A Scalable Architecture for On-Demand, Untrusted Delivery of Entropy

Abstract

This paper presents a scalable software architecture enabling delivery of on-demand entropy by an untrusted third-party, henceforth the data carrier, without risks of front-running or tampering. The entropy is generated by one or more trusted, secure-element based devices featuring remote attestation.

The proposed scalable architecture accommodates the different hardware limitations of embedded Trusted Execution Environments (TEEs) and more modern trusted computing technologies as Intel SGX, and to easily adapt to different use contexts.

In particular, the use in the context of applications interacting with blockchain protocols and smart contracts had a primary role in shaping our design, but the applicability of the technology is not limited to this context only.

Introduction

One of the most compelling and unsolved problems in the context of blockchain protocols and smart contracts is the trustless generation of random numbers. Different methods, some of which will be surveyed later, have been proposed but fell short of their stated goal due to the complexity of the economic incentives infrastructure and the strictly deterministic nature of blockchain protocols.

The lack of a solution to this problem is limiting what smart contracts can accomplish today. Besides the obvious applications of random number generation to provably-fair gaming and gambling systems, there are many others: e.g efficient probabilistic payment channels, random ballots systems and possibly some yet to be invented. Unfortunately, a theoretical solution seems nowhere near of being discovered.

But perfection is the enemy of good. Smart contracts and blockchain protocols not only enable trustless applications, but also applications with clearly defined trusted boundaries.
As a data carrier, Oraclize is well familiar with the concept: any party interacting with a data-dependent smart contract must trust the data source. The aim of a service like Oraclize is to remove itself completely from the trust equation, by providing cryptographically binding data authenticity proofs, leveraging attestation technologies provided by widely recognized actors.
Breaking such proof would require cooperation between the data carrier and one of these actors which, it is assumed, have a lot of reputation at stake.

By applying this hindsight, the scope of this paper is to introduce a *practical* solution to the aforementioned problem, by presenting an architecture which enables smart contracts and blockchain applications to receive a feed of entropy from a trusted, super partes source, without having to rely whatsoever on the data carrier. The architecture is flexible and it can adapt to different types of devices and attestation technologies, therefore with different super partes sources, which can be even used together.
The paper also presents a set of guidelines which should be followed by smart contract developers to leverage the entropy feed without compromising the security of their works.

The applicability of the architecture is not limited to blockchain space only. Thanks to its flexibility, it can be used by any type of applications requiring a random input, while retaining the same security guarantees.

In fact, Random Number Generation (RNG) has always been a very important task for many real-life problems ranging from security and cryptography to traditional gaming and gambling applications, both in online and offline forms. Many different hardware

and software based methods have been used but in most cases the integrity of these systems relies on costly certification processes performed often by small or unknown parties which, ultimately, rely on trust[1].
These processes can fail badly and leave no accountability as it has happened more than once in the gambling industry [2] [3].
Oraclize's team believes that the model presented in the paper offers far higher, mathematically binding guarantees of fairness, thanks to open and third-party auditable authenticity proofs.

[1] "At some point, every certification of software relies on accepting the certified entity's testimony about some part of the system." Synopsys
[2] Lottery Worker rigged System to Win $14 million jackpot New York Post
[3] Russians Engineer a Brilliant Slot Machine Cheat—And Casinos Have No Fix Wired

Background

Terminology

In order to help the understanding of the reader, the next paragraphs will clear up the meaning of some terminology which is widely used in the paper.

Oracle Architecture

The Oracle Architecture is used by the data carrier, also known as oracle, to perform its functions. Three different actors can be identified within the architecture:

- *Data Source*: for the scope of the design, it is the actor which creates the data stream and makes it accessible to the data carrier: e.g kraken.com, weather.com, WolframAlpha etc. Data aggregators like Bloomberg are data sources themselves, since the aggregation is done internally and it based on trust. The data carrier may support different types of datasources.
- *Data Carrier:* it fetches the data and forwards it to the application along with a data-authenticity proof.
- *Data Consumer Application*: the final receiver of the data. In the case of blockchain protocols, it can be a smart contract or a Bitcoin script, but non-blockchain applications leveraging some of the properties offered by the authenticity proofs can be envisioned.

When the data consumer application needs a data piece from a data source, it creates a request to the data carrier. The request is called *query* and each datasource type has different parameters which can be passed as arguments.
The data carrier returns immediately a *queryId,* which is an unique identifier that can be used by the data consumer application to check the status of its query.
Once the data has been retrieved and the authenticity proof created, the query's status is updated and the data carrier execute the expected action: it can sends a transaction, called callback transaction, to a smart contract on Ethereum or Rootstock; create and send a transaction on Bitcoin; simple return the results along with the proofs as an HTTP response.
On Ethereum-based blockchain protocols, the query can be created automatically when the user send a transaction to the data consumer smart contract.
Upon receiving the transaction, the smart contract executes a call to the Oraclize's Connector contract, passing all the query arguments as parameters. The call immediately returns the *queryId.*
The data carrier continuously monitors the blockchain for such event, and it starts computing the query upon seeing one.

Oraclize's documentation has a more exhaustive explanation regarding its Oracle Architecture, which will be the topic of a dedicated paper soon.

Trusted Execution Environment

A Trusted Execution Environment (TEE)[4] is a computational environment strongly isolated from the main operating system running on a given device. The isolation is achieved through software and hardware enforced mechanisms.
A TEE runs a small-footprint operative system, which exposes a minimal interface to the main operative system running on the device in order to reduce the attack surface.
TEEs can run special applications with high-security requirements, such as cryptographic key management, biometric authentication, secure payment processing and DRM.
Some examples of TEEs are ARM Trustzone-based Secure Elements[5], widely used in smartphones, and the more recently released Intel Software Guard Extensions (SGX)[6] Enclaves.

Remote Attestation

Remote attestation[7] is a method by which a device authenticates its hardware and software configuration to a remote host. The goal of remote attestation is to enable a remote party to determine the level of trust in the integrity of the platform running on the device.
Remote attestation is often implemented by having the device produce a signed document, attesting the state of the system. Signing is performed by a special attesting key, embedded in the device during manufacturing and over which the third-party application running on the platform as no arbitrary control. The attesting key is known by the manufacturer of the device, which can therefore verify the signed integrity document and return the status of the platform to the party who initiated the attestation request.

Hardware Random Number Generator

A hardware random number generator (HWRNG) is a specially dedicated piece of circuitry which uses electrical environmental noises and manufacturing hardware defects to generate entropy. Cryptographic keys, extensively used in TEEs, require a high-quality source of randomness, which can be provided through a HWRNG.

[4] GlobalPlatform:Trusted Execution Environment
[5] ARM Trustzone
[6] Intel SGX
[7] Remote Attestation

Known Approaches

As a starting point for the discussion of the benefits of the architecture presented in the paper, the following section will survey previous works in generation or delivery of entropy to smart contracts and blockchain-based applications, in particular on Ethereum.

Using the Block Hash as Source of Entropy

Ethereum is a generalized state machine public blockchain with a global state. Smart contracts are constituted of a number of instructions taken from a predefined set, known as EVM Assembly[8]. When a smart contract is involved in a transaction, its instructions are translated into updates to the global state by the execution of the Ethereum Virtual Machine.
Since all nodes must converge to the same final state after each transaction, the execution is strictly deterministic. The only residual non-determinism accessible from within the EVM is the block hash of previous blocks, retrievable through the specific BLOCKHASH instruction, and some additional informations regarding the current block, such as the timestamp.

A simple example of an end-user application that could use the block hash as a source of randomness is a decentralized lottery: a smart contract could be in charge of selling tickets, collecting the funds and awarding the winner based on the block hash at a specific blockchain height.
Unfortunately, the previous block hash is a poor source of randomness, since the miner of such a block could decide to discard it and try their luck by mining a second block at the same height. If unsuccessful, by doing so, it will incur in the loss of the block reward. In Ethereum, this loss is reduced by up to 87.5% because of the uncle incentivization mechanism[9]: a child-less block, called uncle, included in the heaviest chain, is rewarded but the transactions within the block are not executed and therefore have no effect on the global state. A miner would then act rationally by attempting an attack to lottery if the probability P of winning times the expected payout E would be higher than the difference between the full and reduced block reward R:

$$R - \frac{7}{8} * R < P * E$$

To be consistently successful, the attacks require a miner or a coordinated group of miners to have a significant share of the total network hashrate. The higher the expected payout, the greater are the incentives to reach the level of required coordination, limiting the possibility of having large prizes for on-chain lottery or other types of entropy-based games through the block hash method.

[8] Ethereum Yellow Paper
[9] Yellow Paper Page..

The calculation above doesn't even include consideration regarding selfish mining, which further reduces the cost of a successful attack. [10]

Commit and Reveal Schemes

An alternative method to generate entropy in a deterministic state machine, such as Ethereum's, is to delegate it to an interactive protocol involving many economically incentivized anonymous users. These type of protocols are generally known as commit and reveal schemes and they are composed of three phases:

- **Commit:** Each of the involved users sends to a delegated smart contract a cryptographically secure hash of a bytes sequence of his choice, which is saved in storage. To incentivize pre-image reveals and continuation of the protocol, users are forced to send a deposit which can be redeemed at reveal time.
- **Reveal:** After a predefined number of blocks have been mined, the users can reveal the original bytes by sending another transaction to the smart contract. The bytes sequence is validated against the initial digest.
- **Generation**: All the users values are taken as inputs for a function, whose output is the requested number

This approach has been implemented in the RandDAO smart contract but it suffers of a number of drawbacks. In particular, it is expensive to manage and to run. The participating users must incur in gas costs and opportunity costs, reducing the number of users willing to participate. To date, the RandDAO project is not yet used in any production smart contracts based application[11].

Lately, two-party partial commit and reveal scheme have gained popularity.[12] [13] In this type of schemes, the consumer application operator generates and commits to the hash of a random seed before receiving a second random seed from the user and generating the resulting number from both. This solution resolves some of the costs of the decentralized approach, but it has some additional drawbacks:

- Constant online key management for publishing the hash of random seed by the smart contract operator can be dangerous
- Additional hassle for the user to generate client side, safely, a random number. If the generation is delegated to client-side UI crafted by the application developer, then the user is trusting that the code is not compromised. Even delegating it to external services open the user to a potential collusion attack between the external service and the smart contract operator.
- It doesn't adapt well to wide-spread use cases: for example, most smart contract games are backed by investors. If the casino operator is acting as one of the two party in the scheme, then he could rig the fairness of the game at their expenses.

[10] Selfish Mining
[11] RandDAO
[12] Edgeless Whitepaper
[13] Random Number Generator on Winsome.io

Fetching through an Oracle from Third-Party Services

Another way to access entropy from within Ethereum, it is to fetch it from dedicated, trusted third-party services through an oracle service. For example, random.org provides cryptographically signed random numbers, but as of today the RSA-based signature is not verifiable directly from within an Ethereum smart contract.
By carefully designing the smart contract, it is possible to reduce the attack surface available to the oracle service against it. Oracle services such as Oraclize offer additional guarantees that the data is correct through HTTPS-based authenticity proofs, but today the verification has to be done off-chain and *ex-post*.
Authenticity proofs make it easier to detect occurrences of cheating, but do not provide the ability to prevent it.
If the expected payout is extremely large, only reputational damage can stop the oracle service from cheating, which is a poor security guarantee for a smart contract system. Moreover, the third party service could cheat without any consequences or risk of being discovered.

Architecture

Overview

The architecture presented in the paper will require to use one or more physical devices featuring: a Trusted Execution Environment (TEE); a cryptographic-quality hardware-based Random Number Generator (RNG); support for remote attestation of the device and local attestation of code execution, which will be needed to generate the authenticity proofs. If these requirements are fulfilled, the final user of the data consumer application, will have to trust only the following parties:

- The trusted execution environment designer who could easily introduce a backdoor or create a flawed design, hampering the integrity guarantees of the TEE.
- The chip manufacturer who can introduce a physical backdoor during the fabrication process.
- The trusted execution environment firmware developer: the firmware could be backdoored or be exploitable by third parties due to an implementation flaw. The vast majority of TEEs have closed-source firmware and require extensive NDAs to be signed with the chip producers, so it is not possible to audit and scrutinize their firmware.

In the large majority of cases, the aforementioned roles can be overlapping and therefore reduced to two or even only one trusted party. This is the case with Intel SGX, where the designer, the manufacturer and the firmware developer are the same actor; namely Intel.

The data consumer application can create a query by: accessing the data carrier's HTTP APIs or, in the case of smart contracts running on Ethereum-based blockchain protocols, by a transaction executing a call to the Oracle's Connector.
Each data source has its own query parameters. This data source requires the following:

- A 32-byte long *commitmentNonce*, which it should be a piece of information not accessible by the data carrier before the creation of the query.
- A time *dT*, expressed in seconds, which is the minimal time which has to elapse before the query can be served by the secure application and the random bytes requested returned.
- A public key *deviceSessionPubKey*, which is the public part of the *deviceSessionKeyPair* generated by the secure application and which uniquely identifies a device.
- A number *nRandomBytes* between 1 and 32, which is the number of random bytes to be returned to the application.

Upon reception of the query, the data carrier will issue a request for the random data to the secure application running on the TEE-enabled device.

Implementation Details

As mentioned before, the scope of the paper is to present a scalable architecture that can run both on embedded devices and more powerful trusted computing platforms.
The first device used to implement the architecture is the Ledger Nano S, a cryptocurrency hardware wallet designed and assembled by Ledger Co.
Ledger's devices are interesting because they run an operative system called BOLOS which, unlike many others TEE operative systems, supports third-party developed applications, remote attestation and code attestation.[14]
BOLOS is developed internally by Ledger and it is partially closed-source, mainly due to the many NDAs required by hardware manufacturer.
The Nano S uses in fact a ST31M chip as the secure element, a chip developed and produced by STMicroelectronics. Therefore the user of the data consumer application has to trust that neither Ledger nor STMicroelectronics collaborate with the data carrier with the aim of rigging the random number generation.

The relevant limitations of the Ledger Nano S specifications to the architecture design are:

- 4 Kb of usable volatile memory
- 256 byte message size
- Up to 256 Kb of usable non-volatile memory
- The lifetime of the NVM is 500,000 writes for each 64-byte sector. Currently the device's firmware doesn't support wear leveling, so the secure application implements a simple one.

On the base of the these specifications, the secure application should be engineered to minimize memory footprint, reduce the number of writes, the storage occupied by the persistent state and reduce the number of messages exchanged with the host.
The application code should be open-sourced to enable third-party auditability of its functions and independent verification of the binary hash present in the device code attestation.

Secure Application

The secure application can be decomposed into three functional components:

- A timer, incremented during application execution, so that we can guarantee that the time *dT* specified by the query has elapsed.
- A tamper-resistant, persistent state used to hold: the private key of an elliptic curve key pair, generated at application initialization; an append-only data-structure for storing the queries; the last value of the timer.

[14] Proving Code Attestation on the Ledger Platform

- An asymmetric key pair management component with signing and verification capability, used for authenticating any piece of data exported from the device.

In the actual implementation, only the private key of the *deviceSessionKeyPair* and the nonce *storageNonce* will be saved to the non-volatile memory, while all other values are kept only in the memory state. The *storageNonce* is a monotonic counter incremented every time the volatile memory state is exported.

The application exposes four functions which can be called by the data carrier to produce the result and the authenticity proof:

- Initialization
- Query Insertion
- Volatile Memory State Export and Import
- Query Execution

Initialization

Once the application is installed on the device, the first step to be completed is the initialization.
The initialization function creates an elliptic curve key pair, called *application session key pair*, bound to the current application code and to the device's present state: any firmware update, device resets and application reinstalls would wipe the session key as well. It also sets the *storageNonce* to zero and starts the timer, which on the Ledger Nano S is incremented every 10th of second.

The initialization function returns to the host the attestation chain which certifies the *application session key pair* and the secure application binary hash with a known root-of-trust device key.

In the case of the Ledger Nano S, the application session public key and two signatures are returned:

- A signature over the session public key and the application *codehash* made by the *endorsement_app_key1*. This key is generated on the device at initialization phase and it is attested by Ledger.
- A signature made by the Ledger Root Attesting Key, over the *endorsment_app_key1*.

Once verified that the signing keys are not revoked by Ledger and that the binary hash computed from the compiled application source-code matches the signed code hash one, valid signatures prove that the application is running in the secure environment as expected.

Query Insertion

This function, exposed by the secure application to the host, inserts a new query in the tamper-resistant state.
In order to avoid cheating or unfair behavior from the data carrier, it must be enforced that only one query for a determined *queryId* can exist. Any additional query with the same query*Id*, no matter if including different *commitmentNonces* or *dTs*, will be rejected by the application and won't be appended to the state.

This effectively makes the data structure held within the tamper-resistant state, a table of explicit commitments of the type:

queryId -> Commit(commitementNonce, dT, nRandomBytes)

That's it, for each id only one query can be served and only after *deltaT* time has passed. Each *queryId* is also implicitly committed to a specific *deviceSessionPubKey*, because the returned data will be signed by the corresponding private key. If the signature verification fails, the data must be rejected by the data consumer application.

The data to be stored can be therefore represented by a *key:value* pair of the form:

H(queryId) : H(currentTimer, commitmentNonce, dT, nRandomBytes)

where H is a cryptographic hash function and *currentTimer* is the value of the secure application timer at insertion.

Clearly, with the limited persistent storage available in the trusted execution environment it is impossible to store a state increasing linearly with the number of queries. But thanks to cryptographically authenticated data structures [15], most of the state can be kept off the device. The secure application assumes the role of verifier and keeps only a small digest in memory. The role of the prover is taken by the host, the physical machine in the Oracle Architecture connected to the device, which keeps the entire device state in a *key:value* database.

Every computation or operation performed on the data requires the prover to send the result along with a proof of correct computation, which can be used by the verifier to validate the result and eventually update the digest kept in memory for insertion or deletion operations.

For simplicity of implementation and as a starting point, the secure Ledger application uses a modified trie[16] as an authenticated data structure. The inspiration was taken from

[15] Authenticated Data Structures
[16] Trie

the Merkle Patricia Trie used by Ethereum, with some modification to reduce the complexity of the implementation. To summarize the properties of the trie implemented:

- The key is fixed length (64-chars) and it is the hex representation of the *H(queryId).*
- It's a nibble-wise trie, where a nibble is an half-byte, so one char of an hexadecimal string. Each node has 16 entries, as 16 are the entries in the hexadecimal alphabet. Traversing the trie would mean testing each subsequent nibble in the key and choosing `child[0] .. child[15]` appropriately until the item is found.
- The node's entries contain the hash of next node required to continue the search.
- The leaf node, containing the hash of the values, is reached by following the key till the end.

Therefore this trie can be called a Merkle Nibble-wise Trie. Thanks to this data structure, the application only needs to keep two pieces of data in the volatile memory at runtime:

- The relative timer, which is incremented every second the application is running.
- The hash of the trie root node, also known as *roothash.*

Initially, the trie is initialized with empty nodes and the roothash of such trie is hard-coded in the application code.

The insertion of query starts by computing the hash of the parameters values and creating the leaf node. Then, for each node traversed by reversing the key:

- The current node hash is saved in a temporary variable
- The hash at the i-th *decimal(key[i])* entry is replaced with the child hash
- The new hash of the current node is computed and saved in a temp variable used in the next iteration.

If once reached the root node, the current node hash and the roothash diverge, the insertion is aborted; if they match, the root hash is updated with the new value and the insertion is completed.

Memory State Export and Import

As the device could serve more than one application or simply be restarted, the secure application requires functions which can be used to export and import the volatile memory state. To guarantee its integrity, the state is signed with the device session private key*:* only states with a verified signature made by such key can be imported. The freshness of the memory state is ensured by including the last value of the *storageNonce* in the signature.

When the application reloads, the signature is verified and the *storageNonce* must match the internal value so that the data carrier cannot import memory states older than the last one.

Query Execution

A query can be executed only after its time *dT* has elapsed. The process of executing a query is simple: the secure application signs with the ECDSA deterministic *k* signature algorithm over the hash of the query parameters with the application session private key.
This signing algorithm always produces the same signature for a determined message, in this case the hash of the query parameters, while the regular ECDSA signing algorithm requires a random number *k* to be generated for each signing operation: the signature obtained is different every time.
The *k* used in the ECDSA deterministic algorithm is instead computed from the message by applying an HMAC construction using the private key as an HMAC key. The signature bytes are not determinable *a priori* from the inputs, if not with previous knowledge of the private key which is inaccessible to the data carrier. The requested *nRandomBytes* are the truncated hash of the signature, while the signature itself is returned by the data carrier as part of the authenticity proof.

Thanks to the ECDSA deterministic signing algorithm[17], executing a specific query will always return the same value.

[17] Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)

Applications

The architecture presented in the paper will be offered as a new type of data source offered by the Oraclize's Oracle Architecture: the *random datasource*.

Random Datasource on Ethereum

A smart contract using the random datasource will include, in addition to the Oraclize's API, the following code:

```
oraclize_query("random", commitmentNonce, deviceSessionPubKey, dT,
                                nRandomBytes);
```

where the *commitmentNonce* recommended value is the hash of the current block coinbase, timestamp, gas limit and previous block hash. The next session, dedicated to security consideration, will explain the reason.
All parameters must be saved and stored by the smart contract to verify the authenticity proof sends with the result.

The smart contract developer can decide to either commit all the queries to the same session public key or instead to commit each query to a different key among the ones available in the Connector. In case of high-volume contracts the second option should guarantee faster response time, but at the cost of being more expensive since each query must be linked to a session key in the smart contract storage.

Since devices can become unavailable or new versions of the application can be released, the data carrier can add new session keys to the Connector. The process of adding an application session key requires the verification of its attestation chain and that the codehash matches a list of on-chain whitelisted codehashes. Ultimately, the codehashes should be verified off-chain against the open-source secure application code.

Discussion

Security Considerations and Guidelines

This section will addresses a series of attacks that the data carrier and external collaborating actors could attempt against a smart contract using the *random datasource*.

Front-Running by the Data Carrier

Once the time *dT* has elapsed and the query has been executed, the data carrier can know the result before sending it on-chain.
The data consumer application developer should make sure that having knowledge of the generated entropy before it's public, cannot affect the fairness of the application. This is easily enforceable by making it impossible for the data carrier to act on behalf of this knowledge. For example, in the case of a lottery, the application should close the sale of tickets before the query can be executed by the data carrier.

Selective Publishing

The data carrier could selectively publish only results which are in his favour. The smart contract should protect against this scenario: for example, in the case of a betting game, by not returning the amount wagered to the player when the data carrier doesn't publish the result.

Computation of a Query before Time

In a similar way, the data carrier could try to precompute the query using the last block hash and guess the timestamp, coinbase and the gas limit parameters of the next block, and only participate if the result is in his favour. Without collusion of a miner, the probability of success is low because only one try can be done: in the secure application state there can be only one *queryId*. The probability can be reduced to negligible by setting an appropriate minimal elapse time higher than the current time between blocks. Even if a winning result is found, the authenticity proof will be invalid, because it doesn't include the correct block parameters.

Generating Multiple Values

The secure application allows the insertions of only one query for a specific *queryId* in the application state: therefore, the data carrier cannot arbitrarily add queries with the same *queryId* trying to generate more than one result.
Since the execution of a query is a deterministic signing process, the data carrier cannot even alter the results by repeatedly asking for new signatures over the same data, as it would be possible with the standard, non-deterministic ECDSA algorithm.

It is therefore impossible for the data carrier to generate more than one output for a given *queryId*, and then select which output to send to the data consumer's application. The same *queryId* and parameters could be inserted on more than device, but the smart contract will accept only the authenticity proof signed by the session key to which the specific query has committed.

Tampering of the Result by Miners

As outlined before, the miners can tamper with the fairness of a smart contract, which uses the block hash as a form of randomness, by dropping a mined block if the result is not in their favour. The architecture presented here enables application developers to solve the problem, since block parameters are not the only element which defines the random bytes result.

Data Carrier and Miners Interactions

There are two potential scenarios involving interactions between the data carrier and miner to affect the outcome of the generation:

- The data carrier broadcasting node is unknowingly surrounded by malicious nodes and the transaction returning the data is not broadcast to the whole network. The miner can then try to mine a block to verify that the result is in their favour, and it will only make the block public if it's so.
- The miner can directly bribe the data carrier to obtain the result before the transaction is broadcasted.

Both attacks can be avoided thanks to the minimal elapsed time *dT* and *queryId* commitment to various block information. By having the *queryId* commit to a specific set of information of a block at a given height, such as the coinbase, the timestamp and the gas limit, the *queryId* is implicitly committed to the blockchain history which includes that block.
The minimal elapsed time which has to pass before the query is executed, makes it exponentially more costly to the miner to try these attacks, as the probability of successfully publishing a chain heavier than the most of the network becomes exponentially lower as time passes. The final application developers should take this in consideration when writing smart contracts and pass as a query parameter a *dT* proportional to the amount which is at stake. Future iterations of the paper will include a quantitative analysis, in order to help developers to set an appropriate time.

Denial of Service

The presented architecture doesn't aim to protect the data consumer application from denial of service, intentional or unintentional, by the data carrier.

An application developer may be tempted to resolve the issue by letting the users recollect the amount wagered after a predefined time has elapsed without an action by the data carrier: here be dragons; the data carrier would be incentivized to place bets and only publish on-chain transactions containing a winning result, knowing that he could collect back the amount wagered of the unpublished, losing results.
The application developer could decide to resolve it either implicitly, by not setting a time limit and de facto locking the amount wagered until the data carrier provides a callback transaction, or by actively making the amount wagered not collectable, for example sending it to a burn address.

Ransoming a Winning Player

The data carrier can calculate the outcome of a result before sending it to the smart contract. Potentially, in the case of a lottery, he could ask for a ransom from the winning player to publish the result. The smart contract developer can defend from this attack by allowing anyone to execute the callback function: since only valid result will be executed, there is no need to limit the access only to the data carrier. This makes the authenticity proof fully transferable and the data carrier would have no way of ransoming the winning player without the facto publishing the proof.

Physical Attacks Against the Device

Our architecture bases its security guarantees on the existence of a tamper-resistant state internal to the TEE of the device. Physical attacks against hardware devices are always possible, but for this class of devices they are extremely costly, ranging from many hundreds of thousands dollars to millions, and without success guarantee.
If the consumer application is extremely successful and it is handling an expected payout much larger than the figure reported here, the application developer should ask for more than a single dedicated device, which should linearly increase the cost of attack, and more than one type of device, when available, to increase the security guarantees.
Using more than one source of randomness must be carefully planned to avoid exposing the application to additional attacks, as for example dependency from transaction ordering. The application developers should remember than the transaction ordering within a block is selected by miners.

Conclusion

In this paper we have presented a scalable architecture for untrusted third-party data delivery of trusted generated entropy.
The architecture leverages trusted computing technologies and remote attestation of code execution to run in a secure, tamper-resistant environment an application whose source is published and third-party auditable.
The secure application is used to update in a defined manner a tamper-resistant state, in a way to guarantee that each *queryId* is unique, and to generate the requested random bytes by deriving them from a private key held within the inaccessible storage. The receiving party, which can be a smart contract, possesses the associated public key and can verify, thanks to the accompanying authenticity proof, the origin and the nature of the random bytes sequence before accepting it.

The architecture has been designed to protect the data consumer application of the generated entropy from a number of attacks the data carrier could attempt. Additionally, the paper also suggests a set of good security practices which data consumer applications should follow in order to protect the fairness of any draw when using the presented solution.

According to their own threat model and to the stake, application developers will be able to set the minimal elapsed time whose passing is enforced by the secure application, before the random bytes are returned.