Peter Todd

OpenTimestamps: Scalable, Trustless, Distributed Timestamping with Bitcoin

Sep 15, 2016

I've spent the past few weeks working on[1] an old[2] project of mine, OpenTimestamps, and it's ready for a public alpha release. It's timestamping infrastructure with three big advantages over the existing alternatives:

1. **Trust** — OpenTimestamps uses the decentralized, publicly auditable, Bitcoin blockchain, removing the need for trusted authorities; OpenTimestamps's architecture is designed to support multiple, cross-checked, notarization methods in the future.
2. **Cost** — OpenTimestamps scales indefinitely, allowing timestamps to be created for free by combining an unlimited number of timestamps into one Bitcoin transaction.
3. **Convenience** — OpenTimestamps can create a third-party-verifiable timestamp in about a second; you don't need to wait for a Bitcoin confirmation.

In this post we'll talk about what timestamps can do, what they can't do, how to use OpenTimestamps to timestamp files, and how OpenTimestamps works under the hood. OpenTimestamps can also timestamp Git commits; we'll discuss Git integration in a follow-up post.

Acknowledgements: Verisart funded development work I did last year on an earlier version of OpenTimestamps - now used in their digital art provenance solution. Additionally BTCC funds my Bitcoin development efforts, and while OpenTimestamps isn't specifically part of that, that funding did give me the ability to work on it. Finally, Riccardo Casatta contributed to development and testing, and implemented an earlier version of OpenTimestamps on his Eternity Wall service.

Contents

1 What Can and Can't Timestamps Prove?

A timestamp proves that a message existed prior to some point in time; timestamps are occasionally referred to as "proofs-of-existence". Being able to prove that data existed prior to a point in time is surprisingly useful. Let's look at some use-cases to understand why:

1.1 Record Integrity

> *Our colocation centre informed us that someone broke into the cage the backend servers are located in last night; the last off-site backups were a week ago. The auditors want to know which records the intruders might have changed after they broke in. What do I tell them?*

If we know when the intrusion happened, and have trustworthy timestamps for all records on the servers, we can use those timestamps to prove records existed prior to the intrusion. This lets post-intrusion auditing and recovery efforts focus on a much smaller set of data.

However, timestamps can't help you if you don't know when the intrusion happened, nor can they help for records created after: the intruder could have just as easily made the timestamp on data they've modified. But still, this is an excellent use of timestamping, one that usually helps, and will never make the problem any worse.

1.2 Software Signing and PGP

> *I need to install this two-year-old software package to recover old records, but the developer revoked their PGP key six months ago - apparently their laptop got stolen. How do I know I'm getting the real thing?*

PGP revocations have dates on them, so if the revocation is dated after the signature on the software - and there's a valid timestamp for that signature - you're good to go![3] Again, this is an excellent timestamping use case.

1.3 Evidence Authenticity

> *I found a two-year-old snapshot of the defendant's website on a public archiving service, but is it authentic?*

A timestamp on a website snapshot doesn't necessarily mean the snapshot is authentic, but it does greatly limit the scope of who might have tampered with the data. How likely is it that someone hacked into the archiving service, well before they knew there would be a lawsuit?

> *I run a website archiving service, but I don't have the resources to hire auditors or security consultants. How can I prove our archives haven't been modified after the fact?*

Simple answer: timestamps only make your problem better, not worse!

> *My colleague said X in an IRC chat last year, but ever since the project failed they're now claiming they said Y. How do I prove they're lying?*

A timestamp only proves *a* message existed, not *the* message. Last year, would you have known the significance of X and Y? If so, you could have timestamped two conflicting IRC logs.

> *This anonymous crypto-currency ByteCoin looks pretty cool, the technology is legit, and there's a two-year old community around it. It can't be a recently-created pre-mined scam with a fake community, right?*

Timestamps can't help you here: ByteCoin's creators wouldn't have timestamped anything. Additionally, keep in mind that even if they did, creating a crypto-currency from scratch could have very well taken two years - enough time to create perfectly valid timestamps for things that the public never saw.

1.4 Ownership

> *I want to secure land titles on the blockchain with timestamps!*

Timestamps do *not* by themselves solve the doublespend problem; Bitcoin is a lot more than just a way to timestamp data.

Having said that, timestamps can provide evidence that ownership *records* are accurate; my client Verisart aims to do exactly that in for art provenance records. While a timestamp on an art provenance record such as a gallery sales receipt doesn't by itself prove that record is valid, it can limit the scope of potential fraud, allowing provenance investigators to focus their efforts by eliminating many of the possible fraud scenarios they may be dealing with.

2 Creating and Validating Timestamps

First, get the OpenTimestamps Client, version 0.2.0:

```
git clone https://github.com/opentimestamps/opentimestamps-client.git
cd opentimestamps-client
git checkout opentimestamps-client-v0.2.0
```

and install the dependencies:

```
pip3 install -r requirements.txt
```

There's no system-wide installation process, so we'll run it directly out of the repo. Creating a timestamp is fast and simple:

```
$ ./ots stamp README.md
Submitting to remote calendar https://a.pool.opentimestamps.org
Submitting to remote calendar https://b.pool.opentimestamps.org
```

You'll see that `README.md.ots` has been created. We can't verify it immediately however:

```
$ ./ots verify README.md.ots
Assuming target filename is 'README.md'
Calendar https://alice.btc.calendar.opentimestamps.org: No timestamp found
Calendar https://bob.btc.calendar.opentimestamps.org: No timestamp found
```

It takes a few hours for the timestamp to get confirmed by the Bitcoin blockchain; we're not doing one transaction per timestamp.

However, the client does come with a number of example timestamps which you can try verifying immediately. You'll need a local Bitcoin Core node (a pruned node is fine) with the `rpcuser` and `rpcpassword` options set in `~/.bitcoin/bitcoin.conf` to allow the OpenTimestamps client to connect to your node via the RPC interface. Once that's setup, let's try verifying `examples/hello-world.txt`:

```
$ cat examples/hello-world.txt
Hello World!
$ ./ots verify examples/hello-world.txt.ots
Assuming target filename is 'examples/hello-world.txt'
Success! Bitcoin attests data existed as of Thu May 28 15:41:18 2015 UTC
```

Simple!

3 How OpenTimestamps Works

Those already familiar with Bitcoin and hash functions may want to skip to Commitment Operations.

But how did that actually work? Let's start by adding the verbose flag and trying verify again:

```
$ ./ots -v verify examples/hello-world.txt.ots
Assuming target filename is 'examples/hello-world.txt'
Hashing file, algorithm sha256
Got digest 03ba204e50d126e4674c005e04d82e84c21366780af1f43bd54a37816b6ab340
Attestation block hash: 000000000000000003e892881a8cdcdc117c06d444057c98b6f04a9ee
Attested time: 1432827678
Success! Bitcoin attests data existed as of Thu May 28 15:41:18 2015 UTC
```

3.1 Notaries and Time Attestations

Every Bitcoin block header has a field in it called `nTime`. For a Bitcoin block to be accepted by the network the Bitcoin protocol requires that field to be set to approximately the time the block was created. Exactly how accurate that field must be for a block to be valid is a complex topic, but for our purposes it's fair to say it'll very likely be accurate to within two or three hours - even if a sizable minority of Bitcoin miners are trying to create invalid timestamps - and almost certainly within a day.

Thus we can say Bitcoin is a *notary*, and we can use Bitcoin block headers as *time attestations*: proof that a notary that we trust *attested* to the fact that some data existed at some point in time. That means that if Bitcoin is working correctly these 80 bytes - the block header for block 358,391 - existed on May 28th, 2015:

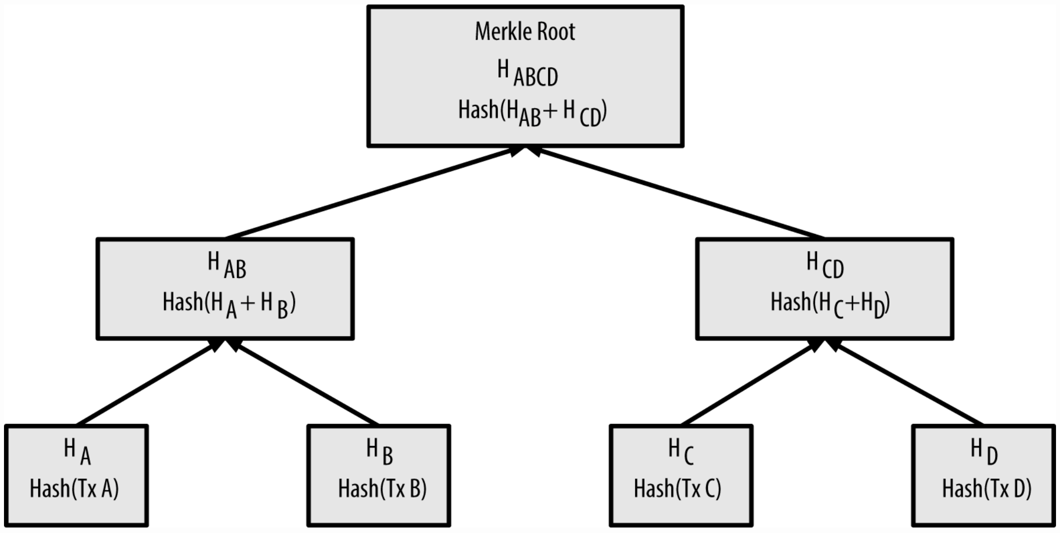
```
02000000b96394585a281b7e5f438fd1c9ed492645a1fd61cb38020400000000000000000007ee445
d23ad061af4a36b809501fab1ac4f2d7e7a739817dd0cbb7ec661b8a1e376755f58616186272def6
```

3.2 Merkle Trees

Of course, our message "Hello World!" isn't included in those 80 bytes! But neither are any transactions! So how does Bitcoin link the transactions in a block to the block header? With another field, called the `merkleroot`, shown here in bold:

02000000b96394585a281b7e5f438fd1c9ed492645a1fd61cb3802040000000000000000000**007ee445**
d23ad061af4a36b809501fab1ac4f2d7e7a739817dd0cbb7ec661b8a1e376755f58616186272def6

The value of the `merkleroot` is calculated by taking all the transactions in a block and creating a merkle tree, shown here in this diagram from the book Mastering Bitcoin:



Each arrow represents a cryptographic hash function, which takes an variable-length message, and transforms it to a fixed-length digest. Importantly, a secure hash function has the property that it's not possible to find two messages that hash to the same digest. Pairs of messages that "collide" to produce the same digest *do* exist, but the numbers involved are so enormous that actually finding collisions isn't believed to be physically possible:

> *These numbers have nothing to do with the technology of the devices; they are the maximums that thermodynamics will allow.* ***And they strongly imply that brute-force attacks against 256-bit keys will be infeasible until computers are built from something other than matter and occupy something other than space.*** *-Bruce Schneier, Applied Cryptography*

This means that if we change any of the transactions at the bottom of the tree, every digest above the modified transaction changes, ultimately changing the `merkleroot` at the tip of the tree. For example, if we changed the last byte of the last transaction in block 358,391 from `0x00` to `0x01` the merkle root changes completely:

```
Original: 007ee445d23ad061af4a36b809501fab1ac4f2d7e7a739817dd0cbb7ec661b8a
Modified: 87808e5a6a196e401bde8ebaf5b453825cdc3b66b87526bb355d519ae52ba42b
```

Finally, it turns out that if you take our message `Hello World!\n` and hash it, we get:

```
RIPEMD160(SHA256('Hello World!\n')) => 1df8859e60bc679503d16dcb870e6ce91a57e9df
```

That digest happens to be in the Bitcoin transaction 7e9f0f7d9daa2d9e51b2e22f4abe814c3f90539afa778a9bef88dc64627cb2ec[4], and thus if were were to change that message, the digest would change, changing the transaction, changing the merkle tree, and finally, changing the block header. Since you can't change the message without changing the block header, we've proven that the message must have existed prior to when the block header was created - this is a timestamp proof!

3.3 Commitment Operations

Another way of describing that proof is to say that the block header *commits* to the merkle tree, and the merkle tree commits to the transactions; the transactions are *committed* by the block header, because changing the transactions changes the header. We can also say that a cryptographic hash function is a *commitment operation*, because the input to the operation can't be changed without changing the result.

An even simpler example of a commitment operation is the `append` operation:

```
"Foo" -> append("Bar") -> "FooBar"
```

It's easy to see why this is a commitment operation: if I change the input it's obvious that the result must change - it contains the input itself!

A timestamp proof in OpenTimestamps is just a list of commitment operations that are applied to the message in sequence. To verify the proof, all that needs to be done is to replay the operations, and check that the final result is a message that you already know existed at a certain time. Since every commitment operation is guaranteed to have a different result for a different input, there's no way to change the message we're timestamping without changing the result and invalidating the timestamp.

We can see the actual operations in our timestamp with:

```
$ ./ots info examples/hello-world.txt.ots
File sha256 hash: 03ba204e50d126e4674c005e04d82e84c21366780af1f43bd54a37816b6ab34
Timestamp:
ripemd160
prepend 0100000001e482f9d32ecc3ba657b69d898010857b54457a90497982ff56f97c4ec58e6f9
append 88ac00000000
sha256
sha256
prepend a987f716c533913c314c78e35d35884cac943fa42cac49d2b2c69f4003f85f88
sha256
sha256
prepend dec55b3487e1e3f722a49b55a7783215862785f4a3acb392846019f71dc64a9d
```

```
sha256
sha256
prepend b2ca18f485e080478e025dab3d464b416c0e1ecb6629c9aefce8c8214d042432
sha256
sha256
append 11b0e90661196ff4b0813c3eda141bab5e91604837bdf7a0c9df37db0e3a1198
sha256
sha256
append c34bc1a4a1093ffd148c016b1e664742914e939efabe4d3d356515914b26d9e2
sha256
sha256
append c3e6e7c38c69f6af24c2be34ebac48257ede61ec0a21b9535e4443277be30646
sha256
sha256
prepend 0798bf8606e00024e5d5d54bf0c960f629dfb9dad69157455b6f2652c0e8de81
sha256
sha256
append 3f9ada6d60baa244006bb0aad51448ad2fafb9d4b6487a0999cff26b91f0f536
sha256
sha256
prepend c703019e959a8dd3faef7489bb328ba485574758e7091f01464eb65872c975c8
sha256
sha256
append cbfefff513ff84b915e3fed6f9d799676630f8364ea2a6c7557fad94a5b5d788
sha256
sha256
prepend 0be23709859913babd4460bbddf8ed213e7c8773a4b1face30f8acfdf093b705
sha256
sha256
verify BitcoinBlockHeaderAttestation(358391)
```

The big advantage of representing a timestamp this way is it doesn't matter *how* you create the timestamp: so long as you can extract a valid list of commitment operations that the OpenTimestamps protocol supports, you can create a timestamp that any OpenTimestamps-compatible verifier can verify.

3.3.1 Timestamp Operation Tree

Additionally, rather than being limited to a linear list of operations, timestamps are actually *trees* of operations, with the root of the tree being the message, the edges the commitment operations, and the leaves the attestations. We'll see how this functionality is currently used when we discuss calendars; in the future this will allow a single message to be timestamped with multiple notaries. For instance multiple proof-of-work blockchains such as Bitcoin and Litecoin could be used simultaneously, as well as notaries offering different trust models and accuracy guarantees, such as traditional RFC-3161 timestamps and Certificate Transparency servers.

3.4 Scalability Through Aggregation

An important way we use the flexibility commitment operations provides us is for scalability. For example, suppose you want to timestamp 10,000 different files: with most existing Bitcoin timestamping solutions, that would require 10,000 Bitcoin transactions - inefficient and expensive. But with OpenTimestamps, you can instead create a merkle tree of those 10,000 files and timestamp the tip of that tree in one transaction. Each per-file timestamp is simply the list of commitment operations that comprise the path up the first merkle tree, then up the Bitcoin block's merkle tree, to finally get to the block header. The verifier doesn't care: to it it's just a series of operations like any other timestamp.

Secondly, we further improve this with a system of aggregation servers: publically available "meeting points" where anyone can submit a digest to be timestamped. As of writing, there are two public aggregation servers, `a.pool.opentimestamps.org`, and `b.pool.opentimestamps.org`

As digests are submitted for aggregation, they're added to a list of pending digests. Periodically that list is combined into a single merkle tree, and then the tip of that tree is timestamped with Bitcoin. The server then returns the individual timestamp proofs for each submitted digest (conceptually speaking; see the next section for how it actually works!). This process can be done multiple layers deep, with merkle trees timestamping tips of merkle trees timestamping tips of merkle trees. An unlimited number of messages can be timestamped every second, and the system can scale from one user to an unlimited number of users without making any changes at all to the way timestamps are verified.

While the aggregation process works most efficiently with centralization, it's still essentially trustless: the worst an aggregation server can do is go offline, an inconvenience. The aggregation system *can't* produce a fake timestamp, because it's Bitcoin, not the aggregation system, that proves the validity of a timestamp.

3.5 Public Calendars

While aggregation servers are efficient, they aren't convenient: you still have to wait for the underlying Bitcoin transaction to confirm. This takes at least 10 minutes on average, and can take significantly longer if you're unlucky. For some applications that's OK, and the OpenTimestamps client supports a `--wait` flag that'll simply wait until the underlying Bitcoin transaction is confirmed.

But for many applications that's nowhere near fast enough. For example, if you're sending a timestamped PGP-signed email, or signing a timestamped Git commit, you want the timestamping process to finish within a second or two at most - that's not even possible with any existing decentralized consensus system!

OpenTimestamps solves this problem with a compromise: public calendar servers. A calendar is simply a collection of timestamps; a calendar server provides remote access to a calendar. As of writing, there are two public calendar servers[5] in operation, `alice.btc.calendar.opentimestamps.org`, and `bob.btc.calendar.opentimestamps.org`

The calendar servers work in conjunction with the aggregation servers: rather than wait until the pending digests are timestamped by Bitcoin, what the aggregation system actually does is aggregates all submitted timestamps in a one second interval into a merkle tree, and submits the tip of that tree - which we call a "commitment" - to a public calendar server. The calendar server makes two promises:

1. Every commitment added to the calendar will be timestamped by the Bitcoin blockchain in a reasonable amount of time.

2. Completed commitment timestamps will be made available to the public and kept indefinitely.

We call a timestamp made using a calendar server an incomplete timestamp. Here's an example:

```
$ ./ots info examples/incomplete.txt.ots
File sha256 hash: 05c4f616a8e5310d19d938cfd769864d7f4ccdc2ca8b479b10af83564b097af
Timestamp:
append e754bf93806a7ebaa680ef7bd0114bf4
sha256
append b573e8850cfd9e63d1f043fbb6fc250e
sha256
prepend 57cfa5c4
append 6fb1ac8d4e4eb0e7
verify PendingAttestation('https://alice.btc.calendar.opentimestamps.org')
```

See the last line? When we try to verify the timestamp, that line tells the verifier to ask the remote calendar "Alice" for the rest of the timestamp:

```
$ ./ots verify examples/incomplete.txt.ots
Assuming target filename is 'examples/incomplete.txt'
Got 1 new attestation(s) from https://alice.btc.calendar.opentimestamps.org
Success! Bitcoin attests data existed as of Wed Sep  7 05:56:43 2016 UTC
```

Thus, with the aid of a publicly available aggregation and calendar infrastructure, we can create the timestamp quickly and conveniently in about one second, yet anyone can verify the timestamp later, using the decentralized, trustless, Bitcoin blockchain.

Of course, this convenience does have a cost: the public calendars *are* a central point of failure. For that reason we call a timestamp that requires the assistance of a public calendar server an *incomplete* timestamps. OpenTimestamps mitigates this risk in a number of ways:

3.5.1 Using Calendars is Optional

First and foremost, you can create timestamps that don't depend on calendars by using the `--wait` option. If enabled, the client simply waits until the timestamp is completely confirmed by the Bitcoin blockchain. The resulting timestamp will contain all the data needed to prove the timestamp with Bitcoin, allowing verification to be done completely locally.

3.5.2 Upgrading Timestamps

An incomplete timestamp can be upgraded later with the 'ots upgrade' command. The complete Bitcoin proof will be downloaded from the calendar and saved as part of the timestamp, with the result being as though you used the `--wait` option in the first place.

3.5.3 Redundancy

By default when creating timestamps two public calendars are used simultaneously, and the timestamp is only saved if we get a response back from both:

```
$ ./ots info examples/two-calendars.txt.ots
File sha256 hash: efaa174f68e59705757460f4f7d204bd2b535cfd194d9d945418732129404dd
Timestamp:
append 839037eef449dec6dac322ca97347c45
sha256
 -> append 6b4023b6edd3a0eeeb09e5d718723b9e
    sha256
    prepend 57d46515
    append eadd66b1688d5574
    verify PendingAttestation('https://alice.btc.calendar.opentimestamps.org')
 -> append a3ad701ef9f10535a84968b5a99d8580
    sha256
    prepend 57d46516
    append 647b90ea1b270a97
    verify PendingAttestation('https://bob.btc.calendar.opentimestamps.org')
```

Note how we're taking advantage of the fact that timestamps are commitment operation trees: the path splits after the `sha256` operation. So long as at least one of the calendars is available the timestamp can be verified.

You are of course able to use your own calendars, both instead of, and in addition too, the public calendars. For example, Example Inc. could setup their own calendar, which in turn uses the public calendars to actually create the Bitcoin proofs (potentially falling back on its own wallet if the public calendars are down). Such a timestamp might look like the following:

```
$ ./ots info <some file>.ots
File sha256 hash: 522cf38077181bf09ce229d2bf49d23b8d7a4e9fcf7e23d7455ad2e665bf17c
Timestamp:
append 839037eef449dec6dac322ca97347c45
sha256
 -> verify PendingAttestation('https://calendar.example.com')
    sha256
    append 1467861d8186c1df176ef2efc08d553d2dd183aadec403335d707f1e90fe3e92
    sha256
    prepend 192e8f15b77bdc5a85e346d081c995f7459dcde9f68a3523e0ae453c5a693ef9
```

```
     -> append 6b4023b6edd3a0eeeb09e5d718723b9e
        sha256
        prepend 57d46515
        append eadd66b1688d5574
        verify PendingAttestation('https://alice.btc.calendar.opentimestamps.org'
     -> append a3ad701ef9f10535a84968b5a99d8580
        sha256
        prepend 57d46516
        append 647b90ea1b270a97
        verify PendingAttestation('https://bob.btc.calendar.opentimestamps.org')
```

The client has a whitelist of calendars it'll contact automatically, so users external to Example Inc. will just ignore their pending attestation and try the public calendars instead; whether or not Example Inc. chooses to make the calendar publicly accessible is up to them, but regardless, they don't have to rely on the public calendars to verify their timestamps.

Software to actually do this isn't written yet on the server side. But once it is written it'll be compatible with existing clients - a good example of the advantages of the commitment operations model.

3.5.4 Caching and Mirroring

The OpenTimestamps client maintains a cache of timestamp data retrieved from calendar servers. We can see this if we verify a timestamp twice:

```
$ ./ots verify examples/incomplete.txt.ots
Assuming target filename is 'examples/incomplete.txt'
Got 1 attestation(s) from cache
Success! Bitcoin attests data existed as of Wed Sep  7 05:56:43 2016 UTC
```

We can also upgrade a timestamp directly from the cache, even if the calendar isn't available:

```
$ ./ots upgrade examples/incomplete.txt.ots
Got 1 attestation(s) from cache
Success! Timestamp is complete
```

The upgraded timestamp can be verified by anyone, using nothing more than their local Bitcoin node.

More generally, timestamp proofs have the property that they're self-validating: regardless of where you got the proof, you can still validate the timestamp. In the near future I hope to take advantage of this property by making it easy for anyone to make mirrors of the public calendars, be it just the subset of timestamps relevant to them, or complete copies. This data is relatively small: with 31 million seconds in a year and about 100 bytes per second (amortised), a complete calendar mirror would consume about three gigabytes per year.

This also means that decentralized calendar mirroring with technologies such as Freenet, Tahoe-LAFS, and IPFS will be very practical; equally the write-once and small size of the data makes DDoS resistant hosting easy to use.

3.5.5 Legal Risks

I expect the legal risks of operating calendar infrastructure to be low for a number of reasons:

1. Calendars aren't authoritative — at worst they can deny service, not produce false proofs.
2. Taking down calendars is pointless — the data (will be) easy to mirror widely.
3. Calendars don't store externally provided data — all incoming digests are nonced, and only commitments to those digests are stored permanently.

Quite likely, the biggest legal risk is software patents.

4 Footnotes

1. By “working on” I pretty much re-wrote the entire thing from scratch, adding a number of features that the previous architecture didn’t easily support. ↩
2. OpenTimestamps is actually my first Bitcoin project: the earliest Git commits date back to June 8th 2012, a year and a half before I quit my job to work on Bitcoin full-time. ↩
3. An interesting nuance to this is someone who has stolen a PGP key could *also* create a revocation, and they could backdate it to *deny* access to previously created signatures; there’s a lot of interesting design questions about how to deal with this with random beacons and the like that are beyond the scope of this blog post. ↩
4. It’s the Bitcoin address 13jUKAuPDfgVPVgsVbeKVNWBv6wAh31vkN. ↩
5. In reality, the two aggregation servers and two calendar servers are combined into two machines, with each one running a combination aggregator and calendar. But that’s an implementation detail that will likely change in the future as the infrastructure is scaled up. ↩

Peter Todd

petertodd
petertoddbtc