U2F-ETH Documentation

CHEN Xiwen
victoria-x@sjtu.edu.cn

April 10, 2019

Contents

1 Program Structure

The structure of the program is shown in Figure 1. It is divided into the following three parts.

- Users interact with the program on a web-based user interface. It enables the user to make operations using a browser.
- The user interface is supported by Javascript frontend that listens to user's actions and respond to their actions. It also interacts with smart contracts as client side. While the real state changes such as deposit, withdraw and transfer events happen on blockchain, basic calculations and most data processing operations are completed in Javascript frontend.
- For basic bank operations (including deposit, withdraw, transfer and lock), the smart contract changes the state and emits the events on blockchain. For U2F functions, the smart contract completes the major part of verifying signature and emits the event of successful verification on blockchain.

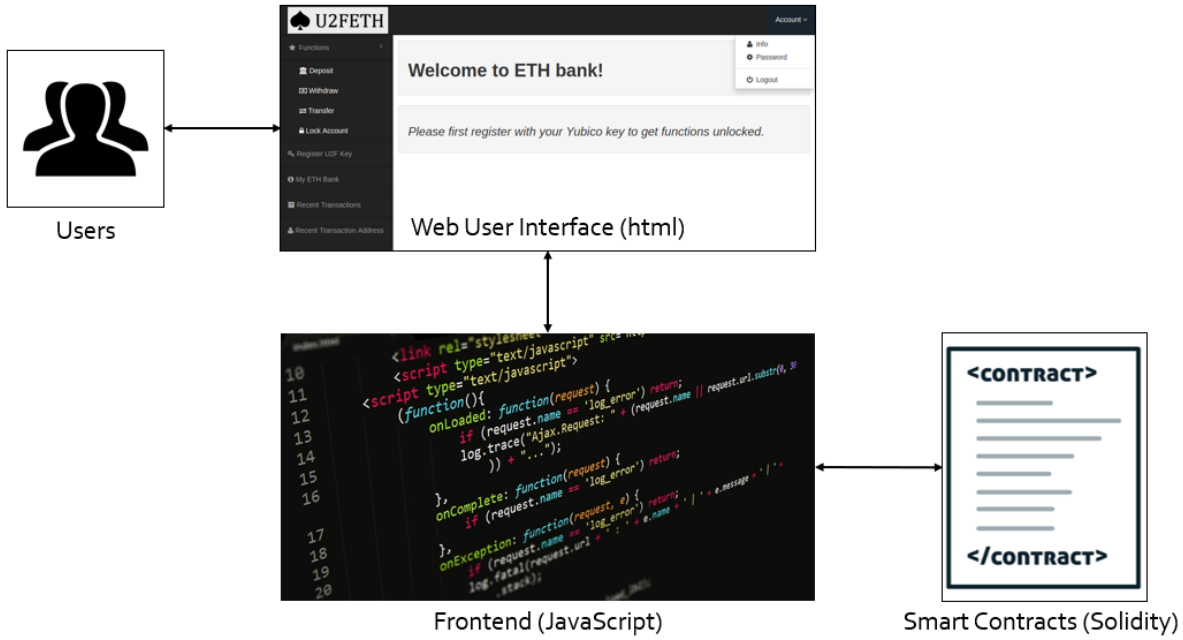

Figure 1: Program Structure.

2 Dependencies and Node Modules

Table 1: Dependencies and Node Modules

Name	Version & Notes
Solidity	0.4.24
Truffle	v5.0.8
Node	v8.10.0
Web3.js	v0.20.7, injected by MetaMask
Ganache CLI	v6.4.1, ganache-core: 2.5.3
npm	v3.5.2
lite-server	-
base64url	-
js-sha256	-
u2f-api	injected in Chrome browser
randomstring	-
query-string	-
MetaMask	v6.3.1

For the following sections, we use a private RPC to run tests for the program. The private RPC is provided by Ganache CLI and is listened by MetaMask.

3 Setup

1. **Ganache CLI.** To setup the private RPC, in a terminal run `$ ganache-cli -l 8000000`. Then ten test accounts with 100 ETH each will be returned.

2. **MetaMask.** Open the MetaMask extension in Chrome browser. Set the network to be `127.0.0.1:8545` (localhost) and import the blockchain account using private key. We use the first account that is returned by Ganache CLI in the last step here, as is shown in Figure 2. (Public address: `0x49a591956f78449e15f4B8099CD1509278dB617A`.)

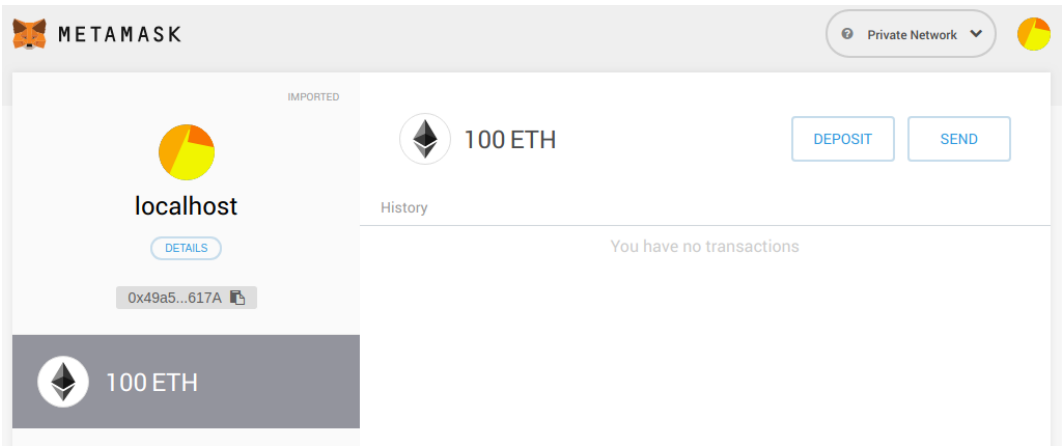

Figure 2: MetaMask Setup.

3. **Server.** In a separate terminal, run `$ npm run dev`. Then the server is setup on `https://localhost:3000` and the starting page should open in the browser.

4 Usage

4.1 Login and Register

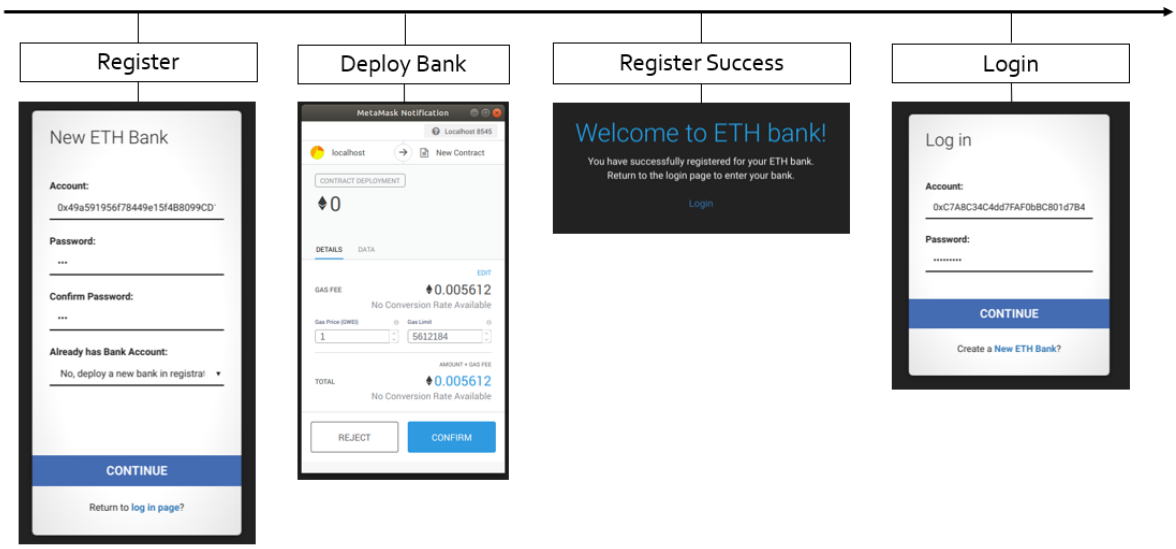

Figure 3: Login and Register.

To get registered,

1. In the register page, input the public address in the *Account* blank and setup the password.
2. Click *Continue*. Then a MetaMask notification should appear. Confirm the transaction in the notification. (There will be 3 transactions for deploying smart contracts in this

test: `Register.sol`, `Authenticate.sol` and `Bank.sol`. In real cases, the first two can be omitted if we already have deployed libraries.)

3. After confirming all transactions, a prompt message indicating successful registration should appear. Then return to the login page, input the public address and password, and click *Continue*. The user will be directed to the homepage of the bank.

4.2 Settings

The basic functions, deposit, withdraw and transfer, are not available before U2F registration. But users can set the transfer limit and the address of the bank contract in the *Info* page.

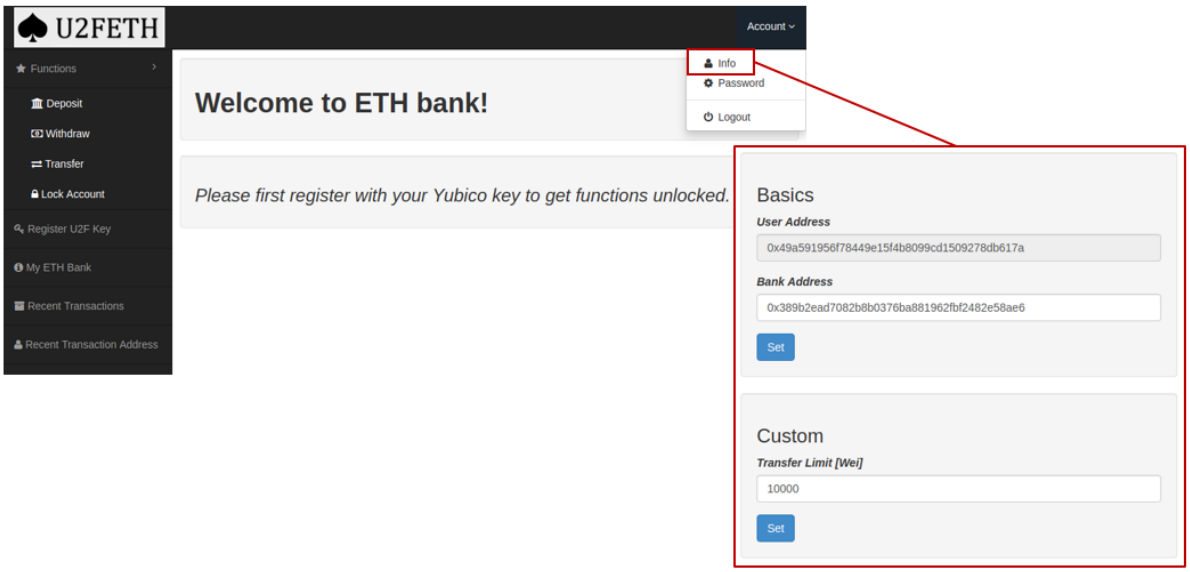

Figure 4: Settings.

To change the address of the bank contract,

1. input the deployed contract address in the *Bank Address* blank;
2. click *Set*;

Then the page will be reloaded (and a new U2F registration is required if the user has already registered the U2F token).

To set the limit,

1. input the customized limit in the *Transfer Limit* blank;
2. click *Set*;
3. a MetaMask notification should appear, confirm the transaction in the notification.

Then the new transfer limit should appear in the *Transfer Limit* blank.

4.3 U2F Registration

To register a U2F device,

1. go to the *Register U2F Key* page and click *Begin*, a MetaMask notification (for begin registration) should appear, confirm the transaction;
2. then a register request record should appear in the bottom, insert the U2F device in a USB port and click *Continue* on that record;
3. at this point, the U2F device should start flashing, tap on the device;
4. then a MetaMask notification should appear (for complete registration), confirm the transaction, and the status on the record should be *Confirmed*, indicating successful registration of the U2F device.

The confirmed registration page is shown in Figure 5.

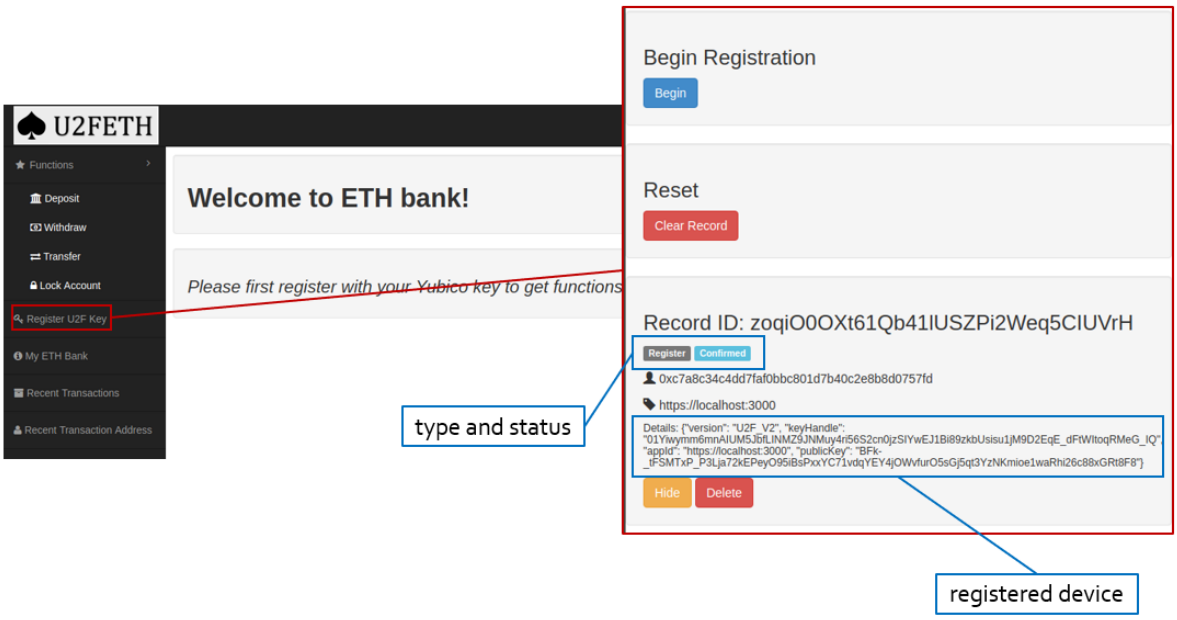

Figure 5: Successful Registration.

4.4 Bank Functions

After U2F registration, the bank functions are available in the *Functions* column, and the pages are shown in Figure 6. If the value involved exceeds the transfer limit, a U2F authentication is required.

To deposit, withdraw, or transfer in the bank,

1. input the amount (and the public address of the recipient for transfer operation) in the blank, if the amount exceeds the transfer limit, make sure that the U2F device has been properly inserted (and the authentication will be completed in a single run).

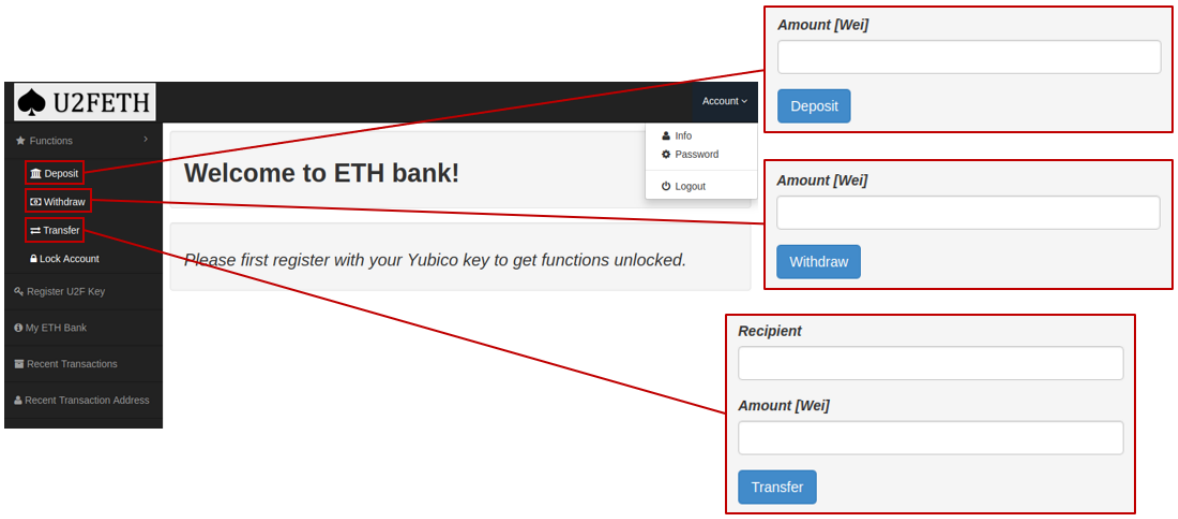

Figure 6: Bank Functions.

2. click the confirm button;
3. if U2F authentication is required, proceed as "confirm transaction on MetaMask → tap U2F device → confirm transaction on MetaMask", and if U2F authentication is not required, this step can be skipped;
4. a MetaMask transaction for the real operation (deposit, withdraw or transfer) is created, confirm the transaction.

Then the deposit, withdraw, or transfer operation should be completed. The transaction history can be viewed on *My ETH Bank*, *Recent Transactions*, and *Recent Transaction Address* page.

4.5 Lock Account

The users can lock their account in the *Lock Account* page. After locking the account, the three bank functions are not available, while the *Info* page is still available. Both locking and unlocking require U2F authentication.

To lock the account, first make sure that the U2F device has been inserted. Click *Lock* and proceed with "confirm transaction on MetaMask → tap on the U2F device → confirm two transactions on MetaMask". Then if the authentication process as normal, the account will be locked. Then the button that previously showed *Lock* becomes *Unlock*.

To unlock the account, first make sure that the U2F device has been inserted. Click *Unlock* and and proceed with "confirm transaction on MetaMask → tap on the U2F device → confirm two transactions on MetaMask". Then if the authentication process as normal, the account will be unlocked and the bank functions will be available again.

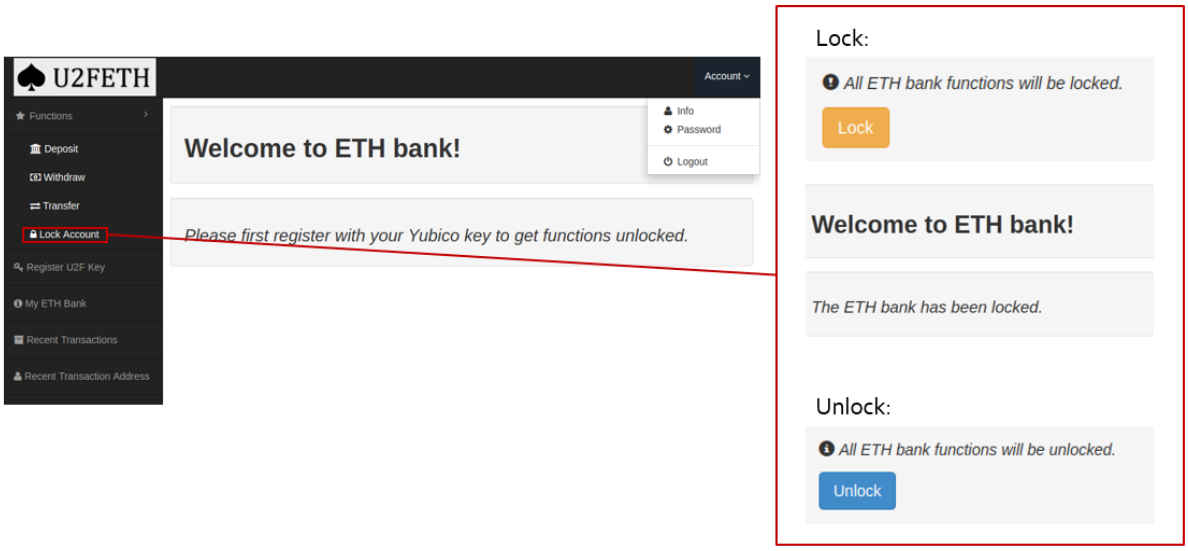

Figure 7: Lock and Unlock Account.

5 Technical Details

In the DApp, the `.html` files make up the interface for the user, and the `.js` files listen to user's actions and are responsible for a part of the computations for the user. To complete the computations, change the states of the smart contract, and emit events on the blockchain, the `.js` files interact with the smart contract and call the corresponding functions. An overview of this structure is shown in Figure 8.

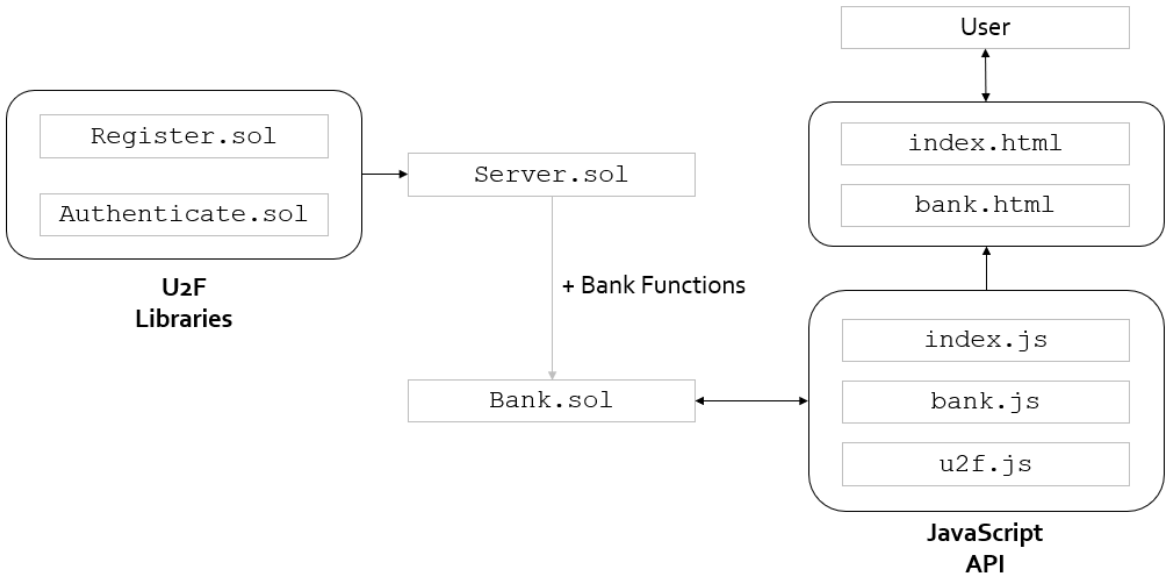

Figure 8: DApp Structure.

5.1 U2F Implementation

When the user click the *Begin* button in the U2F registration page, or starts a transaction with value larger than transfer limit, a U2F register/sign process is initiated. The API of the functions in Javascript and the smart contract is shown in the following subsections. A complete API of `Bank.sol` can be found in Appendix A.

5.1.1 U2F Smart Contract (`Bank.sol`)

```
// ======================== Verify Client Data ========================;
client(clientData, appId, type);
/*
  IN: clientData = {typ, challenge, origin}, appId, type (register/sign);
  OUT: challengeParameter = sha256(clientData);
  EFFECTS: verify same type, same challenge and same appId for the
     request and response, calculate challengeParameter;
*/
// ============================== Register ==============================;
enroll(appId);
/*
  IN: appId = https://localhost:3000;
  OUT: request in JSON string
       {
         challenge = 32 random bytes,
         version = 'U2F_V2',
         appId = origin of url location (https://localhost:3000)
       };
  EFFECTS: generates the 32-byte challenge, store the request string,
     emit begin registration event;
*/
bind(challengeParameter, pubKey, keyHandle, certKey, signature, facet);
/*
  IN: challengeParameter, pubKey (parsed from registrationData),
     keyHandle (parsed from registrationData), certKey (public key
     extracted from certificate), signature (parsed from
     registrationData), facet (origin/appId);
  OUT: registeredDevice in JSON string
    {
      version = 'U2F_V2',
      keyHandle,
      appId,
      publicKey
    };
  EFFECTS: generate the signed message by concatenation of 0x00 :
     appParameter : challengeParameter : keyHandle : pubKey, where
     appParameter = sha256(appId), and verify the signature using ECDSA,
```

```
      store the device information and emit successful registration on
      blockchain;
*/
// ========================== Authenticate ============================;
sign(appId);
/*
  IN: appId = https://localhost:3000
  OUT: request in JSON string
    {
      keyHandle,
      version = 'U2F_V2',
      challenge = 32 random bytes,
      appId = origin of url location (https://localhost:3000),
      publicKey
    };
  EFFECTS: generates the 32-byte challenge, use the device stored for
     the user to generate a request string, store the request string,
     emit begin authentication event;
*/
verify(challengeParameter, signatureData, keyHandle, facet);
/*
  IN: challengeParameter = sha256(clientData), signatureData (from
     response), keyHandle (from response), facet (origin/appId);
  OUT: verification in JSON string
    {
      keyHandle,
      touch = userPresence,
      counter = the number of times the U2F device has signed
    };
  EFFECTS: generate the signed message by concatenation of appParameter
     : userPresence : counter : challengeParameter, where appParameter =
     sha256(appId), and verify the signature using ECDSA, store the
     verification result and emit successful authentication on
     blockchain;
*/
```

5.1.2 U2F Javascript API (`u2f.js`)

```
beginRegistration();
/*
  EFFECTS: listen to user's action of begin registration, get current
     origin of url location, call enroll(origin) from Bank.sol, get and
     store the register request, update web page;
*/
confirmRegistration();
/*
```

```
  EFFECTS:
    1. listen to user's action of complete registration;
    2. get the register request string from Bank.sol;
    3. evoke u2f-api in web browser to get the data signed by U2F device
       and get the returned JSON string response (after websafe decode)
      {
        registrationData,
        clientData
      };
    4. call Bank.sol method client(clientData, origin, 'register') and
       get the returned challengeParameter;
    5. parse the registrationData in the format of
      +-------------------------------+
      | 1 | 65 | 1 | L | implied | 64 |
      +-------------------------------+
      0x05 : pubKey : keyHandleLength : keyHandle : cert : signature
    6. extract certKey from certificate using prefix identifier;
    7. call Bank.sol method bind(challengeParameter, pubKey, keyHandle,
       certKey, signature, origin);
    8. get verfication result and update web page;
*/
beginAuthentication();
/*
  EFFECTS: listen to user's action of begin authentication, get current
     origin of url location, call sign(origin) from Bank.sol, get and
     store the sign request, update web page;
*/
confirmAuthentication();
/*
  EFFECTS:
    1. listen to user's action of complete authentication;
    2. get the sign request string from Bank.sol;
    3. evoke u2f-api in web browser to get the data signed by U2F device
       and get the returned JSON string response (after websafe decode)
      {
        signatureData,
        clientData,
        keyHandle
      };
    4. call Bank.sol method client(clientData, origin, 'sign') and get
       the returned challengeParameter;
    5. the signatureData has the format:
      +-----------------+
      | 1 | 4 | implied |
      +-----------------+
    6. call Bank.sol method verify(challengeParameter, signatureData,
       keyHandle, origin);
```

```
    7. get verification result and update web page;
*/
```

5.2 Bank Functions

5.2.1 Bank Smart Contract (`Bank.sol`)

```
deposit(amount); // unlocked and registered;
/*
  IN: amount of deposit;
  OUT:
  EFFECTS: increment balance, emit event of deposit;
*/
withdraw(amount); // unlocked and registered;
/*
  IN: amount of withdraw;
  OUT:
  EFFECTS: decrement balance, send tokens to owner;
*/
transfer(amount, recipient); // unlocked and registered;
/*
  IN: amount of transfer, recipient's public address;
  OUT:
  EFFECTS: decrement balance, send tokens to recipient;
*/
lock(); // registered;
/*
  EFFECTS: change state to be locked;
*/
unlock(); //registered;
/*
  EFFECTS: change state to be unlocked;
*/
```

5.2.2 Bank Javascript API (`bank.js`)

```
funcDeposit(); // unlocked and registered;
/*
  EFFECTS:
    1. listen to user's request of deposit;
    2. read amount from web page;
    3. if the amount exceeds the limit, evoke U2F authentication process;
    4. call Bank.sol method deposit(amount);
    5. update web page;
```

```
*/
funcWithdraw(); // unlocked and registered;
/*
  EFFECTS:
    1. listen to user's request of withdraw;
    2. read amount from web page;
    3. if the amount exceeds the limit, evoke U2F authentication process;
    4. call Bank.sol method withdraw(amount);
    5. update web page;
*/
funcTransfer(); // unlocked and registered;
/*
  EFFECTS:
    1. listen to user's request of transfer;
    2. read amount and recipient from web page;
    3. if the amount exceeds the limit, evoke U2F authentication process;
    4. call Bank.sol method transfer(amount, recipient);
    5. update web page;
*/
funcLock(); // registered;
/*
  EFFECTS:
    1. listen to user's request of locking account;
    2. evoke U2F authentication process;
    3. call Bank.sol method lock();
    4. update web page;
*/
funcUnlock(); //registered;
/*
  EFFECTS:
    1. listen to user's request of unlocking account;
    2. evoke U2F authentication process;
    3. call Bank.sol method unlock();
    4. update web page;
*/
```

6 Test Data

Using a localhost RPC provided by Ganache CLI and MetaMask for testing, the gas consumption data and costs are shown in Table 2. (We use 1 ETH $\approx$ 177.46 USD.)

Table 2: Experimental Results

Category	Name	Gas Limit [Units]	Gas Used [Units]	Cost [ETH]	USD
Deployment	`Register.sol`	5612184	3741456	0.003741	0.66
	`Authenticate.sol`	7200000	5031960	0.005032	0.89
	`Bank.sol`	6697219	4464813	0.004465	0.79
Settings	Set Limit	40639	27093	0.000027	0.0048
U2F Register	Enroll	1290241	860161	0.00086	0.15
	Bind	7600000	7038845	0.007039	1.25
Deposit	$\leq$ limit	36721	24481	0.000024	0.0043
	$>$ limit	10345809	5899614	0.000590	0.10
Withdraw	$\leq$ limit	48507	32338	0.000032	0.0057
	$>$ limit	10366094	5805285	0.000581	0.10
Transfer	$\leq$ limit	48442	32295	0.000032	0.0057
	$>$ limit	10366482	5814414	0.000582	0.10
Lock Method	Lock	10350543	5827360	0.000583	0.10
	Unlock	10350998	5796939	0.000580	0.10

A Complete API of U2F Implementation

A.1 `Bank.sol`

```
// ================================ U2F ================================;
client(string clientData, string appId, uint typ) public view returns
    (bytes);
enroll(string appId) public onlyOwner;
bind(string appId, string challengeParameter, string pubKey, string
    keyHandle, string certKey, string signature) public onlyOwner;
function sign(string appId) public onlyOwner;
verify(string appId, string challengeParameter, string
    decodedSignatureData, string keyHandle) public onlyOwner;
// ================================ Bank ================================;
setLimit(uint limit) public onlyOwner;
lockBank() public onlyOwner checkSigned;
unlockBank() public onlyOwner checkSigned;
deposit(uint amount) public payable onlyOwner checkReady;
withdraw(uint amount) public payable onlyOwner checkReady;
transferViaBank(address to, uint amount) public payable onlyOwner
    checkReady;
```