Temporal signatures

A W Roscoe
Chieftin Lab, Shenzhen
and Oxford University Department of Computer Science

October 23, 2017

Abstract

We present a new way of constructing cryptographic signatures using the combination of simple hashing and reliable time-stamping as provided, for example, by the blockchain. While achieving the conventional objectives of cryptographic signature, its mode of operation is so different (not being based on public and secret keys) that it does not satisfy the formal definition. Therefore we name it Temporal Signature.

1 Introduction

At the time of writing (2017) we are seeing both challenges and opportunities in the world of cryptography and computer security. One of these is the increasing focus on post-quantum cryptography thanks to the ever wider awareness of the prospective problems caused by future quantum computers. A second is the emerging popularity of blockchains, a model with very interesting timing and security properties.

The work in this paper was motivated by the desire to have an efficient means of proving message authentication, integrity and repudiation that is quantum resistant. Rather than following the route of devising new types of public key cryptography we wanted to find something that depends only on basic symmetric cryptography and hashing as there is a strong consensus that these are quantum resistant.

While methods of hash-based signature, notably one-shot schemes such as Lamport's [**?**] and similar, which can be enhanced by Merkel trees, have been known for a significant time, they are expensive and typically stateful, in the sense that a given key pair is hard to distribute between different instances of an identity.

Ours does not have this disadvantage, but it does make an additional assumption about the availability of a service that can be provided by a blockchain, as we will discuss later.

We present two variants of our scheme: in one each key is *ab initio* associated with a future time, and another these times are created later. We also demonstrate how analogues of PKI and Certification Authorities (CAs) can work in our new space, including providing an analogue of zero-knowledge proof.

This is one of a pair of papers introducing ideas in post-quantum cryptography. In the second we will introduce an analogous idea for key exchange which similarly relies on absolutely standard symmetric cryptography plus an extra piece of technology.

The method we describe is extremely efficient in the amount of cryptographic calculation required and therefore offer prospects of security to applications such as IoT where asymmetric cryptography is barred on cost grounds rather than because of the worry of future quantum computers.

Our schemes are both based on circumstances where A's signature is simply the hash of what is to be signed with a key or nonce that A knows at that time but no-one else does, but which B will know later and further more know it was uniquely associated with A. In that sense it resembles the TESLA stream authentication protocol [] and the interactive authentication scheme of []. However we are able to develop a related idea into a full signature scheme complete with certification authorities primarily because of the popularity of architectures such as the blockchain that establish an unambiguous form of common knowledge and time stamping.

2 Time-based signature

In this section we show how to create a means of "cryptographic signature" based only on hashing. It is not the first such, since there are well established ones such as Lamport Signatures []. Ours is quite different and has the advantages that there is no bound on how many times a given key can be used, and that it is extremely cheap to use.

Our model of signature will work in blockchain systems and ones where there is a trusted third party operating a bulletin board with time-stamps, or indeed any structure that achieves the model of a universally writable time-stamping database where all nodes get a consistent view and, given any time-stamp t, can find a moment beyond which all future data will get a time-stamp strictly greater than t.

We postulate the following model of the blockchain. It is a write-only database to which any party can write and which is agreed in blocks. For each block of data there is a point where it is agreed and becomes an irrevocable part of history to which all parties agree. All events have a time-stamp t (which may just be the block number) which strictly increases from one block to the next.

In most blockchain models there is a notation of a transaction which comprises all or most entries, and the parties making up the blocks (miners) have to ensure consistency of transactions. That is not essential in our world, though our mechanism can still be used in such contexts, and indeed consistency checking can be used to support it as we will see later.

Indeed for our signature scheme to work we do not need a full consensus-driven blockchain. What we require is a trustworthy write-only bulletin board which assigns times monotonically to the items that are posted on it, and where all parties can read what is posted on the board. When node A posts a message it knows it is going to get a later time-stamp than any presently published on the bulletin board or blockchain, and A knows an upper bound on the time-stamp it will get relative to the present time.

In neither case do we require the central database to verify the writer of data placed on it. We think of these as unsigned packets. However the signature scheme proposed below would allow these to be verified shortly after the writes.

We are all familiar with digital signature schemes based on one-way functions to the extent that the very definition of a cryptographic signature assumes the existence of private and public keys. Ours does not. Thus while our scheme satisfies the requirements at the start of the Wikipedia article on digital signatures [**?**]:

> A digital signature is a mathematical scheme for demonstrating the authenticity of digital messages or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (non-repudiation), and that the message was not altered in transit (integrity).

We term this the *extensional* definition, since it sets out what has to be achieved without being specific about how it is to be done.

Our method does not, however, operate in the way implied by the "formal definition" given there:

> Formally, a digital signature scheme is a triple of probabilistic polynomial time algorithms, (G, S, V), satisfying: G (key-

generator) generates a public key, pk, and a corresponding private key, sk, on input 1^n, where n is the security parameter. S (signing) returns a tag, t, on the inputs: the private key, sk, and a string, x. V (verifying) outputs *accepted* or *rejected* on the inputs: the public key, pk, a string, x, and a tag, t. For correctness, S and V must satisfy

$$Pr[(pk, sk) \in G(1^n), V(pk, x, S(sk, x)) = accepted] = 1$$

A digital signature scheme is secure if for every non-uniform probabilistic polynomial time adversary, A

$$Pr[(pk, sk) \in G(1^n), (x, t) \in A^{S(sk,.)}(pk, 1^n), x \notin Q, V(pk, x, t) = accepted] < negl(n)$$

where $AS^{(sk,.)}$ denotes that A has access to the oracle, $S(sk, .)$, and Q denotes the set of the queries on S made by A, which knows the public key, pk, and the security parameter, n.

Note that we require that any adversary cannot directly query the string, x, on S.

We term this the *intensional* definition of a signature, since it strongly implies how the objective represented by the extensional definition is to be achieved. There is an interesting discussion in [**?**] of the relationship of a similar form of signature to the intensional one.

We term this *temporal signature*: it is based on simple but completely different ideas to the usually seen one.

We will couch this signature in terms of the blockchain or TTP running a bulletin board, but it will apply to any situation where players can unambiguously publish packets in a common medium where all get a common view of what has been written and when, and the fact that this is so is common knowledge.

We will assume that the time at the start of operation is 1, and that the blockchain or bulletin board is initialised with a special block or data written by some privileged initialisation or by the individual nodes at some point where they are trusted. This information has time 0 which strictly precedes all times of ordinary written data.

This initial block contains, for each node A, a finite set of key certificates of the form[1] $(hash(k, A, t), t, A)$ where k are chosen at random from a large

[1] In using a standard cryptographic hash function here, we are assuming perfect cryptography in the the forms of the terms used. It may well be wise to use different forms of the terms (e.g. reordering or replicating components of the hashed terms) to counter weaknesses in specific (e.g. iterative) hash constructions. Of course careful and conservative choices of the hash functions themselves is recommended.

set and t varies over future times. Thus each node will typically have keys labelled by a spread of (initially) future times.

The fundamental idea is that the use of k (as opposed to $hash(k, A, t)$ by someone before time t proves that this someone is A, and that A will release k at time t allowing anyone to check such uses, deducing they were by A.

Thus to sign X with k we have A compute $hash(k, A, X)$ and place it on the blockchain or bulletin board before t and so it gets a stamp of less than t.

Of course A can compromise her own signatures by releasing k early, but this would be no different to her disclosing her conventional secret key. Doing so would damage her in much the same way that disclosing her conventional secret key. It is her duty to write k into the blockchain when she knows it will get a stamp of at least t. (So in the case where the time stamp is the index of the block, this could be when block $t - 1$ has appeared.)

Note that once the keys are established, signature only requires the computation of a single cryptographic hash, and verification, where the key belongs to the initial set, requires at most two hashes. These are the hash of the data with the now-revealed key, and the hash that verifies the same key whose hash was initially published.

We have the option to save (at least then) one of these hashes by verifying each key via the consensus mechanism at the point it is published openly. In other words, when A places k openly in the blockchain, it should be apparent to all that this is a reveal of a key, and those responsible for consensus and block creation must verify k before publishing it.

3 Refreshing keys

Since signatures can only be checked once the appropriate time has been reached, in most case it is desirable, when signing a message, to use a nearly expired key. Given that each node is presumably only allowed a finite number of initial keys, such a key from that set is not always available. However it is straightforward to allow A to add further keys.

Suppose A will shortly enter a time-frame where it has no keys or only a few, but it still has a key (A, k, t) that will expire before this. Then she can create as many new keys randomly (or perhaps pseudo-randomly) as she wishes, choosing a future time for each, and sign their certificates (either collectively or individually) as new keys for A, each with their own time beyond t.

The strategy for doing this will depend on the expected time-frame of the service being created, namely for how long will an agent wish to create signatures. This might be for some finite period that can be divided up into equal parts *ab initio* with this division and further subdivisions used to structure the key space. Or is might be indefinite, in which case one can do the same but, near the end of an initially chosen epoch, new keys are signed for another. By (say) doubling the lengths of successive epochs, it is easy to keep the chain required to check each key down to logarithmic length.

So for example each node could be initialised with keys that are revealed at 2^n for $n \leq N$ for some arbitrary $M > 0$. Note that initially the gap between consecutive keys is a power of 2. We will maintain this as an invariant and also the fact that the largest t is a power of 2,

At each time $r = \sum_{i=0}^{n-1} \delta_i 2^i$. When the current time is (say) one less than the last time before a gap of length more than one $(r, r+2^s)$, it creates new keys for the times $r+1, r+2, \ldots r+2^{s-1}$. as far as it has to to maintain this invariant.

So it will create new keys as follows:

- at time 1 for 3,
- at time 3 for 5 and 6,
- at time 5 for 7,
- at time 7 for 9,10,12,
- and so on...

When it reaches $2^N - 1$ it adds a key key for 2^{N+1} at the end.

With this approach, every time a node uses a new key it can check it by following a chain back to time 0 with length bounded by $2log_2\, t$. To understand this, observe that every integer m greater than 0 can be expressed uniquely in the binary form $m = S_r = \sum_{r=0}^{p} 2^{q_r}$ where the q_r are a strictly decreasing decreasing sequence. Notice that both q_0 and p are no greater than $log_2 m$. In general the key for time $S_a = \sum_{r=0}^{a} q_r$ $(0 < a \leq p)$ was signed with the key for time S_{a-1}, and the key for 2^b was created either at time 0 or signed using time 2^{b-1}. Thus the checking chain is composed of two parts neither of which is longer than $log\, m$.

Of course we have the same option about checking keys as before: those building a blockchain are responsible for ensuring that the public versions of keys are consistent with their previously published hashes.

In other words if a new key $(hash(A,k,t),A,t),((hash(A,k,t),A,t,k'),ref(k'))$ is published signed by A using k' which is revealed at $t' < t$ then all 'mining' nodes should check this at t' resulting in a certificate of agreement that k is a good key for A to be revealed at t'.

when k is ultimately released at t, it should be checked as outlined above.

We could even extend the role of the blockchain to verifying all individual signatures (rather than merely signatures on keys), but this is of less obvious benefit.

4 Checking authorship

An interesting question is whether relying, as mooted here, on the blockchain to check the accuracy or keys and signatures (as opposed to each node doing it by themselves) creates any more risk than may be present because of allowing the blockchain to arbitrate time.

In the description above, we have assumed that data placed in the system is not authenticated to its author. Our signature mechanism provides the means for a TTP or consensus protocol to check the authorship of any item before it is openly published, thus providing such authentication. To do this we partition the bulletin board or blockchain space into to two parts, that we will describe as *authenticated* and *provisional*. An ordinary application running in a node can only read *authenticated*, and write to *provisional*. Such writes are accompanied by the temporal signature, for the next time t, of the written data by its name. When the corresponding $k_{A,t}$ is released at t, the TTP or consensus protocol can verify the signature and move the write into *authenticated*.

Of course in such contexts there might still be the need for individual nodes to be able to check other people's signatures, for example when the signed item is sent inside some other cryptographic construct such as encryption.

5 An alternative approach

We have presented a model in which the time of each key is committed in advance. An obvious alternative is for keys to be created and their hashes bound to an identity without a fixed release time. The identity could then sign one of more files with the key in the same way as above: placing the signed document in the blockchain or bulletin board and only releasing the key into the same medium once it has seem all the signed documents there.

Imagine the following scenario: Alice has created key k, signing the key certificate $(A, hash(A, k))$ and placing it on the bulletin board. After the signature of the certificate has been checked, she signs a number of documents with k and places them $(F_1, \ldots, F_n)$ on the bulletin board. Once she has seen they are all there she places k on the bulletin board.

Eve can potentially delay the final write while learning k, and sign further documents with it "on behalf of" Alice before finally releasing k. If it then can block Alice from intervening (or if Alice does not notice quickly) the signatures will be believed.

This model can be rescued: one solution is to have Alice use each such keys only once. By checking that the signed document is on the blockchain before releasing the key, making the blockchain consensus prevents "double spending" of k appears to prevent attacks.

Extending this: Alice can count how many files she has signed with k at the same time as checking they are all there. She then signs a specially formatted message such as *Alice has signed 23 files with* $hash(A, k)$ on the blockchain – signed with k – and checks it is there before releasing k. The consensus protocol only accepts k when the numbers tally. Here we are using the properties of our blockchain/bulletin board in being immutable and permanent once a write is made and agreed. As a further alternative she can replace the count with the simple message *No more signatures with* $hash(A, k)$, again signing it with k. The consensus protocol only accepts the release of k when this consistent. In both cases checking the signature on the message ending k means that Eve cannot successfully end k early.

As an alternative to counting, she can observe the final time stamp t of one of the files signed by k in the blockchain, and place a message on the blockchain which associates this time with the key. Again Alice makes sure this message is there before releasing k. To verify a signature it is verified that the signature has a time stamp no later than t. Essentially here we are replacing the original pre-committed time for each time, by one that is post-committed.

Refreshing keys is now much simpler than in the earlier case.

This method has the advantage of using keys efficiently: no key gets wasted if there is nothing to sign with it at the time it is released. It has the disadvantage of only really working with a blockchain consensus algorithm or something very like it. It also needs more bookkeeping and might not work as well when a given identity has more than one instance generating signatures with a given key.

6 Public signatures, private data

An essential feature of our models is that signatures are placed on a commonly write-able, commonly readable time-stamped database. This might seem to limit their usability, since in many applications agents will wish to sign data that is only meant to be seen by less than all agent, often only one other.

Nevertheless we really everyone to be able to see all signatures and identify their time-stamps and link them to their signing keys. In cases where we want to have signatures checked by external parties such as blockchain consistency, those doing the checking apparently need access to what it being signed.

We can, however, overcome these problems easily by adopting the principle that, other than when signing the keys used in the signature scheme, which have their own protocols, it is $hash(X)$ that is actually signed in the senses set out above.

So we can decide that the signature of X by key k is actually $(hash(X), hash(hash(X), k))$ coupled perhaps with a key certificate for k. This allows the signature to be checked by anyone without them knowing X. And of course anyone who thinks they know X can hash it to check this equals the value in the signature.

Thus we can maintain signatures in public without making the underlying data public too.

Just as with public key signature, if we want to stop an attacker checking a guess at X by hashing it, it may be necessary to add random bits as salt to X to prevent this, naturally communicating such salt to those intended to know X.

7 TKI or "PKI"

It is possible for one party to sign another's keys as a token of validity in our temporal setting just as much as it is with asymmetric signature.

A trusted third party can sign a certificate attesting that one or more timed keys belonging to A are valid. Just as in an ordinary PKI such a "Timed Key Infrastructure" can indicate limitations on the validity of the keys it is signing. It can place a limit on how long or how many times it permits A to refresh its own keys, or indeed ban such refreshing altogether. Time is much more built into the core principles of a TKI than it is into a PKI.

Note that the chain of trust in this case goes back to the start of time through the keys of multiple parties.

- For a key k for A, it might have a chain of trust going back through multiple A keys to one attested to A a time 0, or it might go back to one attested to one signed at a later time by key $kS0$ of a key server $S0$...
- Which is either attested back to time 0 by keys for $S0$, or eventually back to one attested by a second server $S1$,
- and so on. All the key servers themselves must be certified in their roles, just as in a regular PKI, and the chain of trust must get back eventually to time 0.

As discussed above, the system or consensus mechanism can check signatures and roles as time progresses.

In a PKI it is not the secret key but the public key that is signed, which limits the potential mischief a compromised key server can cause. In a TKI the same is true: all the key server has to do is to sign the key certificate with the hashed key in, not the open key. [Of course the mechanism by which the server comes to know which identity the certificate is for, and that it is valid, remains application dependent much as with PKIs.]

In some cases where signatures are validated by the blockchain, it might be appropriate to let the blockchain act like as a CA, in the sense that a node can provide one or more unsigned key certificates and the blockchain votes (presumably in receipt of evidence of some sort) to accept the keys and place them in the valid area. That would be analogous to putting such keys into an extension of block 0.

We have not addressed here the question of how a new node Alice proves her authenticity to a CA. To some extent this is always going to be an ad hoc process which assumes (rightly or wrongly) that at the moment of proof Alice has an authentic channel to CA. One thing that is commonly done with traditional CAs is for Alice to ask CA to sign her public key and at the same to prove that she knows her secret key via a zero-knowledge proof. The zero-knowledge proof approaches we are aware all use constructs in the asymmetric cryptography domain such as exponentiation. The well-known ones are vulnerable to quantum computers. Therefore we offer the following hash-based alternative, which while not providing a *proof* of knowledge, at least gives arbitrarily strong evidence that the intruder does not know the secret key.

In the following Alice is trying to construct something that she can use in our schemes or something similar.

1. Alice chooses some M, which for the sake of convenience is even, and random nonces N_i for $i \in \{1, \ldots, M\}$.

2. She sends these to CA, all in the form $hash(A, N_i, t)$ or $hash(A, N_i)$ depending on whether or not there is a time t attached to this key.

3. CA then picks $M/2$ indices at random from $\{1, \ldots, M\}$, and so has $C(M, M/2)$ choices, or approximately $(2^M\sqrt{2})/\sqrt{(}\pi M)$. It informs Alice of its choice as a challenge set C.

4. Alice then reveals to CA exactly the N_i for $i \in C$, and CA then verifies this choice.

5. CA then signs the rest of the hashes sent in the second step as Alice's key.

6. To sign an item with this key, Alice forms the hash of the object being signed with all the N_i not previously revealed.

In order to impersonate Alice by substituting his own choice of key, that Alice has not picked and so does not know, he would have to guess correctly at the set C. For M=16 this gives him a chance of one in 12,870 and for M=32 one in 601,080,390. The protocol establishes that no intruder can know all the precursors of the hashes not in the challenge except for this small probability. It does not establish that Alice knows of these with the compliment.

Alice can raise the probability to this by adding the following to the above exchange: As well as answering the challenge she picks a new key k_A for herself and places the hash of the concatenation of $Alice, hash(Alice, k_A)$ (and a time if appropriate) with all the N_i not included in the challenge on the blockchain. When it sees this published it releases the set of N_i (whose hashes have been signed by the CA). This does prove, to 1 minus the probability above, that Alice knew all the N_i and establishes $k_A/hash(k_A)$ as a key for her. The CA can now sign $hash(A, k_A)$ safe in the knowledge that it was created by Alice. (Note that more than one k_A can be established in the same message.)

We imagine that this approach to zero knowledge proof might be useful in other quantum-resistant contexts.

8 Temporal signature in support of the blockchain

It is already clear how temporal signature can use the blockchain to establish the temporal relationships required, and how items thus signed (including key certificates, both self-signed and signed by certification authorities) can be placed in it.

We need to maintain the integrity of the blockchain, so the communications that form and agree blocks need themselves to be signed. The mechanisms already described can be used for such signatures so that the integrity of the structure can be checked post hoc. However the inability a node B to check A's signature in the same time frame that it was created is an obstacle to the active use of temporal signature in the formation of blocks.

Fortunately nothing can make B accept a fraudulent signature, only have to defer checking it. We see several possible approaches to coping with this, including the following.

1. Where there is a secure distributed clock, separate out the release times of keys from the individual blocks. In this way a key might be released at a particular time, during the period where some block is being created. Note that no strong security is required for the release of unhashed keys, as they are already secured by the earlier releases of their hashes. B will then be able to check any signature it has seen before the release time of its key.

2. Ensure that all keys that are to be used in support of assembling a block will be released immediately that block is agreed. When it is agreed the formation can be checked. Ensure that any node performs this check on the block formation before using its data.

3. Use a non-temporally bound method of hash-based signature such as Lamport signatures for signing messages in this phase. The keys required for this can be signed by temporal signature provided this is validated before the block in which it is required.

This topic is one for further research. Certainly our new type of signature should be extremely helpful in creating a variety of quantum-resistant blockchains.

9 Conclusions

The mechanisms we have provided here give conclusive proof that the claimed party has signed some data, on the following assumptions

- The blockchain or bulletin board provides a trustworthy service. Standard signature mechanisms do not require a TTP, though of course PKIs do.
- The key structures that have been bootstrapped at time 0 are accurate. Note that the possibility of a TKI means that we do not have to deal with all identities at time 0.
- If it is not to be compromised, a node must obey the rules on when and when not to release things, and the system or blockchain must give reliable information on this.
- The hash function has strong collision freeness, to a high degree of confidence.
- Nodes adequately refresh their own keys or use a TKI to do so.

We have shown how a TKI, the analogue of a PKI, works. Signing and signature checking are now very easy, particularly if the provenance of keys is checked as time progresses. We have shown that there are potentially issues where this form of signature is used as part of the mechanism use in building blocks in the blockchain.

To support this we have created a version of zero-knowledge proof that can establish the relationship between an agent Alice and the key that a CA signs for her.

It is fascinating that a completely different model of signature exists which as far as we aware is new but so simple. We look forward to seeing what effect this discovery has on practice.

Acknowledgements

This work has been improved by discussions with Cas Cremers, Bangdao Chen, Wang Lei and Peter Ryan.