this specification as a Coq module type in Figure 1.

No matter how we implement a module of this module type, the specification above is all that will be exposed. This prevents us from directly analyzing hash values.

There is an obvious way to implement this specification: simply take `hashval` to be `bool` and `hashbool` to be the identity. The remaining components are easy to obtain. We give the corresponding module in Coq code in Figure 2.

Let us breifly consider how this module can be used (and how it cannot be used).

```
Module BoolHashVals : BoolHashValsType.

  Definition hashval : Type := bool.

  Definition hashbool := fun b : bool => b.

  Lemma hashboolinj : forall m n, hashbool m = hashbool n -> m = n.
    intros m n H. exact H.
  Qed.

  Definition hashval_eq_dec (h1 h2:hashval) : { h1 = h2 } + { h1 <> h2 }.
    destruct h1; destruct h2.
    - left. reflexivity.
    - right. discriminate.
    - right. discriminate.
    - left. reflexivity.
  Defined.

  Lemma hashval_ind : forall p:hashval -> Prop,
                      p false ->
                      p true ->
                      forall h, p h.
    intros p H1 H2 [|]; assumption.
  Qed.

End BoolHashVals.
```

Figure 2: Implementation of boolean hashes

We can easily use the induction principle `hashval_ind` to prove every hash value is the hash of some boolean. Let p hold for h if there is a boolean b such that `hashbool` $b = h$. By the induction principle we know p holds for every h if p holds for the hash of `false` and for the hash of `true`. It holds for the hash of `false` using `false` as the witness and holds for the hash of `true` using `true` as the witness. Here is the proof script in Coq.

```
Lemma preimage_exists : forall h:hashval, exists b:bool, hashbool b = h.
  intros h. apply (hashval_ind (fun h => exists b:bool, hashbool b = h)).
  - exists false. reflexivity.
  - exists true. reflexivity.
Qed.
```

Note that this lemma allows us to reason about preimages, but does not allow us to compute preimages.

A slight modification (using Σ-types instead of the existential quantifier) would allow us to compute preimages. That is, it would be a problem if we could complete the following definition.[1]

```
Definition preimage_sig : forall h:hashval, { b:bool | hashbool b = h }.
```

On a first glance it may appear that the same proof script above could be used to synthesize such a term. However, the induction principle `hashval_ind` cannot be applied here since the Σ-type is not a proposition.

For the same reasons, we cannot give a direct function computing the preimage. That is, a direct attempt to prove the following would fail:

```
Lemma preimage_computable : exists p:hashval -> bool, forall h:hashval, hashbool (p h) = h.
```

[1]We will shortly learn that the definition can be completed, and it is something of a problem.

```
Definition preimage_sig : forall h:hashval, { b:bool | hashbool b = h }.
  intros h.
  destruct (hashval_eq_dec (hashbool false) h) as [H1|H1].
  - exists false. exact H1.
  - destruct (hashval_eq_dec (hashbool true) h) as [H2|H2].
    + exists true. exact H2.
    + exfalso. revert h H1 H2.
      apply (hashval_ind (fun h => hashbool false <> h -> hashbool true <> h -> lse)); tauto.
Defined.

Lemma preimage_computable : exists p:hashval -> bool, forall h:hashval, hashbool (p h) = h.
  exists (fun h:hashval =>
            match preimage_sig h with
              | exist b _ => b
            end).
  intros h. destruct (preimage_sig h) as [b H1].
  exact H1.
Defined.
```

Figure 3: Preimages via search

The reason we have hedged by saying a *direct attempt* is that, in fact, preimages can be computed in Coq via a simple search. In the case of booleans, we simply check if `false` works and if not we check if `true` works. If neither works, there is a contradiction. The relevant Coq code is in Figure 3.

It is often the case that if we can enumerate the inputs, then we can search for a preimage without needing to analyze the hash value. A representation for which this is not possible would be preferable. At the moment, it is up to the user to simply avoid using such a search while formalizing algorithms. (Algorithms using such a search would be infeasible in any case.)

3 Hashing Natural Numbers

Hashing booleans is only a toy example to understand a simple case. A more realistic example is given by taking natural numbers instead of booleans. One can obtain an appropriate module type by (for the most part) simply replacing references to `bool` with references to `nat` (Coq's type of natural numbers). The only difference is in the induction principle since we cannot list all the natural numbers. Instead we use a universal quantifier to express that a property holds for all hashes if it holds for the hashes of all natural numbers. Note that we did not need to list the booleans in the previous example and could also have used the universal quantifier there. The module type is given in Figure 4.

Given an implementation of this module type, `hashnat` is an injection from the (infinite) type `nat` into `hashval`. Therefore, the type `hashval` must necessarily be infinite.

There is again an easy implementation by taking `hashval` to be `nat`. The module interface again prevents one from directly analyzing hash values, although one could likely get around this using search.[2]

[2]The author has not implemented such a search algorithm in the Coq code, so this is left

```
Module Type NatHashValsType.

  Parameter hashval : Type.
  Parameter hashnat : nat -> hashval.

  Axiom hashnatinj : forall m n, hashnat m = hashnat n -> m = n.

  Parameter hashval_eq_dec : forall (h1 h2:hashval), { h1 = h2 } + { h1 <> h2 }.

  Axiom hashval_ind : forall p:hashval -> Prop,
                        (forall n, p (hashnat n)) ->
                        forall h, p h.

End NatHashValsType.
```

Figure 4: Interface for natural number hashes

4 Hashing Pairs

In one sense, hashing natural numbers should be enough since we can encode other data structures of interest as natural numbers. However, sometimes we may want to hash data which includes hash values. If we could only hash natural numbers we would need a way to turn a hash value into a natural number. (This is trivial in practice since hash values are bit sequences of a certain length, but would require more work while using some idealized hash values.) Mapping back from abstract hash values to concrete natural numbers seems to negate the philosophy of abstracting the hash values in the first place.

Instead we can extend the module type to allow for hashing of either natural numbers or pairs of hash values. This gives the ability to formalize, for example, Merkle trees [6]. More generally, we can hash structured data by recursively hashing the components. Two examples are given in Section 6.

In order to incorporate hashing of both natural numbers and pairs of hashes, we will need two functions for obtaining hashes both of which must be injective. We must also ensure that numbers and pairs never hash to the same value.

The induction principle says that a property p holds for every hash value if it holds for the hash of every natural number (the leaves of the tree) and holds for the hash of every pair assuming it held for the two components of the pair. The assumption that it holds for the two components of the pair correspond to inductive hypotheses, so this is the first case where it is proper to call the principle an *induction* principle. The module type is given in Figure 5.

The way to implement the module is not quite as obvious in this case, but it is also not difficult. An inductive type `hashval'` of the following form has the properties we expect of `hashval` in this case.[3]

```
Inductive hashval' : Type :=
| hashnat' : nat -> hashval'
| hashpair' : hashval' -> hashval' -> hashval'.
```

as a conjecture.

[3]The modifications to the names is required because of an implementation detail in the Coq module system which is not important here.

```
Module Type TreeHashValsType.

  Parameter hashval : Type.
  Parameter hashnat : nat -> hashval.
  Parameter hashpair : hashval -> hashval -> hashval.

  Axiom hashnatinj : forall m n, hashnat m = hashnat n -> m = n.
  Axiom hashpairinj : forall h1a h1b h2a h2b, hashpair h1a h1b = hashpair h2a h2b ->
                                                  h1a = h2a /\ h1b = h2b.
  Axiom hashnatpairdiscr : forall m h1 h2, hashnat m <> hashpair h1 h2.

  Parameter hashval_eq_dec : forall (h1 h2:hashval), { h1 = h2 } + { h1 <> h2 }.

  Axiom hashval_ind : forall p:hashval -> Prop,
                     (forall n, p (hashnat n)) ->
                     (forall h1, p h1 -> forall h2, p h2 -> p (hashpair h1 h2)) ->
                     forall h, p h.

End TreeHashValsType.
```

Figure 5: Interface for hashes with pairs

5 Addresses

A bitcoin address is essentially a 160-bit number (obtained as a hash of a public key). This is a very different use of a hash than the kind considered before. In future work we will represent the state of a cryptographic ledger as a function from addresses to lists of assets. It will be important that there are only finitely many addresses (even if the number 2^{160} is huge).

Since addresses are fundamental building blocks of cryptocurrencies, we include basic hash functions for addresses as well.

In Coq we can represent addresses as sequences of booleans of length 160. Many functions on addresses (as well as other data structures) will be defined by recursion on the length of the sequence. We can generally define $\mathcal{B}_n$ as the type of bit sequences of length n using a dependent type `bitseq` in Coq.

```
Fixpoint bitseq (n:nat) : Type :=
match n with
| 0 => unit
| S n' => (bool * bitseq n')%type
end.
```

In particular the decision procedure for deciding equality of bit sequences is by recursion on n. (The algorithm is the obvious one: if n is 0, then the two sequences are equal; if n is $n' + 1$, then test if the first booleans are equal and, if they are, recursively call on the tails which are in $\mathcal{B}_{n'}$.)

The set $\mathcal{A}$ of addresses is $\mathcal{B}_{160}$, i.e., `bitseq` 160. Clearly equality of addresses is decidable.

In Figure 6 we show the module type for hashes including natural numbers, addresses and pairs of hash values. As before, one can use an inductive type to implement the module. The details can be found in the Coq code, but the point of ascribing the module to the module type is that the information in the

```
Module Type HashValsType.

  Parameter hashval : Type.
  Parameter hashnat : nat -> hashval.
  Parameter hashaddr : addr -> hashval.
  Parameter hashpair : hashval -> hashval -> hashval.

  Axiom hashnatinj : forall m n, hashnat m = hashnat n -> m = n.
  Axiom hashaddrinj : forall alpha beta, hashaddr alpha = hashaddr beta -> alpha = beta.
  Axiom hashpairinj : forall h1a h1b h2a h2b, hashpair h1a h1b = hashpair h2a h2b ->
                                                                h1a = h2a /\ h1b = h2b.

  Axiom hashnataddrdiscr : forall m alpha, hashnat m <> hashaddr alpha.
  Axiom hashnatpairdiscr : forall m h1 h2, hashnat m <> hashpair h1 h2.
  Axiom hashaddrpairdiscr : forall alpha h1 h2, hashaddr alpha <> hashpair h1 h2.

  Parameter hashval_eq_dec : forall (h1 h2:hashval), { h1 = h2 } + { h1 <> h2 }.

  Axiom hashval_ind : forall p:hashval -> Prop,
                       (forall n, p (hashnat n)) ->
                       (forall alpha, p (hashaddr alpha)) ->
                       (forall h1, p h1 -> forall h2, p h2 -> p (hashpair h1 h2)) ->
                      forall h, p h.

End HashValsType.
```

Figure 6: Interface for hashes with addresses and pairs

module type is all one needs to know in order to use it. Put another way, the information in the module type is all one is allowed to use.

For the remainder of the paper, we assume we have a module of the type in Figure 6 given, so that we can freely use its components.

In mathematical notation, we use $\sharp\cdot$ for all the hashing functions. That is, $\sharp n$ means the hash of the natural number n via `hashnat`, $\sharp\alpha$ means the hash of the address α via `hashaddr`, and $\sharp(h_1, h_2)$ means the hash of the hash pair (h_1, h_2) via `hashpair`.

As a first result we prove the following.

Theorem 5.1. *$\sharp(h_1, h_2) \neq h_1$ and $\sharp(h_1, h_2) \neq h_2$*

Proof. The two conjuncts are proven by indepedent inductions using the induction principle `hashval_ind`. We only prove the first here. (The proof for the second is analogous.)

Let p hold for h if $\forall h_2.\sharp(h, h_2) \neq h$. We must prove p holds for hashes of natural numbers, hashes of addresses, and hashes of pairs (under inductive hypohteses). The result follows immediately for natural numbers and addresses since h cannot be the hash of a pair if h is the hash of a natural number or address. Now suppose h is $\sharp(h', h'')$. Assume $\sharp(\sharp(h', h''), h_2) = \sharp(h', h'')$. By injectivity of `hashpair` we know $\sharp(h', h'') = h'$. This contradicts the inductive hypothesis for h'. ☐

6 Hashing Structured Data

We can define auxiliary hashing functions for structured data. For example we can define a function hashlist to hash a list of hash values by an easy recursion.

```
Fixpoint hashlist (hl:list hashval) : hashval :=
match hl with
| h::hr => hashpair h (hashlist hr)
| nil => hashnat 0
end.
```

An easy recursion verifies that hashlist is injective. In mathematical notation we write $\sharp\overline{h}$ for the hash of a list $\overline{h}$ of hash values.

Next consider the type option hashval which represents all the hash values plus an extra None value.[4] We can naturally lift the notion of hashing to include None by always hashing None to be None.

Here we show how to hash a pair of optional hashes to obtain an optional hash.

```
Definition hashopair (h1 h2:option hashval) : option hashval :=
match h1,h2 with
| None,None => None
| Some h1,None => Some (hashpair (hashnat 0) h1)
| None,Some h2 => Some (hashpair (hashnat 1) h2)
| Some h1,Some h2 => Some (hashlist (hashnat 2::h1::h2::nil))
end.
```

Note that hashing a pair of None values to None. If at least one of the two is a real hash value, then we hash the data along with a natural number which is used to tag whether the first is None, the second is None, or both are real hash values. Again hashopair can be proven to be injective.

Along the same lines one can extend hashing to allow for hashing of more complex structured data such as cryptographic transactions and ledgers.

7 A Precedence Relation

Suppose h is a hash value representing some asset (e.g., a transaction output in Bitcoin). A transaction spending this asset would contain h as part of its data. Suppose h' is the hash of such a transaction. There is an informal sense in which h *precedes* h'. Our goal in this section is to make the definition of the *precedes* relation formal and precise. We write $h \prec h'$ in mathematical notation and subh as the name of the corresponding notion in Coq. Mathematically the definition of $\prec$ can be given using the four rules in Figure 7.

The corresponding definition in Coq is given as an inductive proposition as follows:

[4]In future work, None will be used as the "hash" of empty data. This is important since cryptographic ledgers are, by design, very sparse data structures.

$$\frac{}{h_1 \prec \sharp(h_1, h_2)} \qquad \frac{}{h_2 \prec \sharp(h_1, h_2)} \qquad \frac{h \prec h_1}{h \prec \sharp(h_1, h_2)} \qquad \frac{h \prec h_2}{h \prec \sharp(h_1, h_2)}$$

Figure 7: Definition of $\prec$ by rules

```
Inductive subh (h:hashval) : hashval -> Prop :=
| subh_L h' : subh h (hashpair h h')
| subh_R h' : subh h (hashpair h' h)
| subh_PL h1 h2 : subh h h1 -> subh h (hashpair h1 h2)
| subh_PR h1 h2 : subh h h2 -> subh h (hashpair h1 h2).
```

An easy first result one can prove is that if h is a hash value in a list $\overline{h}$ of hash values, then $h \prec \sharp\overline{h}$. The proof is by induction on the list.

For the remainder of the section we state a number of results about $\prec$ and indicate how the results are proven. The proofs are available in the Coq code.

Theorem 7.1. *$\prec$ is transitive: if $h_1 \prec h_2$ and $h_2 \prec h_3$, then $h_1 \prec h_3$.*

Proof. By induction on $h_1 \prec h_2$ we prove

$$\forall h_3 . h_2 \prec h_3 \rightarrow h_1 \prec h_3 .$$

There are four cases corresponding to the four defining rules in Figure 7. In each case we do a subinduction on $h_2 \prec h_3$. □

Theorem 7.2. *$\prec$ is irreflexive: $h \not\prec h$.*

Proof. This can be argued by the induction principle on hash values. □

Theorem 7.3. *$\prec$ is asymmetric: if $h \prec h'$, then $h' \not\prec h$.*

Proof. This follows easily from transitivity and irreflexivity of $\prec$. □

To end the section we return to the example with which we began. Suppose a transaction is represented by a pair of $\overline{h}$ (a list of hash values representing input assets of the transaction) and $\overline{k}$ (a list of hash values representing outputs of the transaction). Let $h' := \sharp(\sharp\overline{h}, \sharp\overline{k})$ be the hash of this transaction.

Suppose h is the hash value of some asset in the input list $\overline{h}$. Note that $h \prec \sharp\overline{h} \prec h'$ and so $h \prec h'$. That is, h precedes h'.

Going one step further, consider the i^{th} output of the transaction. This output can be represented by the hash value $h'' := \sharp(h', \sharp i)$. Clearly we have $h \prec h''$ as well. By irreflexivity we know h'' is not h. In other words, the assets represented by transaction outputs are always different from the assets spent in the transaction. These kinds of properties will be important in future work when we prove properties of hashes of these basic building blocks of cryptocurrencies.

8 Related Work

Any attempt to list the Coq formalizations that influenced this work must be incomplete. To keep the list short we simply note that several books and lecture notes [3, 4, 11, 9] have provided helpful introductions to Coq and given interesting examples demonstrating a variety of techniques.

One of the main motivations for this work is to provide secure underpinnings to some cryptocurrency ideas. Earlier work formalizing some cryptocurrency notions in Coq was done by Consensus Research [10]. The papers [1, 2] describing a formally verified Proof of Stake simulator are of particular note.

9 Conclusion and Future Work

We have described a way to represent and use an idealization of cryptographic hashes in the proof assistant Coq. The representation ensures hashes have unique preimages and ensures that preimages cannot be computed in a direct manner. In addition we have defined a precedence relation on hash values and proven that it is a strict partial ordering.

In future work we plan to formally define the state of a cryptographic ledger as a function from addresses to lists of assets. Such a state can be modified by transactions which spend some assets from some addresses in order to create new assets at possibly different addresses. The state function can be approximated by Merkle trees and these Merkle trees can be acted upon directly by transactions. We plan for all the results to be proven in Coq and to make the Coq code available.

References

[1] andruiman. PoS forging algorithms: formal approach and multibranch forging, 2014.
`github.com/ConsensusResearch/articles-papers/blob/master/multibranch/multibranch.pdf`

[2] andruiman. PoS forging algorithms: multi-strategy forging and related security issues, 2014.
`github.com/ConsensusResearch/articles-papers/blob/master/multistrategy/multistrategy.pdf`

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[4] Adam Chlipala. *Certified Programming with Dependent Types.* MIT Press, 2011.

[5] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2012. Version 8.4.

[6] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology CRYPTO 87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer Berlin Heidelberg, 1988.

[7] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[8] NIST. FIPS PUB 180-4: Secure Hash Standard (SHS).
`csrc.nist.gov/publications/fips/fips-180-4/fips-180-4.pdf`

[9] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014.
`www.cis.upenn.edu/ bcpierce/sf`

[10] Consensus Research.
`consensusresearch.org`

[11] Gert Smolka and Chad E. Brown. Introduction to Computational Logic (lecture notes), 2014.
`www.ps.uni-saarland.de/courses/cl-ss14/script/icl.pdf`