Blockchain-based Bidirectional Transformations for Access Control and Data Sharing in EMRs

Tao Zan
zantao007@gmail.com
School of Mathematics and Information Engineering,
Longyan University
Fujian, China

Zhenjiang Hu
huzj@pku.edu.cn
Key Lab of High Confidence Software Technologies,
Ministry of Education, Department of Computer Science
and Technology, EECS, Peking University
Beijing, China

Abstract

Electronic medical records (EMRs) are scattered in different hospitals, which hinders the process of data sharing. On the other hand, people with different roles should access different parts of data, thus we need a way to control the accessibility. To address these issues, we propose a blockchain-based data sharing system that gathers EMRs into blockchain and controls data sharing through bidirectional transformation. In our system, EMRs are encoded with a carefully designed data structure that stores not only data, but also data's read/write permission for different roles. We redesigned a bidirectional transformation language based on our previous work BiGUL by taking access control into consideration, and the accessibility are checked during program execution in both direction to avoid un-authorized data access. Further more, each bidirectional transformation program and updates on the shared data are stored in the blockchain in a transaction-like form. The immutability of blockchain guarantees EMR's data integrity.

CCS Concepts: • **Software and its engineering → Application specific development environments**.

Keywords: bidirectional transformation, blockchain, access control

ACM Reference Format:
Tao Zan and Zhenjiang Hu. 2020. Blockchain-based Bidirectional Transformations for Access Control and Data Sharing in EMRs. In *12th Asia-Pacific Symposium on Internetware (Internetware'20), May 12–14, 2021, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3457913.3457915

1 Introduction

Suppose a patient often goes to several hospitals (A near home, B close to work place, and even C for special treatment), and thus his/her medical records are scattered. Since each hospital only has partial information, a doctor needs to inquire about past data from other hospitals which will probably delay the treatment process, as the data shared from other hospitals may not update-to-date. In general, there should be a place where all the medical records are gathered.

On the other hand, patients should be able to see their own medical records, but this does not mean every detail. For example, doctor's psychotherapy notes, or physician intellectual property should not be made available to patients or even other parties. Medical records are also useful for research, but when sharing with researcher we shall not expose patients' private/sensitive information (i.e. name, home address, phone number). Only research related information such as medicine name, mechanism of medicine and the symptom before and after treatment could be shared. We need a way to control the accessibility for different roles.

Several inspiring works [2, 6, 14] have been done, and they all follow a similar approach. In order to gather fragments, instead of storing data in each hospital's local database, sending all the data to a trusted server/cloud which can only be accessed by authorized users. Moreover, a cryptographic hash of a medical record is stored on the blockchain to guarantee data integrity. If the medical record in the server/cloud storage is tampered, the hash value of the modified record will not be the same with the one stored on the blockchain. Smart contract [13] or Chaincode [5] is used to code permissions. Only if a user satisfying the permission can be agreed by blockchain nodes and granted to read/write data.

Since permission handling and data retrieving are two separate processes, even though smart contract or chaincode can be used to store and verify permissions, we cannot guarantee that an actual action (database query/update executed off-blockchain) is conformed with the corresponding permission. On one hand, as an action can be a complicate program, we cannot formally check the program satisfies the given permission; On the other hand, if someone tampers with the program, permission checking on smart contract cannot stop execution of the program.

We think actions should also be stored on blockchain as transactions, to make it transparent and tamper-proof. Further more, permission checking should be embedded into the action, to make their behavior consistent. Instead of checking permission using smart-contract firstly and then executing a separate program to retrieve data, we do it in the other way around. During execution of query/update, when facing basic data (i.e. name, address), we check whether the user has proper permission. If the user has no read permission, then the query action is prohibited. If the user has no write permission, the data will not be updated.

We propose a new framework that combines blockchain and bidirectional transformation. In our system, every thing (record, permission, and query/update operation) is stored on the blockchain. We carefully define the data structure for EMRs that includes both data and permission. There is a one-to-one mapping between each data item (i.e. name, address) and its permission. Each permission contains read/write information for different roles (patient, doctor, researcher).

Bidirectional transformation (BX in short) provides a mechanism that one program can be executed in both forward (*get*) and backward (*put*) directions , which can be used to query (*get*) parts of data as a view from a big source, or reflect (*put*) updates on the view back to the source. Based on our previous work BiGUL [12], we revised its semantics to decompose permission data when decomposing medical record data, and the semantics will further check accessibility of data against corresponding permission. Through get transformation, different person queries different part of data from full EMRs and the get semantics guarantee the person has proper permission to read the data; Through put transformation, all invalid updates (no write permission, or big change that violates put semantics) will be rejected.

We do not directly store full medical records for one patient. Only modifications on view and bidirectional transformation program that computes this view are stored on the blockchain in a transaction-like form. Just as computing account balance in bitcoin, full EMRs can be constructed by computing all related transactions in chronological order.

In this paper, we propose a novel framework that gathers all EMR-related data in one place on blockchain and supports access control and data sharing of EMRs among different people through bidirectional transformation. Our main contributions can be summarized as follows:

- As far as we know, we are the first one to propose a blockchain-based bidirectional transformation framework that combines the merits of both bidirectional transformation and blockchain technology.
- We redesign the bidirectional transformation semantics by taking access control into consideration. For a given BX program, if user has permission to read-/write record data, then the program generates result, otherwise, it will fail.

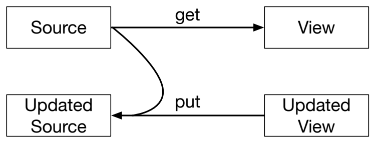

Figure 1. Bidirectional Transformation

- We define a bitcoin-like blockchain to store EMR-related data and bidirectional program. The program and modifications on view are packed as a transaction on the blockchain. The immutability of blockchain guarantees data integrity. All of the transactions are validated before pushing to the chain. The proof-of-work consensus algorithm, the underlying peer-to-peer sharing of chain data, and the longest-chain property prevent transactions from tampering.
- We implemented a prototype system in TypeScript for blockchain-based bidirectional transformation and tested it with a typical medical data sharing example which shows the correctness and usefulness of our system.

The rest of this paper is organized as follows: in Section 2, we formally introduce bidirectional transformation, as well as blockchain technology. In Section 3, we give an overview of system architecture. We present our designed bidirectional transformation language in Section 4, and explain the design and implementation detail of blockchain in Section 5. Section 6 shows a case study of EMRs. Section 7 presents related work. We conclude this paper in Section 8, where we discuss future work.

2 Background

2.1 Bidirectional Transformation

Bidirectional transformation [3] consists of a pair of functions (*get* and *put*) as shown in Figure 1. The *get* function extracts part of information from source to construct a view, and the *put* function takes an updated view and the original source as inputs and outputs an updated source. Any updates on source can be reflected to view through *get*, and modifications on the view can be reflected back to the source through *put*.

Bidirectional transformation provides a mechanism for maintaining the consistency between two related information. To guarantee the correctness, this pair of *get* and *put* needs to satisfy two properties which is called well-behavedness.

$$put(s,\ get(s)) = s \quad \text{(getput)}$$
$$get(put(s,\ v)) = v \quad \text{(putget)}$$

The *getput* property means that a bidirectional transformation is stable: putting back a target immediately after getting it shall yield the original source. The *putget* property

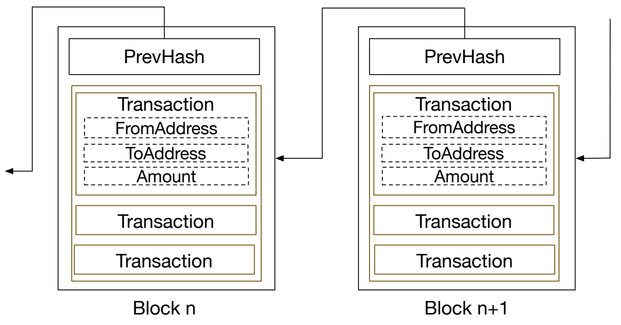

Figure 2. A Chain of Blocks

means that a bidirectional transformation is acceptable and all target updates can be reflected to the source: applying the forward transformation after putting back any view shall return the same target.

Many bidirectional programming languages [4, 7, 9, 11] have been proposed, and we deploy the putback-based approach [10, 12, 17, 20, 21]. Since the programmer only need to write one put transformation, from which a unique forward transformation can be derived for free. We will explain our putback-based language in detail in Section 4.3.

2.2 Blockchain

A blockchain is a type of distributed database that every node has the same copy of transaction data which provides robustness and prevents single-point-of-failure. The blockchain is in the form of chained blocks chronologically, as shown in Figure 2. Each block contains a list of transactions, and each transaction includes sender and receiver's address, together with the amount to be sent. Moreover, each block is marked with the hash (*PrevHash*) of previous block, which makes transaction data immutable and traceable.

In Bitcoin [16], the consensus of nodes is achieved by using proof-of-work mechanism. The proof-of-work is based on an algorithm to find a nonce so that the hash result of a block's data smaller than a certain value. After reaching consensus by 51% of the participants in the distributed network, valid blocks will be added to the blockchain.

3 Architecture Overview

Our system is devised in a two-layer architecture, consisting of on-chain layer and off-chain layer, as shown in Figure 3. The function of each layer is described as follows.

The upper part is a bitcoin-like blockchain, which consists of a sequence of chained blocks, each block stores the hash value of its previous block. Each block's data area keeps a record of signed and validated transactions. Each transaction is signed with the person's (who issued the transaction) private key, the blockchain can verify the signature by using the person's public key derived from the *FromAddress*. Instead of storing the detail of digital assets transferred from A to B in the bitcoin system, a transaction includes role of the actor (patient, doctor, and researcher), modifications on view by calculating the difference between original view and updated view, and the corresponding bidirectional transformation program written in our designed language.

The lower part is the off-chain layer. When a specific bidirectional program is given, such as BX_{doctor}, the off-chain layer can fetch a specific view of some patient's medical records from the blockchain by running the get (BX_{doctor_get}) direction of the bidirectional program. If the doctor modifies one medical record as marked in red color in Figure 3, then the off-chain layer will package the modification, the BX program (BX_{doctor}), as well as the role information into a transaction, and sign the transaction with the doctor's private key. After that, the transaction will be added to the blockchain's pending transaction list if the transaction is valid. If the transaction is a valid transaction (the modification on the records satisfies access permission and the signature is also valid), then the transaction will be mined and packaged into a block in the blockchain.

4 Refinement of BiGUL

4.1 Data Structure for Medical Records

We carefully design the data structure for medical records, by following the principle of simplicity and expressiveness. On top of that, we build a robust bidirectional transformation language to manage the data.

Since the whole system is built using TypeScript language, we will mainly describe the detail using TypeScript syntax. We define an interface for the Source, which contains two parts: the actual medical record data part (*data*) and the description of accessibility (*access*).

```
interface Source {
  data: Data;
  access: Access;
}
```

For simplicity, we use a syntax which is similar to Haskell's datatype definition to describe the data structure. The data can be a list of Data, a Data pair, an object that contains a Name (which is a string) points to its value which is also a Data, and basic data such as a string, number or boolean value.

```
Data = [Data]
     | (Data, Data)
     | {Name: Data}
     | string | number | boolean
```

Note that: (1) there is no pair type in TypeScript, we defined a Pair class, with a left() and right() function to retrieve left and right part of the pair; (2) we enforce there should be only one attribute in an object (we call it one-attribute object), i.e., in the form of $\{Name : Data\}$. Multiple attributes can be implemented using pair. For example, a person with a name and a phone number can be described as $(\{name : "Alice"\}, \{phoneNumber : 0345559584\})$.

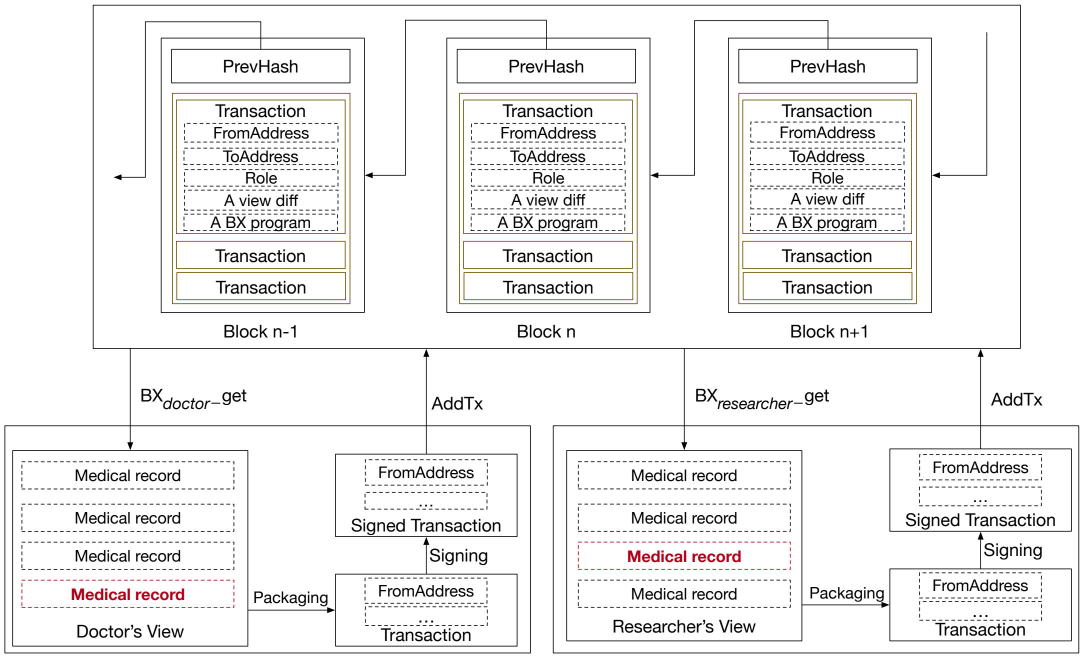

Figure 3. An Overview of Blockchain-based Bidirectional Transformation System

The definition of Access is simpler, which only contains a Role and a pair of Roles. A pair of Roles always goes after a pair of Datas. The access definition for list and one-attribute-object is simplified. Since list is used to store a sequence of data with the same type, so we just use *Role* instead of [*Role*] to define the access information for list. The accessibility for an one-attribute object is the same as its value.

```
Access = (Role, Role)
       | Role
```

Role has three types: researcher, doctor and patient. For a given data, the read and write permission for each role must be defined.

```
interface Role {
  researcher: Authority;
  doctor: Authority;
  patient: Authority;
}
interface Authority {
  read: boolean;
  write: boolean;
}
```

Following is an example that defines data and access for a person's name and phone number. We can see that all of them have read permission for both *name* and *phoneNumber*, and only patient has write permission.

```
{
  data: (
    {name: "Tao Zan"},
    {phoneNumber: 15705832363}
  ),
  access: (
    {researcher:{read:true, write:false},
     doctor:{read:true, write:false},
     patient:{read:true, write:true}
    },
    {researcher:{read:true, write:false},
     doctor:{read:true, write:false},
     patient:{read:true, write:true}
    }
  )
}
```

The data structure for view is simplified that excludes the access information. Note that in the final view presented to end-user, we will use access information to guide and restrict data modification, but for simplicity access data are not stored in this view. If the end-user bypasses the access checking on the view, unauthorized modification will still be prohibited when putting back to the source.

```
interface View {
  data: Data;
}
```

4.2 Formalization of Bidirectional Transformation with Access Control

We slightly modified the definition of *get* and *put* function in bidirectional transformation by adding an extra parameter *Role*. The *role* is defined as an enumerate type that only has three values : *patient*, *doctor*, and *researcher*.

```
bigul ::= Replace | Skip| Iter bigul | upat
        |  RearrS pat env bigul
        |  RearrV pat env bigul
upat  ::= UVar bigul | UIn upat | UConst val
        | ULeft upat | URight upat
        | UProd upat upat
pat   ::= PVar var | PConst val | PIn val pat
        |  PLeft pat | PRight pat
        | PProd pat pat
env   ::= EConst val | EProd env env
        | ELeft env | ERight env
        | EIn val env | EDir dir
dir   ::= DVar var | DLeft dir | DRight dir
```

Figure 4. Language Definition

Lemma 1 (Permission Implication)**.** *For an attribute, if a role has write permission, then it implies the role has read permission.*

The permission implication is quite straight forward: a role has write permission means he/she can modify the data. Before modification, he/she needs to query the data which means he/she has read permission.

Since both the *get* and *put* semantics will check the accessibility for a given role, then the transformation may fail if the permission is denied. We use *fail* to denote the failure of a transformation.

$$get(role, s) = v \text{ or } fail$$
$$put(role, s, v) = s' \text{ or } fail$$

The well-behavedness is still guaranteed:

$$get(role, s) = v \longrightarrow put(role, s, v) = s \quad \text{(getput)}$$
$$put(role, s, v) = s' \longrightarrow get(role, s') = v \quad \text{(putget)}$$

If $get(role, s)$ fails which means we have identified an permission-denied action and the program will terminate. There is no need of checking getput property and it is the same for the other way around. The getput property says if $get(role, s)$ can successfully compute a view v, then we will get the original source s when directly putting this view v back. The putget property says if $put(role, s, v)$ can success successfully compute an updated source s', then we can get the exact view v by executing $get(role, s')$.

4.3 Language Syntax

Having explained the data structure for medical records in our system, we now move to the refined BiGUL [12] language itself. In our refined language, to keep it concise and simple, we removed several unneeded operators. The syntax is the same as the BiGUL, but the semantics are revised by taking access control into consideration.

Replace uses a given view to update the source by replacing it, *Skip* skips the source and the view shall be empty(*null*). Given a source list and a view item, *Iter* will use the *bigul* to update each source item in the list using the view. *upat* performs pattern matching on the source.

RearrS rearranges source by firstly using *pat* to decompose the source, then using *env* to construct a new source, and finally performing *bigul* on this new source. *RearrV* is similar to *RearrS*, except that all the view variables must be used to update the source, and this is necessary for embedding view into source, so we can *get* the view from source.

UProd is used to match pair-structured data of source, *ULeft* and *URight* only match left and right part of the pair respectively, *UConst* matches source against a constant value, *UIn* steps into the source when source is an on-attribute-object, *UVar* defines a hidden variable that points to the matched part of the source which will be updated in the *bigul* program.

pat, *env*, and *dir* are used in *RearrS* and *RearrV* to perform pattern matching on source/view and construct a new source/view.

4.4 Replace

Permission handling is mainly done in *Replace*, and thus we describe its semantics of *get* and *put* in detail.

Definition 2 (get of Replace)**.**

$$get(role, s) = \begin{cases} s, \textit{ if read permission is true} \\ fail, \textit{ otherwise} \end{cases}$$

For *get* direction, if a person has read permission, we construct a view which is the same with source data, otherwise throw an error to indicate the person has no read permission.

Definition 3 (put of Replace)**.**

$$put(role, s, v) = \begin{cases} v, \textit{ if write permission is true} \\ s, \textit{ if read permission is true and v equals s} \\ fail, \textit{ otherwise} \end{cases}$$

For *put* direction, if the person has write permission, then we will update the source data with view data. If the person has no write permission, but he/she can read the data, then we check the equality of source and view data. If they are equal, then we directly return the original source. If not, we throw an error to indicate the view data shall not be modified.

We can easily check the well-behavedness property by hand, and we omit the formal proof.

4.5 Source Rearrangement

RearrS accepts three arguments: *pat*, *env*, and *bigul*. *pat* is short for pattern. Given a source, we performs pattern matching on the source data. *env* is used to construct a new sub-source, and finally a *bigul* program is used to synchronize between this new sub-source and view.

4.5.1 Example.

Suppose we have a source that contains a pair of name and location. For simplicity, we omit the access information here.

```
{
  data: (
    {name: "Tao Zan"},
    {location: "LY"}
  ),
  access: ...
}
```

For this example, the source is a pair of name and location one-attribute object. We can write the following patterns:

- *new PProd*(*new PVar*(), *new PVar*()): the left *PVar* matches the left part, i.e. $\{name : "Tan\ Zan"\}$, and the right *PVar* matches $\{location : "LY"\}$.
- The second pattern:

  $$new\ PProd(new\ PIn("name", new\ PVar()),\ new\ PConst(\{location : "LY"\}))$$

  PIn says we dig into the one-attribute object, and matches the value that the name points to, so *PVar* matches "*Tao Zan*". *PConst* adds a constraint that the right part must be the same as $\{location : "LY"\}$.

Now we construct a new sub-source by retrieving the values that *PVar* matched in the pattern.

- For the first pattern, if we want to shift name and location, we can write a env like:

  $$new\ EProd(new\ EDir(new\ DRight(new\ DVar())),\ new\ EDir(new\ DLeft(new\ DVar())))$$

  This pattern will construct a new pair that the left side is the location object and right is the name object.
- For the second pattern, we could extract the only one *PVar* as the new sub-source:

  $$new\ EDir(new\ DLeft(new\ DVar()))$$

  Since *PConst* occupies one hole in the pattern matching result (the result is like $(val, null)$), so *DLeft* is used to retrieves the left value of the result pair.

Finally, suppose our view only contains name value:

```
{data: "Tao Zan"}
```

Our intention is to keep the location as a constant and changes of the value on the view should be reflected back to the source. We can write the following program to synchronize between source and view. Since the data of the new sub-source is the same as the view, so we just call *Replace* operation.

```
new RearrS(
  new PProd(new PIn("name", new PVar()), new
      PConst({location: "LY"})),
  new EDir(new DLeft(new DVar())),
  new Replace()
)
```

The language syntax and semantics of RearrS might take time to understand. In fact, bidirectional transformation serves as the underline engine of our permission checking and data synchronization parts. In the final system, we can provide a graphic user interface for end-user to select attributes they are interested, and the system automatically generates bidirectional program for free.

5 Design of Blockchain

We implement a bitcoin-like blockchain by changing the contents of a transaction from real transaction to updates. The remain parts are the same, i.e. transaction will be signed by sender, using proof of work as the consensus mechanism, each block stores a SHA256-hash value of the previous block (except the genesis block), the longest chain rule to decide which chain to follow.

5.1 Transaction with Updates

A transaction is always carried out between two parties, and we denote them with *fromAddress*, and *toAddress*. Note that these two addresses can be the same when a patient modifies his/her own records. The *role* attribute specifies the role of the sender (*fromAddress*), which can be *patient*, *doctor*, or *researcher*. The *bx* attribute stores literal representation of a bidirectional program which will be executed on the blockchain's network nodes. *diff* represents the difference between updated view and original view.

```
class Transaction {
  fromAddress: string;
  toAddress: string;
  role: string;
  bx: string;
  diff: string;
  signature: string;
}
```

When constructing a transaction, sender uses his/her private key to sign the SHA256 value of string concatenation of all other attributes' value in the transaction as the signature value. Only a transaction signed by sender can pass the verification using sender's public key (derived from *fromAddress*) before adding to the blockchain or during validating of the blockchain, which prevents malicious tampering of the transaction data. Following is a transaction example.

```
{
  fromAddress: '04384517a2b5b5b336f16c6b84...',
  toAddress: '04384517a2b5b5b336f16c6b84...',
  role: 'patient',
  bx: 'new RearrS(new PProd(new PVar(),...),new
      EProd(new EDir(new DLeft(new DVar())),...),
      new UProd(new Replace(),...))',
  diff:'@@-10,1+10,1@@\n-null\n+"Ibuprofen"\n@@
      -14,1+14,1@@\n-null\n+"one tablet each 4h"',
  signature:'304502207b541194988c10b7fe756...'
}
```

```
{
  data: {
    first: {
      patientId:
        13345552244
    },
    second: {
      first : {
        medicationName:
         null
      },
      second: {
        dosage:
          null
      },
      ……
}
```

```
{
  data: {
    first: {
      patientId:
        13345552244
    },
    second: {
      first : {
        medicationName:
          "Ibuprofen"
      },
      second: {
        dosage:
          "one tablet every 4h"
      },
      ……
}
```

Figure 5. An Example of View Updating

Since the *role* in the transaction is *patient* which means a patient modifies his/her own data, the *fromAddress* and *toAddress* are the same.

5.2 Difference Computation

Even though view is relatively small compared with source, since a blockchain may contains billions of transactions, it still takes lots of disk space if we directly store the whole updated view in the transaction. On the other hand, usually updates on view are small, so we store the difference of updated view with original view based on the classical diff algorithm [15].

In Figure 5, the left side is a view queried by patient's doctor. The doctor adds medicationName and dosage for the patient as shown on the right side. Text marked in red denotes the updated parts.

We compute the difference between these two views as a patch:

```
@@ -10,1 +10,1 @@
-           null
+           "Ibuprofen"
@@ -14,1 +14,1 @@
-           null
+           "one tablet every 4h"
```

Since the original view can be constructed from blockchain through get transformation, patching the original view with the patch, we get the updated view.

5.3 Source Construction

5.3.1 Source Initialization. Patient, doctor, and researcher only see a small view of the whole source data. They always modify the view instead of directly changing the source. We use bidirectional transformation to put the updates on view back to source. So initially there should be a template source. In our blockchain system, we implement an *init* function that initializes a source for updating.

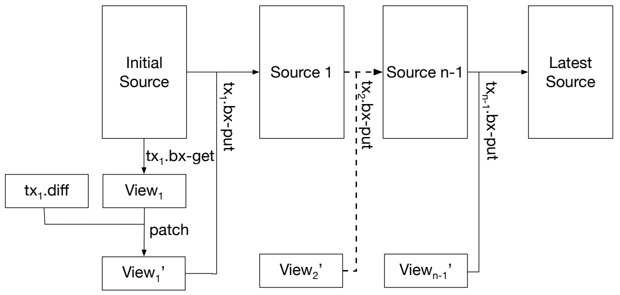

Figure 6. Source Construction through a Sequence of Puts

5.3.2 Latest Source Construction. If there are no transactions related to an given address, then an initial source is the latest source. Otherwise, the latest source can be computed by applying all the related transactions in order, since all the modifications are recorded in transactions. Figure. 6 illustrates how a latest source is computed. Note that we suppose all the transactions in this figure is related to one person, which means the *toAddress* is the same with a given address.

We use *tx* to represent transaction, and $tx_1.bx$ means the bidirectional program stored in transaction tx_1. $View_1$ is the original view computed by executing get direction of $tx_1.bx$ on *Initial Source*, then patching $View_1$ with $tx_1.diff$ to get updated view $View_1'$. *Source 1* is computed by executing put direction of $tx_1.bx$ with $View_1'$ on *Initial Source*. *Source 2* is computed by executing put direction of $tx_2.bx$ with $View_2'$ on *Source 1* and so on. Finally, we get the latest source.

5.4 View Querying

We have described how a source can be constructed in the previous subsection, now we will show how to query a view from the blockchain through bidirectional transformation. In order to get a view from source, we firstly need to have a source. So we call *getEMRofAddress* function which is used to construct the latest source.

TypeScript offers a *transpile* function that takes a program string in TypeScript and translates it into JavaScript. By using JavaScript's built-in *eval* function, the program in text becomes a runnable program assigned to variable *runnable*. Finally, by executing the get function, we get a view.

```
getViewofAddress(address: string, role: string,
    bxStr: string) {
  let source: Source = this.getEMRofAddress(
      address);
      bx = ts.transpile(bxStr),
      runnable: any = eval(bx),
      view = runnable.get(role, source);
  return view;
}
```

5.5 Validity Checking of An Update

Before adding a transaction to blockchain, we will check whether the updated view is valid or not by calling the put direction of the bidirectional transformation program in the transaction. The blockchain will only accept valid transactions. If put fails, it means the updated view does not pass the bidirectional semantics (may break well-behavedness) or violates the permission defined in the source.

```
isValidBX(tx: Transaction) {
  let source: Source = this.getEMRofAddress(tx.
      toAddress),
      bx = ts.transpile(tx.bx),
      runnable: any = eval(bx),
      view = runnable.get(tx.role, source),
      newView = patch(source, tx.diff);
  try {
    source = runnable.put(tx.role, source,
        newView);
  } catch (err) {
    return false;
  }
  return true;
}
```

6 Case Study of EMRs

This is a typical example about sharing medical data between different people from Li et al.'s work [13]. Now we will show how it can be implemented in our designed system.

Table 1 gives two full medical records that contains a person's basic information such as address on the blockchain, patient's ID in the medical system, medication name, patient's address, clinical information (clinical name), prescription (medication name and dosage), details about medication (mechanism of action and mode of action). Table 2 shows different roles' read/write permissions. For example, patient can not read mechanism of action and mode of action, can not change dosage, but can modify address.

6.1 Permission Checking

Table 3 shows a view for researcher. As described in Table 2, a researcher can only read medication name, mechanism of action and mode of action, modify mechanism of action, but he/she cannot change mode of action.

A bidirectional program shown in Fig. 7 can be used to get a researcher's view from source as shown in Table 3, and reflect updates on the view back to source if the modifications satisfy the permission defined on the source. The core idea of this program is to rearrange the source to a new sub-source and matches each element of this new sub-source with view element.

- Lines 2-5 performs pattern matching on the source, resulting in three variables that points to medication name, mechanism of action and mode of action respectively.
- Lines 6-10 constructs a new sub-source using these three variables. For example, the left part of *EProd* is

```
1   new RearrS(
2     new PRight(
3       new PProd(
4         new PVar(),
5         new PRight(new PRight(new PRight(new
              PProd(new PVar(), new PVar()))))),
6     new EProd(
7       new EDir(new DLeft(new DVar())),
8       new EProd(
9         new EDir(new DRight(new DLeft(new DVar()
              ))),
10        new EDir(new DRight(new DRight(new DVar
              ()))))),
11    new UProd(new Replace(), new UProd(new
          Replace(), new Replace())))
```

Figure 7. Bidirectional Update Program for Researcher

new EDir(*new DLeft*(*new DVar*())), which retrieves the medication name in the result of pattern matching.
- The sub-source and view are in the form of $(val, (val, val))$, so our bidirectional program (line 11) uses *UProd* to decompose both sub-source and view, and *Replace* each part with the corresponding view element.

Table 4 gives an example that researcher modifies Mode of Action from *MoA1* to *MoA1′*. According to Table 2, researcher cannot modify Mode of Action. So the *AddTx* function in Fig. 3 will fail as this updated view will not pass the validation step on the blockchain.

```
Update record fail: Error: Source and view data
    are not equal, while the person only has read
    right. Role: researcher
  at Replace.put (~/blockchain/js-bx/src/core.js
      :33:23)
  at UProd.put (~/blockchain/js-bx/src/core.js
      :300:38)
  at UProd.put (~/blockchain/js-bx/src/core.js
      :300:38)
  at RearrS.put (~/blockchain/js-bx/src/core.js
      :227:52)
  at Blockchain.isValidBX (~/blockchain/js-
      blockchain/src/Blockchain.js:164:31)
```

Researcher has no permission to modify Mode of Action, but he/she can change Mechanism of Action, Table 5 shows researcher changes Mechanism of Action from *MeA1* to *MeA1′*. Since researcher has the permission to update this attribute, the updated view will pass the validation. Finally the bidirectional program and view difference will be packed up as a transaction and stored on the blockchain. No one can change the view difference, no one can change the bidirectional program.

```
{
    fromAddress: '04c161ca3dce084decc3...',
    toAddress: '04384517a2b5b5b336f1...',
    role: 'researcher',
```

Table 1. Full Medical Records

Address on Chain	Patient ID	Medication Name	Clinical Data	Address	Dosage	Mechanism of Action	Mode of Action
04384517a...	188	Ibuprofen	CliD1	Sapporo	one tablet every 4h	MeA1	MoA1
04c161ca3...	189	Wellbutrin	CliD2	Osaka	100 mg twice daily	MeA2	MoA2

Table 2. Permission Table

Attribute	Patient		Doctor		Researcher	
	Read	Write	Read	Write	Read	Write
PatientId	true	false	true	true	false	false
Medication Name	true	false	true	true	true	true
Clinical Data	true	true	true	true	false	false
Address	true	true	true	false	false	false
Dosage	true	false	true	true	false	false
Mechanism of Action	false	false	true	false	true	true
Mode of Action	false	false	true	false	true	false

Table 3. Researcher's View

Address on Chain	Medication Name	Mechanism of Action	Mode of Action
04384517a...	Ibuprofen	MeA1	MoA1

Table 4. Modified Researcher's View 1

Address on Chain	Medication Name	Mechanism of Action	Mode of Action
04384517a...	Ibuprofen	MeA1	MoA1'

Table 5. Modified Researcher's View 2

Address on Chain	Medication Name	Mechanism of Action	Mode of Action
04384517a...	Ibuprofen	MeA1'	MoA1

```
    bx: 'new RearrS(new PRight(...), new EProd
        (...), new UProd(new Replace(), ...));',
    diff: '@@ -10,1 +10,1 @@-\n"MeA1"\n+"MeA1\'"',
    signature: '3046022100a6c16d14d34267c8f...'
}
```

6.2 On-chain Data Sharing and Updating

We initialized a blockchain with 5 nodes, and a set of RESTful APIs can be used to manage the chain, e.g. adding transaction, and querying a view from the blockchain. Each node will broadcast its status every one minute for synchronization. Since different nodes are synchronized, querying on any node is the same and transactions pushed to any node will be broadcasted to the others.

We designed an experiment to show the correctness and usefulness of our system, by executing a sequence of queries and updates on the blockchain from different nodes with different roles. The experiment contains 12 test-cases as shown in Table 6. Own to the ability of fine-grained data sharing of bidirectional transformation, we can easily write different BX programs to share different piece of data. The experiment result shows:

- If a role has no read permission for some attributes, then the query action will fail (test-case (5), and (9));
- If a role only has read permission for some attributes, then he/she cannot modify the data (test-case (4), (8), and (12));

Table 6. Experiment Cases

Role	Update	Result	
		Permission check	Well-behavedness
patient	(1) query address, clinicalData	pass	pass
	(2) update address, clinicalData	pass	pass
	(3) query dosage	pass	pass
	(4) modify dosage	fail	-
	(5) query Mechanism of Action	fail	-
doctor	(6) query medicationName, dosage	pass	pass
	(7) update medicationName, dosage	pass	pass
	(8) modify Mechanism of Action	fail	-
researcher	(9) query PatientId, Address	fail	-
	(10) query Mechanism of Action	pass	pass
	(11) update Mechanism of Action	pass	pass
	(12) modify Mode of Action	fail	-

- If a role has write permission for some attributes, he/she can update the data (test-case (2), (7), and (11)).
- If permission check is passed, then the well-behavedness property is also satisfied.

7 Related Work

In bidirectional programming, Foster et al. [8] proposed a novel bidirectional transformation framework to build updatable security views. They enrich types of basic bidirectional combinators to capture notions of confidentiality and integrity. If the data item shall not be modified, the type annotation will have an extra mark E (short for endorsed), and updates will be refused; If allowing updates, the type annotation will have an extra mark T (short for tainted).

The potential of using blockchain technology for handling healthcare data has been discussed widely. The first idea of introducing blockchain into the design of managing medical record data was presented in [19]. They use blockchain to store the medical data directly and provide a healthcare data gateway to enable patient to control and share their data between different entities that may use patient data.

MedRec [2] stores raw medical records locally in separate provider's and patient's databases, and references to assembled medical data are encoded onto blockchain. Authorization data are stored on the blockchain's smart contracts, replicated in each node. MedRec gathers scattered medical data by storing references to medical data on the blockchain, forms a unified access and permission management platform.

Different blockchains [1, 6, 14, 18] are used to manage data sharing, and they all follow a similar approach that stores medical data off-chain and permission data on-chain, then utilizes smart contracts to control the access to medical data.

Li et al. [13] provides an innovative approach that focuses on data sharing by utilizing smart contracts and bidirectional transformation. Smart contracts define permissions for each role. If permitted, then using bidirectional transformation to synchronize related medical data among researcher, doctor and patient's local database off-chain. In our framework, we implement permission checking as part of the language semantics to guarantee the consistency of permission and action. We combine bidirectional transformation with blockchain seamlessly. Data is stored on chain, transformation is also stored on chain, which guarantees the integrity of both data an transformation.

8 Conclusion

In this paper, a blockchain-based bidirectional transformation framework for access control and data sharing for EMRs is proposed. In our system, we do not store full EMRs, which can be construct through a sequence of related transactions by using put direction of bidirectional transformation programs stored in transactions. Using bidirectional transformation, fine-grained data sharing between different entities can be accomplished. The bidirectional transformation semantics control read/write access for a given role, which also guarantee the consistency between permission and action.

In current work, we assume the actor is trustable and we did not check his/her identity. For future work, we can deploy another blockchain to store role authentication information and query from this chain to check the identity before executing get/put transformation on chain. Since information stored on the blockchain are plaintext, we will investigate encryption algorithms to figure out an approach for hiding sensitive information.

Acknowledgements

We would like to thank the anonymous reviewers of Internetware '20 for their valuable comments. This work is partially supported by Ph.D. start-up foundation of Longyan University No. LB2020010 in China and the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099 and (A) No. 18H04093.

References

[1] Sandro Amofa, E. Boateng Sifah, K. O. . Obour Agyekum, Smahi Abla, Qi Xia, James C. Gee, and Jianbin Gao. 2018. A Blockchain-based Architecture Framework for Secure Sharing of Personal Health Data. In *2018 IEEE 20th International Conference on e-Health Networking, Applications and Services (Healthcom)*. 1–6.

[2] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. 2016. MedRec: Using Blockchain for Medical Data Access and Permission Management. In *2016 2nd International Conference on Open and Big Data (OBD)*. 25–30.

[3] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations*, Richard F. Paige (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–283.

[4] Barbosa Davi, Julien Cretin, Foster Greenberg, and Benjamin Pierce. 2010. Matching Lenses: Alignment and View Update. *ACM SIGPLAN Notices* 45 (09 2010).

[5] Alevtina Dubovitskaya, Zhigang Xu, Samuel Ryu, Michael Schumacher, and Fusheng Wang. 2017. Secure and Trustable Electronic Medical Records Sharing using Blockchain. *AMIA ... Annual Symposium proceedings. AMIA Symposium* 2017 (08 2017).

[6] Kai Fan, Shangyang Wang, Yanhui Ren, Hui Li, and Yintang Yang. 2018. MedBlock: Efficient and Secure Medical Data Sharing Via Blockchain. *Journal of Medical Systems* 42 (08 2018).

[7] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (May 2007), 17–es.

[8] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. 2009. Updatable Security Views. In *2009 22nd IEEE Computer Security Foundations Symposium*. 60–74.

[9] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. 2008. Quotient Lenses. *SIGPLAN Not.* 43, 9 (Sept. 2008), 383–396.

[10] Xiao He and Zhenjiang Hu. 2018. Putback-Based Bidirectional Model Transformations. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 434–444.

[11] Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. 2011. GRoundTram: An Integrated Framework for Developing Well-behaved Bidirectional Model Transformations. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 480–483.

[12] Ko Hsiang-Shang, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (St. Petersburg, FL, USA) *(PEPM '16)*. Association for Computing Machinery, New York, NY, USA, 61–72.

[13] Chunmiao Li, Yang Cao, Zhenjiang Hu, and Masatoshi Yoshikawa. 2019. Blockchain-Based Bidirectional Updates on Fine-Grained Medical Data. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 22–27.

[14] Jingwei Liu, Xiaolu Li, Lin Ye, Hongli Zhang, Xiaojiang Du, and Mohsen Guizani. 2018. BPDS: A Blockchain Based Privacy-Preserving Data Sharing for Electronic Medical Records. In *2018 IEEE Global Communications Conference (GLOBECOM)*. 1–6.

[15] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1 (1986), 251–266.

[16] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. *Cryptography Mailing list at https://metzdowd.com* (03 2009).

[17] João Alexandre Saraiva, Pedro Martins, Zirun Zhu, Hsiang-Shang Ko, and Zhenjiang Hu. 2015. BiYacc: Roll Your Parser and Reflective Printer into One. *Fourth International Workshop on Bidirectional Transformations (Bx 2015)*, 43–50.

[18] Qi Xia, Emmanuel Sifah, Abla Smahi, Sandro Amofa, and Xiaosong Zhang. 2017. BBDS: Blockchain-Based Data Sharing for Electronic Medical Records in Cloud Environments. *Information* 8 (04 2017), 44.

[19] Xiao Yue, Huiju Wang, Dawei Jin, Mingqiang Li, and Wei Jiang. 2016. Healthcare Data Gateways: Found Healthcare Intelligence on Blockchain with Novel Privacy Risk Control. *Journal of medical systems* 40 (10 2016), 218.

[20] Tao Zan, Li Liu, Hsiang-Shang Ko, and Zhenjiang Hu. 2016. Brul: A Putback-Based Bidirectional Transformation Library for Updatable Views. In *Bx@ETAPS*.

[21] Tao Zan, Hugo Pacheco, Hsiang-Shang Ko, and Zhenjiang Hu. 2016. BiFluX: A Bidirectional Functional Update Language for XML. *Computer Software* 33 (11 2016), 93–115.