Breeding Unicorns: Developing Trustworthy and Scalable Randomness Beacons

Samvid Dharanikota[1] Michael Toft Jensen[2] Sebastian Rom Kristensen[2] Mathias Sass Michno[2]
Yvonne-Anne Pignolet[3] René Rydhof Hansen[2] Stefan Schmid[1]
[1]Faculty of Computer Science, University of Vienna, Austria [2]Aalborg University, Denmark [3]DFINITY, Switzerland

Abstract—**Randomness beacons are services that periodically emit a random number, allowing users to agree on the same random outcome without trusting anyone: ideally, the randomness beacon is secure (cannot be influenced) and transparent (can be monitored by users). Hence, such randomness beacons can serve as an important primitive for smart contracts in a variety of contexts. In this paper we aim to bridge the gap between theory and practice of public beacon design inspired by the unicorn protocol of Lenstra and Wesolowski using verifiable delay functions. We first present a structured security analysis, based on which we design, implement, and evaluate a trustworthy and efficient randomness beacon allowing users to join at any time. We then compare different implementation and deployment options on distributed ledgers, and report on a Ethereum smart contract-based lottery using our beacon.**

Index Terms—**Random beacon, transparency, smart contract**

I. Introduction

A *randomness beacon* is a service emitting unpredictable random values, defined in 1983 by Michael O. Rabin who used it to add probabilistic security in several protocols [1]. The primary use for randomness beacons are in applications where a group of users needs to agree on some random outcome, but do not trust each other. In particular, the main purpose of the randomness beacon is *not* necessarily to produce "better" random numbers than, e.g., using `/dev/urandom`; it merely allows users to agree on the same random outcome without trusting anyone. Randomness beacons come with many applications, including seeding the generation of elliptic curves, preventing selfish mining, or secure elections, and are generally seen as a "tool of democracy" [2].

While in early versions of randomness beacons, the beacon operator itself needed to be trusted (i.e., it is an unbiased third party), with obvious implications for security, recent literature has sought to design beacons where the need to trust the beacon operator is reduced or removed entirely.

In keeping with this trend, in this paper we design and implement a randomness beacon that works on the most pessimistic assumption possible: everybody (in particular, this includes the beacon operator) is secretly colluding against the user and is willing to invest money and resources towards manipulating or biasing the randomness. Specifically, we seek a design that minimizes the trust required by a user and also allows each user to decide how much they *want* to trust the beacon such that, a user will know that under self-chosen trust assumptions, the randomness has not been manipulated.

Randomness beacons are of particular interest in the context of distributed ledgers and smart contracts, steering the interaction of mutually distrusting parties. In such scenarios trustworthy randomness can speed up computations and break symmetries. Although many potential implementations and practical solutions are discussed in the literature on randomness beacons, very few actual implementations of public, general-purpose beacons have been published or made available.

Our Contributions. In this paper we bridge the gap between theory and practical solutions by designing and implementing a secure, trust-minimizing randomness beacon based on the *transparent authority* model which relies on *user input*. It is based on the unicorn protocol devised by Lenstra and Wesolowski [2]. It allows users to join any time and at low overhead. Our implementation relies on parallelized computation, which minimizes the possibility of malicious operation while avoiding idle periods. Furthermore and unlike other approaches of transparent authorities, the beacon operator in our beacon design has no private information: all inputs are hashed and are released to the public in batches before the computation. The beacon also offers users to make subtle decisions on when to trust the output. Our beacon uses Merkle trees as the data structure for inputs to reduce the computation proof size. Our experiments with a first prototype demonstrate the scalability of our approach. We further illustrate how this beacon can be deployed on distributed ledger platforms. We compare different (partial) on- and off-chain deployment options and discuss our experience and evaluation of Ethereum smart contracts for a lottery with our beacon.

To ensure reproducibility of our results as well as to facilitate follow-up work, we share our implementations on https://github.com/randomchain/randbeacon.

II. Basic Beacon Concepts

We first provide an overview along the two main concepts of a beacon: *input source(s)* and *beacon operation*. The input source(s) describes what input sources to use, while the beacon operation describes the design of the protocol, i.e. how to perform the computation and publish the output.

Input sources can be split into three categories. A beacon can use its *private source* of data to produce randomness. This potentially allows users to produce randomness of high quality at a high rate, but denies users access to inspect the process and thus requires users to trust the beacon and its randomness. It does not align with our stated 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 [3] which observes quantum mechanical effects to produce what is claimed to be high-quality randomness. As such, the users need to blindly trust the beacon operator, i.e., NIST in this case [4], [5], [6]. Beacons based on *publicly available sources* cover input from sources that are publicly available and which everyone can agree on the value of, e.g., bitcoin block hashes or lottery numbers. The users must trust the source to be sufficiently random, which may be fine for the examples mentioned. Finally, beacons can also rely on *user input* in which a user is allowed to directly provide input to the beacon. The idea is that a user provides a value that they believe is sufficiently random. The beacon then performs an operation on the set of user-supplied inputs, yielding an output that allows all users *a*) to *verify the inclusion* of their input and *b*) 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, users know their input was not knowingly "counteracted" by another user.

We can distinguish between three models for beacon operation, detailed below. In the *autocratic collector* model, a beacon is run by a party which requires blind trust from the users. As such, the computation is a black box with no possibility for proof of honesty. An alternative is to use *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. Despite significant work in the field, this approach is difficult to scale to large groups since any addition or removal of a user requires a new setup phase [7], [8]. This type of beacon is therefore not well-suited for public settings, but might fit in a controlled private context. Finally, in a *transparent authority* model, a single entity collects inputs and publishes them with a focus on transparency. Users can, by observing the beacon, verify that it behaves according to the protocol. This does not directly prevent Byzantine behavior, but rather makes it difficult to hide such behavior. This type also supports a wide variety of implementations, and can be scaled to a public setting.

We thus focus on transparent authorities and provide a scalable implementation of such a randomness beacon.

III. DESIGN

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.

- *Transparent Operation* Users should be able to oversee that the beacon operates according to the protocol and thus catch any deviations from it. Being able to verify whether their own input has been used, allows users to determine whether they should trust the output. Furthermore, users 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.
- *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 denial of service (DoS) protection. The protocol should be secure meaning that even if only a single user is honest, the output is still unpredictable.
- *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.
- *Practicality* Scalability of all components is important to be suitable for many use cases. Therefore, it should scale to at least several thousand users contributing with user input in every output. 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.

Security Design A major security concern is the operator's ability to predict or manipulate the output. Our solution for this problem is to ensure that each published output is paired with a commitment which can be used in the verification of the beacon. As a novel design decision, the commitment must contain all data required for the computation and all inputs.[1] The transparency allows any party to compute the randomness alongside the beacon operator. It ensures that the operator cannot 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.

To further decrease the possibilities of the operator trying different commitments before releasing them, we use a *verifiable delay function*. Delay functions can be seen as black box 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 that the operator cannot try more than one commitment before running out of time. As such, the operator is unable to perform the input manipulation attack

[1] In Unicorn [2], the beacon operator commits to an input that is revealed when publishing the next output

in a meaningful way. In order to avoid excessive computation by users performing verification, delay functions used in randomness beacons should be hard to compute and easy to verify, i.e., they must be *asymmetrically hard*. The delay function also protects against *last-draw attacks*, where an adversary attempts 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.

We use the delay function *sloth* [2]. As mentioned earlier, there is no secret input to the delay function in our design. Note that in [2], a different attacker model is used. More precisely, the beacon operator wanted to safeguard against adversaries trying to manipulate the outcome. In this work, we consider the beacon operator as potentially malicious. Therefore, we proposed that the operator produced a commitment to a set of inputs, while also revealing the inputs. This effectively means that anyone can calculate the delay function, and potentially be faster than the operator. We deemed that by having the operator include a secret input, to prevent anyone from computing the outcome before himself, the trust implications are too severe, as a user would have to trust that the operator did not try multiple secret values in parallel and chose the most beneficial outcome. In our design, an adversary may know the outcome earlier than an honest participant that waits for the beacon operator to announce it. However, the adversary cannot bias the outcome, as long as there is at least one honest party.

Rational Trust Assumptions In our approach we want to push beyond the need for honest operators and naïve users. To achieve this we extend the work of [2] 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 to the input for the delay function. This can be condensed to

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

where 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. This relation between the time taken to compute the delay function and the time before a commitment is seen allows users to flexibly adjust their willingness to trust the outcome has not been biased against them. A similar threshold is also described by [2], where they advise a ratio of no more than one fifth of the computation time spent collecting inputs. In their paper, the authors furthermore state that 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.

Parallelization Beacon 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 with a pipelined approach, meaning that several delay functions run in parallel but offset in time and on different input.

IV. Prototype Implementation

In this section we give a brief overview of the implementation of our beacon design. Our prototype has been implemented mainly using Python 3 with a few subcomponents written in C for performance. The message passing infrastructure of our SOA is implemented using the *ZeroMQ* framework for asynchronous message passing and concurrency. We can directly employ the "publish/subscribe" pattern provided by *ZeroMQ* between computation nodes and publisher. This pattern handles the message routing based on subscription prefixes, resulting in less traffic on the network. Furthermore, the fan-in for input collectors is implemented with a "push/pull" socket pair which ensures 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.

To avoid implementing heavy service discovery functionality and to simplify configuration, we deploy proxies at key points in the pipeline: one between input collectors and the input processor and one between computation and publishers.

Combining Inputs One of the most important tasks of our implementation is to combine the (hashes of the) collected input both as a preparation for the computation phase, but also to derive commitment data that can be verified by users. As a novel contribution, our implementation uses a *Merkle tree* for this purpose. A Merkle tree is a special binary tree where the value of each node is the hash of the concatenation of its two children; here the leaf nodes are the hashes of user inputs and the root node is then the condensed output.

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 $O(\log n)$ where n is the number of leaf nodes in a Merkle tree. The commitment data consist of an ordered list of the leaf nodes.

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

```
procedure INITIALIZATION( )
    CONNECTTO(input processor, publishing proxy)
end procedure
procedure MAINLOOP( )
    repeat
        SENDTOINPUTPROCESSOR(READY)
        if OK received before timeout then
            $W \leftarrow$ RECEIVEWORK( )                    ▷ blocking call
            if $W$ is valid then
                SENDTOINPUTPROCESSOR(OK)
                STARTCOMPUTATION($W_{\mathrm{INPUT}}$)
                SENDTOPUBLISH($W_{\mathrm{COMMIT}}$)
                wait for computation to finish
                $C \leftarrow$ COLLECTCOMPUTATIONRESULT( )
                SENDTOPUBLISH($C_{\mathrm{OUTPUT}}, C_{\mathrm{PROOF}}$)
            else
                SENDMESSAGE(ERROR)
            end if
        else
            continue
        end if
    until the end of time
end procedure
```

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 changes the root node in the Merkle tree.

Parallel Computation As discussed, 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 available, the input processor will just continue collecting input. If no computation service becomes available within a given threshold, the input processor will give a warning to the system operator.

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.

Delay Function For the computation phase we implement a delay function based on *sloth*, the function proposed to be used in the unicorn protocol [2]. The general idea behind *sloth* is to iterate through modular square root permutations of a large prime number and thereby construct a time hard algorithm, while containing a trapdoor for fast reversal, i.e., verification. Essentially, the verification calculates squares of the output

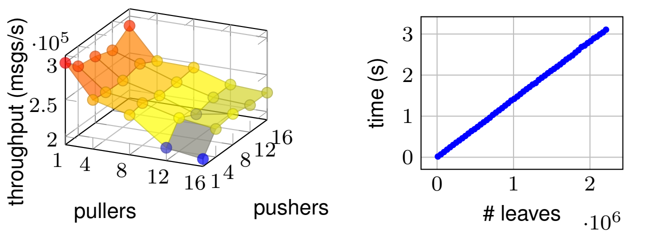

Figure 1. (Left) 64 bytes message throughput per second of *stream* proxy, with different numbers of pullers and pushers. (Right) Correlation between number of leaves and the time it takes to build a Merkle tree with those leaves.

from the computation. 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.

V. Performance Evaluation

We conducted several experiments to explore potential system bottlenecks to gauge reasonable throughput. We also investigate our chosen delay function *sloth* and different configurations of it. All experiments are executed on a server with an *Intel Core i7-2600* CPU, which runs at 3.40 GHz. The server has four cores and can hence run 4 simultaneous sloth computations. We use *SHA512* as the hashing algorithm in both the Merkle tree and in the *sloth* delay function.

Bottleneck Analysis We examine the potential bottlenecks which require the most effort to scale horizontally: the proxies and the input processor.

1) Proxies.: As discussed, our beacon contains two proxies. While the *forward* proxy between computation and publishers is unproblematic in any real world randomness beacon deployment (it only forwards outputs, commitments, and proofs), the *stream* proxy situated between input collectors and input processors may become a bottleneck, as it has to handle a constant stream of input messages. Recall that 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 message length.

In Figure 1 we see how the aforementioned different configurations affect the throughput of messages in the proxy. Firstly, every combination shows a throughput of at least 200k messages per second: likely sufficient even for popular real world beacons. 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

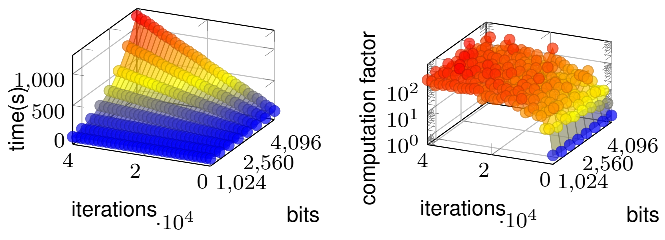

Figure 2. Execution time of *sloth* computation and verification with different parameters. The computation time grows exponentially as the number of bits is increased, and linearly as more iterations are performed. Verification time grows far less. (a) bits and iterations vs time of computation (b) computation time vs verification time with the logarithmic z-axis.

in throughput can be seen, suggesting that fair distribution is easier with more suppliers. Another observation we can make from Figure 1 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 muckiest more expensive task than fan-in — a fortunate fact, since a deployment of our beacon most likely consists of considerably more pushers than pullers.

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

2) Input Processor — Building Merkle Trees.: The most expensive task in our input processor is building the Merkle tree. This task is done periodically when it is time to compute a new random output. It is critical that this computation is fast, as this step can extend the time between the last seen input and publishing the commitment. As such, we examine how the number of leaves, i.e. inputs, affects the building time of the Merkle tree. In Figure 1, 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. Well over 2m leaves are needed to result in a build time over $3s$.

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.

Sensitivity Analysis of *sloth* The computation and verification time of the delay function, *sloth*, can be configured by adjusting two parameters: (1) the number of bits of the prime number used in the computation; and (2) the number of times to iterate through the permutation process of said prime.

To evaluate the *sloth* delay function and its sensitivity on the parameters, we run a series of tests of the algorithm. During the tests we sample multiple rounds with random inputs and take the average. Section V-2 illustrates the correlation between the two parameters, and the time it subsequently takes to do a computation with a given combination of bits and iterations. An increase in the 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.

While computation time is important for the delay function, another significant metric is verification time — especially in relation to the computation time. Section V-2 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 one hundred. This means that in configurations with more than roughly 3k iterations, the computation time is always more than two orders of magnitude larger than the verification time.

We also observe that the number of bits does not affect the factor except for some irregularities in the data. These irregularities are 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.

VI. Blockchain Applications and Implementations

As a case study, we consider the application of our beacon for distributed ledgers. Smart contracts between mutually distrusting parties can benefit from unbiased trustworthy randomness to speed up computations and break symmetries, e.g., in games. Since openness to any users is key in our design, an implementation providing randomness on a public permissionless blockchain makes most sense. We first compare different implementation options and the report on our blockchain-based implementation.

Design Choices There are essentially two options for the implementation:

- The actual beacon operator is run as a smart contract.
- The beacon operator runs separately from the blockchain, but publishes some of its artifacts on the blockchain.

While a fully blockchain-based solution offers benefits in terms of decentralization (no single point of trust/failure, more robust to attacks including DoS, ...), this solution is costly as each computation in the smart contract consumes virtual currency. This ties the beacon into the monetary incentive structures that dictate smart contract behavior. Due to the high cost, this option requires many users to compensate for the large on-chain computation cost.

The second implementation option offers several variants with different trade-offs. E.g., the verification can be done either on-chain in a smart contract, or by each interested user on their own. This has the advantage that expensive on-chain computations are avoided. Which artifacts and computation are on-chain and what parameter size are appropriate is an application-specific tradeoff between security and costs. Using blockchain-based distributed hashtables or other storage solutions such as IPFS (and e.g. storing only a pointer to the off-chain data on-chain) can further reduce the monetary cost: cost is proportional to the size of data stored on-chain (e.g., gas cost in Ethereum). Despite storing data off-chain, if the pointer is on-chain, we can still provide guarantees that tampering with the commitment and output will be detected.

The time necessary for a proposed transaction to be in a block that can be considered immutable may be highly variable depending on the nature of the underlying blockchain. Since the time interval between submitting an input and receiving a commitment from the beacon operator is the basis of trust for a user, the blockchain latency must be taken into account when configuring the delay function. Moreover the variability of the blockchain latency may tarnish the trust assumption of users. In addition, since parts of the beacon would still be off-chain, those parts will depend on an operator and are vulnerable to DoS attacks.

We study a *lottery* application based on our beacon, and to compare different implementations more systematically, we consider the following 3 players:

- *Owner* - Runs the lottery (e.g. smart contract owner)
- *User* - Takes part in the lottery by sending a small payment to the lottery smart contract
- *Beacon* - Beacon operator that provides a random value for the drawing of a lucky winner

The main goal of the lottery owner is to shave off some of the users' participation payments as a reward. In other words, not all of the user payments are given to the lucky winner, some of it is transferred to the lottery owner. Users only want to participate in a lottery when they have a reason to trust that the random value provided by the beacon is not biased, i.e., if they sent some input to the beacon to influence the generated random value and received a commitment within their trust time bound (on or off-chain).

We consider the following implementations, ordered according to increasing on-chain smart contract complexity:

- *Maximum off-chain (OFF)*: In this implementation all beacon-related logic is off-chain: only the lottery logic is on-chain. The users send their inputs to the beacon off-chain and obtain the commitment off-chain. Thereafter they send the lottery payment to the smart contract. When the off-chain beacon value computation has finished, the lottery smart contract fetches the value with an oracle. Using this value, it then determines the winner of the lottery. Users can verify if the beacon matches the commitment and complain off-chain and decide not to trust this beacon in the future. This has no influence on the outcome of the current draw of the lottery. Both the owner and the winning user receive rewards through the execution of the smart contract, while the beacon operator is remunerated off-chain.
- *Adding beacon incentive and on-chain commitment (ITV)*: To allow the beacon operator to be compensated with the smart contract, the following changes can be made to the simple contract proposed above. In a first step, the beacon publishes its public key and a nonce and locks some funds in the smart contract. Users send their input to the beacon off-chain and once they see their input included in the commitment, they send (i) the Merkle tree root signed by the beacon operator together with the nonce locked earlier and (ii) their participation payment to the smart contract. In this scenario, the beacon operator submits the next beacon value to the smart contract directly or it fetched with an oracle call. The verification of the correct execution of sloth on the Merkle root is performed on-chain. The beacon loses its locked funds if verification fails. If the verification succeeds, the owner, beacon and winner receive rewards.
- *On-chain inputs (INP)*: This version moves the input inclusion verification done by the user in the previous versions to the smart contract. In this case the user can send their input on-chain together with the lottery payment and it is then up to the beacon operator to send the corresponding commitment in time to avoid losing its locked funds. Thus the user does not have to worry about the commitment after selecting an input. Verification and reward distribution is analogous to ITV. To reduce the cost for on-chain memory and computation, a sequential commit representation is advantageous in this and the following variant.
- *Optimistic (OPT)*: Since verification is costly and needs to be paid by the smart contract owner, the beacon operator and the users, another option is to add a *complaint phase* instead of carrying out the verification computation for every draw. In this case, an entity can submit evidence within a certain time frame to the smart contract that shows that the beacon value has not been computed correctly. Upon the successful verification of the evidence, this entity then receives part of the beacon funds currently stored in the contract, the beacon loses its funds and all users get reimbursed an equal fraction of the remaining beacon funds and their lottery fees. If the complaint phase expires without such evidence being presented, the value is assumed to be valid and the winner, owner and beacon operator are rewarded correspondingly.

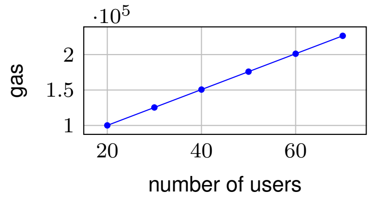

Figure 3. OFF gas consumption for different number of users.

The different variants as well as their advantages and disadvantages are summarized in Table I.

Implementation and Evaluation We have implemented an Ethereum smart contract for the OFF and ITV models (INP and OPT are very similar to ITV from an implementation point of view) on a private test network and analyzed the gas costs for the implementations. In both the cases, the smart contract uses an oracle service to obtain the necessary data from the beacon.

Figure 3 shows the gas costs for fetching the value from the beacon and drawing a winner for different number of users in the lottery, for the OFF model. As expected, we observe a linear increase of the gas cost with number of users.

The main implementation difference between the OFF and the other variants is the fact that in the later, the verification computation can be done on-chain. Thus we compute the gas requirements for the verification (modular squaring for sloth verification) in isolation. Using delay functions that require modular squaring for verification in smart contracts is discouraged [9], owing to the high gas consumption. But

	Delay Function on-chain	Delay Function off-chain, growing contract complexity			
Step \Version	**Full on-chain beacon**	**OFF**	**ITV**	**INP**	**OPT**
Preprocessing	-	-	Beacon locks fund and nonce on-chain (to be released if not enough users participate within a certain time frame)	Like ITV	Like ITV
User Input	On-chain, together with lottery fee payment	Off-chain	Off-chain	On-chain, together with lottery fee payment	Like INP
Beacon Commitment	Not necessary	Off-chain. After users see their input committed in time, they send lottery fee payment	Users send Merkle root obtained off-chain (with beacon signature on root and nonce) with lottery fee payment to contract	Commitment is stored on-chain, if delivered timely and including all inputs in the commitment, else users are refunded and beacon loses fund	Like INP
Beacon Computation	On-chain	Off-chain, independent of lottery	Off-chain, after commitment is stored on-chain.	Like ITV	Like ITV
Beacon Output	On-chain	Store beacon value on-chain	Like OFF	Like OFF	Like OFF
Post-processing	-	User verify beacon and complain off-chain, may decide to not trust this lottery in the future (no influence on outcome of this draw)	On-chain verification. If verification is unsuccessful, beacon forfeits funds and users get lottery fees back	Like ITV	If evidence submitted by user, on-chain verification, if successful, users receive beacon funds and lottery fees, beacon forfeits funds
Reward	Owner and winning user receive rewards	Owner and winning user receive rewards	Owner, beacon and winning user receive rewards	Like ITV	Like ITV
Pros	Users do not have to worry about verification	Simple to implement, low gas consumption	Beacon compensated for its service	Beacon compensated, user only needs to interact with the contract	Beacon compensated, verification only executed on chain if someone complains
Cons	Requires many users to offset the on-chain computation cost, simpler and cheaper solutions without delay functions are possible for this scenario	Owner and user must know and adhere to timing of beacon, trust stems from incentives to repeat lottery execution. Beacon operator remunerated off-chain.	User interacts with off-chain beacon operator and smart contract. All honest users submit the same data. Verification executed on-chain for every draw	Verification executed on-chain for every draw, even though the beacon would typically be incentivised to be honest in this scenario	User must execute verification off-chain fast enough to react within the complaint window

Table I. Lottery implementation options using the transparent randomness beacon.

the addition of a 'pre-compiled' contract to perform modular exponentiation as a part of EIP198 [10] significantly reduces the gas cost required to perform verification. The gas needed for modular exponentiation can be calculated based on the formula given in [10].

Table II shows the gas requirements for sloth verification performed for different sizes of witness, prime modulus and number of iterations. The values in the last row of the table show that for the largest evaluated witness and modulus sizes, the sloth verification cost amounts to around 5 times the cost for the rest of the smart contract with 70 users. For ITV and INP the lottery smart contract owner must set the participation fee high enough to be able to make a profit despite the verification cost. In the OPT variant, the verification computations are only executed on chain if someone submits a complaint. Thus with OPT, the owner can set a much lower participation fee as long as the locked funds by the beacon can cover the bounty and the computation cost of a successfully verified complaint.

Note that in addition to the increase due to sloth verification computation, the amount gas required for parsing, preprocessing and validating beacon inputs, commitments, output and proof parameters including their signatures on chain has to be considered. Parsing and preprocessing can be done in multiple ways (e.g., by making multiple calls to the oracle to obtain each value individually, or making a single call and parse the returned data on-chain, and so on). It also depends on how the beacon values are encoded when sent to the contract. In addition to this, the gas costs to use the oracle service depends on the amount of data fetched. However, this part of the gas cost is dominated by far by the verification cost, so we do not report on these numbers.

Size of Witness (bits)	Size of Prime Modulus (bits)	Iterations	Gas
512	512	1024	159,129
1024	1024	1024	517,171
512	512	2048	368,844
1024	1024	2048	1,198,745

Table II. Verification gas cost for different parameter sizes.

Discussion When using a blockchain to run (parts of) a randomness beacon, the incentive structure of all involved parties needs to be considered in a security analysis, which may include miners in public permissionless blockchains. E.g., for the trust assumption of everyone being against the user, he or she would have to mine blocks itself to guarantee interaction with the beacon, which is a steep requirement.

We also note that using smart contracts interacting with an off-chain beacon, a beacon can also be used on a deeper level of a distributed ledger, namely as a means to speed up consensus with shared randomness. If all the members in a distributed environment trust and agree on the random value generated by the beacon, it can be used to select leaders, committees and/or rank block proposals in an otherwise trust-lacking blockchain scenario. Recent consensus algorithms leverage this idea [11], [12] with MPC beacon generation. If and how a transparent authority beacon can be applied in this context is an interesting open question.

VII. Related Work

Randomness beacons have been studied intensively in the literature already, see [2], [3], [7], [8], [13], [14], [15], [16], [17], [18], [9] to list but a few example. An interesting example of a MPC protocol is *Drand* [8], which relies on a distributed randomness beacon daemon [8]. However, in contrast to our approach, Drand requires knowledge of all participating nodes at the setup phase (initiated by a single leader). This makes Drand static, i.e. new nodes cannot join an already running protocol. Another interesting solution is the "zoo approach", [2], a protocol reminiscent of a beacon which collects data from a variety of sources before running them through 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 *sloth* delay function is a also key part of our randomness beacon. However, the supporting structures driving the beacon are designed differently and we analyse the security of both the protocol and the beacon operator in more detail, in particular, we assume the beacon operator can be malicious. A *unicorn* protocol is then used to combine input collection from multiple sources and then compute the output of a delay function. This protocol resembles that of the *transparent authority* beacon computation model, and is done by a single entity. Lenstra and Wesolowski suggest feeding *sloth* with an aggregation of user inputs However, 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. A final protocol named *trx* is presented, which utilizes the output of the unicorn.

There exist other verifiable delay functions beyond sloth. Bünz et al. [9] evaluate the computation and verification of delay functions based on modular square roots and the hashing functions Keccak-256 (SHA3) and SHA-256. Subsequently, [19] formalized the notion and present functions that achieve an exponential gap between evaluation and verification time. Note that sloth could be replaced by these functions in our implementation and most likely achieve better performance. Since the focus of this paper is on more general system aspects, we omit an evaluation of these functions in this paper.

VIII. Conclusion

We designed, implemented and evaluated 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. To this end, we refined and extended the work in [2].Our implementation 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. Our beacon is attractive for applications based on smart contracts and distributed ledgers with minimal trust assumptions, illustrated with an Ethereum lottery application.

References

[1] M. O. Rabin, "Transaction protection by beacons," *Journal of Computer and System Sciences*, vol. 27, no. 2, pp. 256–267, 1983.

[2] A. K. Lenstra and B. Wesolowski, "A random zoo: sloth, unicorn, and trx." Cryptology ePrint Archive, Report 2015/366, 2015.

[3] National Institute of Standard and Technology, "NIST randomness beacon." Available online https://www.nist.gov/programs-projects/nist-randomness-beacon. Last accessed December 6, 2017.

[4] J. Huergo, "Nist removes cryptography algorithm from random number generator recommendations." NIST News announcement, 2014.

[5] N. Perlroth, "Government announces steps to restore confidence on encryption standards." Available online https://bits.blogs.nytimes.com/2013/09/10/government-announces-steps-to-restore-confidence-on-encryption-standards/, 2013. Last accessed June 6, 2018.

[6] N. Perlroth, J. Larson, and S. Shane, "N.S.A. able to foil basic safeguards of privacy on web." Available online http://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html, 2013. Last accessed June 6, 2018.

[7] I. Cascudo and B. David, "Scrape: Scalable randomness attested by public entities," *IACR Cryptology ePrint Archive*, vol. 2017, p. 216, 2017.

[8] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *Symposium on Security and Privacy*, pp. 444–460, 2017.

[9] B. Bünz, S. Goldfeder, and J. Bonneau, "Proofs-of-delay and randomness beacons in ethereum," *IEEE Security and Privacy on the blockchain (IEEE S&B)*, 2017.

[10] "Eip 198: Big integer modular exponentiation."

[11] T. Hanke, M. Movahedi, and D. Williams, "Dfinity technology overview series consensus system. whitepaper."

[12] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*, pp. 357–388, Springer, 2017.

[13] I. Bentov, A. Gabizon, and D. Zuckerman, "Bitcoin beacon," *CoRR*, vol. abs/1605.04559, 2016.

[14] M. J. Fischer, M. Iorga, and R. Peralta, "A public randomness service," in *Proceedings of the International Conference on Security and Cryptography*, pp. 434–438, 07 2011.

[15] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source," *IACR Cryptology ePrint Archive*, vol. 2015, p. 1015.

[16] T. Baignères, C. Delerablée, M. Finiasz, L. Goubin, T. Lepoint, and M. Rivain, "Trap me if you can — million dollar curve," *IACR Cryptology ePrint Archive*, vol. 2015, p. 1249, 2015.

[17] J. Clark and U. Hengartner, "On the use of financial data as a random beacon," *EVT/WOTE*, vol. 89, 2010.

[18] randao, "Randao: A dao working as rng of ethereum." Available online https://github.com/randao/randao. Last accessed December 11, 2017.

[19] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions.," *IACR Cryptology ePrint Archive*, vol. 2018, p. 601, 2018.