Summary

We examine the topic of randomness beacons, services that provide public randomness, and identify a gap in the current literature. The gap consists of the practical implementation and security analysis of randomness beacons. We seek to design and implement a secure beacon that can be used in real life. We explore a set of use cases to motivate the use of randomness beacon, and argue for the necessity of security under these circumstances.

This work is a continuation of the work we did in the previous semester: We analysed the structure of beacons and how they are commonly structured and used. This culminates in three operational models for beacons, the *Autocratic Collector*, *Specialized MPC* and *Transparent Authority*, and three input sourcing models, *private input sources*, *publicly available sources*, and *user input*. These models are referenced throughout the paper as a means of describing key approaches to analysis, design, and implementation.

To better understand how to design a secure beacon, we analyze the threats towards a randomness beacon. We identify threats both from outsiders, i.e. the users of the beacon, and insiders, also called the beacon operator, and estimate the different severities of each threat. The estimation of the threats is driven by the *DREAD* method, which we slightly alter to fit the analysis of randomness beacons.

Based on the threats discovered to beacons, as well as the operational and input models previously examined, we construct a set of requirements for our own beacon. These requirements address the criteria for a randomness beacon, given our world view — nobody can trust anyone but themself. The requirements form the base of our design, which encompasses both the architecture and security properties of the beacon.

We scrutinize which factors define how users should trust the beacon, and design our major contribution; a beacon protocol that uses a series of time offset delay functions to provide randomness at regular intervals. We show it enables users to have a probabilistic level of trust in the beacon, meaning that each user will use their own assumptions about the world we live in, to assign a certainty to the trustworthiness of the randomness beacon. All this is done, while ensuring a scalable and easily deployable beacon.

We also go in depth with the implementation of the beacon, and detail which choices have been made in the process. This includes the tools and frameworks used in the construction, and we also consider their effects on the security of the beacon. We break down the beacon and explain how each component works, and how they live up to the previously established requirements and design.

To evaluate the performance of our beacon, we select key areas to examine, such that reasonable statements can be made about the randomness beacon as a whole. This performance evaluation includes benchmarks of our expected bottlenecks, and an assessment of the intricate computations of the beacon. We find that our beacon fulfills our expectations when it comes to performance, and will be able to handle virtually any real world usage.

To contextualize the beacon we designed and implemented, we present a series of applications of our beacon, detailing how the beacon should be used to guarantee security. This involves perceiving the randomness beacon as a cryptographic primitive used to build secure ceremonies. We discuss topics that have come up during the process of developing our beacon, as well as any future work relevant to our beacon and randomness beacons in general. Lastly, we conclude on how our design and implementation fulfill the requirements, and how the beacon mitigates the threats found in the security analysis.

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
http://www.cs.aau.dk

Title:
Developing a Trustworthy Randomness Beacon for the Public

Subtitle:
Ensuring Rational Trust in a Hostile World

Project period:
Spring semester, 2018

Project group:
deis1014f18

Participants:
Michael Toft Jensen
Sebastian Rom Kristensen
Mathias Sass Michno

Supervisors:
René Rydhof Hansen
Stefan Schmid

Pages: 28

Date of completion:
June 8, 2018

Abstract:

Randomness beacons are services that emit a random number at a regular interval. A recent trend in these beacons is making them transparent or secure such that no party can covertly influence the output without detection. Existing literature on the subject lacks a bridge from their respective theoretical solution to a practical implementation suitable for deployment and usage. Much of the existing literature also lacks a structured security analysis.
We close these gaps by designing and implementing a randomness beacon supporting a broad range of use cases. The randomness beacon is designed to be practical in the real world while prioritizing the security and integrity of the output.
Our randomness beacon is based on a service oriented architecture and supports a multitude of input and output channels. The transformation from input to output utilizes a Commit-Compute-Output (CCO) workflow combined with a delay function. Together, these provide good security guarantees even under the assumption that everybody else is colluding against you. Our beacon greatly minimizes the amount of trust needed in such a way that each user can decide how much they want to trust. As such, even fastidious users can be serviced by our randomness beacon.
Lastly, we explore a variety of applications for our randomness beacon as a cryptographic primitive, and discuss how to use it in a secure way.

The contents of this paper are freely available, but publication (with reference) may only be pursued with permission from the authors.

Developing a Trustworthy Randomness Beacon for the Public

Ensuring Rational Trust in a Hostile World

Michael Toft Jensen
micjen12@student.aau.dk

Sebastian Rom Kristensen
sromkr13@student.aau.dk

Mathias Sass Michno
mmichn13@student.aau.dk

Abstract

Randomness beacons are services that emit a random number at a regular interval. A recent trend in these beacons is making them transparent or secure such that no party can covertly influence the output without detection. Existing literature on the subject lacks a bridge from their respective theoretical solution to a practical implementation suitable for deployment and usage. Much of the existing literature also lacks a structured security analysis.

We close these gaps by designing and implementing a randomness beacon supporting a broad range of use cases. The randomness beacon is designed to be practical in the real world while prioritizing the security and integrity of the output.

Our randomness beacon is based on a service oriented architecture and supports a multitude of input and output channels. The transformation from input to output utilizes a Commit-Compute-Output (CCO) workflow combined with a delay function. Together, these provide good security guarantees even under the assumption that everybody else is colluding against you. Our beacon greatly minimizes the amount of trust needed in such a way that each user can decide how much they want to trust. As such, even fastidious users can be serviced by our randomness beacon.

Lastly, we explore a variety of applications for our randomness beacon as a cryptographic primitive, and discuss how to use it in a secure way.

1 Introduction

A *randomness beacon*, i.e. a service emitting unpredictable random values at fixed intervals, is not a new concept. In 1983 Michael O. Rabin coined the term and utilized one to add probabilistic security in several protocols [21]. In this definition, a randomness beacon was to be seen as a third-party trusted to be unbiased towards any outcome. As such, you should trust the beacon operator (the entity running the beacon service) to not be biased since you can not verify that they were unbiased.

For quite a while, randomness beacons did not receive much attention, likely because alternatives to Rabin's protocols not requiring a trusted beacon were used instead (such as [2]). In circa 2010, a renewed interest in beacons was seen as an increase in beacon-related literature; the trend in this new literature was to remedy the need to trust the beacon operator. We believe it was a reaction to revelations like the National Security Agency (NSA) whistle-blower leaks that diminished people's trust in authorities. In other words, people had their eyes opened to the fact that *trust* can be an issue in itself and that removing the need for trust in any one entity could be beneficial. As an example, cryptocurrencies have flourished in recent years alongside a sharp rise in the popularity of blockchains — two technologies seeking to facilitate cooperation of mutually distrustful users.

Conceptually, randomness beacons seem to fit this environment of minimizing the need to trust. However, the *old generation*, requiring users to trust the operator, simply shifts the trust issue to the centralized entity, i.e. the beacon operator. In *new generation* beacons described in the recent literature, the beacon is acting as an impartial party.

The merit of a randomness beacon lies in contexts where a set of users needs to agree on some random outcome, but do not trust each other or a third party to make the decision. Therefore, a randomness beacon is *not* required in the case a user needs randomness for just themself. In this case, standard ways, e.g. using `/dev/urandom`, of generating ran-

domness on a computer are far easier. Similarly, if users trust each other or a third party randomness generation is also trivial. A randomness beacon is not necessarily generating "more random" numbers — it merely allows users to agree on the same randomness without trusting anyone.

Even though the literature theoretically argues for a variety of solutions, we have not seen many implementations of randomness beacons. We believe that bridging a theoretical design and practical implementation is uncharted territory. Additionally, the literature which somewhat explore this bridge either culminate in a highly specialized solution unfit for a public context [8, 25], or disregards a thorough, structured security analysis [7]. Throughout this paper we design, implement, and evaluate a randomness beacon, borrowing from existing research while introducing novel ideas; both in the system but also the operation of it.

In any case, randomness beacons are interesting as a concept, and we feel it needs further exploration to find real world applications. This paper will be a step in that direction.

Regarding terminology, we use the terms "randomness beacon" and "beacon" interchangeably in this work.

1.1 Security Goals

Using an *old generation* beacon simply shifts the issue of trust to the beacon. Therefore, we strive to design and implement a *new generation* beacon that works on the most pessimistic assumption possible: "Everybody is secretly colluding against you and is willing to put an unlimited amount of money and resources towards manipulating or biasing the randomness. As such, you can only trust yourself."

These assumptions describe the mindset we take on while designing and implementing the beacon. We have not seen any work on beacons that can guarantee a completely trustless beacon. Therefore, such a trustless system may not be practically feasible today. As we also need to account for the feasibility of the system in real world applications, we consider degrees of trust, i.e. we perceive it as a system with variables. We seek to minimize the trust required and ultimately let each user decide how much they *want* to trust. In essence, a user will know that under self-chosen trust assumptions, the randomness has not been manipulated.

1.2 Beacon Context

As stated, beacons are relevant in contexts where several users want to agree on some random outcome, but do not trust anyone to solely decide that outcome — a pattern which fits a number of use cases.

Consider the generic use case of sampling. Essentially, sampling is about selecting representative data points, potentially with high stake consequences. It would not be far-fetched to imagine someone wants to bias this sampling process in order to skew the results. One such sampling process is lotteries, which need to randomly sample a pool of participants to draw winners.

The field of cryptography also contains use cases. Many cryptographic schemes require some constants in the design of algorithms, and it has been shown that some schemes have been intentionally built with specific constants in order to facilitate a backdoor [13]. Selecting constants with a randomness beacon can prove to the users of such cryptographic schemes that the constants were not manipulated and thus are unlikely to contain backdoors [1]. One could even in some use cases imagine a "refreshing" algorithm where constants are variables which change with new beacon outputs.

Staying in the field of cryptography, zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) require a lengthy process of initial bootstrapping. In systems such as *zcash*[1], this bootstrapping has been performed by a multi-party computation (MPC) [5]. However, the MPC scales poorly because of the many rounds of communication needed between parties alongside big amounts of data to ensure a fair output. Bowe, Gabizon, and Miers [6] suggest avoiding the heavy communication and computation, and instead propose a simpler MPC where users directly contribute a random number. To avoid the last user carefully choosing their input to manipulate the bootstrapping to their benefit (a so-called *last-draw attack*), an output from a randomness beacon is applied as the last input. As such, they decrease the number of rounds in the MPC protocol from four (plus several subprotocol rounds) to just two, decrease the amount of communication between users, and decrease the complexity of the computation. Thus, a lengthy MPC is substituted with a quick round of user input and sealing the deal with a randomness beacon.

[1] https://z.cash

1.3 Beacon Concepts

A randomness beacon emits an unpredictable random value at a regular interval, e.g. every five minutes. Figure 1 shows the workings of a simple, generic beacon. The beacon performs *some* computation on an input source in order to generate an unpredictable number. The result of this computation is sent to users. This workflow is repeated indefinitely at the specified interval.

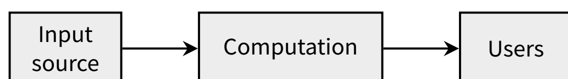


Figure 1: Abstract randomness beacon

Examining existing beacons [1, 3, 4, 7, 8, 9, 11, 14, 18, 22, 25], a few common ways of composing the input sourcing and computation are discernible and can be described as specific models. Here, we distinguish between an input source model and operational model. The input source model describes the way the beacon sources its input, while the operational model describes the design of the protocol, i.e. how to perform the computation and publish the output. These models are based on our earlier work [17], a survey of different approaches to randomness beacons as presented in literature.

1.3.1 Input Source Models

Based on a survey of randomness beacons we made in previous work [17], there are three sources of input:

Private Input Sources A beacon can use a private source of data to produce randomness. This allows them to produce randomness of high quality at a high rate, but since users are seldom present to physically inspect this private source, it requires users of the beacon to trust the beacon and its randomness. This does not align well with our aforementioned security goals, since inputs cannot reliably be distinguished from carefully crafted values that appear to be random.

An example of this input source model is the National Institute of Standards and Technology (NIST) randomness beacon [18] which observes quantum mechanical effects to produce what they claim is high-quality randomness. Ultimately it requires trust, since the observations cannot be repeated, and therefore users cannot make sure that the value is indeed from observing the quantum mechanical effects. As such, the users need to blindly trust the beacon operator, which in the case of NIST can be hard given their history [13, 19, 20].

Publicly Available Sources This input source is using publicly available sources that everyone can agree on the value of, such as bitcoin blocks, stock market data, or lottery winning numbers from several international lotteries. The user must trust the source, and this is reasonable since these sources are governed by some guarantees, and often it is easy to see the freshness of these values. In case of bitcoin, the blocks have a monetary value and are virtually impossible to forge. Users have to interact with the source to indirectly influence the beacon and prevent biased outputs. However, it may also be harder for adversaries to bias the beacon through the source unless they are in complete control of the given source.

User Input A user can be allowed to directly provide input to the beacon. The idea is that a user provides a value that they firmly believe is sufficiently random, such that nobody could have predicted the value. In other words, users know their own value is fresh, and no other party could realistically have used this value before. There exists no concept of ownership of specific values, which means that users should trust the input they give, and not the fact that it is their own. However, for the sake of simplicity we, throughout this paper, refer to an input trusted by a given user to be said user's input.

The beacon performs an operation on a set of user-supplied inputs. The output of the beacon is structured in a way that *a*) allows all users to verify the inclusion of their input and *b*) allows all users to verify the validity of the computation.

If these are satisfied, the user knows that a value they trust to be random has been part of the random output generation. The computation performed by the beacon should ensure that users cannot knowingly bias the output to anyone's disadvantage. As such, the user knows that his input was not knowingly "counteracted" by another used.

1.3.2 Operational Models

Alongside presenting input source models in our previous work [17], we also identify three ways in which a beacon is typically operated:

Autocratic Collector A beacon is run by a party, which deems it irrelevant to prove honesty, thus requiring blind trust from the users. As such, the computation is a black box with no possibility for proof of honesty. This type lies in the trust-requiring category of beacons, i.e. the old generation.

Specialized MPC Users utilize multi-party computation (MPC) to collectively produce randomness, typically from their own inputs. Given an honest majority, this type of beacon produces randomness that is not biased against the participants, and although work has been done in the field, they are difficult to scale to large groups since any addition or removal of a user requires a new setup phase [8, 25]. This type of beacon is therefore unsuited for public settings, but might fit in a controlled private context.

In Section 1.6.1 we describe the state of the art of specialized MPC randomness beacons.

Transparent Authority A single entity collects input and publishes it with a focus on transparency. Users can, by observing the beacon, verify that it behaves according to protocol. This does not directly prevent byzantine behavior, but rather makes it difficult or nearly impossible to hide such behavior. This type also support a wide variety of implementations, and is potentially scalable to a public setting.

1.4 Delimitations

We want to create a public randomness beacon that is secure under the assumption that everyone may be colluding against a given user, as per our security goals in Section 1.1. As such, we can immediately see that the autocratic collector is not suitable for our security goal, because it requires users to trust its claimed honest operation. The MPC model does not scale well enough for general use in public randomness beacons [25]. As described by Damgård and Ishai [10], MPC protocols either assumes an honest majority, which is a weaker assumption than our security goals, or cannot guarantee fairness and output delivery, a likewise undesired behavior. Even as more scalable MPC protocols has been developed, they still need some assumptions about the protocol participants, which we cannot guarantee in a public randomness beacon. An example of this is the MPC protocol designed by Damgård and Ishai [10], which they claim is the first scalable general MPC protocol. This protocol still only allows for corruption of some constant fraction of the participants, and furthermore assumes a computationally simple pseudorandom generator to maintain a constant number of rounds needed to complete the protocol.

Since our world view of “everybody is colluding against you” is far more pessimistic than the guarantees provided by the aforementioned protocols, MPC is not suitable for us, although the model could fit in a more controlled or private environment. This leaves us with the transparent authority model which we adopt. This choice adds a role to our environment: the beacon operator. Our security goal of everybody colluding against a user must therefore be expanded such that “everybody” includes the beacon operator.

Regarding input source models, we can immediately discard private input sources as they are tied to the autocratic collector model, and as such do not work for the transparent authority. The guarantees by publicly available sources are weak compared to user input. If sufficiently paranoid, the user will want to bias these publicly available sources to make sure all other users are not colluding. Therefore, user input is the simplest solution and provides the strongest guarantees for the user.

1.5 Contributions

We bridge the gap between theoretical solutions and the real world by designing and implementing a secure, trust-minimizing randomness beacon based on the transparent authority model with user input. The design is based on a structured analysis of threats to a randomness beacon. We design the beacon from the ground up based on tried and tested methods as well as novel ideas. Specifically, we differ from previous transparent authority approaches by the following:

- We describe a novel way of parallelizing the computation in the beacon protocol to minimize possibility of malicious operation while avoiding idle periods.
- Unlike all other approaches of transparent authorities we have seen, the beacon operator in our beacon design has no private information — all inputs are hashed and are released to the public in batches.
- We choose to use Merkle trees as the data structure for inputs to allow reducing the computation proof size.

- We allow multiple input channels and output channels to be instantiated for reliability, distribution of workload, easy scaling, and convenience for the users.

Combined, our novel ideas significantly decreases the need for blind trust and limits the severity of a myriad of attacks. We evaluate our work both performance-wise and whether the design and implementation satisfy the requirements. Lastly, we explore usage of our beacon in context. As such, we describe several use cases and how the beacon output is used as a cryptographic primitive in a way that aligns with and extends our security goals.

1.6 Related Works

To establish an idea of the field of randomness beacons, both current literature and implementations, we present some related work.

1.6.1 Drand — A Distributed Randomness Beacon Daemon

The *Decentralized and Distributed Systems Research Lab* (DEDIS) at EPFL in Switzerland has developed an open source distributed randomness beacon called *Drand*[2]. The beacon uses the *Specialized MPC* computation model and deploys a set of limitations and assumptions which makes it well-suited for a private setting. Drand links nodes together to periodically and collectively produce what they claim is "publicly verifiable, unbiasable, unpredictable" random values. The beacon shares many authors with, and is based on, another paper regarding distributed randomness [25].

At its core Drand consists of two phases, a setup phase which requires knowledge of all participating nodes, and a randomness generation phase which must be initiated by a single leader. The setup phase and requirements for a leader to initiate the randomness generation makes the operation of Drand static, i.e. new nodes cannot join an already running protocol. However, due to the mechanisms underlying the randomness generation, faulty or unavailable nodes may not hurt the availability of the beacon, provided a defined threshold of running nodes is achieved. The details of the two phases are described in Appendix C on page 31.

[2] https://github.com/dedis/drand

While our beacon does not borrow many ideas from Drand, we believe that understanding a state of the art specialized MPC based randomness beacon is beneficial to underline the contrast and thus why we choose a transparent authority as operational model. One such contrast is the static nature of Drand and its participation scheme where we choose to have no notion of regular participants, but instead aim for a dynamic set of different users.

1.6.2 A random zoo: sloth, unicorn, and trx

Lenstra and Wesolowski [14] implement a protocol reminiscent of a beacon as a way to generate random numbers and parameters for elliptic-curve cryptography (ECC). They produce random numbers by collecting data from a variety of sources before running it through a time-hard delay function called *sloth*. Sloth is a strictly sequential function which is orders of magnitude faster to inverse for verification. The time-hardness prevents last-draw attacks, as attackers have to dedicate large amounts of time to compute how to bias the output, during which new inputs can render their efforts pointless.

The combination of input collection from multiple sources and then computing the output of a delay function, is presented as the *unicorn* protocol. This protocol resembles that of the *transparent authority* beacon computation model, and is done by a single entity. In the paper, Lenstra and Wesolowski suggest feeding *sloth* with an aggregation of user inputs, such as tweets, and private input sources such as a sampling of weather data. While they guarantee random unpredictable outputs even if all other users are malicious, they do not explore the scenario of a malicious operator, who colludes with adversarial users. Furthermore, the unicorn protocol lacks a concrete implementation and security analysis.

To generate the aforementioned ECC parameters, a final protocol named *trx*[3] is presented, which utilizes the output of the unicorn — thus completing the zoo analogy.

The *sloth* delay function will be a key part of our randomness beacon. However, the supporting structures driving the beacon will be different. We go in greater detail with the security of both the protocol and the beacon operator, and in particular assume the beacon operator can be malicious. We expand on this throughout the paper.

[3] pronounced like the T. rex dinosaur

1.6.3 Proofs-of-delay and randomness beacons in Ethereum

As an extension to the ideas presented by Lenstra and Wesolowski [14], the work of Bünz, Goldfeder, and Bonneau [7] uses a delay function and the bitcoin blockchain as a public available source. The idea of using a blockchain as a source of randomness is also seen in other previous work. They use a delay function to mitigate issues of biasing the blockchain in anyone's favor, and to limit the benefits of a block-withholding attack. These two attacks are claimed to be prevalent in naïve blockchain based randomness beacons without usage of a delay function.

The operation of the beacon is based on operator election, with the option for anyone to become the new operator. Outputs can be publicly contested, prompting the operator to verify correct execution. They present an incentive structure for operating the beacon and fulfilling verification, which relies on the beacon being operated as a "greater good". The contesting and verification is implemented in an Ethereum smart contract, which attaches a cost to contesting correct operation. The usage of a smart contract limits the availability for users not invested in the Ethereum blockchain, and restrict the possible delay functions to sequential hash chains. They consider the *sloth* delay function to be a state-of-the-art delay function, but use sequential hash chains as they are cheaper to verify in a smart contract.

Compared to this approach, our beacon exists without the need for smart contracts and buying into various cryptocurrencies. This simplifies the beacon, but also removes a convenient way of disbanding a dishonest beacon operator. However, we believe that the added complexity of relying on e.g. Ethereum will repel many potential users, who do not want to get involved monetarily to use a randomness beacon.

2 Threat Analysis

We now turn to considering possible threats to a generic randomness beacon in order to understand the surrounding environment. These threats assume the *user input* model of input as well as a beacon based on the *transparent authority* model.

Randomness is the fundamental resource that adversaries would attempt to threaten and control. It is considered and used as a fair determinant, and adversaries can seize control of it to control the outcomes it is used to determine. Once in control of the randomness, an adversary can bias it towards their own benefit, ensuring that otherwise fair outcomes will consistently favor them. Alternatively, an adversary colluding against a user will only have to make sure the randomness is either biased against or not available to that user.

2.1 Threat Discovery

To facilitate the process of discovering threats to a general beacon, we classify threats as they are found in a two-dimensional matrix. This helps us discover and explore new threats from multiple sides.

The important part of a beacon that must be protected is the output. Adversaries can harm the output in two ways: Either a threat targets the *availability*, i.e. making the output unavailable, or it targets the *integrity*, i.e. reducing the quality of the beacon output potentially to a state where the output should not be used at all. What the threat targets is as such one of the dimensions.

The other dimension depicts who is able to exploit a given threat. Here, we distinguish between *insiders* and *outsiders*. An insider is anyone with the capabilities of the beacon operator, which ideally is just the beacon operator, but for all intents and purposes may as well be anyone gaining inside access to the beacon, e.g. by hijacking it. Because the beacon operator should not be trusted either, we see no reason to distinguish between a legitimate beacon operator, a malicious beacon operator, or an adversary maliciously acquiring access to the inside of the beacon. An outsider is anyone who can influence the beacon operation from the outside network, and thus does not possess inside access.

Figure 2 visualizes these dimensions. After listing the found attacks in the following sections, we relate them to this matrix.

	Insider	Outsider
Threats to availability		
Threats to integrity		

Figure 2: A matrix visualizing the two dimensions we classify by. Threats target either availability or integrity of the beacon, and can be performed by adversaries either inside or outside the beacon.

2.2 Scoring Framework

We consider a wide variety of threats, and analyze their severity according the DREAD framework with some modifications. DREAD consists of five metrics [16], that we score on a scale of 1 to 3 (low, medium, high): DAMAGE: How bad would such an attack be? REPRODUCIBILITY: How easy is it to reproduce such an attack? EXPLOITABILITY: How little work is required to launch the attack (3 being the least)? AFFECTED USERS: How many users will be impacted? DISCOVERABILITY: How easy is it to discover the threat? Discoverability can, however, be considered to reward security through obscurity. As such we will not consider it, as an adversary with unlimited resources is expected to have knowledge of all possible exploits.

In the following section, all threats are accompanied by a DREAD score which is portrayed as follows:

D R E A Σ
2 3 2 3 10

The first four numbers each corresponds to the first four DREAD metrics. The fifth, discoverability, is omitted. The last number, Σ, is the sum of the prior numbers, and used as a severity indicator.

We go more in-depth with our use of the DREAD framework and explain our scoring of some specific threats in Appendix A.

2.3 Threats

This section lists the threats we are able to find alongside our best educated guess on a DREAD score. The list is split into two; first attacks threatening the availability of the beacon, and then attacks that threaten the integrity of the output. A summary of the threats can be seen in Figure 3 on the following page.

2.3.1 Threats to Availability

If the beacon is not available, users will potentially not have a stake in the output in the first place. The special case is an output withholding attack and eclipsing the beacon before output, both making users believe there is going to be an output, but it never comes. In these cases, users will have placed their stake in the beacon output, but the protocol fails. Another thing to consider is that users' faith in the beacon will erode each time it fails to output. This in turns makes it less attractive for operators to attack the availability, as they will slowly drive their user base away.

Input Flooding (D R E A Σ: 2 3 2 3 10) Outsiders can overwhelm the beacon with inputs to prevent other users from contributing their own input, thus denying service. Another approach could be to perform a denial of service (DoS) attack on the central server of the beacon to prevent operation. This is quite a big threat as users can temporarily be denied service, and the attack is quite easy to execute — any determined adversary could rent a botnet to flood input collectors.

Shutdown (D R E A Σ: 2 2 2 3 9) A malicious beacon operator can shut the beacon down, completely denying availability. This threat is impossible to prevent for a beacon run by a single operator, as the operator can always shut any part of the beacon down, but will only be able to get away with it a finite number of times.

Withholding Output (D R E A Σ: 2 2 2 3 9) The operator can withhold outputs that are not favorable to his interests. This threat is also quite severe as it not only denies an output, but also ruins the beacon reputation. More gravely, it can be blamed on technical mishaps such as crashes to conceal malicious behavior, while remaining easy for the operator to execute.

Eclipsing the Beacon (D R E A Σ: 2 1 1 3 7) An outsider can deny all users from providing input or receiving the output by infiltrating the inbound and outbound connection to the beacon. We believe it is a difficult attack to execute, but if successful the outsider can potentially eclipse the beacon from all users.

Eclipsing (Select) Users (D R E A Σ: 2 1 1 1 5) An outsider can deny select users from accessing the beacon to provide input or receive output. This is a quite small threat, as it is extremely hard to completely prevent a determined party from accessing the beacon, and such an eclipse would still only affect that party.

2.3.2 Threats to Integrity

These threats are far more damaging if not detected. Where availability is binary and users obviously cannot use a missing output, successful integrity attacks provide an output, that appears legitimate, but is biased. We consider using a biased output the worst thing for any user, which makes these threats critical.

Input Biasing D R E A Σ 3 3 2 3 11 An outsider can provide input that knowingly biases the output to their benefit or others' disadvantage. In this attack the outsider constructs an input such that it affects the output in a known way despite other users contributing input later. If the outsider has the capability of providing the last input, it is a last-draw attack.

This is a severe threat to the beacon, as the adversary is able to freely manipulate the output with their input, and violates the unpredictability of the random number as the adversary now knows more than everybody else. The attack can be executed by anyone with access to the input collectors given that they have the ability to pre-compute outputs.

Output Degradation D R E A Σ 2 3 3 2 10 Adversaries can supply "bad" input to reduce the quality of the output. This is also a serious threat as it will affect the quality of randomness provided to all users, a randomness which may not even be usable. In addition, it is easy to do given access to the input collectors, and could even happen by accident.

Input Manipulation D R E A Σ 3 3 2 3 11 The operator can manipulate the input to bias the output of the beacon. He can also selectively exclude inputs from certain users to deny them availability. This threat is quite severe as the operator has direct access to manipulate the inputs, and may even be able to do so in a way that cannot be detected. It is also easy for any operator capable of pre-computing the output, and affects the randomness given to all users.

Man in the Middle D R E A Σ 3 1 1 3 8 Adversaries can intercept and change data sent between user and beacon. This threat could be significantly damaging but also extremely hard to execute for adversaries.

Due to the nature of beacons we recommend using them when you need to agree on some random number — thus, to intercept and manipulate inputs and outputs, the adversary would have to distribute the manipulated number to all users, as they would otherwise disagree on the numbers, leading to the manipulation being discovered.

Emitting False Output D R E A Σ 2 1 2 3 8 A malicious operator can output false results of the computation that benefit him. While this is technically a threat to the integrity of the beacon, the effects should be similar to those of a withholding attack. This is due to the fact that simply publishing false output would rapidly be discovered in a transparent authority beacon, making the output unusable, but also removing any faith in the operator.

Leaking Output D R E A Σ 3 3 2 3 11 The operator can give access to the output earlier to some parties than others — potentially selling early access. This threat can be quite severe, as we do not know how early access can be granted compared to when the randomness is used. It also violates the unpredictability property of the beacon, and is easily executable for any malicious operator of the beacon. In the worst case it would affect all users.

Cryptography Exploit D R E A Σ 3 3 1 3 10 Weaknesses or exploits may exist in the cryptographic techniques that protect the beacon. While we estimate it will be hard to find such exploits, they would likely be quite easy to apply once found, and would affect all users. In this case one might also consider the effect quantum computers would have on the use of cryptography, which could also threaten the beacon.

2.4 Summary

Figure 3 shows the above threats in the matrix. To reiterate, an insider can always perform outsider attacks, and outsiders can perform insider attacks if they obtain sufficient privileges, e.g. through hijacking the beacon.

	Insider	Outsider
Threats to availability	Shutdown Withholding output	Input flooding Eclipse beacon Eclipse (select) users
Threats to integrity	Input manipulation Leak output Emit false output	Input biasing Output degradation Man in the middle Cryptography exploit

Figure 3: The identified attacks in the previously defined matrix

3 Requirements

This section lists the requirements for a randomness beacon suitable for our security goals and the threats that exist towards beacons. We decided on using the *transparent authority* type of beacon, which requires a high level of transparency, and as such we build requirements on top of that. The requirements presented here will serve as a foundation of our design.

3.1 Transparent Operation

Users should be able to oversee that the beacon operates according to protocol and thus catch any deviations from it. This in turn requires all aspects of the protocol to somehow publicly announce or display their work for users to verify. Fundamentally, users should be able to see which inputs have been used to produce randomness and verify that they do produce the published output. This benefits the integrity of the beacon as some previously mentioned attacks would be detected in this setting. It is also a necessity for our chosen type of input model, *user input*, which requires users' ability to verify that their own input is used to produce outputs.

Firstly, users should be able to see which inputs are used to produce an output. Being able to verify whether their own input has been used allows users to determine whether they should trust the output. If their input has not been used, they should not trust it. Secondly, they should be able to repeat the process on their own computers as a means of verification. This also requires the process to be deterministic. However, the output should still be unpredictable, even to the beacon operator.

3.2 Open and Secure Protocol

Anyone should be able to easily contribute to the beacon protocol to influence the random generation. There should be no requirements imposed on users to limit their contribution rate besides DoS protection. The protocol should be secure meaning that even if only a single user is honest, the output is still unpredictable.

3.3 Timely Publishing

The protocol should enforce that input, output, and any data needed for verification of an output is published as soon as possible to make the beacon more transparent. By having a requirement of timeliness at the protocol level, we restrict the time a malicious operator has available to diverge from protocol before users will suspect them.

Giving users all the tools to replicate and oversee the process makes it difficult for adversaries to covertly manipulate the beacon to their benefit, and allows users to complete output computation themselves if the beacon stalls. This in turn mitigates one of the greatest threats from the operator, input manipulation. A beacon that does not reveal which inputs were used before publishing the output will essentially be admitting that they picked the inputs to bias the output.

We should also note that despite having this property the beacon does not guarantee outputs on any specific wall-clock time, e.g 12:00:00, 12:01:00 & 12:02:00. Instead, it will output as soon as possible after each period of input collection. Barring any attacks, this will provide a regular stream of outputs.

3.4 Practicalities

A part of the goal is to create a beacon that is practical and implementable in the real world. As such, we value requirements that other purely theoretical approaches may not consider. Scalability of all components is important as we envision a general beacon suitable for many use cases. Therefore, it should scale to at least several thousand users contributing with user input in every output. Here, other approaches usually only focuses on scalability of the core theory, but we will consider all parts.

If users lose confidence in an operator, a new operator and thus new beacon can be used instead. As the beacon is expected to be run as a greater good, nobody should need to make any large investment to deploy a beacon instance and become operator of it. As such, we value easy deployment and installation of the beacon, in order for it to not be a hindrance for deciding to run a beacon.

It will be beneficial to allow different channels for input and output, both to make the beacon easier to access for users, but also to make it resilient to having any single channel attacked. Should a single channel be attacked, input could still be submitted to another. This requirement further increases usability since different users may prefer different input and output channels. We also consider fault tolerance a valuable property to have, and having multiple channels still allows users to input if one fails.

4 Design

This section describes our beacon design, the majority of which is concerned with mitigating threats as security is an important aspect of any randomness beacon. The design choices are based on fulfilling the requirements and incorporating mechanisms to prevent as many of the threats identified in the threat analysis.

4.1 Architecture

To meet the requirements of modular input and output and fault tolerance, we use a service oriented architecture (SOA) in the beacon design. This architecture splits systems into application components, also called services. These services serve a single purpose, i.e. they each logically represent part of the activity needed for the entire system and have a specified outcome. Communication between services is done according to a well-defined protocol.

This architecture provides loose coupling in the system and also allows for easier fault tolerance since services, being black boxes, are easily replaceable on failure. Furthermore, each service can be scaled as needed.

The beacon could also have been designed as a monolithic program running on a single machine. This would likely be more efficient initially, but would fail to scale to meet large demands, and would likewise represent a single point of failure for the entire randomness beacon.

4.2 Services

An instance of a randomness beacon designed in the SOA pattern will consist of a number of services. To fulfill our requirement of modular input and output methods, we will allow multiple different input collectors and output publishers to exist in the system. Lenstra and Wesolowski [14] mention having several input sources, but to our knowledge we are the first to generalize it in the design to allow virtually any input source. We believe this is superior as it increases redundancy, increases usability, and spreads the load to several smaller services.

A number of INPUT COLLECTOR services collect input from a myriad of different sources. These sources could for example be email, irc, a bot for your favorite instant messaging service, tweets with a specific hashtag, raw TCP or HTTP connections, a pretty website, SMS, Morse telegraph, or carrier pigeon[4]. An INPUT PROCESSOR service acts as an aggregator of the input from all collectors and hands it over to the COMPUTATION service, which commits to the aggregated input and runs the computation to generate an output. Finally, various PUBLISHER services publish the commitment, output, and any relevant proofs to different outlets. These outlets could be the same media as input collectors use but can be different.

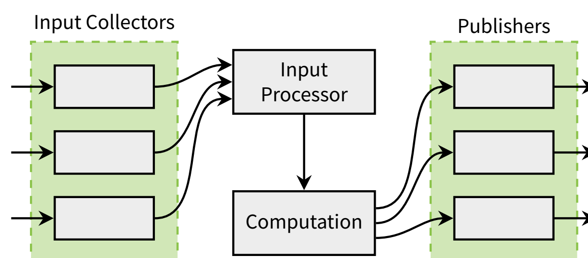


Figure 4: An abstract beacon architecture based on services. Solid boxes illustrate services and arrows represent data flow.

A simple randomness beacon illustrating these services and their relationships can be seen in Figure 4. Being a loosely coupled system, the arrows in the figure are not a given. Services need a way to know about other services. As such, some kind of service discovery is needed in the implementation. Service discovery can be a single point of failure but is mitigated by using redundancy [15]. Alternatively, a peer-to-peer method of service of discovery could be used to have services connect directly to each other.

4.3 Pipeline

In our beacon, as illustrated in Figure 4, data only flows one way through the system and each component performs an independent transformation. This effectively means that a beacon is a *pipeline* where data flows into the system as inputs, is processed, transformed to a random output, and lastly published.

The pipeline architecture can be seen as a specialization of the SOA pattern, since each step in the pipeline is a service as defined in a SOA. When considering the system as a pipeline some restrictions are imposed compared to the more generic SOA, because data only flows in one direction. This means that if data is lost due to failure, e.g. a component crashing, we cannot inform previous pipeline components to

[4]For carrier pigeon we recommend the *IP over Avian Carriers* proposal: `https://www.ietf.org/rfc/rfc1149.txt`

resend the data. However, since our beacon design is meant to operate in a forward-only manner, this loss of data should be tolerated, and even expected in some cases. This underlines the fact that users should always verify both inclusion of their input and correct computation of output — users should not assume that a submitted input is always included in the next output.

4.4 Security Design

From a security perspective, new attack surfaces may be introduced by splitting up the system from a monolithic self-contained architecture to a service-oriented kind. When designing a composable system such as our randomness beacon it is important to take the inter-component communication into account. For example, the architecture can potentially make it possible for adversaries to block out parts of the system, by means of DoS attacks. The protocol used to communicate from service to service must be secure in a way that prevents adversaries from being able to covertly manipulate the messages.

In the case of a randomness beacon, the security also embodies the operator's ability to predict or manipulate the output. This means we need a mechanism to prevent the operator from disguising last-draw attacks as regular user inputs, and from excluding certain inputs to alter the output. We also want to prevent the operator from initiating multiple beacon computations, and then only publish the output which benefits the operator the most.

4.4.1 CCO Workflow

Our solution for this problem is to enforce what we will call a Commit-Compute-Output (CCO) workflow in the beacon protocol. We have designed this workflow to govern the security of the beacon, and is one of our contributions. It means that each published output is paired with a commitment which can be used in the verification of the beacon. The operator must publish the commitment a significant amount of time before the output is published — otherwise, the beacon operator could just publish a commitment to any desired output. Furthermore, the operator is limited to a single commitment — otherwise the operator could publish several commitments and only publish the most desirable output.

Contrary to other approaches of transparent authorities, we have decided on an even more transparent approach: The commitment contains all data required for the computation and all inputs. To our knowledge, we are the first to take transparency to this extreme.

The transparency allows any party to compute the randomness alongside the beacon operator. It ensures that the operator can not cause much damage by withholding output or by deciding not to open a traditional (e.g. hashed) commitment. In essence, it reduces the "market value" of the output, making it less attractive to leak output (i.e. sell early access to the output) because everyone can just compute it. While it does not prevent the operator of performing a withholding attack, it minimizes the effects of it, as others can compute the output from the commitment and still obtain an equally valid output.

4.4.2 Delay Functions

To decrease the possibilities of the operator trying different commits before releasing them, we use a *delay function*. Delay functions can be seen as black box hash functions that require a given amount of time to run and are inherently sequential, meaning they cannot benefit from parallel execution. It ensures that the output cannot be instantly computed, and ensures that the operator cannot try more than one commit before running out of time. As such, the operator is unable to perform the input manipulation attack in a meaningful way. The operator is of course able to exclude or change output, but not in a way that knowingly benefits anyone because the effect of the manipulation is hidden behind the delay function.

When deploying delay functions in randomness beacons, it is important to keep verification in mind. A user should be able to run the delay function in reverse to confirm that an output matches the commitment. To avoid having to require each user to execute the full delay function, we use a flavor of delay functions which is *asymmetrically hard*, i.e. hard to compute but easy to verify. In the normal case only the operator runs the full delay function, resulting in much CPU time saved globally and thus electricity, too. In the case that the operator is (maliciously or not) performing an output withholding attack, users still, because of the CCO workflow, have all the information needed to run the delay function themselves. It might even be imagined that a few volunteers will run their own "mirror beacon", each mirroring the computation of the main beacon operator. This adds redundancy in the computation.

The delay function also protects against last-draw attacks by adversaries. A last-draw attack would attempt to bias the output by crafting an input to produce favorable randomness. The adversary needs to compute the result of adding a specific input as the last input. Delay functions make this significantly more difficult to attempt due to the time needed to compute the result. Given a delay function that takes five minutes to complete, an adversary must dedicate five minutes of processor time to any given input he attempts to use. This means he must dedicate large amounts of resources to perform any significant amount of attempts, and more importantly if a single input is added to the beacon within that five minute period, all of his work will be null, and he will be forced to restart.

4.5 Exploring Trust Assumptions

Given the precautions taken in the design, the amount of trust required to use the beacon should be minimal or non-existent. We propose two scenarios, which explore the users' trust assumptions towards the beacon. In both of these scenarios, we focus on a user named Alice, while all other users are regarded as potentially colluding adversaries with malicious intent. The only assumption about adversarial users, is that they can interact with the beacon in the same manner as Alice; i.e. send inputs to the beacon. In the second scenario the beacon operator is also an adversary and thus has total control of the beacon.

Because network latency is dependent on many variables in a system and virtually impossible to verify, Alice should not trust any claim from the beacon operator regarding latency. To protect herself, Alice should assume that any inputs she sends is received immediately by the beacon, and vice versa — any message from the beacon is received instantly by her. This can be considered a "worst-case" time. This means that the beacon will not be able to claim different timings than what Alice observes — and as this observation is all she can trust, she should base her decisions on that.

Scenario #1: An Honest Operator This is the best case scenario for Alice. The operation of the beacon is honest, which means that *a*) the beacon operator will accept all inputs, i.e. not exclude any; *b*) the commitment is published as soon as possible, i.e. right after a batch of inputs has been processed; and *c*) the output and proof is published immediately after the computation, i.e. the delay function, is done. This operation is also depicted in Figure 5.

Any adversarial users can try to manipulate the output by providing their own inputs, but Alice can disregard this, as long as she can verify the presence of her own input. With this honest beacon, Alice knows exactly how long the computation took, and can trust the output to not be manipulated in any predictable way. However, assuming that the beacon is honest is not advised, since it leaves Alice vulnerable by exploiting this trust. If Alice trusts the beacon operator to be honest, she will not suspect them to act according to scenario #2.

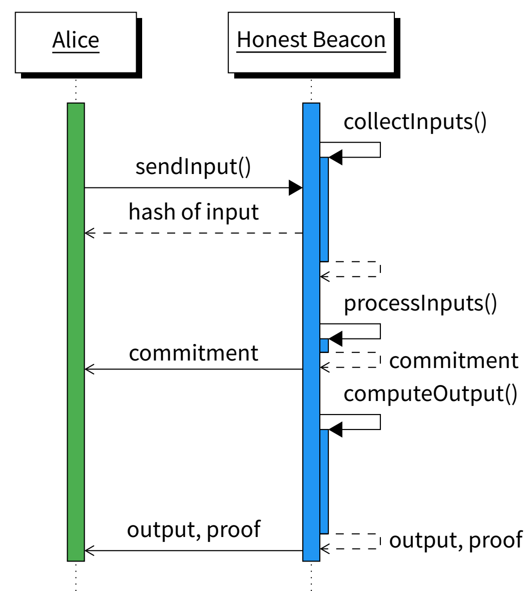


Figure 5: Sequence diagram depicting one beacon iteration from the perspective of Alice. The beacon operator is honest.

Scenario #2: A Malicious Operator This is the worst case scenario for Alice; a beacon operator which is trying to choose the output, yet still make it appear valid to Alice. The malicious operator will not exclude Alice's input, since they are interested in fooling her to trust a forged output. Besides, the malicious operator should be expected to collude with all other users against Alice — she is effectively alone in Wonderland.

When the operator acts maliciously, they try to manipulate the output, while displaying correct operation outwards. This means that Alice still receives a commitment, output, and proof, which she can use to verify the correctness of the output. She will receive these messages at seemingly the same timing as described in the honest operator scenario. The

main difference here is that Alice should not assume correlation between the timing of received messages and timing of beacon process.

Assume the following behavior of a malicious operator: *a*) the beacon operator will stop input collection after receiving Alice's input; *b*) they will attempt to publish the commitment, output, and proof when it is expected by Alice; *c*) the operator will use unlimited resources to pre-compute possible outputs to seemingly valid commitments; *d*) the operator will use pseudo inputs to affect outcomes, which will give the impression of input collection after Alice's input; *e*) out of the pre-computed outputs, the malicious operator will choose the one which benefits them the most. This behavior is also depicted in Figure 6.

In this scenario the operator will effectively carry out a last-draw attack against Alice. However, if the malicious operator cannot compute an outcome they deem beneficial, they can claim disrupted operation before publishing any commitment. This will leave Alice without any output, a withholding attack, and she will not know if the operator was malicious or disrupted by a third party adversary.

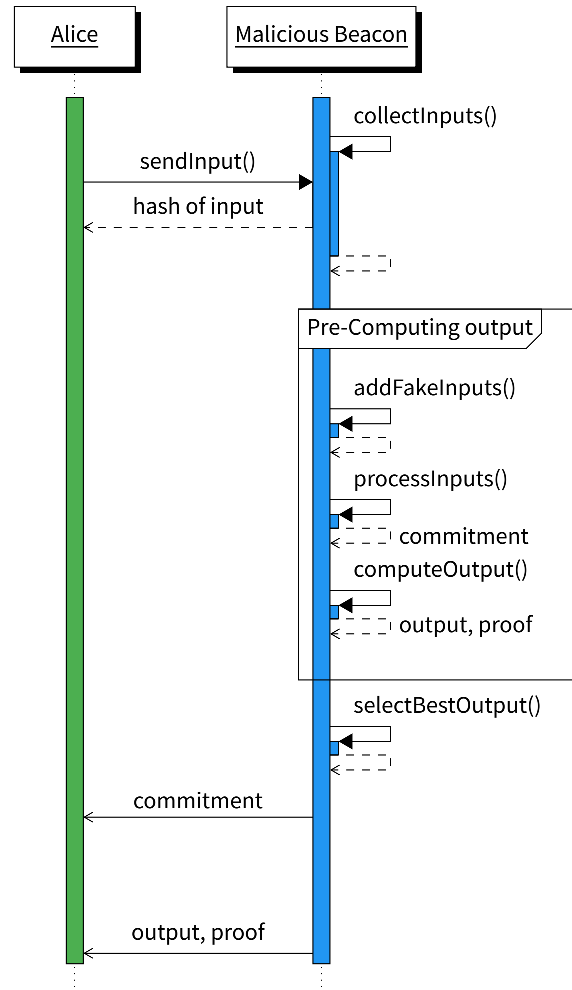


Figure 6: Sequence diagram depicting one beacon iteration form the perspective of Alice. The beacon operator is malicious.

4.6 Rational Trust Assumptions

In our approach to a randomness beacon we want to push beyond the need for honest operators and naïve users. To achieve this we extend the work of Lenstra and Wesolowski [14] to quantify trusting the beacon and determine thresholds for reasonable behavior when using delay functions. This provides a measure of rational trust, where users decide for themselves if what they observe is adequate.

We present a property which, if satisfied, means a user can trust that the beacon operator is not capable of fooling them. This property is true if the user determines that nobody is able to compute the delay function in the time between the users input and the user receiving the beacon's commitment. This can be condensed to:

$$t_{\text{COMMITMENT}} - t_{\text{INPUT}} \quad < \quad T_{\text{DELAY FUNCTION}}$$

given that t_{INPUT} is the time when the user sent the input, $t_{\text{COMMITMENT}}$ is when the user received the commitment, and $T_{\text{DELAY FUNCTION}}$ is the fastest computation of the delay function. So for users to be more likely to trust a beacon, the time between sending the input and receiving the commitment must be significantly smaller than the time between the commitment and the output. In fact, it must be smaller than the shortest time the user thinks the operator could compute the delay function.

An example could be that a user believes that the world's fastest computer can compute the delay function in 2 minutes. In this case the user can trust the output if he sees a commit to a set of inputs containing his input within 2 minutes of his input, because then he knows that nobody could have had time to run the computation on his input before choosing to release a commitment or not. This relation between the time taken to compute the delay function and the time before a commit is seen allows users to flexibly adjust their willingness to trust the outcome has not been biased against them.

This threshold is also described by Lenstra and Wesolowski [14], where they advise a ratio of no more than $\frac{1}{5}$ of the computation time spent collecting inputs. In their paper, Lenstra and Wesolowski

furthermore state that smart participants will always try to minimize the time between their input and the commitment. We see this as potentially problematic, since such behavior can create congestion in the system, which might result in some inputs not being used in the intended output computation. This means that users whose inputs were not included cannot trust the output of the given beacon iteration.

Taking all this into consideration we present a beacon operation protocol which can be adjusted to increase or decrease the ratio and thereby the limit for probabilistic trust. The operation must be sequential which means that we must collect input before computing the delay function. However, because we want to spend more time computing than we are collecting input, a strictly sequential beacon will contain dead spots where no user is submitting input. This may be acceptable in some scenarios, but we want to design a beacon which always accepts inputs and will not be suspected of malicious operation. To achieve this we parallelize the beacon protocol, meaning that several delay functions run in parallel but offset in time and on different input. In Figure 7 this is illustrated, where these offset but parallel beacons are seen.

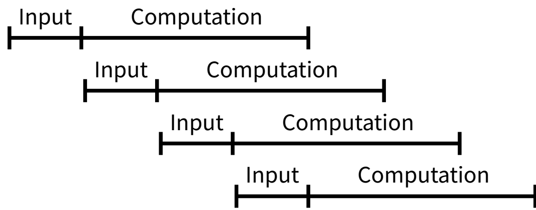


Figure 7: Parallelized beacon protocol, with offset input collection and overlapping computation. After every computation the output is published.

We observe that no input collection is run in parallel nor overlapping, which resembles a constant stream of input collection. In addition, the computation components can eventually be reused for future beacon computations, thereby eliminating the need for spinning up new computation services. These observations are depicted in Figure 8, where the beacon would output at each circle shown in the diagram.

4.6.1 Number of Computation Nodes

The number of computation nodes required in this fashion is the duration of the delay function divided by the duration of input collection:

$$\text{Number of Nodes} = \left\lceil \frac{T_{\text{DELAY FUNCTION}}}{T_{\text{INPUT COLLECTION}}} \right\rceil$$

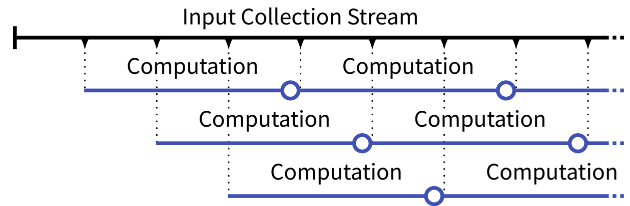


Figure 8: Parallelized beacon protocol, with input collection stream and overlapping computation. A circle denotes outputting at the end of the computation.

As an example, an input collection time of 2 minutes and a delay function of 10 minutes will require 5 computation nodes to always begin a computation every 2 minutes.

However, the delay function is not guaranteed to precisely take e.g. 10 minutes — the computation nodes are expected to be running other processes requiring CPU time such as an operating system. Therefore, the delay function can be expected to from time to time finish a bit later than naïvely anticipated. This will over time cause the beacon output to be increasingly more skewed compared to the initial output frequency, since they can only be delayed and not catch up by being faster sometimes.

To remedy this, a number of additional nodes should be kept at hand. Therefore, we update the prior equation to take this extra time for each delay function into account. Let δ be this extra time additional to the delay function.

$$\text{Number of Nodes} = \left\lceil \frac{T_{\text{DELAY FUNCTION}} + \delta}{T_{\text{INPUT COLLECTION}}} \right\rceil$$

If, for example, the delay function is expected to always finish at most 2 minutes later than the expected time of 10 minutes (i.e. a worst-case time of 12 minutes) and the input collection is 2 minutes, 6 nodes in total are necessary to guarantee a node is always ready every 2 minutes, given a maximum of 2 minutes expected delay.

The beacon operator should find a sensible number of nodes that maximizes the chances of a ready computation node given expected delays, while minimizing the idle time of the nodes.

4.6.2 Expected User Behavior

Based on what Lenstra and Wesolowski [14] write about smart users always trying to input as close to the commitment as possible, we admit that our solution of parallel offset computations will not prevent

such behavior. However, the goal of our approach is not to eliminate this user strategy, but to minimize the need for it.

In our beacon we do not expect to give guarantees about deadlines, such as commitment and output publishing, since such guarantees only would serve as false reassurance to the user. Instead, our beacon is adjustable, such that a ratio of input collection and computation time, which most users finds reasonable, can be deployed.

Users are still welcome to contribute input as close to a deadline as possible in order to gain a better probabilistic guarantee. This may, however, be tricky as the beacon will not announce deadlines as to not encourage users to input at the same time. Instead, we propose another strategy for the smart user: Instead of supplying one input, multiple unique inputs should be supplied spaced apart in time. For example, a user supplying an input every 10 seconds until receiving a commitment will have a better chance of coming close to the deadline.

4.7 Scalability

We consider the scalability of the beacon to be a significant factor in both performance, operation, and deployment. Potential bottlenecks in the system should be easy to mitigate, and beacon operators should not be burdened by the architecture and design choices deploying and expanding their randomness beacon.

Our choice of pipeline and SOA fits well with our intentions for scaling the beacon. Individual services in the SOA can be scaled as needed, as they are designed to be stateless and loosely coupled with each other. As long as the overall contract of the pipeline is respected, i.e. the order of component types, individual steps in our design can be scaled as needed.

In some scenarios a beacon may consist of multiple computation services as a mean of redundancy, as long as each computation is run on the same input. The same can be said about input processors, where a beacon may need redundancy to likewise avoid a single point of failure. This presents the issue of consensus about which input to use, but that is out of scope for this report.

4.8 Review of Requirements

We established a set of requirements for our beacon, and we now briefly describe how each requirement is fulfilled by our design.

We designed the beacon in a SOA. The architecture facilitates the scalability of our beacon, and allows for multiple interchangeable components making it easy to have multiple channels of both input and output.

The beacon operation is structured around a CCO workflow, which makes it transparent. There are no requirements on users, which makes it easy for anyone to contribute. As such, it is an open protocol. The commits contains enough information for users to compute the output alongside the beacon.

Using delay functions makes the beacon more secure, as the output is harder to pre-image due to the time it takes to compute. The *sloth* function, being asymmetrically hard, also enables faster verification of the result. It also facilitates a secure protocol, in the sense that any single honest user will render the output unpredictable. The delay function provides a measure of timeliness to the output, as it will always take some regular wall-clock time to compute for the beacon. Timely commits also form the basis of our rational trust assumptions, that helps users decide whether to trust a specific beacon output.

The design contains succinct and well-separated components. And because of this we believe fault tolerance will be easy to implement, and it will facilitate ease of deployment and installation. We leave these areas to be fulfilled in the implementation, as they are tightly coupled to which tools we use.

4.9 Review of Threats

In the process of designing our beacon, we have considered the threats to a beacon and designed measures to mitigate them. We present the threats we consider to be successfully mitigated with our design, and list them in Figure 9 on the next page.

Output Degradation D R E A Σ 2 3 3 2 10 The beacon will additively aggregate inputs. Because of the amount of inputs, the input space will likely be larger than the output space. Utilizing a hashing algorithm with diffusion and confusion properties, any input, no matter the quality, will unpredictably affect the output. It is not possible to statistically reason about the output of the beacon related to the input, besides being well-distributed.

Input Manipulation D R E A Σ 3 3 2 3 11 We have designed the beacon around delay functions specifically to prevent this attack. As previously mentioned, it requires

an adversary to spend significant resources to compute a single pre-image before releasing a commitment, which is essential to this type of attack. This is not possible under our CCO workflow and reasonable trust assumptions by the users. This also makes any attempt at this attack from the operator equivalent to a withholding attack, as users should not use an output they did not see a timely commit for.

Input Biasing D R E A Σ 3 3 2 3 11 This threat is mitigated by the same means as the previously addressed threat mitigation — input manipulation.

Leaking Output D R E A Σ 3 3 2 3 11 Our delay function based CCO workflow also mitigates the operator leaking outputs that give any significant advantage. An output will never be used unless a commit for it is seen, and the commit contains all the data required to compute the output alongside the operator — thus the operator can only leak outputs that are already predetermined, removing their "market value".

Withholding Output D R E A Σ 2 2 2 3 9 The CCO workflow accompanied by a delay function minimized the consequences of this type of attack. Using delay functions and requiring the beacon to publish a commit to a set of inputs before computing, prevents malicious operators from pre-computing outputs, and withholding if they are not beneficial. The operator could still withhold the commit to prevent availability, but they can not know whether the output favors them or not.

Man in the Middle D R E A Σ 3 1 1 3 8 This threat is rendered ineffective because of the CCO workflow. Assuming the adversary is not able to compute the delay function significantly faster than anyone else, a man in the middle attack will only affect the availability of the beacon for the targeted user.

The user must receive a timely commitment and verify inclusion of the input in the output, and if an adversary is capable of this the adversary has performed the same work as an honest beacon operator would have performed. As such, the user can actually use the output. However, if an adversary has the resources to compute the delay function in time to release a timely commitment, they can potentially present a seemingly valid commit and output to the user. In this case the attack resembles that of the input manipulation threat, since the user will be unable to distinguish the adversary from the beacon operator.

Normal man in the middle mitigation using certificates will be redundant here. We do not care *who* the beacon operator is. As long as a user's input is included in a valid output, that output is good to use for that user — no matter who did the computation.

Emitting False Output D R E A Σ 2 1 2 3 8 Users should not trust an output that cannot be verified both for inclusion of input and correctness of computation, and as such emitting a false output will simply be an availability attack — the output cannot be used and as such could be non-existent. While the beacon operator practically could easily do this, it would not cause much harm, since our CCO workflow ensures that users can compute the delay function themselves.

	Insider	Outsider
Threats to availability	Shutdown ~~Withholding output~~	Input flooding Eclipse beacon Eclipse (select) users
Threats to integrity	~~Input manipulation~~ ~~Leak output~~ ~~Emit false output~~	~~Input biasing~~ ~~Output degradation~~ ~~Man in the middle~~ Cryptography exploit

Figure 9: The state of mitigated threats (striked through)

4.9.1 Unmitigated Threats

There are unmitigated threats, particularly the threats concerning availability by outsiders. A variety of existing solutions for mitigating input flooding attacks already exist, and these problems are not particular to randomness beacons.

Both eclipsing attacks are difficult for us to mitigate. Essentially, it requires control of the connections to all the users. To launch this kind of attack, it would require the role of system administrators at ISP level or similar, and thus it is deemed out of scope. Eclipsing the beacon could be mitigated by having several redundant forms of communication with the outside world. We also cannot prevent the operator from shutting down the beacon, but this attack will eventually drive users away from the malicious operator.

Cryptography exploits is the last threat we have not, and quite possibly cannot account for. Our beacon uses a variety of cryptographic components like hash functions and delay functions. If exploits in these were found, it would change the assumptions on the beacon. A shortcut to computing the delay function would allow adversaries to perform many of the other attacks. We have partly mitigated it by allowing the hashing algorithm to be switched out.

5 Implementation

In this section we present the general idea behind the implementation of our beacon design, technologies used in said implementation, and a discussion of implementation details which affect the beacon protocol. We seek to realize the design introduced in Section 4 while making reasonable trade-offs where necessary. This means that some parts of the implementation will be seen as future work, in the interest of time. However, the implemented beacon will be a functional proof of concept (PoC), with an underlying infrastructure suitable for real world deployment and usage, and as such a good infrastructure is something we will prioritize.

In the implementation of the beacon, we choose not to focus on usability applications, such as allowing a user to track their input automatically through the beacon. Instead, we implement a beacon with simple, secure, and succinct operation. For some concrete details about the implementation beacon, including the file structure, see Appendix B on page 31.

5.1 Overview

This chapter explains the choices leading to the specific implementation seen in Figure 10. As such, the figure can be used as a reference as we explain the various parts of the implementation.

At a glance, the figure shows the services and how they interact. All services are controlled by the beacon operator. Examples of input collectors are shown, and each of them hash inputs as they are received and send them to a known proxy (fan-in pattern). It also responds to the user with the hash of the given input. The proxy forwards any message to the input processor. The input processor adds each input to a Merkle tree structure. Here the pipeline fans out.

A computation node that is ready to take on work signals its readiness to the input processor (1). Here, the input processor internally adds the computation node to a queue. The input processor will at an interval equal to the input collection duration send the current Merkle tree (2) to the computation node in front of its queue and starts building a new Merkle tree with new input. This ensures that e.g. every minute a computation node is starting a computation based on the Merkle tree built during the past minute. Once given the Merkle tree, the computation node releases a commitment (3) to this input, and runs the delay function on it (4). Eventually the output (5) is sent to the output proxy (fan-in pattern). This proxy forwards the output to each publisher. A publisher then packages the received output and sends it to a specific outlet.

5.2 Framework and Language Choice

To achieve the service oriented architecture (SOA) and pipelining presented in Section 4, we utilize a framework for asynchronous message passing and concurrency. This will allow us to develop the components separately and to gradually implement business logic by mocking not-yet-complete services, as long as the inter-component communications protocol and

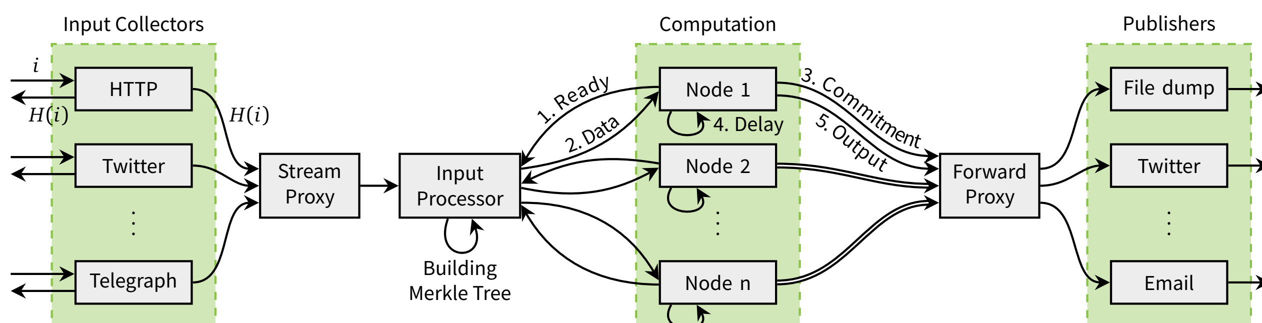


Figure 10: Significant data flows in the beacon. Arrows signify data flow. Only the top-most arrows are labelled, but labels on the top arrow apply to all arrows beneath.

message passing method is agreed upon. We choose to use *ZeroMQ*[5] as this framework for message passing and concurrency.

5.2.1 ZeroMQ

ZeroMQ is language agnostic with bindings for virtually all programming languages, and that it is fast, flexible, and scalable. The name *ZeroMQ* hints at its alternative approach to a messaging framework, in that no (or *zero*) broker is needed between components. This means that our beacon implementation does not rely on any centralized broker for passing around messages — no single point of failure in that aspect.

The pipelining fan-in and fan-out patterns are implemented using the primitives of *ZeroMQ*. These primitives provide useful patterns for communication, e.g. pipeline and publish/subscribe patterns. Using something like *ZeroMQ* and not developing our own ad-hoc solution, allows us to leverage well-tested technologies, and focus on the beacon itself instead.

Another reason for choosing *ZeroMQ* is the guarantees it provides us in relation to communication. Messages sent and received are atomic, meaning that we either receive everything or nothing at all. Loosing a message is not critical in our use case, and when messages are used to drive actions (e.g. control and status messages), we simply retransmit if an acknowledgement is not sent back. Furthermore, we can enable authorization and authentication protocols on *ZeroMQ*, which limits who can send and receive which messages, using elliptic curve cryptography and certificates. However, as we are implementing a PoC we are not using any authentication and authorization, for simplicity and fast prototyping. Previously, some security vulnerabilities have been found in *ZeroMQ* regarding privilege escalation where crafted packages could downgrade the version of the protocol being used, but these were all patched in prior releases, as evident in their public repository[6]. As of writing this, no existing security issues with *ZeroMQ* are published.

5.2.2 Python

The components of our beacon are mainly implemented using Python 3, both for fast prototype turnaround, but also because of a state of the art *ZeroMQ* library. However, in some cases the performance overhead in Python is unsuitable for the task at hand. Fortunately, Python programs can easily be extended with C code, and due to our SOA entire components can be implemented in this fashion if deemed necessary.

[5] http://zeromq.org/

[6] https://github.com/zeromq/libzmq/

5.3 Infrastructure

Establishing a solid and scalable infrastructure is a significant part of implementing our beacon. We choose not to rely on running all components of the beacon on the same machine.

Because of this, we use TCP sockets through *ZeroMQ*. This means that operations such as reading bytes from the sockets and reconnecting in case of networking issues is handled by *ZeroMQ*, which also maintains a local queue per socket to mitigate network congestion. Sockets in *ZeroMQ* require one end to bind and the other to connect, and usually it is recommended that the most stable or consistent component binds, while dynamic or unstable components connect. Only one component can bind to a given socket while many can connect to that same socket.

Between computation nodes and publisher, the "publish/subscribe" pattern provided by *ZeroMQ* handles the message routing based on subscription prefixes, which means less traffic on our network. Furthermore, the fan-in pipelining is implemented with a "push/pull" socket pair which ensure fair operation, thereby avoiding starvation of components. Lastly, *ZeroMQ* guarantees atomic delivery of messages, which means that we can assume all parts of a message or none at all. This is the most desirable scenario for us since lost messages should be relatively insignificant, while malformed messages usually means strenuous error and edge case handling.

5.3.1 Proxies

In the interest of rapid iteration and ease of configuration, we deploy proxies at key points in the pipeline. This allows us to add and remove components easily from the network, since components then do not need to know of each other — they only need to know of the proxies. An alternative would be to use some form of service discovery to allow components to discover and connect to each other directly.

We insert two proxies in the network: one between input collectors and the input processor and one between computation and publishers, as depicted in

Figure 10 on page 17. Another benefit of using proxies is the ability to have a many-to-many connection, since the proxy binds both its frontend and backend socket, which components then connect to.

These proxies are written in C for performance and since they utilize a *ZeroMQ* primitive for the actual forwarding, their operation is quite stable. It should be noted that these proxies inherently introduce single points of failure in our beacon, which could be detrimental for continuous and stable operations. In Section 5.3.2, we describe a way to mitigate this.

While the two proxies serve the same purpose, i.e. pass along traffic between components, their workloads are vastly different. The proxy between computation and publishers is a *forward* proxy, which transparently facilitates publish/subscribe pattern. In our beacon, this proxy will never see a high frequency of messages, since the number of outputs, commits, and proofs are limited by the beacon output frequency. The proxy will, however, be subjected to significantly larger messages, due to especially commits which must contain every input used in the output computation. Even with the large message, the infrequency of them is far from saturating the capacity in this proxy.

Between the input collectors and input processor, we have a *stream* proxy, which ensures that the fan-in and fan-out patterns are executed fairly, i.e. no connected components are starved. This is facilitated through a round-robin message distribution. In contrast to the previous, this proxy is required to be able to handle a substantial amount of messages, since every input submitted to the beacon will pass through it. While the messages are remarkably smaller (64 bytes of application data), the amount of messages can become a problem, when the fair distribution and no starvation policy must be enforced.

In the later Section 6.1 on page 22, we evaluate the performance of our proxies.

5.3.2 High Availability

One way to mitigate single points of failures could be to implement a pattern called "binary star" by *ZeroMQ*. Here a component is configured with two instances, a primary and a backup. The backup can then take over and signal for a new backup to be started if the primary disappears from the network. This pattern can potentially be applied to all components exposed as a single point of failure and would be sufficient in most cases of crashes. Some scenarios where this pattern can improve availability are hardware failure, instability or disappearance of the network link, or the component code crashing.

The success of such a "binary star" pattern also depends on the assumption that both the primary and backup will not fail at the same time, which might prove difficult to guarantee if our system is under attack. However, as we deploy a pipeline architecture, the randomness beacon will generally only move forward; missing inputs or computations is considered an affordable loss.

Avoiding availability issues when encountering byzantine components is often more troublesome than mitigating crashes, since adversarial components will try to display correct operation. We deem this as outside the scope of our randomness beacon, where we instead opt to make suspicious activity such as manipulated packages detectable.

Another measure to provide high availability in the beacon is our delegation of user interaction to input collectors and publishers. In the system, user interaction is the most demanding task regarding availability, and the statelessness of the components means that adding new instances is as easy as executing a shell command as the operator.

Summarizing, the reliability of the beacon could be greatly improved by eliminating single points of failure, such as the proxies and input processor. However, changes as these are fairly trivial to implement, and we deem them unnecessary for a PoC random beacon.

5.4 System Interface

As previously mentioned, the system boundaries, i.e. where users and the outside world interacts with the beacon, are handled by input collectors and publishers. We implement these and the surrounding infrastructure, as well as vertical scaling if the load becomes too high on a single component.

To limit the space of potential messages and message sizes passed around inside of our system, we sanitize the user inputs by hashing them at the entry point. Realistically, allowing *any* input could be seen as an invitation by some users to post messages or even files, e.g. illegal or inappropriate content. Our choice of hashing at entry point will mitigate this.

Given a substantial amount of users, receiving and hashing inputs may become a costly affair performance-wise. Fortunately, the state of an input collector is only relevant to a single input request, meaning that scaling and even distributing across

many machines is a trivial task. When we hash an input, as a convenience we return the hashed input as a response. As such, they will later be able to confirm that their hashed input was used in the output of the beacon. To allow users to verify correct hashing, the hashing algorithm should be made publicly known.

Currently we use the SHA512 hashing algorithm since its digest size is 64 bytes, which gives us reasonably sized messages flowing through the system, while still having 2^{512} possible different values. It could be argued that the 32 bytes of SHA256 are more than enough for any use case. However, SHA512 is actually roughly 1.5 times faster than SHA256 on a 64-bit CPU [12]. Therefore, we see no reason to limit the possibilities to 2^{256}, since we do not expect 512 bits per input to be too much data. We implement the system such that the chosen hashing algorithm can be configured at beacon start.

Notice that a beacon outputs twice per cycle. Before the delay function is applied, a commitment is released from the computation node and published through all publishers. Specifically, this commitment contains all leaf nodes of the Merkle tree, which corresponds to all inputs. After the delay function, the output is sent to the publishers. The output consists of two parts: *a*) the result, computed from the commitment; and *b*) a witness, which can be used to rapidly verify the result. The commitment and output are correlated by an arbitrary sequence number.

The publishers publish to multiple different outlets. We implement several publishers, with different capabilities. This means that e.g. a JSON publisher can dump all messages, while a Twitter publisher only is able to post messages with 280 characters.

5.5 Input Processing and Computation

The "core" of the beacon, i.e. input processing and computation, is what collects and compiles the user inputs, and then computes the random output. In our beacon implementation we separate these steps into the two distinct components as described in the design of the beacon.

We develop these steps to be independent of each other, besides a well-defined contract consisting of two messages from the input processing to the computation, specifically: *a*) condensed output from input processing, which is the input to the computation; and *b*) data from input processing, which is the commitment to the computation.

5.5.1 Combining Inputs

One way to combine the inputs is the simple operation of concatenating them. This is then used as commitment data, while a hash of the commitment data can be used as the condensed output. This processing method requires the users to acquire the full commitment, if they want to confirm the inclusion of their input — which can be suboptimal in cases of significantly many users.

Although our beacon implementation allows for virtually any input processing method, we choose to focus on a Merkle tree approach. A Merkle tree is a special binary tree where the value of each node is the hash of the concatenation of its two children.

In our implementation this means that the leaf nodes are user inputs, which are already hashes, and the root node is the condensed output. For consistency, the hashing algorithm used to construct the tree is the same as the one applied to sanitize each input (SHA512). Truncating a SHA512 to any desired length is safe [12].

Merkle trees as commitment data allows third-party applications to provide verification, since the inclusion of a given leaf node in a Merkle tree can be verified by providing all siblings to the nodes on the path up to the root. This greatly limits the amount of data which the user needs to fetch and process to $\log n + 1$ where n is the number of leaf nodes in a Merkle tree where all levels are filled, i.e. there is a power of two number of leaves. The data consists of $\log n$ sibling nodes in the path to the root, and for comparison the root node as well (the +1). The commitment data consist of only the leaf nodes. This is possible if the ordering of the leaves is retained, and the algorithm to construct the tree is publicly available.

Another property of the Merkle tree is that, like hashing a concatenation of all collected inputs, each leaf node equally affects the root node, due to the diffusion property of the hashing algorithm. This means that any change to the set of inputs completely changes the root node in the Merkle tree.

To the best of our knowledge Merkle trees has never been used in previous beacon implementations as a means of combining inputs. However, they are used in other cases where it is undesirable for users to fetch all data for verification, e.g. in bitcoin where a Merkle root of all transactions in a given block is stored in the block header.

5.5.2 Parallel Computation

As we discussed in Section 4.6, we need parallel and time offset computations in the beacon. This is achieved by letting the input processor handle the scheduling of computations.

The beacon is configured to process inputs at a lower bounded interval, which means that the input processor will send work at fixed times, given an available computation component. It should be noted that if no such computational component is freely available, the input processor will just continue collecting input. Does no computation service announce itself within a given threshold, the input processor will give a warning to the system operator. This scenario should be unlikely since the beacon operator should configure the system to always have available computation components waiting for work.

The worker announcements and subsequent work assignments are facilitated with *ZeroMQ*'s "router/dealer" socket pair which allows asynchronous addressed messaging. When a computational node connects to the input processor it sends a `READY` message, receives an `OK`, and proceeds to wait for incoming work; this process, accompanied by what follows inside the computational node, can be seen in Pseudocode 1. The input processor then keeps track of each announced worker, and when the time comes, sends condensed processing output and commitment data to the next free worker.

If the worker does not acknowledge the work with an `OK` response, the inputs are reprocessed, and the next free worker is assigned. This cycle continues until a worker accepts the work, while new incoming inputs are included in each reprocessing of inputs. Having duplex communication between the input processor and the computation nodes is a practical compromise between a strict pipeline pattern and a monolithic input processor/computation node.

5.5.3 Delay Function

For the computation we implement a delay function based on *sloth* by Lenstra and Wesolowski [14]. The general idea behind *sloth* is to use modular square root arithmetics to construct a deterministic time hard algorithm, while containing a trapdoor for fast reversal, i.e. verification. The computation of *sloth* iterates through modular square root permutations of a large prime number. This is a significantly more expensive operation than its inverse, which is used in the verification process. Essentially, the verification calculates squares of the output from the computation.

Pseudocode 1 Specification of computational node outlining the communication pattern with the input processor.

```
1  procedure INITIALIZATION( )
2    CONNECTTO(input processor, publishing proxy)
3  end procedure
4  procedure MAINLOOP( )
5    repeat
6      SENDTOINPUTPROCESSOR( READY )
7      if OK received before timeout then
8        W ← RECEIVEWORK( )                ▷ blocking call
9        if W is valid then
10         SENDTOINPUTPROCESSOR( OK )
11         STARTCOMPUTATION(W_INPUT)
12         SENDTOPUBLISH(W_COMMIT)
13         wait for computation to finish
14         C ← COLLECTCOMPUTATIONRESULT( )
15         SENDTOPUBLISH(C_OUTPUT, C_PROOF)
16       else
17         SENDMESSAGE( ERROR )
18       end if
19     else
20       continue
21     end if
22   until the end of time
23 end procedure
```

When implementing delay functions in systems that rely on their time guarantees, it is important to focus on performance, since an obvious yet undeployed optimization of execution time would compromise the "time hardness" of the algorithm. Because of this, and the fact that Python is not the best performing language, we implement *sloth* as a Python module with a C-extension for the actual algorithm. In the C-extension the GNU MP library[7] is used to perform integer arithmetics with extremely large numbers.

In Section 6.2 on page 23 we evaluate using *sloth* as our delay function, and how execution time of the delay function can be adjusted.

6 Performance Evaluation

One way of evaluating our beacon is to examine the performance of the key parts. In this section we explore the performance of potential system bottlenecks to gauge reasonable throughput. We also investigate our chosen delay function *sloth* and different configurations of it.

[7] https://gmplib.org/

All tests are executed on a server with an *Intel Core i7-2600* CPU, which runs at 3.40 GHz and has 4 cores. As such, it can run 4 simultaneous sloth computations. Further, it supports simultaneous multithreading, “hyper-threading”, but irrelevant to our system. While it might improve total throughput of multiple computations, it also does not speed up individual runs of sloth. We use *SHA512* as the underlying hashing algorithm in both the Merkle tree and in the *sloth* delay function.

6.1 Bottlenecks

We examine two potential bottlenecks in our beacon. These are components which require the most effort to scale horizontally, and as such for simplicity we want to discover the current limits before scaling. Furthermore, it is important to determine if these are actual bottlenecks before attempting to scale, as we want to avoid premature optimization.

6.1.1 Proxies

As presented in Section 5.3 on page 18, our beacon contains two proxies. We believe that the *forward* proxy between computation and publishers never will be a bottleneck in a real world randomness beacon deployment, as the data passing through it only consists of outputs, commitments, and proofs. However, the *stream* proxy situated between input collectors and input processors must be equipped to handle a constant stream of input messages.

As previously mentioned, this proxy facilitates fan-in and fan-out pipelining with fair message distribution using a round-robin strategy. Hence, we test the throughput of the proxy in different configurations of input collectors and input processors. For simplicity and benchmark consistency, we utilize “dummy” components for this. The input collectors are referred to as *pushers* and fan in at the proxy, while the input processors are called *pullers* and fan out. In the tests we transmit messages which resemble those of an actual beacon in size, i.e. 64 bytes of application data plus any *ZeroMQ* packaging; in this case one byte which serves as a flags field, and one byte to denote the length of the message body[8].

In Figure 11 we see how the aforementioned different configurations affect the throughput of messages in the proxy. Firstly, no configuration combination

[8] As per the framing specification in `https://rfc.zeromq.org/spec:23/ZMTP`

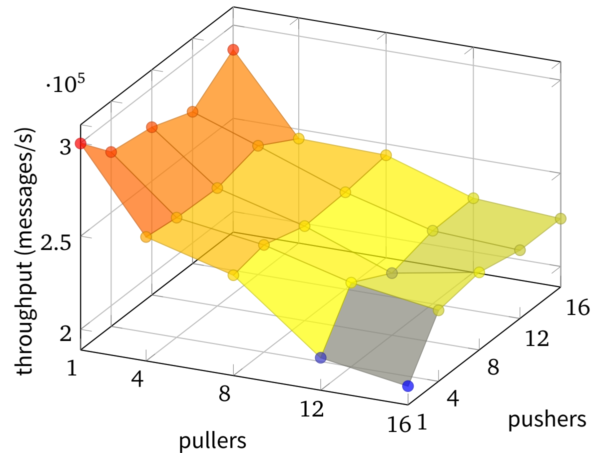


Figure 11: 64 bytes message throughput per second of *stream* proxy, with different numbers of pullers and pushers. Overhead of round-robin message distribution can be seen affecting throughput.

measured results in a throughput below circa 200,000 messages per second. We presume this is significantly higher than the number of inputs a real world beacon would ever be constantly subjected to — as this also would cause significant problems further down the pipeline, e.g. the sheer amount of data contained in a commitment.

It is the scenario of one pusher to sixteen pullers that results in the lowest throughput, which can be caused by the overhead of the fair message distribution enforcement. However, as we add pushers at sixteen pullers, a slight increase in throughput can be seen, suggesting that fair distribution is easier with more suppliers.

Another observation we can make from Figure 11 is that increasing the number of pushers does not affect the throughput as much as adding pullers does. This evinces that fan-out is a considerably more expensive task than fan-in — a fortunate fact, since a deployment of our beacon most likely will consist of remarkably more pushers than pullers.

We can conclude that the proxies in our system are extremely unlikely to be bottlenecks, and we should rather look further down the pipeline for issues; hence we examine the input processor.

6.1.2 Input Processor — Building Merkle Trees

The most expensive task performed in a bottleneck is building the Merkle tree in our input processor. This task is done periodically when it is time to compute a new random output. It should not take a significant amount of time, since this would extend the time be-

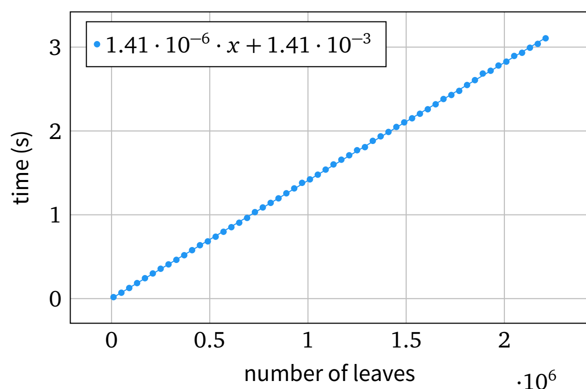


Figure 12: Correlation between number of leaves and the time it takes to build a Merkle tree, with those leaves.

tween the last seen input and publishing the commit. As such, we examine how the number of leaves, i.e. inputs, affects the building time of the Merkle tree.

In Figure 12, a linear growth in build time is seen as a factor of the number of leaves. The growth is slow and is negligible in our beacon. There needs to be well over 2 million leaves to result in a build time over three seconds. We can describe the relationship as follows, where N is the number of leaves:

$$1.41\,\mu s \cdot N + 1.41\,\text{ms}$$

Admittedly, the build time could be a problem if significantly many inputs are used. However, in this case one might reimplement the input processor in a more performant language than Python, e.g. C. In addition, the construction of Merkle trees is trivially parallelized. Our implementation of Merkle trees does not take advantage of this fact, and so building subtrees in multiple processes and merging them to form the final tree will likely provide a significant speed-up with a factor close to the number of available CPU cores.

6.2 Parameters of *sloth*

The computation and verification time of the delay function, *sloth*, can be configured by adjusting two parameters. These are *a*) the size of the prime number used in the computation, in bits (must be a multiple of 512); and *b*) the number of times to iterate through the permutation process of said prime.

To evaluate the *sloth* delay function we run a series of tests of the algorithm. During the tests we sample multiple rounds with random inputs and take the average. This is done to mitigate testing inputs, which are significantly faster to find primes for.

In Figure 13a on the following page we illustrate the correlation between these two parameters, and the time it subsequently takes to do a computation with a given combination of bits and iterations. We see that an increase in number of bits used for the prime number results in an exponential growth of the computation time, while an increase in number of iterations cause a linear growth. The data points in the plot are highly regular as expected as it only depends on single-core performance. It shows that the computation time is reliable and grows as expected without significant deviations, despite running the tests on a machine with a fully fletched operating system.

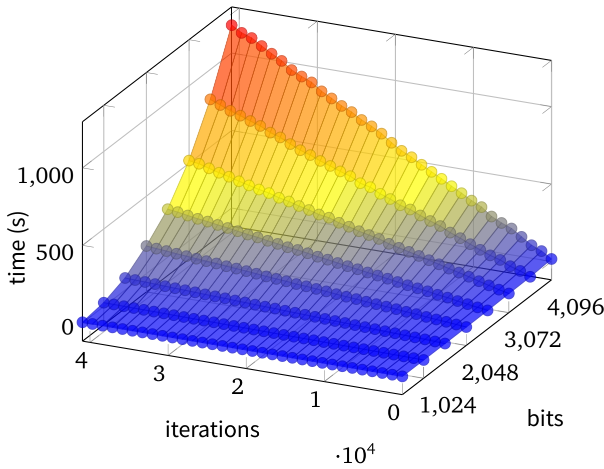
While computation time is important for the delay function, another significant metric is verification time — especially in relation to the computation time. Figure 13b on the next page illustrates this relationship, where the z-axis shows how many more times it takes to compute the output relative to how long it takes to verify. Although the data is more scattered than in the previous figure, we see a trend where the growth of this factor levels out just above 10^2. This means that in configurations with more than roughly 3,000 iterations, the computation time is always more than two orders of magnitude larger than the verification time.

We also see that the number of bits does not affect the factor except for some irregularities in the data. These irregularities are believed to partly be caused by the extra time it potentially can take to initially find the prime number; an operation which can vary in time depending on how close the numeric representation of the hashed input string is to a prime. Since larger primes (given by number of bits) can be more difficult to find, the data fluctuates more at larger number of bits.

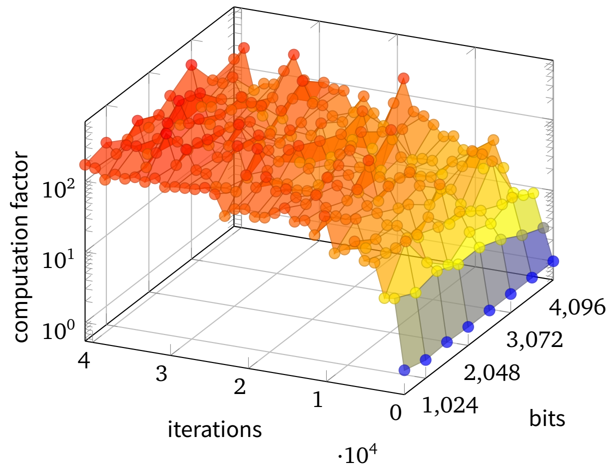
7 Applied Use Cases

To demonstrate the usefulness of our randomness beacon we present a series of use cases, where it may be utilized. The fundamental use case of a beacon is to generate a random value that multiple parties can agree is not biased to anyone's advantage or disadvantage.

Our beacon can be thought of as a *cryptographic primitive* that can be used to obtain randomness. Users can then use the beacon in a way that fits their specific application of randomness. Conducting *cere-*



(a) Correlation between bits and iterations in relation to time of computation.



(b) Computation time as a factor of verification time. Note the logarithmic z-axis.

Figure 13: Execution time of *sloth* computation and verification with different parameters. The computation time grows exponentially as the number of bits is increased, and linearly as more iterations are performed. Verification time grows far less.

monies around the usage of a beacon output is critical to properly apply the randomness produced by our beacon. In essence, the spirit of our security goals for the beacon itself must be carried on to the use cases.

Thus, we present not just the use cases, but how our beacon can be used to securely provide randomness for them.

In all use cases, a mapping from the beacon output space (in our case a 512-bit string) to the desired application space, must be known before the output is announced.

7.1 Lotteries

A lottery is an example of a sampling use case. Lotteries exist in many shapes and sizes, from traditional lotteries with a grand prize to military conscription lotteries, in which many of the picked men would later be sent to the Vietnam War [24]. The example is also generalizable to other sampling use cases, such as sampling counties to perform election recounts [23]. Lotteries have clear incentives, as they have direct notions of prizes, money, advantages, or disadvantages; and because they are also general and describes many use cases, lotteries make a good example of beacon application ceremonies.

We consider ceremonies for two types of lottery applications: *a*) between a group of friends ("small lotteries") and *b*) with a weekly lottery service ("large lotteries"). We illustrate the differences between the two and how to structure the use of the beacon to obtain randomness in line with our security goals.

7.1.1 Small Lotteries

Consider a group of three friends that organize a small lottery among themselves. To prevent cheating they decide to use our beacon to pick the winner. Recall that a user can only trust an output that they have inputted to — so all three must be able to find their inputs in a single output in order to use it.

We also do not guarantee any single output to contain the inputs of all three friends, so we advise users to repeatedly submit their inputs unless they see a commit with their input and a corresponding output.

Each friend commits to a specific input before repeatedly inputting it to the beacon, until a commitment containing all three inputs is released by the beacon. They should then use the output that corresponds to that commitment and use that to determine who wins. How the three friends maps the output space of the beacon to their three outcomes, should be agreed upon before engaging in the lottery.

7.1.2 Large Lotteries

While the approach of having everyone input works for small groups, we cannot reasonably expect every participant in large groups to repeatedly input to the beacon until a common commitment containing all inputs is found.

We consider a weekly lottery that is open for anyone to purchase tickets. Since we cannot wait for simultaneous inputs from all customers, we instead use the set of beacon outputs produced during the lottery lifetime, giving customers a larger time-frame

to influence the outcome. To determine the lottery lifetime, and more specifically when to stop collecting beacon outputs, a transparent way to signal the last beacon output to use is needed. Practically, the lottery must commit to a *stop message*.

The lottery then collects all beacon outputs published in the duration of the lottery, and combines them for use in the final decision, e.g. by use of a hashing function. This combining should be predetermined such that the output of said operation is as unpredictable as the beacon outputs themselves. Once it is time to draw the winners, the lottery sends a signed version of the previously mentioned stop message as input to the beacon. The stop message must be cryptographically secured by signing it, such that no other party can send the message and thus stop the lottery prematurely. When a commit from the beacon containing the signed stop message has been seen the lottery entity, the lottery entity then announces the signed version of the message. The beacon output containing the stop message is then the final one used in the aforementioned output combining, which can now determine who wins.

This scheme relies on the lottery being able to end their collection of input securely, and gives four guarantees to users. *1*) users can verify the presence of the signed stop message in the output by checking the beacon commitment; *2*) users can verify that the stop message was sent by the lottery, as they can verify the signature; *3*) users can be certain that the lottery did not craft the stop message to bias the input in a last-draw attack, as it was committed to at the beginning of the lottery; and finally *4*) users can be sure that adversaries did not craft a last-draw attack around the stop message, as they did not know the signature of the message beforehand.

Users will also have a large opportunity to input to the beacon to influence the final draw. This ceremony extends the “three friends’ lottery” to accept inputs from a large group over an extended period.

7.2 Cryptography

Some cryptographic concepts can benefit from using a randomness beacon, namely parameter generation and protocol bootstrapping. Many cryptographic protocols and schemas require some parameters to initialize. Choosing these can be a lengthy process [6], but also requires a great deal of trust as they can contain backdoors if crafted meticulously [13].

Alternatively the parameters could be pseudo-randomly generated by a generator that only generated good parameters. The generator could be seeded by a randomness beacon, as described by Baignères, Delerablée, Finiasz, Goubin, Lepoint, and Rivain [1]. This could be accomplished much like the large lottery ceremony by announcing a collection period and stop message. Using all inputs collected within that period as the seed would then make a wide variety of interested parties able to input to the parameters, giving them some measure of trust in the protocol.

7.2.1 Bootstrapping Protocols

Another use case is bootstrapping for zkSNARK systems. Such systems require a *common reference string* that must be generated as part of the bootstrapping process. Generating this string can be an extremely complicated process as the trust of the entire system rests on the string. Should any party possess the complete data from which the string was generated they can fake proofs of anything, undermining the system.

The process requires users to trust at least one participant of the bootstrapping ceremony. Because of the complexity it is difficult to scale to more than a handful of participants. Using a randomness beacon allows the process to scale far beyond the norm, as demonstrated by Bowe, Gabizon, and Miers [6]. They present a MPC protocol for zkSNARK bootstrapping. The protocol operates across two rounds and each includes an input from a randomness beacon.

Practically, each round could be organized around a number of beacon outputs containing specific *round number* messages. These messages are signed and committed similarly to the stop messages explained in Section 7.1.2. This extends the period of the rounds beyond that of our beacon’s output intervals, and ensures that the completion of the bootstrapping protocol is not disrupted by a missing beacon output.

8 Discussion

In this section we discuss some questions, which came up during our work. This includes presenting some alternatives to our approaches, and exploring threats which we did not succeed in mitigating.

8.1 Output Dependency

We suggest a variety of use cases for our beacon, but we must include one important caveat: users should not critically depend on the output.

This is because we can not always guarantee that there will be an output due to the possibility of attacks on the availability. Instead, users should aim towards being flexible with when they need to use an output. An example could be to use the *next* output that they see a timely commit containing their input to. Only then will they be certain that the output is not biased.

8.2 Alternative Delay Functions

We currently use a time-hard function to compute our randomness. This provides us some indication of the computational effort needed to correctly produce a random number, and thus makes it harder to cheat. However, requirements on processing power is not an insurmountable obstacle for motivated attackers. An excellent example of this is the bitcoin blockchain, where mining consists of solving a computational puzzle by repeatedly hashing. Here, Application-Specific Integrated Circuits (ASICs) have allowed significant speedups in the mining process, which has resulted in raised mining difficulty, and ordinary computers becoming comparatively useless for mining purposes. If any party was to develop ASICs for our delay function, they would be able to solve it much faster than any other party. This would diminish the security provided by the delay function, and open up for last-draw attacks from the party with the ASIC.

One way to mitigate this would be to increase the difficulty of the delay function, like it has been done in bitcoin. However, this would have the side effect of making the function much more costly to compute for any party without ASICs. This could even include the beacon operator, which would impact operations.

Another way to mitigate this would be to use a delay function that was also memory-hard, i.e. required large amounts of memory to compute. This would make it resistant to ASIC-equipped adversaries, as these have small amounts of memory to optimize for speed. While the function requires more resources, it should still be computable for an ordinary computer.

8.3 Salting

One thing we considered in the design of the beacon was having the operator add a salt to the inputs. This would make the operator the only party capable of computing the output, which would prevent outsiders from pre-imaging and performing last-draw attacks without help from the operator.

On the other hand, this would give the operator even more power. They could perform more damaging withholding attacks, as they would be the only party capable of producing the beacon output. This could be mitigated by having the operator publish a timed commitment to the salt alongside the inputs — this second commit could then be revealed by outsiders given enough time, which would prevent the operator from withholding once both commits were published.

8.4 Smart Contracts

A distrustful environment is an obvious setting for a randomness beacon, and one of the most obvious types of these are public blockchains. We therefore find it interesting to consider implementing our beacon in a smart contract on a blockchain. We consider implementing it in a public blockchain such as the bitcoin or Ethereum blockchains.

There would be some definite benefits to this, namely that it would remove the need for a single operator, which removes some threats towards the beacon. In addition, it would be much harder to DoS attack the beacon, as it would be run by the entire network. However, a central aspect of our beacon is practically incompatible with the nature of smart contracts, as delay functions are computationally intensive. Smart contracts need to pay for each computation they make, which would make the beacon costly to run.

This would tie the beacon into the monetary incentive structures that dictate smart contract behavior. However, some parts, like the verification process, are not nearly as intensive, and could potentially be implemented in a smart contract.

In addition, since parts of the beacon would still be off-chain, those parts would still be dependent on an operator and vulnerable to DoS attacks.

Another thing to consider is that everything that occurs on a blockchain occurs because someone put it into a block. This is typically the job of miners, who have different interests than other users. They are incentivized to include the transactions that give the greatest rewards for the block. Thus, a user would have to pay a competitive fee to interact with a beacon on a blockchain.

If we also consider the trust assumption of everyone being against the user, they would have to mine the block themselves to guarantee their interaction, which is a steep requirement.

9 Conclusion

This work was based partly on interest in randomness beacons, and partly on wondering why these services are not implemented for real world usage.

While we took inspiration from previous work and literature, designing a randomness beacon from scratch was no simple feat. We designed a system which is based on simple and succinct principles. However, simple systems still have many details in their implementation. We made conscious choices regarding communication framework, communication patterns, compromises between verbosity and bare essentials in communication, and put a lot of thought into the practicalities of deploying and operating our beacon. In particular, we parallelized the beacon operation to have a continuous input stream and regular output, and outlined a formula for finding the number of computation nodes. We also scaled input collection by enabling horizontal scaling, and improved usability of the system by allowing these input collectors to collect from a variety of sources with easy access for users.

We showed the possibility of instantiating a randomness beacon with sensible guarantees for any single user; i.e. given their random input to the beacon, they can easily and rapidly verify the computation, and decide if they deem it trustworthy.

We performed a security analysis of randomness beacons to identify threats towards them, and incorporated counter measures for them in the design of our own beacon. Some threats were impossible to counter, and these were discussed.

We refined and extended the work of Lenstra and Wesolowski [14], who propose a delay function and provide a short discussion on the repercussions of the users' trust assumptions by using a delay function. Our refinement allows all users to run the delay function in parallel with the beacon operator, or to run it if the beacon operator (maliciously or not) performs an output withholding attack. We extended the discussion of users' trust assumptions by providing a succinct formula, which only depends on two timestamps and what the user believes is the fastest possible computation of a given delay function.

Further, we have proposed a parallelization technique for the beacon pipeline, such that input collection is a continuous stream. This allows the output frequency to be considerably increased, and enables adjustments to the ratio between input collection time and computation time. Simultaneously, we noted that computation components can be reused for subsequent computations instead of instantiating new services, thereby removing startup time. For redundancy, reducing bottlenecks, and usability we allow multiple input and output channels.

We also explored the applications of our randomness beacon as a cryptographic primitive in a variety of use cases. We presented ceremonies for using our beacon to securely obtain public randomness. The use cases range from lotteries to crypto system bootstrapping, and each has a ceremony associated with it. As such, we not only investigated the gap in literature of a practical beacon implementation, but also discussed the practicality of the beacon in context.

We believe our contributions fills a hole in current literature, and as such can be seen as a step forward. It is, however, clear that there is need for more work that is not necessarily technical in nature. No matter how much we believe in randomness beacons as a concept, it might prove hard to convince users to use a beacon. While we argued that the beacon is not necessarily expensive to run for the beacon operator, incentives to run a beacon still needs further thought. Finally, usability of the beacon needs to be improved if we expect normal users to use it.

In the future, trusting any entity with decisions will, in our opinion, be more far-reaching than it is today. We see more and more corporate giants virtually controlling whole industries. Before empowering a few world-wide corporations to make decisions that affect us, we, as normal users in society, need ways to ensure fairness. Randomness beacons might be one such way.

10 Future Works

This section outlines next steps to explore regarding our randomness beacon.

10.1 Usability Applications

To ease users interaction with the beacon and the process of verification, some usability applications could be implemented.

This could be in the form of a "client" application, which facilitates input submitting and output verification. Interacting with such an application could from the users' perspective be as simple as running it in the background, while the application constantly inputs to the beacon, and verifies both inclusion of said

input and correct computation of the output. The user could then at any time poll the application for a *verified* beacon output. In this scenario, the user would inherently need to trust the application to not be acting malicious.

Another usability application, taking a far simpler approach, could be a simple verification service. It allows users to painlessly check if their input was used in a beacon output. Since our beacon uses a Merkle tree for storing the inputs, the verification service could provide users with proofs of inclusion, without them needing to fetch the entire set of used inputs. In fact, user would at most need to download $\log n + 1$ nodes in the Merkle tree, to validate the proof; this is further explained in Section 5.5.1 on page 20. Users could then correlate the valid proof with the output of the beacon for complete verification.

10.2 Incentives and Use Cases

We have assumed that our beacon will be run by an interested authority as a greater good. In our design, the speed of the beacon operator's computer will not be significantly important. The computation is only a service to save all users the CPU time of computing it themselves. As such, the computation of the delay function should only happen so fast that users will not be annoyed and run the delay function themselves — there is no loss of security by waiting a bit for the beacon operator's output. The only thing that matters in the design of our beacon is the time from a user inputting and the user receiving a commitment. This duration should be lower than what the user believes is the fastest possible execution of the delay function by any computer in the world.

Because of the fact that speed does not matter *much*, it enables a relatively low entry barrier. We have imagined universities or privacy-minded corporations to run the beacon as a public good. It will not require much besides a reasonably powerful server, where the single-core performance and number of cores will be the main resources to consider. These authorities' willingness to run a beacon is purely speculative, and as such we do not know if these authorities are even interested in running a beacon. Therefore, incentives for running a beacon will need to be investigated further. A strong incentive to run a beacon is compelling use cases. As such more use cases, and incentives for using a beacon as a cryptographic primitive, should also be further explored, beyond our discussions in Section 7 on page 23.

11 Acknowledgements

We extend our warmest thanks to René Rydhof Hansen & Stefan Schmid for supervising our project and providing insightful discussions and feedback, and Benjamin Wesolowski for providing a reference implementation of the *sloth* delay function. Furthermore we would like to express our gratitude to our colleagues Rune Willum Larsen & Alex Grøndahl Frie for sharing knowledge and an office with us, and for helping us maintain our physical health.

References

[1] Thomas Baignères, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Tancrède Lepoint, and Matthieu Rivain. “Trap Me If You Can — Million Dollar Curve”. In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1249.

[2] Michael Ben-Or, Oded Goldreich, Silvio Micali, and Ronald L. Rivest. “A fair protocol for signing contracts”. In: *Automata, Languages and Programming*. Ed. by Wilfried Brauer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 43–52. ISBN: 978-3-540-39557-7.

[3] Iddo Bentov, Ariel Gabizon, and David Zuckerman. “Bitcoin Beacon”. In: *CoRR* abs/1605.04559 (2016).

[4] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. “On Bitcoin as a public randomness source”. In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1015.

[5] Sean Bowe, Ariel Gabizon, and Matthew D. Green. *A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK*. Cryptology ePrint Archive, Report 2017/602. 2017.

[6] Sean Bowe, Ariel Gabizon, and Ian Miers. *Scalable Multi-party Computation for zk-SNARK Parameters in the Random Beacon Model*. Cryptology ePrint Archive, Report 2017/1050. 2017.

[7] Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. “Proofs-of-delay and randomness beacons in Ethereum”. In: *IEEE Security and Privacy on the blockchain (IEEE S&B)* (2017).

[8] Ignacio Cascudo and Bernardo David. "SCRAPE: Scalable Randomness Attested by Public Entities". In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 216.

[9] Jeremy Clark and Urs Hengartner. "On the Use of Financial Data as a Random Beacon". In: *EVT/WOTE* 89 (2010).

[10] Ivan Damgård and Yuval Ishai. "Scalable secure multiparty computation". In: *Annual International Cryptology Conference*. Springer. 2006, pp. 501–520.

[11] M. J. Fischer, M. Iorga, and R. Peralta. "A public randomness service". In: *Proceedings of the International Conference on Security and Cryptography*. July 2011, pp. 434–438.

[12] Shay Gueron, Simon Johnson, and Jesse Walker. "SHA-512/256". In: *Information Technology: New Generations — ITNG 2011* (2011), pp. 354–358.

[13] Jennifer Huergo. *NIST Removes Cryptography Algorithm from Random Number Generator Recommendations*. 2014. URL: https://www.nist.gov/news-events/news/2014/04/nist-removes-cryptography-algorithm-random-number-generator-recommendations (visited on June 6, 2018).

[14] Arjen K. Lenstra and Benjamin Wesolowski. *A random zoo: sloth, unicorn, and trx*. Cryptology ePrint Archive, Report 2015/366. 2015.

[15] N. R. May, H. W. Schmidt, and I. E. Thomas. "Service Redundancy Strategies in Service-Oriented Architectures". In: *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*. Aug. 2009, pp. 383–387. DOI: 10.1109/SEAA.2009.59.

[16] J. D. Meier, Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla, and Anandha Murukan. *Improving Web Application Security: Threats and Countermeasures*. 2003. URL: https://msdn.microsoft.com/en-us/library/ff649874.aspx (visited on Apr. 19, 2018).

[17] Mathias S. Michno, Michael T. Jensen, and Sebastian R. Kristensen. *A Survey of Randomness Beacons, or: How I Learned to Stop Worrying and Love the Blockchain*. 2017. URL: https://github.com/randomchain/beacon-taxonomy-paper/releases/download/final/main.pdf.

[18] National Institute of Standard and Technology. *NIST Randomness Beacon*. Dec. 6, 2017. URL: https://www.nist.gov/programs-projects/nist-randomness-beacon (visited on June 6, 2018).

[19] Nicole Perlroth. *Government Announces Steps to Restore Confidence on Encryption Standards*. 2013. URL: https://bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/ (visited on June 6, 2018).

[20] Nicole Perlroth, Jeff Larson, and Scott Shane. *N.S.A. Able to Foil Basic Safeguards of Privacy on Web*. 2013. URL: http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html (visited on June 6, 2018).

[21] Michael O. Rabin. "Transaction protection by beacons". In: *Journal of Computer and System Sciences* 27.2 (1983), pp. 256–267.

[22] randao. *RANDAO: A DAO working as RNG of Ethereum*. Nov. 2, 2016. URL: https://github.com/randao/randao (visited on Dec. 11, 2017).

[23] Chris Skelton. *Bush v. Gore, 531 U.S. 98 (2000)*. 2000. URL: https://supreme.justia.com/cases/federal/us/531/98/ (visited on Mar. 5, 2018).

[24] Norton Starr. "Nonrandom risk: The 1970 draft lottery". In: *Journal of Statistics Education* 5.2 (1997).

[25] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. "Scalable bias-resistant distributed randomness". In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017, pp. 444–460.

Appendices

A DREAD Details

This section serves to further expand our use of the DREAD framework for threat evaluation. We further explain the different measures we use, and examine certain threats closer.

A.1 DREAD components

The first component of DREAD is damage, how damaging an attack is. We consider the most damaging attack to be one that successfully makes users use a biased input. This way users will be tricked into using an input that favors certain parties, which is the exact thing the beacon is supposed to prevent. This requires the attacker to bias the outcome without breaking the beacon protocol, as the output would otherwise not be used by observant users. A slightly smaller threat is DoS attacks. While this does prevent users from using it, the damage it causes is still less than it would be from using a biased output. Finally, the damage caused by attacks revealed through the transparency of the beacon cause negligible damage, as they are unlikely to be used by anyone but the most careless users.

The second component is reproducibility, how easy the attack is to reproduce. Generally, many of the attacks on the beacon are easily reproducible, but we consider it to be lower for malicious operators. This is because the power of malicious operators relies on users of the beacon. Whenever they diverge from protocol, or deny an output, whether by withholding or crashing, users will lose trust in that operator. As a result, they may eventually find themselves with no users of their beacon — this limits their ability to reproduce attacks. On the other hand, outsiders that want to bring the beacon down can use this fact to undermine even legitimate operators by continually DoS-attacking them.

The third component is exploitability, and describes how little work is required to launch the attack. This is essentially the initial investment required by the adversary to launch the attack, and the smaller that investment is, the greater the threat it poses.

The fourth and final component we use is affected users, which describes how many users are affected by a given attack. Here, we distinguish between an attack affecting all users, some users, or only a few to determine the score.

A.2 In-Depth Threats

We have selected a few threats to describe in depth. The threats were selected based on their severity and their ability to exemplify their respective categories.

Input Manipulation D R E A Σ 3 3 2 3 11 The operator can manipulate the inputs received to produce a biased input, that still appears legitimate to verifiers. The damage of the attack is severe, as all users will use the biased output without suspicion. The attack is also completely reproducible, as long as the operator has the ability to execute it and is not somehow caught red-handed — something that would be extremely difficult to do. Thus he will theoretically be able to bias every single output of the beacon to his own benefit while still appearing as an honest operator. The attacks does require being the operator, but otherwise evaluates identically to the input bias attack performed by outsiders. This is because an outsider with the power to execute such an attack would likewise be able to bias every single output.

Emitting False Output D R E A Σ 2 1 2 3 8 As a contrast to the previous attack, here the operator forgoes the process of making it look legitimate, and simply emits a biased output. This output should never be used by any critical users, and so will not cause much damage by itself. In fact, the attack is more akin to a withholding attack, as it effectively denies users the output they have input to. It is also has low reproducibility as it would significantly impact the credibility of the operator. This attack will most likely tarnish the reputation of the operator, and as such the beacon will be used by an ever decreasing number of users.

Shutdown D R E A Σ 2 2 2 3 9 The operator can at any time shut the beacon down to deny operation to all users. This is quite damaging, but ultimately not as bad as making them use a biased input. This attack is also easy to reproduce, but limited by the fact that users will lose faith in a beacon that shuts down often, which eventually drives them away. While the attack is trivial to execute for an operator, we consider becoming the operator of a beacon a minimum investment in and of itself, hence why the E is a 2.

Input Flooding D R E A Σ 2 3 2 3 10 When it comes to availability attacks, outsiders are ultimately the greater threats, as they do not have a vested interest in the beacon. Besides, the beacon operator has easier attacks in his arsenal with the same outcome from the perspective of the users. Hence we see that this attack is slightly more reproducible, as the users would eventually abandon the beacon, which would be a success for the outsider.

B Implementation Structure

Code for the beacon can be found at our GitHub repository[9], and latest commit as of writing this is `47beb4ec03031c22b34d06db85d6fcc9c7bb1fd4`.

The concrete implementation of our PoC randomness beacon is structured as a Python package named *randbeacon*. See Figure 14 for a visual representation. In this package, each group of components, i.e. input collecting, input processing, computation, and publishing, is its own module. Even though no single component relies on any other, with the exception of the `utils` module for code reuse, this module structure makes it convenient to deploy a beacon instance during the development phase.

To orchestrate said deployment, we use the terminal multiplexer *tmux*[10] and a Python program called *tmuxp*[11], which allows us to easily specify a beacon configuration in a yaml file. This way of deploying through *tmux* is intended as a means of debugging and demoing the beacon.

Dependencies are managed with *pipenv*[12] and a Pipfile, thus encouraging the use of virtual Python environments. Besides the *randbeacon* package, the beacon relies on a proxy written in C, which can be found in the `proxy` directory. Outside the repository for the randomness beacon, we use our implementation of the *sloth* delay function[13] and our own fork of *pymerkletools*[14]. External dependencies are fetched from the official Python package repository[15].

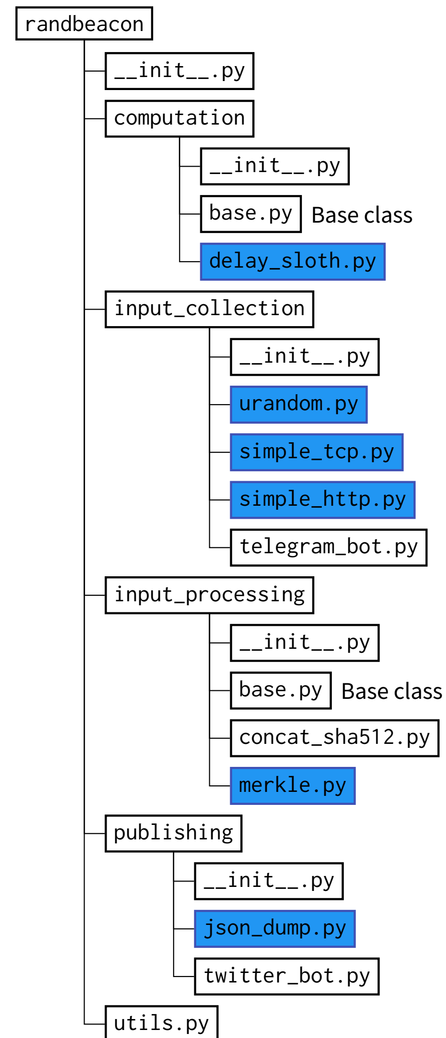


Figure 14: Structure of core beacon components. Placed under one python package, with modules for each group. The blue files indicate the components used in the current PoC deployment demo.

C Details of Drand Phases

Setup Phase During the setup phase each node to be included in the beacon generates a public/private key-pair, to be used long-term. A file called the *group file* is then created, consisting of all participants' public keys, and configuration metadata regarding the beacon operation.

The *group file* is then distributed amongst the nodes, whom then participate in a distributed key generation (DKG) protocol. This protocol creates a collective public key, and a sharded private key, with each node in possession of a unique shard, used for the internal cryptographic operations of Drand.

Randomness Generation Phase Any node may function as a leader and initiate the randomness generation phase by broadcasting a message consisting of a time stamp to all nodes. This time stamp message is then signed by all participating nodes with a threshold version of the Boneh-Lynn-Shacham (BLS) signature scheme. The threshold version allows any node to construct the full signature, which is the random output, given that enough nodes has provided their shard signature.

Output values can be verified by using the public key generated in the distributed key generation.

[9] https://github.com/randomchain/randbeacon
[10] https://github.com/tmux/tmux
[11] https://tmuxp.git-pull.com/
[12] https://docs.pipenv.org/
[13] https://github.com/randomchain/pysloth/
[14] https://github.com/randomchain/pymerkletools/
[15] https://pypi.org/