Building a Beacon Format Standard: An Exercise in Limiting the Power of a TTP

No Author Given

No Institute Given

Abstract. We discuss the development of a new format for beacons–servers which provide a sequence of digitally signed and hash-chained public random numbers on a fixed schedule. Users of beacons rely on the trustworthiness of the beacon operators. We consider several possible attacks on the users by the beacon operators, and discuss defenses against those attacks that have been incorporated into the new beacon format. We then analyze and quantify the effectiveness of those defenses.

1 Introduction

A randomness beacon is a source of timestamped, signed random numbers; randomness beacons (among other kinds) were first described by Rabin in [7].

At high level, a randomness beacon is a service that regularly outputs randomness, with certain cryptographic guarantees and with associated metadata, including a timestamp and a cryptographic signature. Each output of a beacon is called a *pulse*, and the sequence of all associated pulses is called a *chain*. An individual pulse from a given beacon can be uniquely identified by its `timeStamp`, and a pulse must never become visible before its timestamp.

In 2013, NIST set up a prototype beacon[5] service. Recently, a new beacon format has been developed with a number of new security features. Several independent organizations are currently planning to adopt this format, which offers the opportunity to have multiple independent beacon operators (in different countries) supporting an identical format, and supporting the generation of random values using multiple beacons inputs.

A beacon is a kind of *trusted third party*(TTP)–an entity which is trusted to behave properly, in order to get some cryptographic protocol to work securely. It's very commonly the case that we need a TTP to get a cryptographic protocol to work, or to make it efficient. However, this introduces a new security concern: the TTP is usually in a position to violate the security guarantees of the system. In the case of a beacon, users of a beacon gain the benefits of a convenient source of public random numbers, but must now worry that the beacon operator may reveal those random numbers in advance to favored parties, manipulate the public random numbers for his own purposes, or even "rewrite history" by lying about the value of previous beacon pulses.

The design of the new beacon format was largely an exercise in trying to limit the power of a TTP (the beacon) to violate its users' security. We believe this effort has some lessons for anyone trying to design a cryptographic protocol or standard that requires a trusted third party.
In the remainder of this document, we first propose a few principles for designing protocols with trusted third parties to minimize their potential harm. We then discuss the current NIST beacon and the general applications of public randomness. Next, we illustrate our TTP-limiting principles with specific aspects of the design of the NIST beacon format. We conclude by considering the ultimate effect of our efforts, and consider some possible improvements for the future.

2 Trusted Third Parties

A trusted third party (TTP) is an entity in some cryptographic protocol who must behave properly, in order for the protocol to meet its security goals. For example, in a public key infrastructure, a certificate authority (CA) is a TTP. The CA's job is to verify the identity of users, and then to issue users a certificate that binds their identity to a public key. A CA which is careless or malevolent can issue certificates to the wrong users–for example, giving an attacker a certificate that lets him impersonate an honest user. In most currently-used voting systems, the voting hardware is a kind of trusted third party; if it misbehaves, it may undermine the security of the election. A good discussion of the downsides of TTPs appears in [12].
Many important real-world cryptographic systems require TTPs to function. However, a TTP in a system is always a source of vulnerabilities to attack–if the TTP misbehaves, the security of the system will be violated. Because of this, anyone designing or standardizing a cryptographic system with a TTP should try to minimize the potential for abuse or misbehavior of the TTP. In general, this requires going through a few steps:

1. Enumerating each way that the TTP might misbehave so as to undermine the security claims of the system.
2. For each such potential attack, adding features to mitigate the attacks as much as possible.
   (a) In some cases, the attack may be possible to eliminate entirely. This essentially means removing one area of required trust from the TTP.
   (b) Sometimes, it may not be possible to eliminate the attack, but it may still be possible to limit its power or scope.
   (c) In other cases, the design of the TTP or protocol can be altered so that misbehavior creates evidence that can be shown to the world.
   (d) In still other cases, the design may be altered so that misbehavior is at least likely to be *detected* when it is attempted, even if this produces no solid evidence that can be shown to others.
   (e) Sometimes, an external or optional auditing step can be added that makes the misbehavior likely to be detected.

(f) In the worst case, it may be impossible to make this kind of misbehavior impossible or even detectable–in that case, we can at least document the risk of this kind of misbehavior.

Each of these steps is valuable. Once the designer of a system realizes that the TTP could misbehave in some way, it is often possible to work out mechanisms to move up the list–to make an undetectable attack detectable, or to make some kind of misbehavior by the TTP leave evidence that can be shown to the world. Even the final step, documenting the risk of misbehavior, is worthwhile, as it allows users of the TTP to understand exactly what risks they are accepting.

When there is some auditing step added to the protocol to keep the TTP honest, it is very important to consider how practical the auditing step is. If it is extremely computationally expensive or otherwise inconvenient, it may never be done even if it is, in principle, possible.

2.1 Incentives for the Honest TTP

An entity that wants to run an honest TTP has many strong incentives to try to reduce its own scope for abuse.

Fear, Uncertainty, and Doubt When silent, undetectable misbehavior is possible, there will always be suspicions swirling around that it is being done. This can undermine the TTP's reputation, or even convince people to avoid the system that relies on the TTP. Sometimes, this will happen even when the alternative is obviously less secure.

Insider and Outsider Attacks An organization trying to run a TTP honestly must deal with the possibility of both insider and outsider attacks. An insider might cause the TTP to misbehave in some way, despite the good intentions of the organization running the TTP. Or an outsider might subvert the security of the TTP in some way. Both of these have the potential to damage the reputation of the organization running the TTP.

Coercion The organization running a TTP must also worry about the possibility that they will be coerced in some way to misbehave. This might be done via legal means (such as a national security letter), or extralegal ones (a mobster threatening the head of the organization if some misbehavior isn't done). It might involve financial pressure (a business partner threatening some dire reprisals if the misbehavior isn't done), or a change in management of the organization in the future. It could even involve some intense short-term temptation to do the wrong thing that might be hard to resist in the future. The best way to resist this coercion is to have bad behavior simply be unworkable; what you cannot do, you cannot be coerced to do.

This follows the general pattern noted in [9], in which it is sometimes possible to improve one's position by limiting one's options. When it comes to the design and operation of a TTP, both the users and the operators of a TTP benefit from having a TTP and surrounding system designed to minimize the scope for abuse, and to make any abuse instantly detectable.

3 Public Randomness and the NIST Beacon

As described above, a beacon is a source of *public random numbers*, released in self-contained messages called *pulses*. The numbers should be random–unpredictable and impossible for anyone to influence. Each pulse contains a timestamp, and neither the pulse nor its random values may be released before the time indicated in the timestamp. Pulses are digitally signed and incorporated into a hash chain to guarantee their integrity. In order to be useful, a beacon should issue pulses on a fixed schedule, and should keep a database of all old pulses which it will provide on request.

3.1 Public Random Numbers?

Most discussions of random numbers in cryptography involve secret random numbers, such as those used to generate keys. However, there are many places where it is more useful to have *public* random numbers–numbers which are not known by anyone until some specific time, and then later become known (or at least available) to the whole world.
Why would we want *public* random numbers?

- Demonstrate we did something randomly (outside our own control).
- Bind something in time. (It couldn't have happened before time T.)
- Coordinate a random choice among many parties.
- Save messages in a crypto protocol.

The added value of a beacon is that these random numbers can be shown even to parties who were not present or involved at the time some event occurred. For example, if the random numbers from a beacon pulse are used to select a subset of shipping containers to open for inspection, then people who weren't present and weren't participating can verify that the selection was random–assuming they trust the beacon.
There are a number of properties that users need from public random numbers:

- Genuinely random.
- Unpredictable to anyone until they become public.
- Verifiable by everyone after they become public.
- Impossible for anyone to control or manipulate.
- Impossible to alter past values.

3.2 A Beacon as a TTP

All these applications work well with a beacon[1]. However, users of the beacon (including anyone trusting the randomness of the choices made) need to trust that the beacon is behaving correctly. A corrupt beacon might do all kinds of bad things, such as:

[1] There are other ways to get public random numbers. Many also depend on some trusted third party; others introduce other practical problems–ambiguity about correct values, lack of a fixed schedule, etc. Overall, we believe beacons are the best way to get practical public randomness for real-world applications

– Reveal future random values early to favored people.
– Control or influence random values to undermine the security of some user.
– "Rewrite history" by altering old pulses, thus changing claimed historical results.

The NIST beacon is a practical source of public random numbers that has been running continuously (with occasional downtime) for five years. The NIST beacon consists of several parts:

– Engine: the device that actually generates the pulses.
– Multiple independent cryptographic RNGs used to generate random values.
– A commercial hardware security module (HSM) used both as one source of random bits, and also to sign pulses.
– Frontend–the machine talking to the world, providing the random numbers. The frontend must support various requests from users.

The beacon format used since 2013 had only a few fields:

– Version
– Frequency
– Seed Value
– Previous Output
– Signature
– Status
– OutputValue

The new format has added many new fields. In figure 1, the new fields appear in red.

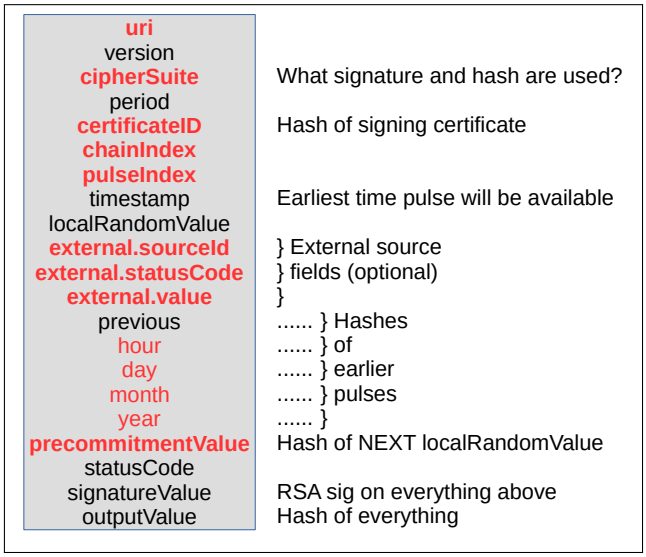

Fig. 1. The new pulse format

Despite the complexity of the new format, there are only a few fields in each pulse which are actually important for users:

- `timeStamp` – the timestamp (UTC) at which the value will be released.
- `localRandomValue` – the internal random value produced once a minute.
- `signatureValue` – a digital signature on the pulse contents.
- `outputValue` – the output of the pulse–the hash of all other fields of the pulse.

The other fields (including all the new fields) add security or improve convenience in using the pulses.

3.3 Timestamps

The `timeStamp` in the new pulse format is always in UTC, so that the time at which a pulse is created is easy for any user to determine, without the need to determine time zone or daylight savings time rules. The beacon promises never to release a pulse (or any information about the `localRandomValue` or `outputValue` field from the pulse) before the time in the timestamp, and never to issue more than one pulse with a given timestamp. The beacon also issues pulses on a fixed timetable–each timestamp is separated by `period` milliseconds. In the new format, `timeStamp` always appears as a string in RFC3339[**RFC3336**] format.

3.4 Where the Random Numbers Come From

Each pulse contains a new random value, called `localRandomValue`. This value is generated using multiple independent, strong random number sources. At present, the NIST beacon uses two RNGs–the Intel RNG included in the Beacon Engine machine, and the hardware RNG inside the HSM. We are currently working on adding a third RNG based on quantum phenomena (single-photon detection), custom built by NIST scientists.

Suppose the beacon has k different RNGs. Let R_i be the 512-bit random output from the i-th RNG. Then, we have the following formula:

$$\texttt{localRandomValue} \leftarrow \texttt{SHA512}(R_0 \parallel R_1 \parallel \ldots \parallel R_{k-1}) \tag{1}$$

That is, we take 512 bits from each independent RNG, concatenate them, and then hash them with SHA512[10]. The resulting 512-bit value becomes `localRandomValue` in the pulse. If *any* of the internal RNGs generate unpredictable, random values, then the result will also be unpredictable and random. Even if one of the RNGs is flawed or backdoored, the value of `localRandomValue` will still be secure, as long as at least one of the RNGs is good.

However, there is no way for any outside observer to verify `localRandomValue` is generated in this way. If the beacon began simply putting any value it wanted in that field, there would be no way for users to detect this.

3.5 Signing Pulses

At any given time, the beacon is using a particular signing key. The beacon also has a corresponding certificate, issued by a well-known commercial CA. (The beacon frontend has an entirely different certificate and key used so that it can support TLS; there is no relationship between these keys.)
In the new format, each pulse contains:

- `signatureValue` is an RSA[11] signature over the entire contents of the pulse.
- `certificateId` contains the `SHA512`() of the X.509[3] certificate of the signing key used for pulses. (The full X.509 certificate is available from the beacon frontend.)

Having each beacon pulse digitally signed accomplishes two security goals: First, it ensures that impersonating the beacon will be difficult. Second, it ensures that a beacon which issues invalid pulses is creating permanent evidence of its misbehavior.
In the NIST beacon, the RSA key used for signing the pulse is kept inside a commercial HSM. The HSM is designed to prevent the beacon engine from extracting the RSA key. This limits the ability of some attackers (both insiders and outsiders) to carry out some attacks, as discussed in detail below.

3.6 The Output Value

The current beacon format (like the previous one) defines the output value of the pulse as the final field in the pulse, labeled `outputValue`. The output value is defined as

$$\texttt{outputValue} \leftarrow \texttt{SHA512}(\text{ all other fields in the pulse }) \tag{2}$$

Because this includes the value of `localRandomValue`, `outputValue` contains the all the randomness in the pulse. However, users of the beacon can always verify that `outputValue` was computed correctly. As described below, the fact that users of the beacon will normally use `outputValue` has important security consequences.

4 Limiting the Beacon's Power

4.1 Attacks

Above, we noted the sorts of bad behavior possible for a corrupt beacon operator:

- Reveal future random values early to favored people.
- Control or influence random values to undermine the security of some user.
- "Rewrite history" by altering old pulses, thus changing claimed historical results.

A good design of the beacon format and the normal operations of a beacon should ideally make such violations impossible; if not, it should at least make them harder, or at least detectable. In this section, we discuss mechanisms in the new beacon format which either add difficulty to misbehavior by the beacon, or which support users of the pulses protecting themselves from misbehavior.

4.2 Prediction: Revealing the Value of `outputValue` in the Future

The first way a beacon operator can misbehave is to reveal future outputs to selected people.

A beacon that is operating correctly generates a new random value very soon before issuing a given pulse, and so can't possibly tell anyone a value of the pulse very far into the future. However, a misbehaving beacon operator can, in principle, generate its pulses far in advance. All the computations needed to compute a pulse are relatively cheap, so computing a year's worth of pulses in advance can easily be done in an hour or two.

There are two limits on the ability of a corrupt beacon to reveal future pulses:

The Certificate ID The `certificateId` is the `SHA512`() of the certificate currently being used to sign pulses. Certificates normally have a fixed validity period–for example, one year. A corrupt beacon operator cannot predict the value of a certificate until he has seen it. Therefore, the validity period of the certificate imposes a (very generous) limit on the ability of the beacon to precompute future pulses.

Suppose each certificate has a validity period of one year, and is issued one month ahead of its validity period. Then the beacon is limited to precomputing pulses no more than 13 months in advance.

The External Source The new format allows beacons to optionally incorporate an *external source* into pulses. For example, a given beacon could incorporate the winning Powerball lottery numbers twice per week, or the closing value of the DJIA at the end of every stock trading day. By incorporating an external source that is outside the control of the beacon, the beacon operator can demonstrably lose a huge amount of power.

Each pulse contains a field called `external.value`, which contains the `SHA512`() of the current external value. It is updated only when a new external value appears; otherwise, it retains the value it had previously.

Each pulse also contains a field called `external.sourceId`, which contains the `SHA512`() of a text description of the external source and how and when the external value will be updated. This should almost never change.

Suppose a beacon incorporates a lottery drawing which occurs once per day as its external source; the result is incorporated six hours after the result is made public. In this case, the beacon is limited to precomputing future values no more than 30 hours in advance.

Prediction: Summary

1. A corrupt beacon can precompute its pulses far in advance and reveal the output values to friends.
2. Incorporating an optional `external.value` into the pulse limits the precomputation (and thus the ability to reveal future outputs) to the time between updating the `external.value`.

4.3 Influencing Outputs

Recall that the output of the pulse is computed as:

$$\texttt{outputValue} \leftarrow \texttt{SHA512}(\text{all other fields of the pulse})$$

Further, the recipient of the pulse can verify that this value is correct by recomputing the hash. This means that, while the beacon operator can completely control the value of `localRandomValue`, he cannot directly control the value of `outputValue`. Instead, in order to exert control over this value, he must try many different inputs; each input leads to a different random output value. With 2^k tries, he can expect to get any property he likes for `outputValue` which has a probability no lower than 2^{-k}.

For concreteness, in the rest of this discussion, we will assume that the beacon operator wants to force the least significant bits of a given pulse's `outputValue` to be zeros. Thus, with 2^k work, he can expect to force the low k bits to zero. This generalizes; an output value with any property that has a 2^{-k} probability can be found about as easily.

From [1], we can get benchmarks for an 8-GPU desktop machine doing brute-force hashing[2]. The listed system can do about 8.6 million SHA512 hashes per second, which is about 2^{23} hashes per second. An attacker with such a system could do about 2^{39} hashes per day, or 2^{47} hashes per year. Because the whole process is parallelizeable, spending N times as much money will get an N-way speedup. However, in order to add one bit of control of `outputValue` with the same hardware, an attacker must double the time taken for the computation.

At the time of this writing, the most powerful attacker imaginable might conceivably have the computing resources to compute 2^{90} SHA512 hashes in a year. This puts an upper bound on the beacon operator's control of `outputValue`. However, for concreteness, we will assume in the rest of this discussion that the attacker has the 8-GPU desktop machine described in [1], and is trying to control bits of `outputValue` using it.

There are several components of the new pulse format which limit the attacker's control of the output bits.

The Certificate ID As described above, the beacon operator can't predict the value of its next certificate, incorporated into every pulse in `certificateId`. This limits the attacker to no more than about 13 months of precomputation–with only about a year to do his attack, he is limited to controlling 47 bits of `outputValue`.

[2] That system is doing password cracking attacks. Trying to control some bits of the output of the beacon is very similar to password cracking.

The Signature Each pulse contains an RSA signature, `signatureValue`, over all fields except the `outputValue`. This affects the attack in two ways: First, it means that each new value of a pulse will require a new RSA signature–adding slightly to the work needed per new output value computed. This probably reduces the attacker's ability to control `outputValue` by a small number of bits.

More importantly, in the NIST beacon, the RSA signing key is stored inside an HSM, and should be very difficult to extract. If the attacker doesn't have access to the signing key[3], then the attacker must involve the HSM in every new computation of an output value.

Suppose the HSM can do 100 RSA signatures per second. Then, the attacker is limited to only about 2^{32} trial values for the output in a year of trying, and so can control only about 32 bits of `outputValue`.

`external.value` As described above, the new format allows beacons to optionally incorporate an *external source* into pulses.

Once again, suppose the external source is updated from a lottery drawing carried out each day. The lottery drawing results become public six hours before the value is incorporated into the `external.value` field. (This must be described precisely in the text whose hash appears in `external.sourceId`.) The attacker's ability to control outputs is now enormously diminished–his precomputations can now run for at most 30 hours.

An attacker who has extracted the RSA key from the HSM and has the 8-GPU system can thus carry out no more than 2^{40} computations, and so can control about 40 bits of `outputValue`. An attacker who hasn't extracted the RSA key from the HSM can control only about 23 bits of the output value.

This makes a pretty good argument for the idea that the private key for the beacon should be generated inside the HSM once and never released to anyone, even encrypted. In the best case, we'd have some kind of guarantee that even the beacon operator could never see the private key.

Influence: Summary Consider an attacker trying to control `outputValue`:

1. A vastly powerful and well-funded attacker might be able to control as many as 90 bits given a year of computation.
2. An attacker with a powerful setup for password cracking can control 47 bits in a year of computation.
3. When the attacker is unable to extract the RSA key from the HSM, his attack is limited by the speed of RSA signatures from the HSM. If the HSM can do 100/second, then the attacker can control about 40 bits of the output value with a year of computation.
4. If the beacon incorporates external values, the attacker's power is massively diminished. Assuming an external value known to the attacker for only 30 hours:

[3] This might be an outsider who has compromised the beacon engine, or an insider with access to the engine but not the HSM or RSA keys.

(a) With the RSA key from the HSM, the attacker can control 40 bits.
(b) Without the RSA key from the HSM, the attacker can control 23 bits.

It is important to note that there is no way for any user to detect these attacks from the outside–as far as the user is concerned, the beacon is behaving normally, and the pulses look just like any other pulses.

4.4 Combining Beacons

Having multiple beacons operating at the same time gives users a choice of which beacon to trust. However, multiple beacons allow a user to *combine* outputs from two or more beacons. The goal of combining beacons is to reduce trust in the beacon operators: if we combine pulses from N beacons and at least one is honest, then the resulting random value should be random, unpredictable, and outside anyone's control. Instead of having to trust a single beacon operator, the user can end up needing to trust that any one of two or three operators is honest.

Notation for talking about beacons In order to discuss combining of beacons, we need to introduce some notation: Suppose A and B are beacons:

- $A[T]$ is beacon pulse at time T from A.
- $B[T]$ is beacon pulse at time T from B.
- $B[T+1]$ = next pulse, $B[T-1]$ = previous pulse.
- $B[T]$.`localRandomValue` is the `localRandomValue` field of the pulse at time T from beacon B.

Combining Beacons: What doesn't *work?* A natural approach would be to XOR the `outputValue` fields from two beacons, getting

$$Z \leftarrow A[T].\texttt{outputValue} \oplus B[T].\texttt{outputValue}$$

For simplicity, let's suppose A is a corrupt beacon and B is an honest one.

Combining the beacons even in this simple way prevents the prediction attack: A can't know what random number B will produce at time T, so he cannot reveal the future Z to his friends.

Unfortunately, A can still exert control over the combined output Z. A uses the following strategy to exert control over the bits of Z, despite B's genuinely random contribution:

1. B does a precomputation of 2^{32} possible values of $B[T]$.`outputValue`.
2. A sends pulse $A[T]$.
3. B observes $A[T]$.`outputValue`
4. B chooses $B[T]$ to control low 32 bits of

$$A[T].\texttt{outputValue} \oplus B[T].\texttt{outputValue}$$

.

This attack led us to add a new field to the beacon format: `precommitmentValue`.

$$A[T-1].\texttt{precommitmentValue} = \texttt{SHA512}(A[T].\texttt{localRandomValue})$$

That is, $A[T-1]$ **commits** to `localRandomValue` in $A[T]$. Once a user has seen $A[T-1]$, A has **no choice** about the value of $A[T]$.`localRandomValue`. Note that `precommitmentValue` requires that the beacon engine compute the value of `localRandomValue` one pulse in advance.
By forcing the beacon to commit up front to the next pulse' random value, we can construct a protocol for combining beacons that is much more secure.

Combining Beacons the Right Way In order to combine beacons from A and B, the user does the following steps:

1. Receive $A[T-1]$ and $B[T-1]$.
2. Verify that both pulses:
   (a) Are valid
   (b) Were received before time T
3. Receive $A[T]$ and $B[T]$.
4. Verify that the `precommitmentValue` fields are in agreement with the `localRandomValue` fields:

$$A[T].\texttt{precommitmentValue} = \texttt{SHA512}(A[T].\texttt{localRandomValue})$$
$$B[T].\texttt{precommitmentValue} = \texttt{SHA512}(B[T].\texttt{localRandomValue})$$

5. Compute combined value by hashing together:
   - `outputValue` fields from $T-1$
   - `localRandomValue` fields from T

$$Z \leftarrow \texttt{SHA512}(A[T-1].\texttt{outputValue} \parallel B[T-1].\texttt{outputValue} \parallel A[T].\texttt{localRandomValue} \parallel B[T].\texttt{localRandomValue})$$

This is a somewhat more complex way of combining beacons, but the added complexity adds security:

1. By incorporating $A[T-1]$.`outputValue` and $B[T-1]$.`outputValue`, we incorporate the external value, RSA signatures, and other security mechanisms described above.
2. By incorporating $A[T]$.`localRandomValue` and $B[T]$.`localRandomValue`, we incorporate a fresh random number from each beacon. Any honest beacon will provide a completely unpredictable random value.
3. Thanks to the `precommitmentValue` fields in $A[T-1]$ and $B[T-1]$, neither beacon can change the value of their `localRandomValue` fields after they see the random value from the other beacon.

As long as the beacons output their values on time, the result is that if at least one beacon is honest, Z is random, it can't be predicted by anyone before time T, and it cannot be influenced by either beacon.

Hitting the RESET button Combining beacons massively improves security, but isn't quite perfect–a corrupt beacon still has a very small amount of influence.

Once again, suppose A is an evil beacon, and B is an honest one. The attack works as follows:

1. Both beacons send out pulses for time $T - 1$.
2. At T, B sends out $A[T]$.
3. A computes combined output Z.
4. If it doesn't like Z, it simulates a failure.

Note that this attack follows a common pattern: simulating a failure that really could happen for innocent reasons. If the corrupt beacon is willing to do this once for a given attempt to combine beacons, then it gets a very small amount of influence on Z–it gets one opportunity to veto a value of Z it finds unacceptable. Along with massively decreasing the power of a corrupt beacon, this attack also forces a corrupt beacon trying to influence the combined output to take a visible action–it must simulate a failure and stop producing pulses for awhile. This is very visible to the user trying to combine beacon pulses, and also leaves a permanent record in its sequence of pulses.

Summary: Combining Beacons

1. Combining beacons massively decreases the scope of a corrupt beacon–as long as at least one beacon used is honest, the user's security is massively improved.
2. Prediction attacks become impossible–a corrupt beacon cannot reveal the value of a future combined value to a friend even one minute in advance.
3. Influence attacks become much harder–a corrupt beacon can veto one combined value, but must do so in a way that is obvious and leaves a permanent record in the chain of pulses.

4.5 Rewriting History

Both the old and new beacon pulse format have two fields that make it impossible for a beacon operator to "rewrite history" without leaving permanent evidence that it has done so.

First, each pulse has a `signatureValue` field, signing the pulse with a key for which the beacon operator has the key and certificate. Second, each pulse contains the `outputValue` of the previous pulse, in its `previous` field. Since the output value is the hash of the contents of the pulse, this means that a sequence of pulses forms a *hash chain*.

A hash chain has the property that introducing a change anywhere in the chain requires changing every block *after the chain.* Figure 2 shows the hash chain (with most fields omitted for clarity) when a change is made to a pulse.

The result of this is that if the beacon operator makes such a change to some pulse in the past, then that pulse's output value becomes inconsistent with the later pulses in the chain–pulses that users have already seen and stored, and that have valid signatures from the beacon operator. A change to any pulse leaves an inconsistency in the chain. Examining the chain will always detect the inconsistency.

4.6 Changing a pulse changes all future pulses!

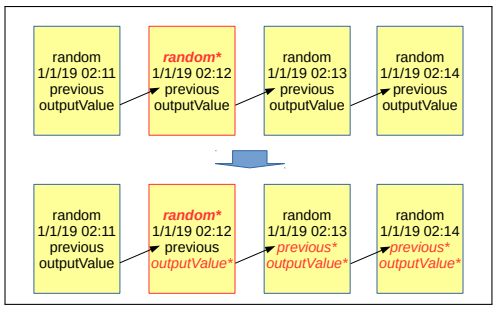

Fig. 2. Altering a pulse affects every future pulse in the hash chain

The result of this is that any attempt to "rewrite history" by changing the value of some past pulse creates permanent evidence of the beacon's misbehavior. Unfortunately, verifying that the beacon hasn't rewritten history can be expensive. Consider the situation where a user know the value of 2022-10-04 17:35, and wants to use this to verify the value of 2019-11-29 22:58. There are about 1.5 million pulses between these two, so a user wanting to do this verification would have to request 1.5 million pulses from the beacon, and then verify the entire hash chain.

Skiplists An auditing step that is unworkably complex and expensive will seldom be done. In order to make this auditing step more efficient, we added several fields to the pulse format, in order to support a much more efficient way to verify that a given pulse has not been changed, by returning a sequence of pulses we refer to as a *skip list*. A skiplist between two pulses, TARGET and ANCHOR, guarantees that pulse TARGET is consistent with the value of pulse ANCHOR–the beacon cannot construct a valid skiplist if the value of TARGET has been changed.
A hash chain used to verify the value of the pulse at 2019-11-29 22:58 given knowledge of the value of the pulse at 2022-10-04 17:35 would require about 1.5 million pulses; the skiplist requires nine pulses. More generally, when the known (ANCHOR) pulse and the pulse to be verified (TARGET) are Y years apart, the skip list will be about $Y + 46.5$ pulses long on average.
The construction of skiplists is described in detail in appendix A.

5 Conclusions and Open Issues

In this article, we have described the design of the new NIST beacon format from the perspective of minimizing the power of a beacon operator to misbehave. By adding fields to the pulse format, and defining a protocol for combining pulses from multiple beacons, we have massively

decreased the power of the beacon operators to misbehave without detection. In the case where a user can combine pulses from two independent beacons, he can be extremely secure from misbehavior by any one beacon operator.

Attack	Resources and Modifications	Vulnerability
Prediction[4]		
	Original format	unlimited
	No external value	13 months
	External value	30 hours
	Combined pulses	attack blocked
Influence[5]		
	Original format	At least 47 bits
	No external, RSA key known	47 bits
	No external, RSA key unknown	32 bits
	External, RSA key known	40 bits
	External, RSA key unknown	23 bits
	Combined pulses	1 bit, but attack is visible
Rewriting History		
	Original format	Permanent evidence, expensive verification
	New format	Permanent evidence, cheap verification

Table 1. What can an evil beacon operator do?

Table 1 shows how various ways for the beacon to operate and be used affects the vulnerability of the users to attacks by a corrupt beacon operator. The important points from this table are:

1. A beacon that includes an external value *enormously* reduces the its own scope for misbehavior.
2. Combining pulses from multiple beacons gives very high security, assuming at least one beacon is honest. (If all the combined beacons are corrupt, then it makes no difference.)

There are two possible directions for further work: we can make changes to future versions of the beacon format, and we can add new recommended protocols for users of the beacon.

5.1 Improving the Beacon Format

The influence attacks described above are computationally expensive. If the output value were computed using a memory-hard function designed to foil password cracking attacks, such as scrypt[6] or Argon 2[2], these attacks would become enormously less practical. Even implementing scrypt with a minimal set of hardness parameters would easily cut ten or more bits off the ability of an attacker to influence outputs.

The "hitting the RESET button" attack exploits the fact that when a beacon commits to its next pulse's `localRandomValue` field, it can still refuse to disclose that. In [4], the authors propose a number of techniques to foil this attack, including the use of time-lock puzzles[8] for the random value in each pulse, and the use of "Merlin chains" to take away any ability for a beacon operator to simulate a failure without permanently shutting down the beacon. We could consider adding these to a future beacon format.

5.2 Improving Protocols and Recommendations for Users

A simple way for a user to protect itself from a corrupt beacon is simply to pre-commit to how it will use the beacon pulse at time T, and also to pre-commit to a secret random number kept by the user. The user reveals his random number at the same time the pulse is revealed, and derives a seed by hashing the pulse's output value and his own random number together. Note that this completely blocks any influence or prediction by the beacon operator, assuming the user is honest.
Instead of adding a memory-hard function to the calculation of the `outputValue`, we could also recommend using a memory-hard function for the computation of the final seed. By choosing the hardness parameters to make this seed calculation take 30 seconds on a reasonably fast desktop machine, the user can massively increase the difficulty of influencing attacks by the beacon operator.

References

[1] *8x Nvidia GTX 1080 Hashcat Benchmarks*. [Online; accessed 09-July-2018].

[2] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications". In: *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. 2016, pp. 292–302. DOI: 10.1109/EuroSP.2016.31. URL: https://doi.org/10.1109/EuroSP.2016.31.

[3] David Cooper et al. "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile". In: *RFC* 5280 (2008), pp. 1–151. DOI: 10.17487/RFC5280. URL: https://doi.org/10.17487/RFC5280.

[4] Peter Mell, John Kelsey, and James M. Shook. "Cryptocurrency Smart Contracts for Distributed Consensus of Public Randomness". In: *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*. 2017, pp. 410–425. DOI: 10.1007/978-3-319-69084-1_31. URL: https://doi.org/10.1007/978-3-319-69084-1%5C_31.

[5] *NIST Randomness Beacon*. https://www.nist.gov/programs-projects/nist-randomness-beacon. [Online; accessed 09-July-2018]. 2018.
[6] Colin Percival and Simon Josefsson. "The scrypt Password-Based Key Derivation Function". In: *RFC* 7914 (2016), pp. 1–16. DOI: 10.17487/RFC7914. URL: https://doi.org/10.17487/RFC7914.
[7] Michael O. Rabin. "Transaction Protection by Beacons". In: *J. Comput. Syst. Sci.* 27.2 (1983), pp. 256–267. DOI: 10.1016/0022-0000(83)90042-9. URL: https://doi.org/10.1016/0022-0000(83)90042-9.
[8] R. L. Rivest, A. Shamir, and D. A. Wagner. *Time-lock Puzzles and Timed-release Crypto*. Tech. rep. Cambridge, MA, USA, 1996.
[9] T. C. Schelling. *The strategy of conflict*. Oxford University Press, 1960.
[10] National Institute of Standards and Technology. *FIPS 180-4, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-4*. Tech. rep. DEPARTMENT OF COMMERCE, 2015. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.
[11] National Institute of Standards and Technology. *FIPS 186-4, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 186-4 Digital Signature Standard (DSS*. Tech. rep. DEPARTMENT OF COMMERCE, 2013. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.
[12] Nick Szabo. *Trusted Third Parties are Security Holes*. [Online; accessed 09-July-2018]. 2001.

A Skiplists

In order to support skiplists, we added four additional fields to the pulse format which contain the `outputValue` of previous pulses. (`previous` was defined in the old beacon format.)

`previous`	`outputValue` of previous pulse.
`hour`	`outputValue` of 1^{st} pulse of *hour* of previous pulse.
`day`	`outputValue` of 1^{st} pulse of *day* of previous pulse.
`month`	`outputValue` of 1^{st} pulse of *month* of previous pulse.
`year`	`outputValue` of 1^{st} pulse of *year* of previous pulse.

By adding these fields, we ensure that instead of a sequence of pulses having a single hash chain, any long sequence of pulses has a huge number of different hash chains running through them, of varying lengths, as shown in figure 3.
The algorithm for extracting a minimal chain (what we call a skiplist) is illustrated in figure 4.
A more precise description of the algorithm appears in algorithm 1.

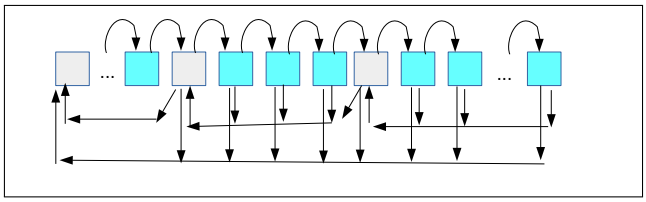

Fig. 3. A long sequence of pulses has a huge number of different hash chains

Fig. 4. Constructing a skiplist from TARGET to ANCHOR

In this case, the result is that instead of needing to verify a chain of 1.5 million pulses, the user requests a skiplist of nine pulses from the beacon frontend, and verifies them very efficiently. If the beacon has altered the TARGET pulse, it will not be able to provide a valid skiplist from TARGET to ANCHOR.
In general, a skiplist between a value of TARGET and ANCHOR that are Y years apart will be about $Y + 46.5$ pulses long.

Algorithm 1 Construct a skiplist from TARGET to ANCHOR.

```
1: function MAKE_SKIPLIST(TARGET, ANCHOR)
2:     path ← []
3:     current ← TARGET
4:     while current< ANCHOR do
5:         path ← pp ||current
6:         if current is first pulse in its year then
7:             current ← first pulse in NEXT year
8:         else if current is first pulse in its month then
9:             current ← first pulse in NEXT month
10:        else if current is first pulse in its day then
11:            current ← first pulse in NEXT day
12:        else if current is first pulse in its hour then
13:            current ← first pulse in NEXT hour
14:        else
15:            current ← NEXT pulse
16:    path ← path ||ANCHOR
17:    return (path)
```