Temporal signature in the blockchain

A W Roscoe
Chieftin Lab, Shenzhen, and
University College Oxford Blockchain Research Centre

Abstract

Every blockchain requires a form of cryptographic hash to maintain its basic integrity, linking blocks as well as contributing to cryptographic signature. A secure and long lasting blockchain provides a secure and linear notion of timestamping. In this paper we explain how the latter can be used in constructing novel hash-based signatures and investigate whether such signatures can be used to maintain the structure of the blockchain.

1 Introduction

A blockchain is a linked list of data blocks, each being within some size limits (typically up to a few Mb) built using a specified format though with no general limitation on what it can contain, with each containing a strong cryptographic hash of its predecessor. The identity of the blockchain contains the contents of block 0. With that as its root, the blockchain is thus a tree rooted at Block 0: it is limited to the set of compliant blocks where the hash pointers eventually lead to this root. Each blockchain is also associated with rules about how new blocks are created in a decentralised and distributed environment, which the said rules always encouraging agreement on a linear sequence of blocks starting with Block 0, where all branching away from that sequence is eventually disregarded. The hash function is assumed to be preimage resistent and collision free, meaning that for the life of the blockchain it is believed that it is essentially impossible to calculate two values that hash to the same image, or a single value that hashes to any image that has not itself been calculated with the function. By this means, if we know a series of blocks where each contains the hash of the previous one, and someone else contains a second such series, and one contains block 0 at the end, the two series are the same between any member the two have in common and their common end at Block 0. When B_n is in this sequence in the nth position, and no further branching is possible at its own or a lower level according to the protocol for building blocks, then we say that B_n is immutable. Looking at the structure from the outside or from the perspective of a single node, the immutable part can only grow. The immutable sequence seen by a node may either equal or be a prefix of the immutable sequence we see from the outside.

One of the core assumptions about blockchains is that they can contain a limited number of bad nodes who do not follow protocol. Depending on the nature of the protocol it may be possible for such a node to post a block at any time whose hash field is any chosen block and therefore a contender for its successor, even though it may already have a successor and the new one has no hope of being accepted. When we say that a block is immutable in such cases, we more precisely mean that no block added by anyone at this or an earlier place would be believed by any trustworthy node running the blockchain according to its rules.

The linear order that emerges amongst immutable blocks creates a natural temporal order amongst them. The node assembling the block can, if the protocol calls for it, add a time stamp to the block which is close to real time. We will discuss some options for this, as well as options for incorporating times within blocks, later in this paper.

We make two further assumptions about time. The first is that any node has a reasonably accurate idea of what the time is at any moment, thanks to an inbuilt clock potentially reinforced by (a) long term observations of the system and (b) observations of external sources such as GPS signals. This inbuilt clock is at least an order of magnitude more accurate than the typical gap between block time stamps. Thus any significant untruths told by untrustworthy nodes about the time should be apparent to all. The second is that blocks will continue to be generated while the blockchain is current, and that a block will appear with a time stamp at least T within some bound ϵ of the actual time T. If there were a lower bound on the size of a block that was greater than that of one empty aside from housekeeping and formatting, this second assumption would map to one about the rate at which material is generated to put in the blockchain. We require no further assumptions about how blocks are agreed, so we cover both public and private models. It goes without saying that where the security of a signature model depends on the accuracy or security of a blockchain, these need to be assessed carefully. In the next section we set out our assumptions about blockchains and the nature of time within them. Our aim in the first sections of this paper is to build efficient models of cryptographic signature that exploit the blockchain model set out above. We then examine corresponding models of services akin to PKI for these.

2 Blockchain assumptions

For us, a blockchain is a series of blocks each of which contains the strong cryptographic hash of its predecessor, and where each of the blocks contains a timestamp which is strictly greater than that of the previous block and which is closely related to the real time at which the block was posted. Specifically the time at which block B_n is posted is within $\epsilon > 0$ of the timestamp therein, and that any attempt to post a block with a more remote timestamp will not be successful.

For blocks posted at and around the true head of the blockchain, we expect this to be managed by the general consensus and approval process. Any attempt to post a block remote from the current head can be countered, for example, by the techniques proposed in [4] which show how to make attempts to add a successor to a long-standing block readily detectable. We believe that the signature mechanisms used there *must not* depend on the time-based methods set out in this present paper, if these are supported directly or indirectly by the same blockchain.

The best-known way of limiting the rate of block creation is Proof of Work (PoW), but it should also be possible to limit it by adding an agreed rate of creation to the constraint already expressed about accuracy of timestamps. It may be agreed that the next block's timestamp must be at least D greater than the previous one's *as well as accurate*. One way of managing this is via the *hash clock* of [5] in which previous block creators securely vote on when the time for the next block has come.

3 Extending the life of a signature

Suppose that Alice has an established key using a signature algorithm that we can now be confident will be secure until future time T. An example of this would be an RSA key where we can be confident that the base cannot be factored either by conventional means (because the time between the key's original publication and T is less than the minimum reasonably possible for keys of the given length) or on a quantum computer (because we are confident that no sufficiently powerful one will be built before T).

There is no particular benefit in making T large here, and making a correspondingly stronger assumption about the signature algorithm. All we really require is that it will still be secure at $t+\epsilon$, where t is the timestamp of the block it is placed in.

If we can give evidence that the signature was created before T, together with evidence of exactly what was signed then, we can safely accept the signature as secure, even at a time $T' > T$ when the signature algorithm is known to be insecure if used afresh. This provides an interesting form of signature. Suppose that $S(A, X)$ is the signature algorithm referred to above, and that $hash(\cdot)$ is a hash function that we are confident will remain secure well beyond T. To sign X, place $S(A, X), hash(X))$ on the blockchain, ensuring it gets a timestamp less than $T - \epsilon$. As an alternative to $hash(X)$ we could use X, but of course that might take a lot more space in general and it would make X public. Then for as long as $hash(\cdot)$ is second preimage resistant, and was collision free at the time T_0 the signature was created, we can be confident in the signature. Suppose $T' > T$ is such a time.

This inference is valid because (a) as no collisions were known at T_0, we can disregard the possibility that X is in one of the few collision pairs that might be known by T'. Thus the intruder will, at T', know no value Y other than X such that $hash(X) = hash(Y)$. Furthermore, this is true even if the signature is made by an A under the control of the intruder. (b) We know that A signed something that hashed to $hash(X)$ at the time when this signature was secure. (c) Thus anything presented by anyone at T' as X must actually be the X that A signed.[1] We can envisage that the choice of S might be made from options laid down in the blockchain itself which change from time to time. The signature S might be the compound of several different functions. As the composition of the choice varies, it is almost certain that any agent A will have to publish new public keys. It might choose to do this by interacting with a certification authority in a PKI, or by publishing the new public key(s) signed in the blockchain at a time when the means of signature are still clearly valid.

Thus the blockchain timing model gives us a robust way of updating public keys. This is an issue which we will explore more in subsequent sections, where key renewal is necessary in a systematic way.

The above begs the question about what happens when we fear that the hash used may become compromised. One solution to this is for anyone who knows the X of a signature pair $(S(A, X), hash(X))$ that is in future danger to place $(hash(X), hash'(X))$ on the blockchain within the validity of $hash(X)$ for a stronger $hash'$. This can be checked by anyone else

[1]The inference here is easier if $S(A, X)$ is actually a signature of $hash(X)$.

who knows X, and combination of the earlier $(S(A, X), hash(X))$ and the later $(hash(X), hash'(X))$ are equivalent to (S(A,X),hash'(X)) at the earlier time. By and large, however, it is probably better to avoid this by not compromising on the strength of hash.

The methods set out in this section demonstrates how our timing assumptions about blockchains can reliably extend the life of signatures, creating a form of *temporal signature*, though we will reserve this term for the sharper use of time in subsequent sections.

We remark that this method of extended signature certainly uses the assumptions about time stamps set out in Section **??**, but not sharply since for conventional signatures there is no known drop-dead date, but rather informed guessing. In the next section we will see a technique where drop-dead times become a reality.

4 Hash-based temporal signature

While methods of hash-based signature, notably one-shot schemes such as Lamport's [1] 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 timing service that can be provided by a blockchain. Indeed in this paper we will generally assume it is a blockchain satisfying the assumptions of Section **??**.

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 later providing an analogue of zero-knowledge proof.

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, or simply where a lot of things need to be signed.

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 [2] and the interactive authentication scheme of [3]. 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.

We now 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 [1]. 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 create, use and store.

Our model of signature will work in blockchain systems using their intrinsic time-stamping, and others 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.

Usually, though not invariably, parties writing to the blockchain will sign the objects they put there.

We are all familiar with digital signature schemes based on one-way functions to the extent that the very definition of a cryptographic signature frequently 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 [3] 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, 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.

Inter alia this initial block contains, for each node A, a finite set of key certificates of the form[2] $(hash(k, A, t), t, A)$ where k are chosen at random from a large set and t varies over future times. Thus each node will typically have keys labelled by a spread of (initially) future times.

[2]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, or using hash combiners that combine multiple hash values into a single 'super-hash') to counter weaknesses in specific (e.g. iterative) hash constructions. Of course careful and conservative choices of the hash functions themselves is recommended.

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 would. Is her duty to broadcast k when she knows that no further block will appear with a timestamp less than t. She can do this by writing it into the blockchain or releasing it by some other means. Because k has been precommitted there is no need for A to sign this release.

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. Of course this can only be done if the publication is in the blockchain. 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.

This is an instance of a general fact: if the hash of any value v is already in the blockchain but v has not been generally released, then releasing v gives it the same status as $hash(v)$: if it is immutable then so is v itself, without placing v unhashed on the blockchain. Anyone who sees v is able to verify its correspondence to $hash(v)$.

4.1 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) with k 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 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 of this style (rather than merely signatures on keys). Thus this style of temporal signature could become totally part of the blockchain and its mining protocol, rather than something that individual nodes check for themselves.

5 Alternative temporal signature

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 and only releasing the key (whether into the blockchain or not) 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 blockchain. 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 blockchain. Once she has seen they are all there in such a way that they are all immutably present there she releases k.

This is not secure: 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 this is there before releasing k, which we will now assume is done by placing it on the blockchain. The consensus protocol only accepts k when the numbers tally. Here we are using the properties of our blockchain 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 medium such as a blockchain. 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 need 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 refreshment 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. Where a key is signed (whether by a TTP or via self refreshment) we would expect to see a checkable statement setting out the authority on which this is done.

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: the unrevealed N_i thus constitute the key.

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.

8 Augmented temporal signature

The obvious disadvantage of the hash-based temporal signatures we have introduced is that signatures cannot be checked immediately. In some applications this will not matter, but there is a straightforward way of overcoming it where it does. That is to couple any such temporal signature with a conventional one which we can be confident will be valid until the temporal signing key is released. The algorithm and public key associated with identity Alice can be updated on Alice's TKI entry (together with upper limits on when these can be used and verified), and each time she places a signature on the blockchain it is accompanied by the then-appropriate conventional one if she wishes it to be immediately verifiable.

Again, the conventional signature would be of a hash of the signed object – for checking purposes the same hash used in the temporal signature.

It is debatable whether the temporal signature is necessary at all here, since the proven-time use of a weak signature (provided augmented with a hash of the underlying value) would fall within the scope of the methods of Section **??**. We believe it is still valuable because it acts as a checkable verification of the weak signature, and is practically free in terms of calculation.

9 Temporal signature in support of the blockchain?

Consider the following facts:

- We have demonstrated that temporal signature is a viable mechanism for signing entries placed in secure blockchains where times can be believed.
- Blockchains and their integrity depend on attributing actions to known agents, and thus to signatures, and may well have other signatures built in, as for example in support shown for blocks etc.
- Any false part of a blockchain introduced later with fake timestamps earlier than reality could easily spoof signatures using temporal keys that had in fact been released where the fake time stamp makes it appear they have not.

We conclude that temporal signature should not be used in any aspect of blockchain construction or contribution to its basic security, and especially not in mechanisms that contribute to the prevention of the belief by legitimate nodes of spoof forks.

In other words, though there may be some exceptions that can be established with great care, temporal signature is a way of taking advantage of the high-integrity record keeping and agreed timings established by blockchains, but not a way of creating these things themselves.

10 Conclusions

We have introduced two distinct approaches to using blockchain and similar timing models for signature: one for extending existing mechanisms, and one for enabling a new one. All depend on the timing assumptions made in Section **??**.

The first approach is a systematisation of the truth that a signature provably made before the algorithm used is broken, remains valid even after that point. The second uses the strong degree of coordination and common knowledge achieved by blockchains to make the extremely efficient form of authentication (based on knowledge of who knew what, when) already present in the TESLA stream protocol practical for a true signature scheme, which nevertheless falls outside the usual intensional definition of signature. We have therefore termed it *temporal signature.* This remains limited, however, by the delay in the verifiability of signatures.

We have shown how a TKI, the analogue of a PKI for temporal signature, 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.

Blockchains are of course an application of cryptography. We have shown here that, in turn, blockchains can contribute to the efficiency and persistence of cryptographic signature.

Acknowledgements

The work reported in this paper was done at Chieftin Lab in 2017, It was been improved by discussions with Cas Cremers and Peter Ryan.

References

[1] L. Lamport. Constructing digital signatues from a one-way function, Technical Report SRI-CSL-88, 1979

[2] A. Perrig, R. Canetti, J.D. Tygar and D. Song, The TESLA broadcast authentication protocol, RSA Cryptobytes, **5**, 2005,

[3] R. Anderson, F. Bergadano, B. Crispo, J. Lee, C. Manifavas, and R. Needham. A new family of authentication protocols. ACM Operating Systems Review, 32(4):9–20, October 1998

[4] A.W. Roscoe and Wang Lei. *Taking the work out of blockchain mining*, Available from `www.tbtl.com`

[5] A.W. Roscoe and Bangdao Chen *The greening of blockchain mining*, Available from `www.tbtl.com`