Reputation Network technical notes

Traity

Abstract

This document describes the technical specification of the Reputation Network. The design principles are introduced to understand the resulting architecture, and the responsibilities of the different players are specified. A sample scenario is also described in order to enable the reader understand the underlying protocol.

Contents

1 Introduction

At the present time, consumers lack control and transparency when dealing with their credit/risk scores. This is mostly a result of the reduced number of players in the field. The Reputation Network is a blockchain-based protocol that provides means for consumers to safely use apps that process their personal data. It is mostly focussed on reputational data that can be transformed into risk/credit scores, but it could be used for a wide variety of different applications.

In a nutshell, the Reputation Network enables a person (the *subject*) to use a service (the *client*) that requests some data processing to third-party services (the Reputation Network *apps*) in a safe environment thanks to the underlying protocol, which involves a trusted third-party, the *verifier*, and keeps a history on *blockchain*.

A brief description of the involved actors is:

- Subject: the person whose personal data will be processed. Usually, the subject will be a person who wants to get access to some product which requires some data processing. E.g., a mortgage might require the subject's credit score, which implies processing some personal data to compute it.

- App: an automated service in the Reputation Network which either provides personal data and/or processes personal data. E.g., an e-commerce could be an app that provides purchase data about their users. Also, there could be an app that processes these purchase data from different sources and produces a credit score. As seen, Reputation Network apps can be composed to feed and reuse each other.

- Client: someone or something (i.e., a person or an automated process) that uses a Reputation Network app. E.g., a bank employee might use a scoring app to know the credit score of a person. This app could also be used by some automated process. And this app might be the client of another apps, which provide personal data about the subject.

- Verifier: an automated middleman which verifies that Reputation Network

apps run as expected. It mostly prevents data leaks in an app by ensuring the output complies with a previously agreed schema. It is an actor that is trusted by both the subject and the involved apps.

- Blockchain: the decentralized ledger which keeps record of personal data. It enables the subject to review who has used some data and when. It also enables rewarding the apps and verifiers for their work, while making sure some rules are followed.

The Reputation Network can be used by banks to use next-generation credit scores that safely use personal data from different data sources. It can also be used simply as a safer alternative to OAuth. It can be used to perform simple checks on personal data, like income checks based on credit card data, without letting the client know the subject's actual monthly income. Potentially, it can deal with identity data to do age checks without revealing the subject's age (as Civic can do), but the Reputation Network is better suited to continuosly changing data that needs to be retrieved from external data sources.

2 Assumptions

2.1 Vision

Decentralized technologies have allowed breakthroughs like decentralized money with Bitcoin, or decentralized storage with IPFS. Of course, decentralization often comes at a cost. E.g., Bitcoin consumes power and has a limited number of transactions per second that the platform can process, while IPFS file persistency depends on file demand, which limits certain applications.

Also, private companies sometimes have limited incentives to decentralize their infrastructures. Taking Amazon as an example, they could easily allow users to store their delivery addresses and payment information locally, instead of keeping them on Amazon servers, and also use key pair credentials to log in, to avoid Amazon from storing passwords (even if they are hashed). Additionally, Amazon keeps browse and

purchase history to recommend other products. Decentralizing the recommender system is not an obvious task that would be challenging for them as a company, and which would only have a return of investment for their users' privacy concerns.

Similarly, decentralizing social networks is equally challenging. Again, the main players have little incentive to attempt such technical work, which would be especially hard in the case of private social networks such as Facebook. Alternatives such as Diaspora are not fully decentralized, and its iniciative has not proven to be a sufficient incentive for the most users to switch.

Additionally, incumbents have intellectual property which might be jeopardized by switching to fully decentralized architectures. Examples are ranking algorithms in search engines, ad networks, recommender systems, and others. A credit scoring company would not Open Source their algorithm, even if they have exclusive access to the data that enabled the training of such algorithm.

As a result, we believe data silos will stay in the medium term. Incumbents have little incentive to get rid of the control they have over data and business logic. Additionally, it has some benefits such as speed of development and ease of management. Also, the average user is comfortable with having their data in the cloud, both because of not having to deal with data management, and also because of inherent trust in organizations, which enable functionalities such as data deletion and "forgot my password" to be taken as accountabilities by the companies.

For example, if a person passes away, a relative can request Facebook to delete or hide this person's account, but this would be tough in a decentralized solution if this situation was not anticipated in its smart contract. Of course, the opposite is true as well, and companies can take decisions which users might not like, and there is no smart contract all parties have accepted.

Despite several data leaks and abuse from companies in the past, most consumers accept certain degree of centralization, and this might take some time to change. Until this change happens, data silos will be there, with consumers trusting different providers for keeping their data and executing their different rights.

Still, decentralized technologies can be of help to ensure many of these rights in

order to make consumers safer and educate them on the benefits of being in control of their data.

2.2 Federated Internet

The Reputation Network accepts the fact that consumers will have their data scattered across different data providers. It is a common assumption in decentralized solutions to try to change this fact by providing decentralized storage solutions. As mentioned before, it can be a challenge to change incumbents' infrastructure and data practices.

The Reputation Network is fine with both decentralized and centralized storage approaches by assuming any storage can count as well as yet another data provider. Therefore, personal data is assumed to be scattered across a federated network of data providers with heterogeneous technologies and ownership.

In the Reputation Network, these data providers can be given access by the subject, and their data can be processed by other systems, again implemented by different organizations and technologies. Both data providers and data processors can be seen as Reputation Network apps, as introduced previously.

2.3 Trust matrix

Typically, a scenario in the Reputation Network will consist of, at least:

- a subject, whose personal data will be involved,
- a client, who wants to use a Reputation Network app,
- a processor app, which could be a credit score that processes the user's personal data, and
- a data app, which could be some site used by the user which has the personal data the processor app would like to process.

↱ trusts →	Subject	Client	Processor app	Data app
Subject	N.A.	✗	✗	✓
Client	✗	N.A.	✓	✗
Processor app	✗	✗	N.A.	✓
Data app	✗	✗	✗	N.A.

Table 1: Trust matrix

Table 1 summarises the trust relationships among the different actors in the system.

More specifically:

- The subject trusts the data app, which will typically be a web site where they have signed up and has (for better or worse) accepted the data app's terms and conditions to have their data managed. On the other hand, the subject does not even know the processor app, and very likely does not trust it nor the client for managing their data.

- The client does not need to trust the subject (that might actually be the reason why the client is looking for a credit score or some other data processing). It trusts the processor app to provide the deserved processing. It also does not even know about the existence of the data app, and therefore does not need to trust it in any way.

- The processor app trust the data app, as it is a dependency for it to provide the result to the client and get rewarded for it. The processor app is simply a client of the data app. The processor app does not need to know about the subject, and also not about the client, nor trust any of them.

- The data app has an accountability about the subject's data security and does not necessarily trust anyone in the system.

Although it has been assumed one processor app and one data app in table 1, there could be many of them, or a combination (i.e., an app that provides data while doing a processing as well). A data app is essentially a Reputation Network app which requires knowing the user's identity to provide its service (i.e., returning some data of the subject), while a processor app simply processes data provided by some data app upon the consent of the subject. There could be a pipeline of processor apps that are combined before a final result leaves the Reputation Network and is given to the client.

As mentioned, we will see that two elements are included as part of the Reputation Network: the verifier and blockchain's smart contract. A smart contract is trustless, in the sense that its own architecture guarantees that its execution can be trusted by every party. The verifier acts as a common trusted party between the apps, the subject, and the client. It is not trustless by design, but there could be potentially many available verifiers in the system to choose from. As its name suggest, the verifier's purpose is to control other parties, so it does not trust anyone else in the system, and, like apps, would want a reward for its work.

2.4 User control

Another principle in the design of the Reputation Network is the full control of consumers about any action affecting their data. Some credit scoring systems are based on data bureaus built without explicit user consent.

But this has implications that go beyond accepting sharing some data from one data source. As seen in scandals such as Cambridge Analytica's, consumers might accept sharing data, but the purpose needs to be respected during the whole lifetime of the data.

As a result, the Reputation Network requires the subject to give permissions to the different actors to use their data. This permissions can only be active for one specific session, and it needs to be ensured that the data have been used for the specific purpose that the permission was given for.

2.5 Design principles

As a result from the previous discussions, here is the full list of principles that guide the design of the Reputation Network:

1. Consumers have personal data scattered across several web sites.
2. Web sites have no incentive to decentralize their infrastructure or data.
3. Web sites are accountable for properly managing consumers' personal data.
4. Consumers trusts web sites to properly manage their data and execute the usual rights on their data, such as right to deletion.
5. Personal data must only be used with explicit permission from the subject.
6. Personal data must only be used for the specific purpose the permission was given for.
7. Clients trust the Reputation Network apps they directly use, as any other service they would pay for.
8. Clients do not necessarily trust the subject nor apps they are not directly using.
9. Apps do not necessarily trust other Reputation Network apps they are providing data to.
10. Consumers, clients, and Reputation Network apps trust a previously agreed verifier to run apps properly.
11. The verifier does not trust any of the parties in the system.
12. A smart contract in blockchain is trusted by all parties in the system.
13. Reputation Network apps and verifiers might want to be rewarded for their work.
14. Good data usage practices must be guaranteed whenever possible, with blame clearly pointed upon bad practices otherwise.

3 Sample interaction

A sample interaction is outlined in this section to highlight the different actors in the system and their roles. The scenario consists of a subject, Alice, who will give permission to a client (a bank) to access a scoring app, which will process some personal data of Alice from two data apps to build her score.

It is assumed that Alice has some personal data in two web sites. They can be social networks or e-commerce web sites, which have valuable information to build a credit score. These web sites control the two data apps, which simply expose some of her data to be usable inside the Reputation Network.

3.1 User experience

The expected user experience is as follows:

1. Alice might want to purchase a financial service in her bank. For example, let's say she is interested in getting a mortgage, so she logs in to her bank and clicks on the mortgages section of her dashboard.

2. Her bank knows not much from her, as she turns out to be a relatively new customer with no track record. Therefore, the bank wants to query a scoring app in order to know more about the ability to pay of Alice.

3. The bank's dashboard shows her a dialog similar to the one in figure 2, which asks for permission to use a Reputation Network app. It also highlights where the scoring app will get its data.

4. Once Alice gives permission, the bank will use the Reputation Network to safely query the scoring app and get Alice's score.

5. Hopefully, Alice will get her mortgage thanks to the alternative credit scoring that her bank boasts.

Alice, do you give permission to Your Bank to use Scoring App? Scoring App will get access to Data App 1 and Data App 2.	
Accept	Decline

Table 2: Permission dialog shown to Alice before getting a mortgage

3.2 Permissions flow

The permissions flow in this scenario is shown in figure 1. The figure shows the following steps:

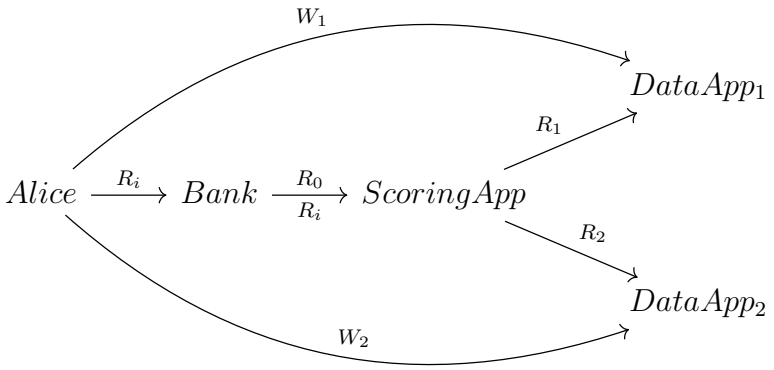

Figure 1: Permissions flow

1. Alice gives write permissions W_i to the two data apps she's registered in. This essentially provides Alice's public key so that the data apps can return Alice's data upon some other app's request.[1]

2. Alice gives three read permissions to the bank, at the bank's request. The read permissions are R_0, which enables the bank to query the scoring app about

[1]As will be discussed, knowing the public key is not enough for an app to make statements about the subject. The smart contract will need to validate that the write permission matches the right app and another corresponding read permission.

Alice's data, R_1, which enables the scoring app to query the first data app about Alice's data, and R_2, which enables the scoring app to query the second data app about Alice's data.

3. The bank then uses read permission R_0 to query the scoring app, and also hands over the remaining permissions to the scoring app, so that it can do its job.

4. The scoring app uses read permissions R_1 and R_2 to retrieve Alice's data from the two data apps.

The idea could just be that once apps have the right permission, they can return the requested data (or the result of processing the data) to their respective clients. This simple approach would leave many open questions and risks that the reader might have already noticed:

- The apps might not validate that the permissions have been built by Alice. It could be assumed that the data apps might be validating the read permission, as they're guarantors of Alice's data (according to section 2.5) but, still, it should not be their job. The scoring app has no incentive in validating that the read permission has been built by Alice.

- It is not defined how the apps would be rewarded. Also, a data app might not know about Alice at all, or might have data but not the write permission from her. The scoring app might want to reward the data app per hit, not miss. Similarly, the bank might not want to reward the scoring app if no data app has provided data about Alice (and hence the scoring app cannot produce a decent score).

- Alice expects that the scoring app just returns a score to the bank. However, the scoring app could leak all the data it is getting from the data apps to the bank. Actually, the bank might be glad to receive such extra information, which is against Alice's will.

To overcome these challenges, the smart contract and the verifier come into play. The smart contract offers a way to reward apps while validating permissions in a trustless way. The verifier makes sure there are no leaks in the data that is returned by the data apps to the bank, so that the scoring app returns a previously agreed schema.

3.3 Verifier interaction

As mentioned in the previous section, the permissions flow will not be exactly the one that is shown in figure 1. Logically, permissions and data follow the flow outlined in such figure, but physically the behaviour is a bit different, as the verifier acts as proxy on apps. This is shown in figure 2.

$$Client \underset{response}{\overset{request}{\rightleftarrows}} Verifier \underset{response}{\overset{request}{\rightleftarrows}} App$$

Figure 2: Verifier flow

The verifier checks that the app's response complies with the schema specified in Alice' permission to prevent unexpected data leaks. The Reputation Network can have many available verifiers. The verifier to be used is proposed by the subject, and implicitly accepted by the rest of parties by using it.

As will be seen in the next section, the verifier gets rewarded for its task with a commission of the app's reward. Therefore, the verifier has the incentive to make sure the request is well built, so that the smart contract accepts the transaction and rewards both the app and the verifier.

3.4 Smart contract interaction

The smart contract is in charge of handling the payments whenever permissions match. It uses state channels, so the steps that the scoring app needs to perform a transaction with one of the data apps is broadly as follows:

1. The scoring app funds the channel it has with the data app. This can be done just once for many transactions.

2. The data app posts the transaction to the smart contract to cash out the corresponding part of the funds.

The transaction that is posted to the smart contract to cash out has two parts: the request and the proof. The request is mostly the intent of the scoring app to use the data app, which needs to include the read permission that Alice built.

The proof contains Alice's write permission that the data app owns. This way, the score would only pay for hits, not misses. This proof can be used by the scoring app to cash out the corresponding channel that goes from the bank to the scoring app.

3.5 Summary and discussion

Figure 3 summarises the interaction between all elements. Numbers show the order of the interaction. Bear in mind that the interactions of the scoring app with the two data apps can be done concurrently, and also that the interactions with the smart contract are not blocking and can be done later to cash out anytime. Additionally, it is important to note that the verifier is a single element (i.e., the same verifier needs to be used for all the interactions), but it is shown three times in the figure just for clarity.

The different steps are as follows:

1. Alice provides the required permissions to the bank to access her score at the scoring app.

2. The bank contacts the verifier to use the scoring app.

3. The verifier validates the request and forwards it to the scoring app.

4. The scoring app contacts the verifier to use a data app.

5. The verifier validates the request and forwards it to the data app.

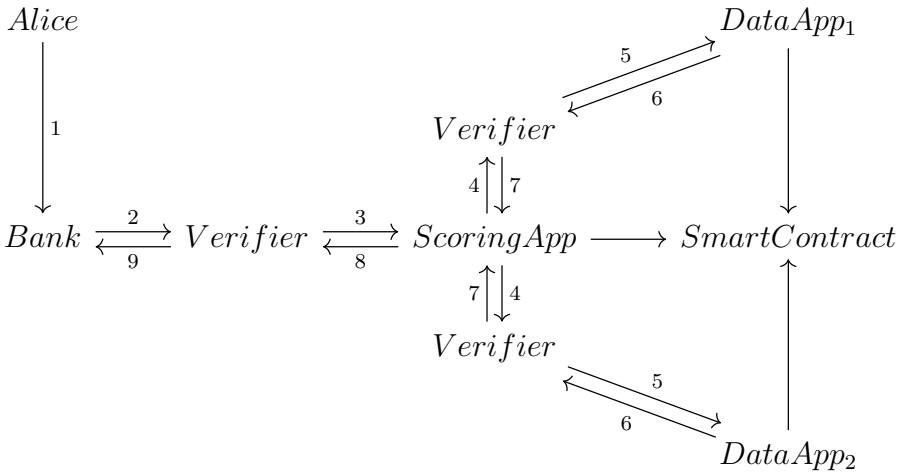

Figure 3: Interaction between all elements

6. The data app returns Alice's data in an encrypted way to the verifier so that only the scoring app can read it.

7. The verifier validates the encrypted data returned by the data app to confirm that it complies with the schema, and hands it over to the scoring app. The verifier can see the structure but not the actual contents.

8. The scoring app performs its computation and returns the data in an encrypted way to the verifier so that only the bank can read it.

9. The verifier validates the encrypted data returned by the scoring app to confirm that it complies with the schema, and hands it over to the bank. Again, the verifier can see the structure but not the actual contents.

It is assumed that the state channels between bank and scoring app, scoring app and data app 1, and scoring app and data app 2, were properly funded. After all the interaction, the apps can submit the transaction to the smart contract and cash out to get their reward.

Some observations about the incentives of each player when following this protocol:

- Alice only trusts the verifier and the data apps that have her data. Alice has the incentive to use the Reputation Network because a trusted third-party will make sure that her data is used according to the permissions given. Also, Alice knows that rewards are controlled by the smart contract, a trustless actor which will only let the rewards go through if all permissions are in order.

- Additionally, the data apps are accountable for safely keeping Alice's data. As a result, the data apps have the incentive to track when they provided data about Alice, including Alice's explicit, signed permission, and the fact that the verifier validated the transmission of data. All this information is stored in blockchain and lets the data app prove they handled Alice's data properly. Additionally, the data app would get a reward after posting the transaction to the smart contract.

- The verifier cannot really see data, but just the structure of the data (actual data are encrypted). It gets rewarded for the verification, so it has the incentive to offer high availability and performance in its service to ensure apps allow it (apps might blacklist certain verifiers if they consider that such verifiers are malicious or that they perform poorly).

- The scoring app has the incentive to track in blockchain when a computation using Alice's data was done, as the blockchain would also state that the verifier validated that there were no data leaks. Additionally, the scoring app would get a reward for computing the score by posting the transaction on blockchain.

- As will be seen later, permissions show multiple obfuscated identities, which reduces the probability of matching the identity of Alice between calls. This eliminates the incentive for the scoring app to store or cache any data from the data apps, thus removing the possibility of private data leaks caused by human errors or by hackers accessing the scoring app's databases.

- The verifier is accountable for not letting data leaks in the form of unexpected data in the agreed schema that apps return. If there was a data leak, it would be either the fault of the verifier (which was trusted by Alice) or the app (simply by publishing unexpected data). Both cases (i.e. the verifier not really verifying, or apps publishing stuff in alternative channels) are clearly not human errors but deliberate malicious activities, and both actors would be clearly identified, thanks to the information stored in blockchain. The only way to completely avoid data leaks would be operating with homomorphic encryption, which poses challenges such as limiting the kinds of operations that can be done on the data, and is left as future work (see section 5).

4 Elements

This section defines the specification for the different elements of the protocol. As mentioned, there are Reputation Network apps, a verifier, and a smart contract, as well as app clients, and a subject. In this section, we will also introduce the gatekeeper, a module that facilitates building Reputation Network apps by already providing a reference implementation of the protocol.

4.1 Subject

The subject is the first actor to start the interaction in a Reputation Network session by building the necessary permissions for apps to use their data.

The identity used for the subject features identity obfuscation. We use *child key derivation* functions to obtain a deterministic set of *n* key pairs $\{(p_{c1}, s_{c1}), (p_{c2}, s_{c2}), \ldots, (p_{cn}, s_{cn})\}$ from the subject's identity key pair (p_k, s_k).

A subject needs to submit the write permission set to data apps in order to keep them identified. The write permission set is a list of permissions that are signed with each of the child identities. The main public key needs to also be shared so that the recipient can validate that each of the sub-permissions belong to the same key pair set.

Thanks to this approach the smart contract will be called with a different p_{ci} for each score request, so no external observers know p_k and thus they ignore the full set of child keys $\{p_{c1}, p_{c2}, ..., p_{cn}\}$.

4.1.1 Write permission

The structure of a write permission set is shown in table 3. This write permission is used by the subject to let an app use its identity as input to, e.g., provide data about the subject or do a computation based on some data about them. As mentioned, it consists of a list of write permissions, each of them referencing a different child key. The write permission set includes the subject's main public key, and it is signed using such main key.

field name	**field type**
subject	*address*
write permission 1	
write permission 2	
...	
write permission N	
signature	*signature*

Table 3: Write permission set

A write permission's structure is shown in table 4. It includes the writer's address, which is the address of the app that will provide data (or *write*) about the subject, as well as the subject's child key and signature.

The consistency checks that should be made on the write permission set are:

- All writer addresses in the write permissions must be the same.
- The subject addresses in each of the write permissions must be exactly the N first derived keys of the write permission set's subject address.

field name	field type
writer	*address*
subject	*address*
signature	*signature*

Table 4: Write permission

- The write permission set must be signed by the write permission set's subject address.

Additionally, the only consistency check that should be made on each of the write permissions is that each write permission must be signed by the write permission's subject address.

4.1.2 Read permission

The read permission is generated by the subject to enable an app to use the subject's identity as input in order to, e.g., return some data or computation about the subject. Its structure is shown in table 5.

field name	field type
reader	*address*
source	*address*
subject	*address*
manifest	*hash*
expiration	*epoch*
signature	*signature*

Table 5: Read permission

It includes the following fields:

- Reader: the party, identified by a public key or address, which will have access to the data provided by the app.
- Source: the app that will return the data to be accessed.
- Subject: the subject, identified by one child key randomly chosen by the subject when building the read permission.
- Manifest: the hash of the app's manifest file. The manifest file is a specification of what the app does, and including its hash in the read permission is a way of committing to a specific version of the app. The manifest file is described in section 4.6.
- Expiration: the permission's expiration as an epoch in seconds.

The only consistency check that should be made on the read permission is that it must be signed by the read permission's subject address.

4.1.3 Schema

The output of apps need to comply with a schema specified in the read permission. Apps are expected to return a JSON structure with specific keys. The objective is to avoid data leaks due to unexpected keys and values returned.

The checks to be done on an app's output are on keys, which are hierarchically white-listed, and on values, whose serialization length need to be smaller than some specified number. The latter is done to avoid leaking unexpected data no matter the data type being used. E.g., both numbers or strings can represent the same data while using different serializations, so the only possible checks are to be done on data entropy, which is represented by data length. Letter `"a"` has 97 as ASCII code, so potentially a fake credit scoring app could be used to leak the first letter of a name, masked as a number. The only countermeasure to this consists of checking the actual lengths of values regardless of their data types, to prevent mass data leaks.

A schema is defined by a JSON document with the following restrictions:

- All values are either an object, an array, or a number. Numbers in the schema represent the expected length of serialized values in the app's output.
- All arrays must be of length two. The first element is the expected structure of the elements in the corresponding array of the app's output. The second element is the maximum length of the array.

Listing 1 shows a sample schema. Listings 2 is an app's output that complies to the schema, while listing 3 is an app's output that presents data leaks and thus does not comply. It can be observed that the valid app output can have absent keys, and that there are no kinds of type checks, as the purpose is solely preventing data leaks.

```
{ "purchases": [{ "value": 10,
                  "category": 20,
                  "currency": 5 }, 10]
}
```

Listing 1: Sample schema

```
{ "purchases": [
    { "value": 320, "category": "electronics",
      "currency": "EUR" },
    { "value": "80", "category": "toys",
      "currency": "USD" },
    { "value": 300, "currency": "EUR" },
  ]
}
```

Listing 2: Valid app output

In the invalid app output, the keys *name* and *item* are not valid, while the value of the *category* key is too long. The value checks are done on the stringified length of values, which prevents leaking what could be a JPG image (or any other kind of data) masked as a Base64-encoded string. Additionally, the invalid app output has too many purchases in the array, which according to the schema, can only be of length 10.

```
{ "purchases": [
    { "value": 0,
      "category":
        "Bfd2vNGnv...AFbhj", /* 6000-byte string */
      "currency": "EUR",
      "item": "Camera" },
    ... /* 100 ommitted elements in array */
    { "currency": "EUR" }
  ],
  "name": "John␣Smith"
}
```

Listing 3: Invalid app output

The algorithm to check whether some app's output complies with a schema is shown on algorithm 1.

The schema's *total entropy* can be computed as the sum of its values' lengths. As there can be 10 elements in the purchases array, the total entropy is 350 bytes. This number means that an app output with up to 350 bytes of data can comply with the schema. It can be used by the subject to assess the safety of using an unknown app. Apps whose schema has a high total entropy have more potential to leak data, whilst apps with a small number can hardly leak any data. E.g., a credit score can be modelled as an integer between 0 and 1000, whose size is just a few bytes, accounting for a negligible risk to the subject.

Schemas are announced by apps as their API specification in the manifest file (see section 4.6). Including the manifest hash in the read permission enables the subject to only agree on certain personal data being returned. As will be seen, app output will be encrypted so that only the recipient can read it. The schema should therefore consider value lengths of encrypted data, which will be longer than raw, unencrypted data.

Algorithm 1 Schema check algorithm

```
1:  function checkSchema(data, schema)
2:      if type(schema) = integer then
3:          if length(json(data)) > schema then return false end if
4:      else if type(schema) = array then
5:          if type(data) ≠ array ∨ length(data) > schema[1] then
6:              return false
7:          end if
8:          for all item ∈ items(data) do
9:              if ¬checkSchema(item, schema[0]) then return false end if
10:         end for
11:     else if type(schema) = object then
12:         if type(data) ≠ object then return false end if
13:         for all key ∈ keys(data) do
14:             if key ∉ keys(schema) then return false end if
15:             if ¬checkSchema(data[key], schema[key]) then
16:                 return false
17:             end if
18:         end for
19:     end if
20:     return true
21: end function
```

4.1.4 Session

The session identifies all the transactions involved in a call to multiple apps using a set of read permissions. Its objective is setting up the verifier, its commission, and enabling traceability of transactions. As will be seen, the session also enables hit proofs to be unique (proofs are used to prevent apps from getting rewards without having used data about the subject).

Table 6 shows the structure of a session. It includes the following fields:

- Subject: the subject, identified by a random child key. The subject address needs to be the same one in all the data structures involved in the session.
- Verifier: the verifier to use in the session.

field name	field type
subject	*address*
verifier	*address*
fee	*integer*
nonce	*integer*
signature	*signature*

Table 6: Session

- Fee: the fee that the verifier will earn from the reward to each app, in parts per million.
- Nonce: a random integer representing the transaction for future traceability.

The only consistency check that should be made on the session is that it must be signed by the session's subject address.

4.2 Client

A client can use a Reputation Network app whenever having the necessary read permission(s) and a session from the subject. Once ready, the client can submit a request to an app to run it, while providing

4.2.1 Request

The request represents the intent of a client to run a Reputation Network app. It includes the appropriate read permission that grants the access to the app, as well as the session and some additional fields to perform the reward to the app.

Table 7 shows the structure of a request. It includes the following fields:

- Read permission: the read permission which states that the client is allowed to query this app.

field name	field type
read permission	*read permission*
session	*session*
counter	*unsigned integer*
value	*unsigned integer*
signature	*signature*

Table 7: Request

- Session: the session that identifies this request and any previous or subsequent ones in the chain.
- Counter: an incremental value between the client and the app, which is used to prevent double cash-outs in the state channel.
- Value: the amount of reward to be given to the app.

The consistency checks that should be made on the request are:

- The read permission must pass its own consistency checks.
- The read permission's reader must be the client's address.
- The read permission's source must be the app's address being called.
- The session must pass its own consistency checks.
- The session's subject and the read permission's subject must match.
- The request must be signed by the client's address.

Any additional read permissions that the app might need to run (by, for example, calling other apps), do not need to be reflected in the request and can be supplied as extra data in the call.

4.3 Verifier

The verifier is the trusted third-party by apps and the subject, and acts as middleman between the client and the app. Therefore, a client does not connect to an app, but to a verifier. Obtaining the actual URL to connect to is done through the registry, which is a decentralized actor where apps and verifiers can post the HTTP addresses where they give service (see section 4.6).

Overall, the verifier expects a request from the client, and it is expected to return the app's output. To obtain the app's output, it needs to forward the request to the app, including any extra permissions that the app might need to run properly. Once the app's output has proven to comply with the expected schema, it will return the output to the client, and will submit the signed transaction to the app so that the app can cash out through the smart contract to get the client's reward. This process is detailed in algorithm 2.

Algorithm 2 Receiving a request at a verifier

1: **function** receiveRequestAtVerifier($request, extra$)
2: $source \leftarrow request.readPermission.source$
3: **if** $\neg contract.validateRequest(request, source)$ **then**
4: **throw** $InvalidRequest$
5: **end if**
6: $session \leftarrow request.session$
7: **if** $session.verifier \neq self.address \vee session.fee \neq self.fee$ **then**
8: **throw** $InvalidVerifierParameters$
9: **end if**
10: $\{url, schema\} \leftarrow readRegistry(source)$
11: $\{data, proof\} \leftarrow callApp(url, request, extra)$
12: **if** $\neg checkSchema(data, schema)$ **then throw** $InvalidOutput$ **end if**
13: $transaction \leftarrow \{request, proof, sign(\{request, proof\})\}$
14: **if** $\neg checkTransaction(transaction)$ **then throw** $InvalidProof$ **end if**
15: $submitTransaction(url, transaction)$
16: **return** $\{output, proof\}$
17: **end function**

It can be observed that checking the request is delegated to the smart contract.

This has the advantage that strong checks are made, and not just consistency ones. For example, as will be seen in section 4.5, to cash out there needs to be enough funds in the channel between client and app. The smart contract makes this check, as well as making sure that the counter is set properly in the request. It can be noted that these kinds of checks are free and fast on a smart contract, as they are just read operations.

4.3.1 Transaction

A transaction represents a successful interaction between a client and an app, and is validated by a verifier. It consists of a request, generated by a client, and a proof, returned by an app, along with the verifier's signature, as shown in table 8. As will be seen in section 4.4.1, the proof is a way of checking that the app had access to personal data from the subject to properly track a hit vs. a miss.

field name	**field type**
request	*request*
proof	*proof*
signature	*signature*

Table 8: Transaction

The consistency checks to be done on a transaction are:

- Both the request and the proof should pass their own consistency checks.
- The request's session and the proof's session need to be exactly the same.
- The transaction must be signed by the session's verifier address.

4.4 App

Reputation Network apps receive a request from the client (along with optional extra read permissions), which has been forwarded by the verifier. They respond with

an output that needs to comply with the schema announced in the registry (section 4.6 covers how an app publishes a manifest file with their schema and additional information). The will also need to attach the proof that they have used personal data about the subject, which will be required to get any reward when cashing out through the smart contract. The process that apps need to follow to provide their service is described later on in section 4.4.2.

4.4.1 Proof

The proof represents the fact that some personal data about the subject has been used when running an app. A proof can be built by an app if they own the subject's write permission set (and therefore, they own the write permission for the child key employed in the request's read permission). Alternatively, a proof can be bypassed from one app to the next one. One example of the latter could be a scoring app that gathers personal data from a data app – the data app would build the proof out of the write permission, and return it to the scoring app, which would simply pass the proof on to the client.

In essence, a proof is a write permission attached to a session. Proofs are used to cash out through the smart contract by showing there was a hit and not a miss. By making proofs depend on the session, it is not possible to cash out with previous hits.

The structure of a proof is shown in table 9. As said, it is just a write permission and a session, along with the write permission's writer's signature, which is the actor that built the proof in the first place.

field name	**field type**
write permission	*write permission*
session	*session*
signature	*signature*

Table 9: Proof

The consistency checks that a proof must pass to be valid are:

- Both the write permission and the session should pass their own consistency checks.
- The session's subject and the write permission's subject must match.
- The proof must be signed by the write permission's writer address.

4.4.2 Gatekeeper

The gatekeeper is a tool to run apps in the Reputation Network. On one side, it can be configured to run as a verifier. Since all verifiers in the network perform the same function, it only requires setting up the desired fee for this purpose.

It can also be configured to run as an app. As each app will have different behaviours and functions, the gatekeeper acts as a proxy to the actual implementation of an app.

The main idea is that the gatekeeper handles all the permission checks and interactions with the smart contract, thus letting companies focus on the actual business logic of the app. The implementation process for a Reputation Network app would simply consist of building a private, unsecured, service that processes a request, and deploying the gatekeeper to give public access to the private service in a secure way.

The actions to be done by an app were introduced in section 4.4. Algorithm 3 shows the process of dealing with a request before passing it to the private, unsecured service.

A few observations about algorithm 3:

- As in the case of the verifier, checking the request is delegated to the smart contract to make sure that strong checks are made, and not just consistency ones. Since the gatekeeper is the point of no return before providing the actual service for the desired reward, it is important to check the request here and not trust that the verifier has submitted a valid request.
- The gatekeeper can be configured to reject verifier fees above certain value, in order to make sure that a considerable amount o reward is received. It also

Algorithm 3 Receiving a request at an app's gatekeeper

```
1:  function receiveRequestAtGatekeeper(request, extra, sender)
2:      if ¬contract.validateRequest(request, self.address) then
3:          throw InvalidRequest
4:      end if
5:      if request.value ≠ self.value then throw InvalidReward end if
6:      session ← request.session
7:      if session.verifier.address ≠ sender then
8:          throw VerifierIsNotSender
9:      end if
10:     if session.verifier.address ∈ self.blackListedVerifiers then
11:         throw InvalidVerifier
12:     end if
13:     if session.verifier.fee > self.acceptableFee then
14:         throw InvalidVerifierFee
15:     end if
16:     {data, proof} ← callUnsecuredApp(self.url, request, extra)
17:     reader ← request.readPermission.reader
18:     encryptedData ← encryptValues(data, reader)
19:     signature ← sign({encryptedData, proof}, self.address)
20:     return {encryptedData, proof, signature}
21: end function
```

features a blacklist of verifiers to avoid malicious actors.

- The gatekeeper needs to make sure that the actual sender is the same verifier that is stated in the session. Otherwise, any verifier could be used with any request no matter what was predefined in the session. Signing the HTTP request serves the purpose of tracking the sender.

- Data values are encrypted using the data recipient address, which is the reader in the read permission. This was anticipated in section 4.1.3, and is used to prevent verifiers from viewing raw personal data while letting them from perform their verification task.

- The returned data includes the proof along with a signature of both. This en-

sures that the verifier cannot modify the encrypted data with an alternative payload from a previous request.

Additionally, the gatekeeper supports receiving a transaction signed by the verifier. The gatekeeper simply submits the transaction to a smart contract to cash out the reward. This closes the loop for providing all the smart-contract-related tasks to developers, in order to relieve them from having to implement functionality that is specific to the Reputation Network's protocol.

The gatekeeper is an optional module in the Reputation Network architecture. It is simply a reference implementation of the protocol at the app's side to facilitate deploying apps in the Reputation Network. It would still be possible to implement an app without the gatekeeper, but it would imply replicating the functionality provided by it.

4.5 Smart contract

The smart contract is the actor that is in charge of keeping a consistent history of transactions and accepting rewards to the involved actors. Transactions consist of a request and a proof, so what is stored in the blockchain are mostly permissions. As identities are obfuscated by using child key derivation, an external observer cannot conclude much by taking a look at what is stored in the blockchain. Actual data are transmitted off-chain, so there is no personally identifiable information that is stored on blockchain.

The smart contract's duty is to persist consistent transactions and submit rewards. To achieve this, state channels are used so that cash out operations can be done asynchronously. I.e., apps receive their reward after they have returned their output, but they have a receipt that assures they will be able to cash out. The procedure to use the smart contract is:

- A client that wants to use a Reputation Network app funds the channel between themselves and the app.

- To fund the channel, the client calls the Reputation Network some amount of tokens, as well as an expiration date of the deposit. This is a way of being able to recover the deposit if it is not used.
- Once the channel is funded, the client can issue a receipt to the app. Actually, the receipt is the request described in section 4.2.1. This is what will let the app obtain the reward.
- The app can use the smart contract to verify that the request is well formed. This was mentioned already in section 4.4, and is a way to delegate validation checks.
- The app needs to submit a transaction to the smart contract in order to cash out. A transaction consists of a request and a proof, signed by a verifier, so the app needs to wait for the verifier to return the signed transaction (see line 15 at algorithm 2).
- Once the signed transaction is in the hands of the app, it can submit it to the smart contract and get the reward.

This structure enables apps to cash out asynchronously without blocking the responses, but being sure they will get their reward (their only dependence is waiting for the verifier's signature, but it is an actor they trust).

4.5.1 Fund function

The first function to fund the channel is shown in algorithm 4. The function simply allows to add funds to a channel and set an expiration for the deposit. The list of funds is kept in a dictionary of the different channels (the total sum is stored inside the contract itself). It is important to note that the expiration can only be extended; otherwise the recipient would never be safe about the expiration time of a deposit.

Algorithm 4 Smart contract's fund function

```
1: function fund(to, value, expiration)
2:     channel ← {sender, to}
3:     balance ← self.balances[channel]
4:     if balance.expiration < expiration then
5:         throw CannotReduceExpiration
6:     end if
7:     transfer(sender, self, value)
8:     self.balances[channel] ← {balance.value + value, expiration}
9: end function
```

4.5.2 Release function

The second function is the release function, which enables an actor to recover their funds if the deposit has expired. This is shown in algorithm 5. After checking whether the deposit is expired or not, the function resets the channel's funds and expiration and transfers backs the funds to the sender.

Algorithm 5 Smart contract's release function

```
1: function release(to)
2:     channel ← {sender, to}
3:     balance ← self.balances[channel]
4:     if balance.expiration > now then throw DepositNotExpired end if
5:     transfer(self, sender, balance.value)
6:     self.balances[channel] ← {0, 0}
7: end function
```

4.5.3 Validate request function

The third function is the validate request function, which is used by apps and verifiers to confirm that a request will be accepted in a specific channel. This involves not just checking the consistency of a request (as described in section 4.2.1), but also that the channel is properly funded, as well as making sure the counter is above the last one used. Requests need to have incremental counters to avoid double cash-outs. To

achieve this, a dictionary that keeps all counters for the different channels is stored in the smart contract.

Algorithm 6 Smart contract's validate request function

```
1:  function validateRequest(request)
2:      if ¬consistentRequest(request) then throw InvalidRequest end if
3:      if request.readPermission.source ≠ sender then
4:          throw StolenRequest
5:      end if
6:      permission ← request.readPermission
7:      channel ← {permission.reader, permission.source}
8:      if self.counters[channel] < request.counter then
9:          throw InvalidRequestCounter
10:     end if
11:     if self.balances[channel].value >= request.value then
12:         throw NotEnoughFunds
13:     end if
14:     return true
15: end function
```

4.5.4 Cash out function

The last function is the one used by apps to store the signed transaction to blockchain and get their reward. It requires validating the consistency of a transaction (request, proof, and verifier's signature), as well as making sure the channel is properly set up (so the checks done by the validate request function in algorithm 6 must be done).

Bear in mind that, in this case, the sender (i.e., the one interacting with the smart contract) is the actor that will be receiving the reward. Another relevant actor is the verifier, which will be getting a part of the reward according to the agreed fee, which is expressed in parts per million. After transferring the reward to the app, and the fee to the verifier, the smart contract logs the transaction onto its transaction history.

Algorithm 7 Smart contract's cash out function

1: **function** cashout($transaction$)
2: **if** $\neg consistentTransaction(transaction)$ **then**
3: **throw** $InvalidTransaction$
4: **end if**
5: $validateRequest(transaction.request)$
6: $request \leftarrow transaction.request$
7: $value \leftarrow request.value$
8: $verifier \leftarrow request.session.verifier$
9: $fee \leftarrow request.session.fee \times request.value/1000000$
10: $channel \leftarrow \{request.readPermission.reader, sender\}$
11: $balance \leftarrow self.balances[channel]$
12: $transfer(self, sender, value - fee)$
13: $transfer(self, verifier, fee)$
14: $balance' \leftarrow \{balance.value - value, balance.expiration\}$
15: $self.balances[channel] \leftarrow balance'$
16: $self.counters[channel] \leftarrow request.counter$
17: $self.history.log(transaction)$
18: **end function**

4.6 Registry

The registry is a decentralized actor where Reputation Network actors (i.e., both apps and verifiers) can inform about their services. It is simply a smart contract where actors can update their manifest URL in JSON format. Therefore, checking the registry consists of reading the smart contract and subsequently checking the manifest file on the corresponding URL. The registry also stores the manifest's hash (computed out of the file's raw data) to enable versioning and to circumvent DNS attacks when downloading the file.

The manifest file must be serialized as JSON, and its structure is shown in table 10. It has the following fields:

- Version: a version number of the app.
- Name: a human-readable name for the actor, which will be used in permission dialogs.
- Description: a human-readable description for the actor, which will be used in permission dialogs along with the name.
- Homepage URL: a URL pointing to the site that provides the app or where the app's data (if any) can be managed.
- Picture URL: a square picture that represents the actor, which will be used in permission dialogs along with the name and description.
- Address: the public key of the app.
- Verifier URL: a URL where to expect verifier functionality. If the actor is not a verifier, it can be omitted.
- Verifier fee: the expected fee for the verification service, in parts per million. If omitted, zero fee is assumed.
- App URL: a URL where to expect app functionality. If the actor is not an app, it can be omitted.

- App callback URL: a URL where the verifier would post verified, signed transactions. Only applies to apps.
- App reward: the expected reward for the app's service. If omitted, zero reward is assumed.
- App schema: the schema to be used by the app when returning its output.
- App dependencies: a list of address for whom the app needs read permissions to properly run.

field name	**field type**	**default value**
$version$	*string*	(required)
$name$	*string*	(required)
$description$	*string*	(required)
$homepage\_url$	*URL*	(required)
$picture\_url$	*URL*	(required)
$address$	*address*	(required)
$verifier\_url$	*URL*	$null$
$verifier\_fee$	*unsigned integer*	0
app_url	*URL*	$null$
$app\_callback\_url$	*URL*	$null$
app_reward	*unsigned integer*	0
app_schema	*schema*	$null$
$app\_dependencies$	*address list*	[]

Table 10: Manifest structure

A sample manifest file is shown on listing 4. It belongs to a hypothetical scoring app that shows the endpoint to run it, the expected reward, its dependencies, and the

schema it will produce (in order to verify its proper execution). This does not imply that additional documentation will not be needed in order to truly understand what the app actually does and how it works.

```
{ "version": "1.0",
  "name": "Super␣Scoring␣App",
  "description": "This␣app␣provides␣great␣risk␣scores",
  "homepage_url": "http://example.com/scoring-app-homepage",
  "picture_url": "http://example.com/logo-square.png",
  "address": "c25b4ff9eb6f52392eef1e103daacc7519795f01",
  "app_url": "http://api.example.com/super-scoring-app",
  "app_callback_url": "http://api.example.com/super-scoring-app/callback",
  "app_reward": 10,
  "app_schema": { "score": 5 },
  "app_dependencies": [
    "31bb9d47bc8bf6422ff7dcd2ff53bc90f8f7b009",
    "88032398beab20017e61064af3c7c8bd38f4c968"
  ]
}
```

Listing 4: Sample manifest file

5 Future work

One possible improvement in the system is making input data completely opaque to apps. The objective would be to completely prevent apps from leaking data from other sources.

However, this would limit the operations that can be applied to data, and would also leave threats such as leaking the decryption keys, which would be in the hands of the app expecting the output data.

In such case, the main deterrant would be again the information stored in the blockchain, which would point blame where it is deserved. Overcoming the threat of encryption keys leakage is not immediate unless completely migrating to a fully

trustless architecture where only Alice is in charge of homomorphic decryption keys.

This would limit the applications, as Alice would have a power to interrupt sessions when the output is disadvantageous for her, something likely to happen in scoring applications. I.e., Alice would have the power to decide not to return a credit score if it is too low for her interests. Although she could certainly check her credit score before sharing her permissions with a third-party, leaving the decryption task to the subject leaves room to repetitively denying the output to clients that are paying to a Reputation Network app and not getting the decrypted output. A escrow mechanism could be employed to deincentivize these behaviours, but it would increase the complexity for consumers.

As a result, this is an area left for research. The current approach does not fix privacy concerns in a trustless way, but aligns incentives, defines accountabilities for good data practices, and points the blame appropriately in the case of data leaks.