Runtime Hook on Blockchain and Smart Contract Systems

Wei-Ting Lin
Management Information Systems
National Chengchi University
Taipei, Taiwan
106356001@nccu.edu.tw

Shun-Wen Hsiao
Management Information Systems
National Chengchi University
Taipei, Taiwan
hsiaom@nccu.edu.tw

Fang Yu
Management Information Systems
National Chengchi University
Taipei, Taiwan
yuf@nccu.edu.tw

***Abstract*—Using hard-fork on the blockchain to recover the losses caused by attacks contradicts the immutable characteristic of a blockchain system. To prevent malicious transactions from getting into blockchains in advance, we propose a runtime hook technique to synchronize ongoing transactions exposed to the Ethereum transaction pool. Having a complete view of ongoing transactions, we are able to identify and enforce abortion of malicious transactions and prevent losses due to attacks being executed and recorded in the blockchain. Specifically, we modify the Ethereum source code to instrument the entry point of a node to synchronize information, import information into our local database, and systematically scan suspicious patterns in transactions to identify potential attacks. As a proof-of-the-concept, we show how to deploy the proposed runtime hook system on a private blockchain system, such that we can detect and prevent transactions of double spending on the 51% attack.**

***Index Terms*—Blockchain, Ethereum, Runtime hook**

I. Introduction

Blockchain is the underlying technology used by Satoshi Nakamoto for creating Bitcoin [1]. Blockchain is a distributed technology that does not rely on trust parties to store and verify data. It forms a peer-to-peer communication network through several decentralized nodes. A blockchain can be viewed as a decentralized database but it works with untrusted P2P nodes to maintain a trusted data repository. When a blockchain stores transaction data, we view it as a decentralized ledger. Anyone who joins the blockchain P2P network can store their data under the same protocol at any time. Whenever a data entry is added, it will be recorded on the ledger through a consensus process among all P2P nodes, and all nodes will synchronize with each other to ensure that all the local ledgers of every nodes are synchronized. In addition, a consensus algorithm will ensure that the recorded entries cannot be modified, so we do not need a trusted third-party entity to keep maintaining the ledger.

A blockchain protocol is a set of rules that specifies how participants (a.k.a. nodes) in the P2P network should validate a new transaction and add it to one of the block. The agreement leverages cryptography, game theory, and economics to motivate nodes to act correctly and run the blockchain collaboratively, rather than attacking the blockchain network. If a blockchain is set up correctly, this system can make accepting an malicious transaction extremely difficult and expensive. However, it is relatively easy to validate and accept a normal transaction. Such secure design makes the blockchain technology so attractive to many industries, especially in finance. The upcoming services from well-known institutions, such as Fidelity Investments, New York Stock Exchange, and Intercontinental Exchange, will begin to integrate their existing financial systems with the blockchain. Even some central banks are considering using blockchain to issue new digital currency (a.k.a. cryptocurrency).

However, the more complex a blockchain system is, the more vulnerabilities and possible errors may exist. On February 5, 2019, the company that is responsible for Zcash cryptocurrency revealed that it secretly fixed a "subtle cryptographic vulnerability" [2] that occasionally appeared in their protocol. Attackers can take the advantage on it to create unlimited fake Zcash. Fortunately, it seems like that actually no attacker did so. Moreover, some high-profile hackers do not attack the blockchain, but they attack the cryptocurrency exchanges. Many cases of looting cryptocurrencies may be attributed to poor basic security measures, such as well-known website vulnerability or misconfiguration. In January 2019, the 51% attack on the Ethereum Classic [3] prompted a change in the robbery.

The 51% attack is most likely to cause a double spending attack. Dual payments are those in which miners who control most of the network's mining capabilities in some way can deceive other users by sending them a transaction and then create an alternate version of the blockchain. Payments for miners who deceive other users in the alternative version have never occurred. This new version of the blockchain is called a fork. An attacker who controls most of the mining capabilities can make the branch a backbone and use the same cryptocurrency again.

However, the cost of attacking a blockchain of the current mainstream cryptocurrency, such as Bitcoin or Ethereum, is very high. According to Crypto51% [4], an attacker needs to spend 270,000 dollars per hour to rent enough mining capacity to attack Bitcoin, and needs 70,000 dollars per hour to attack Ethereum. But the cost of attacking other non-mainstream cryptocurrency is low because the value of the cryptocurrency is low. Since the value is low, the number of miners is relatively small; so the blockchain network is less protected. It is easier for the attacker to perform 51% attack on these lower-value cryptocurrencies. David Vorick, the co-founder of blockchain-based file storage, Sia, predicts that 51% attack will continue to grow in frequency and severity.

Attacks on blockchains are often found and caused actual losses. Cryptocurrency holders usually only count on hard-fork to recover the losses. However, it violates the design of a blockchain, which is that a blockchain should be immutable. If we can find some solutions to prevent attacks or detect attacks before losses get on chains, we can reduce hard fork risks. One of the approach to observe the operation of the blockchain is to connect the observer from the outside to the blockchain by using web3.js or ethclient API. However, the outside observer has delay to reveal an attack within the blockchain. In this paper, we plant our observer in the Ethereum's EVM (Ethereum Virtual Machine) and collect every transaction from a node before it gets into the transaction pool (i.e., before it gets mined). We also synchronize information of blocks. Having these information stored in our local database, we are able to scan malicious patterns and detect possible attacks in advance.

In sum, we provide a runtime hook mechanism to collect and scan all online transactions before they are on-chain. Users (victims) can be notified in advance when a potential transaction gets aborted or an attack has been detected, so that they can avoid expectations on transactions not to be taken on chain in the real world.

II. Related Works

A. Blockchain monitoring

The method of monitoring a blockchain [5] is usually using the blockchain client API interface to retrieve the data in the blockchain.

There are a number of approaches [6]–[10] to limiting inappropriate blockchain transactions. One approach [9] is to filter and discard a transaction before the transaction being committed to the blockchain, and possibly redact transactions from blockchain. Another approach [6]–[8], [10] is to redact a transaction after it has mined in the blockchain. It will not change the immutability characteristic of the blockchain, but these approaches limit the accessibility to the transaction or they hide the transaction. So that even though the transaction still exits in the blockchain, it is not accessible or viewable by the users.

IBM [9] applied for a patent on monitoring blockchain that uses a combination of hardware and software to maintain the security of the blockchain. The detection is based on the word filter, media filter, and Custom filter set by the administrator and other users. The Content processor will receive and process the transaction content and determine if the content should be added to the blockchain. If there is no problem with the content, it will be added directly to the blockchain. If the content causes a potential attack, the content is redacted by the redaction contract set in the genesis block. Similar to deploying a custom filter, we inspect nonce, from address and to address of transactions to detect double spending in Ethereum. In addition to redacting transactions, our design informs the to address of redacted transactions. We further generate an event log that lists information of aborted transactions. The event log can be searched effectively.

B. Blockchain attacks

The problems that blockchains often encounter are roughly divided into nine categories [11].

1) $51\% attack$: The 51% attack [12] occurs when the blockchain adopts Proof of Work for consensus. If a miner or group of miners have more than 51% of the total computation power, then the miner can deliberately select preferred transactions from the transaction pool to generate new blocks [13]. Therefore, such an attacker can trigger double spending attack [14]. In addition, the 51% attacker can generate a longer branch that enforces the blocks and transactions in the shorter branch need to be re-mined.
2) $Double\ spending$: Double spending is an attack that a customer uses the same cryptocurrency token multiple times. [15].
3) $Private\ Key\ Security$: Users private key is regarded as the identity and security credential. Hartwig et al. [16] discover a vulnerability in ECDSA (Elliptic Curve Digital Signature Algorithm) scheme, through which an attacker can recover the users private key and use that to obtain illegally income.
4) $Criminal\ Activity$: Since the usage of Bitcoin is anonymous, Bitcoin has been used in illegal activities, such as ransomware [17], underground [18] and money-laundering [19].
5) $Vulnerabilities\ in\ smart\ contract$: Nicola et al. [20] conduct a systematic investigation of 12 types of vulnerabilities in smart contract code. A famous example of the attack is TheDAO [21].

In this paper, we focus on the first two categories, i.e., 51% attack and double spending. The reason why we focus on the categories is that they don't have much to do with the real world and they attack the blockchain to make a profit rather than use the blockchain as part of criminal activities.

C. Background: Double spending

A double spending attack of blockchain [22] may work as follows.

1) Attacker A requires a product or a service provided by Victim B.
2) A creates two transactions spending the same digital token. One of the transactions is paid to B and the other one is paid to A. The only difference between the two transactions is the recipient address, and all the other fields are the same.
3) A broadcasts the "A to B" transaction to other mining nodes, and then secretly mines another block (B_s). When B_s is successfully mined, A will continue to mine

new blocks and append them to B_s. A plans to include "A to A" transaction in this branch to cause double spending.

4) B delivers the product or service to A after "A to B" transaction has been issued. However, B does not wait for enough confirmations [23] before delivering.
5) A has enough computing power to generate a longer branch than the original branch so that the transactions in the original branch will be re-mined and then be included to the longer branch.
6) However, the token of "A to B" transaction has been spent by "A to A" transaction previously. Hence, "A to B" transaction will be discarded by nodes. That is, actually B does not receive the token while the goods or the services has been delivered.
7) When B realizes that the payment is discarded, A might left the scene.

III. Research Method

Our analysis framework consists of four steps: 1) Tracing transactions on Ethereum, 2) Hacking entry points, 3) Information collection and 4) Property checking. In the first section, we introduce the process of Ethereum's transaction. The second section will show how we hook on the Ethereum node and where we obtain all information from the block, transaction, and receipt. The third section introduces the information we obtain in the last section. The last section shows how we use the information to detect a potential attack.

A. Ethereum: Transaction-based

Ethereum can be seen as a transaction-based state machine. It starts with a genesis state and then gradually changes to the final state through transaction processing. The information contained in the status is address, account balance, input data, etc.

Figure 1 simply describes the process of launching a transaction into a block or broadcast to another node. The new Transaction is issued by the Ethereum user and sent to the transaction pool after signing and submission. After new transaction entering the transaction pool, the node check whether transaction pool already has an old transaction that its nonce and from address are the same as new transaction's.If there is an old transaction, node would check whether the gas price of new transaction is higher than the old one. If the gas price of new transaction is higher, node would discard the old transaction and add the new one into the Pending. If there is no old transaction having same nonce and same from address in Pending, the node would check if there is an old transaction having same nonce and same from address in Queue. If there is, node would check if the gas price of the new transaction is higher than the old one. If so, node would discard the old one and add the new one to Queue. If there is no same nonce and from transaction in Queue, node would add the new transaction into Queue directly. The transaction pool then broadcasts the new transaction to other nodes and sends the event for the new transaction to the Worker if the node is mining. The worker

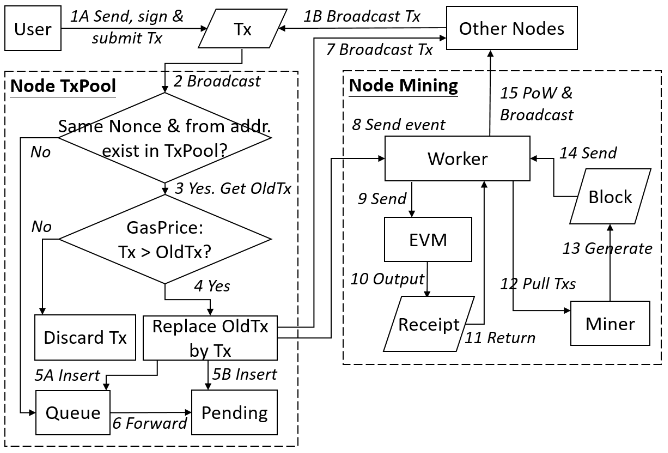

Fig. 1. Transaction process

uses the EVM to execute the transaction and obtain Receipt. Miner collects the transactions through the worker and tries to generate a new block. If a new block is generated, it would be sent to the worker. The worker does Proof of Work and the new block will be broadcast to other nodes.

Other nodes will decrypt the received blocks to obtain transactions when node synchronizes. Then transactions will be converted into message objects. The node then builds the EVM and passes the message objects to the EVM to execute. EVM is an emulation environment created by Ethereum during executing transactions. EVM can calculate the consumption of Gas, and create a receipt object and return it after the transaction is executed completely. There are three types of the operation represented by the transactions:

1) $Ether\ transferring$: User transfers Ether between two addresses.
2) $Creating\ smart\ contract$:User creates a smart contract address.
3) $Calling\ smart\ contract$:User calls smart contracts function. Transaction's data contains function name and input data.

If the transaction is transferring Ether, EVM would directly modify the corresponding account balance in the StateDB. If the transaction is a smart Contract creation or calling smart contract, EVM would load and execute the byte code of the contract and the StateDB is queried or modified during transaction execution. The EVM responses for verifying the correctness of the state. After EVM confirming the state is correct, the message is stored from the StateDB of the EVM into the StateDB of the node. In the future, if a new node requests synchronization, the node's StateDB is provided for the new node.

B. Entry point searching

1) Sync procession: When running the node, our node will request synchronization to the nearby node. During the synchronization procession, node will sync the blocks and then

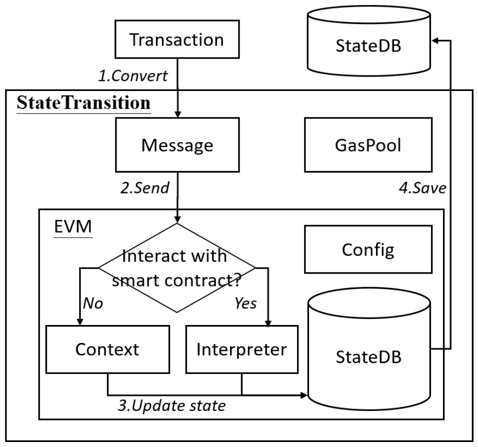

Fig. 2. EVM execute transactions

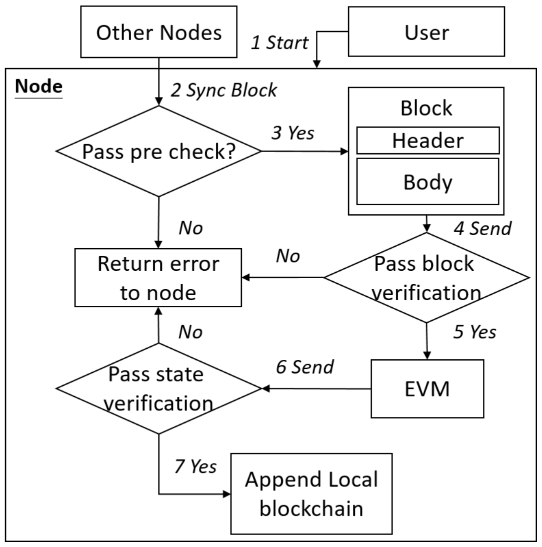

Fig. 3. Sync process

node will obtain the interaction with the smart contract or the Ether transferring from the transactions in the blocks.

So we search where the node executes all transactions. In order to do this, we are using normal sync (i.e., full sync) instead of fast sync when doing the blockchain synchronization.

Figure 3 is a blockchain synchronization flowchart traced from the target log. When an user starts the node, the node sends a synchronization request to other nodes on the network. After receiving the blocks from other nodes, node does a pre-check to ensure that the blocks to be inserted are ordered link. After pre-check is passed, the node will verify the header and body of received block. After the previous verification is completed, the status is read from the parent block and the transactions contained in the blocks are executed by Process function's EVM. EVM will update the state and verify the updated state to check whether it is consistent with the header. If the verification is successful, node would append blocks to the blockchain.

According to the previous synchronization process, we can know that the code of executing all transactions is in the Process function. The Process() calls ApplyTransaction() to execute the transactions in the blocks. The state stores information about each account, such as account balance and contract code (for contract accounts only).

```
func (p *StateProcessor) Process(block *types.Block,
statedb *state.StateDB, cfg vm.Config)
(types.Receipts,[]*types.Log,uint64,error) {
    ......
    for i, tx := range block.Transactions() {
        receipt, _, _ := ApplyTransaction(p.config,
        p.bc, nil, gp, statedb, header, tx, usedGas,
        cfg)
        receipts = append(receipts, receipt)
        allLogs = append(allLogs, receipt.Logs...)
    }
    ......
}
}
```

Listing 1. Process()

The ApplyTransaction() does the follows.

1) ApplyTransaction() calls AsMessage() to generate core.Message with tx. The implementation is to store some fields in tx into Message and decrypt the tx sender from the digital signature of tx.
2) Then it calls NewEVMContext() to create an EVM to execute vm.Context
3) ApplyTransaction() calls NewEVM() to create an EVM. Its main function is to assemble the previous information and build a code interpreter, which will be used to interpret and execute the contract code.
4) Finally, it calls ApplyMessage() to apply the above information to the current Ethereum's state and generate the receipt object.

```
func ApplyTransaction(config *params.ChainConfig,
bc ChainContext, author *common.Address, gp *
    GasPool,
statedb *state.StateDB, header *types.Header,
tx *types.Transaction, usedGas *uint64, cfg vm.
    Config
)(*types.Receipt, uint64, error) {
 msg, err := tx.AsMessage(types.MakeSigner
 (config, header.Number))
 if err != nil {
   return nil, 0, err
 }
 context := NewEVMContext(msg, header, bc, author)
 vmenv := vm.NewEVM(context, statedb, config, cfg)
 _, gas, failed, err := ApplyMessage(vmenv, msg,
 gp)
    ......
 receipt.TxHash = tx.Hash()
 receipt.GasUsed = gas
 if msg.To() == nil {
   receipt.ContractAddress =
   crypto.CreateAddress(vmenv.Context.Origin,
   tx.Nonce())
 }
 receipt.Logs = statedb.GetLogs(tx.Hash())
 receipt.Bloom =
 types.CreateBloom(types.Receipts{receipt})
 return receipt, gas, err
}
```

Listing 2. ApplyTransaction()

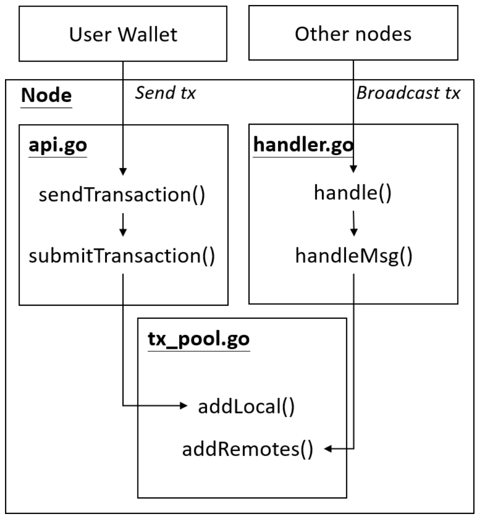

Fig. 4. Receiving transaction process

When the state machine is changed, the Receipt lists and associated logs are returned for the next stage. So we insert our monitoring program before ApplyMessage() and check it before changing the Ethereum's state.

2) Transaction pool: According to Fig. 1, the transaction pool is a collection of all transactions that have not been mined in the block. Whether it is a transaction initiated by a local node or a transaction broadcast by other nodes, all transactions that can be processed and not processed can be stored here. Once transactions are mined into the block, they are removed from the transaction pool.

Figure 4 indicates the process by which Txpool receives transactions which are sent from the local node and other nodes. When the user initiates a transaction at the local node, it will encapsulate the transaction information through json. Then convert it into golang through json-RPC and pass it into sendTransaction(). sendTransaction() will use the sender to get the relative wallet, then generate the transaction structure object based on the transaction information, and finally sign the transaction. After the signature is completed, the node submits the transaction through the submitTransaction() to addLocal() to add the transaction into the TxPool. The transaction event sent by other nodes will be listened to by handle(). When there is a new transaction event, node calls handleMsg() to process and sends the transaction to addRemotes() to add it into the TxPool.

C. Data introduction

1) Block: A block in a blockchain is that stores valuable information. This is the essence of any kind of cryptocurrency.

The Header is the core of the Block. Its member variables are all public, and it is convenient to provide the caller with operations on the Block property. The terminology and notions used in this paper is the same as the notions used in the official yellow paper [24].

2) Transaction: The following is the structure of txdata in Transaction, which is the Transaction information we can get. There is no initiator of the transaction here because the initiator can get the information by signing:

1) $AccountNonce$: The number of transactions the sender has sent.
2) $Price$: The gas cost of this transaction.
3) $GasLimit$: This transaction allows the maximum amount of gas to be consumed.
4) $Recipient$: The recipient of the transaction.
5) $Amount$: The Ether carried by the transaction.
6) $Payload$: The data carried by the transaction.
7) V, R, S: The signature of the transaction.

3) Receipt: Finally, Receipt, the record information after the Transaction is completed:

1) $PostState$: Stores the current state of all accounts in the entire block when the Received object is created.
2) $Status$: The status of whether the execution was successful.
3) $CumulativeGasUsed$: The total amount of gas spent in the implementation of this Transaction in Block.
4) $Bloom$: Used to verify that a given Log is in the existing Log array of Receipt.
5) $Logs$: Log-type arrays, where each Log object records a small step in the Transaction.
6) $TxHash$: This transaction's hash.
7) $ContractAddress$: The address of the smart contract.
8) $GasUsed$: How much gas this Transaction costs.

D. Property check: double spending

This section attempts to use the collected data to detect if a double spending attack can be observed.

There are two types of situations in which an attacker can cause double Spending:

1) Sending 2 Transactions in different time: The attacker first sends a transaction to the victim. When the transaction is confirmed by the victim but has not yet been mined, the attacker then sends another transaction that uses the same digital token with a higher gas price to the address of the attacker. The miner node will preferentially mine the subsequent transaction because there is more gas price. The blockchain will have the transaction to the attacker, and the victim will not receive any digital token.
2) With 51% attack: The attacker sends 2 transactions at the same time, and the content of transactions is the same except the recipient. When the victim's transaction is mined, the attacker privately prepares another malicious fork to mine another transaction. After the length of the malicious fork exceeds the original fork length, the malicious fork will replace the original chain as the main chain, and the victim's transaction will be discarded so that the victim does not get the digital token in the transaction.

We can observe the second type of double spending after it happened. So we try to detect the first type of double spending.

We set the following rules for the first type of double spending before the attacks happen.

1) There is an old transaction in the transaction pool.
2) If the new transaction has same nonce, same from address, and its to address is same as from address, the sender may want to replace the old transaction and don't send the ether.
3) If the new transaction with the same nonce, same from address, and its to address is different with from address, the sender may have incorrect information like wrong to address in the old transaction and want to replace it or the sender may want to cause double spending.

Our model has the following assumptions: 1) The system is used in private chain system. 2) All participating nodes use this version of the program. 3) Wallet cannot exist without a node. 4) There is a smart contract which is used to use event logs to record replaced transaction in genesis block. According to these assumptions, the nodes should comply with the rules we set.

We insert the model into the code of transaction replacement. Fig. 5 is the monitor model. The steps of the model are as follows:

1) The new transaction is sent from the User's Wallet to the node via the Ethereum client API or from another Ethereum node.
2) The new transaction checks whether it already exists in the transaction pool or fails to pass the basic validation before entering the transaction pool. New transactions that already exist or fail to pass basic validation are discarded directly. If transactions that do not exist and pass basic validation, node proceeds to the next step.
3) Node checks whether transaction pool already has an old transaction that its nonce and from address are the same as new transaction's. If not, node inserts it directly into the transaction pool and save it in the database. If so, node goes to the next step.
4) Node checks whether the from address and the to address of the old transaction are the same. If so, node ignores the limit of gas price and directly replaces the old transaction with new one. If not, node does the next step.
5) Node checks whether the from address and to address of the new transaction are the same. If they are the same, node discards the new transaction directly, if not, node proceeds to the next step.
6) Node checks whether the gas price of the new transaction is higher than the old one. If not, discard new transaction directly. If new transaction's gas price is higher, the node sends an alert transaction to the old to address to notify that the transaction is replaced and calls deployed smart contract to use event logs to record the replaced transaction. Then node replaces old transaction with new transaction.
7) After the replacement is complete, the new transaction is inserted into the transaction pool and recorded in the database.

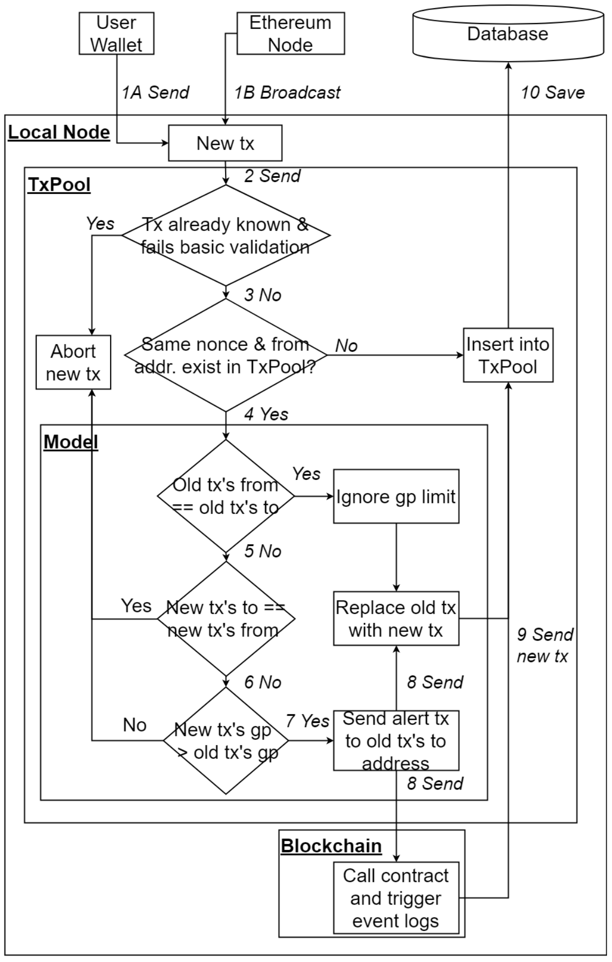
User Wallet
Ethereum Node
Database
1A Send
1B Broadcast
10 Save
Local Node
New tx
2 Send
TxPool
Yes
Tx already known & fails basic validation
3 No
Abort new tx
Same nonce & from addr. exist in TxPool?
No
Insert into TxPool
4 Yes
Model
Old tx's from == old tx's to
Yes
Ignore gp limit
5 No
Yes
New tx's to == new tx's from
Replace old tx with new tx
9 Send new tx
6 No
8 Send
No
New tx's gp > old tx's gp
7 Yes
Send alert tx to old tx's to address
8 Send
Blockchain
Call contract and trigger event logs

Fig. 5. Research model

IV. Experiments

We use the newly created private chain and database to experiment with the model. This private chain has 4 addresses. The first address is sender which is as known as eth.coinbase, the second address is smart contract and the other addresses are receivers.

1) Sender: 0x07e7c8904f8b6cab9cb4b0b9393d...80f2
2) Smart contract: 0xfcf1538ab751126e47641b4d...c579
3) Receiver A: 0x884d0838de824345653c81b72ca5...1aa8
4) Receiver B: 0x5d0d08f8061c31dc179882cf153c...69c1

In the beginning, there were no transactions in the transaction pool:

```
> txpool.content
{ pending: {}, queued: {} }
```

Listing 3. Listing transactions in transaction pool

We first send a new transaction that from address is sender, to address is sender, and the nonce is 1. At this time, because there is no transaction that nonce and from address is the same as the new one in the transaction pool, the new transaction is directly added to the transaction pool.

```
> eth.sendTransaction({from: eth.coinbase, to:eth.
    coinbase, value: web3.toWei(5, "wei"), nonce:
    1})
INFO [05-28|10:44:55.703] Submitted transaction
                             fullhash=0
    x7289275d6b7432c0782aba00a753cff7340a47445b85f4
f89fffd38938b87453
recipient=0x07E7C8904F8b6cab9cb4B0B9393Dd767289E80f2
"0x7289275d6b7432c0782aba00a753cff7340a47445b85f4f89
fffd38938b87453"
> txpool.content

{
 pending: {
  0x07E7C8904F8b6cab9cb4B0B9393Dd767289E80f2: {
   1: {
    from: "0x07e7c8904f8b6cab9cb4b0b9393d...80f2",
    nonce: "0x1",
    to: "0x07e7c8904f8b6cab9cb4b0b9393d...80f2"
   },
```

Listing 4. Add transaction to the transaction pool

Then we send the second transaction that from address is sender, to address is receiver A, the nonce is 1 and the gas price is the same. The model enters the transaction pool and detects the transaction with the same nonce and from address, and starts the comparison of to address and from address between the new transaction and the old transaction. After the model comparison is completed, the to address and from address of the old transaction are found to be the same, so the old transaction is directly replaced by the new transaction and the restriction of the gas price is ignored.

```
> eth.sendTransaction({from: eth.coinbase, to:"0
    x884d0838de824345653c81b72ca5516aabb01aa8",
    value: web3.toWei(6, "wei"), nonce: 1})
Old transaction's to and from the same, rep, caption
    ={Add transaction to the transaction pool}]lace
     it without gasPrice.
INFO [05-28|10:49:56.220] Submitted transaction
                             fullhash=0
    xced4d0a086bc215102e082a9c66cc20cc0e187004652f86
4c2efd749b51f85c6
recipient=0x884D0838dE824345653C81B72Ca5516aABb01Aa8
"0xced4d0a086bc215102e082a9c66cc20cc0e187004652f864
c2efd749b51f85c6"

> txpool.content
{
 pending: {
  0x07E7C8904F8b6cab9cb4B0B9393Dd767289E80f2: {
   1: {
    from: "0x07e7c8904f8b6cab9cb4b0b9393d...80f2",
    nonce: "0x1",
    to: "0x884d0838de824345653c81b72ca5...1aa8"
   },
```

Listing 5. Replace old transaction

We then send the third transaction that from address is sender, to address is sender, the nonce is 1 and higher gas prices, trying to replace the previous transaction. After the model comparison is completed, the to address and from address of the new transaction are found to be the same, so the new transaction is directly aborted from replacing the old transaction. We look at the transaction pool and confirm that it has not been replaced.

```
> eth.sendTransaction({from: eth.coinbase, to:eth.
    coinbase, value: web3.toWei(6, "wei"), nonce: 1,
    gasPrice:10000000000000})
New transaction's to and from the same, abort it.
INFO [05-28|10:57:25.817] Served eth_sendTransaction
                         reqid=43 t=776.409  s   err="
    replacement transaction underpriced"
Error: replacement transaction underpriced
    at web3.js:3143:20
    at web3.js:6347:15
    at web3.js:5081:36
    at <anonymous>:1:1
> txpool.content
{
 pending: {
  0x07E7C8904F8b6cab9cb4B0B9393Dd767289E80f2: {
   1: {
    from: "0x07e7c8904f8b6cab9cb4b0b9393d...80f2",
    nonce: "0x1",
    to: "0x884d0838de824345653c81b72ca5...1aa8"
   },
```

Listing 6. Confirm transaction replacement

Finally, we send the fourth transaction that from address is sender, to address is receiver B, the nonce is 1 and higher gas price, trying to replace the previous transaction. After the model comparison is completed, the to address and from address of the new transaction are found to be inconsistent, so the new transaction is checked for the gas price and then replaced if it passes the check. At this point, the program will send a alert transaction to the old to address as a notification.

```
> eth.sendTransaction({from: eth.coinbase, to:"0
    x5d0d08f8061c31dc179882cf153c1b2a815969c1",
    value: web3.toWei(8, "wei"), nonce: 1,gasPrice
    :10000000000000})
INFO [05-28|10:58:41.441] Submitted transaction
                         fullhash=0
    xe1b51eff65383f4e56971af41653c940f9403b59f9d1f
48f6dee0955df95c19e
recipient=0x5d0d08F8061c31Dc179882cF153C1b2a815969C1
"0xe1b51eff65383f4e56971af41653c940f9403b59f9d1f
48f6dee0955df95c19e"

> txpool.content
{
 pending: {
  0x07E7C8904F8b6cab9cb4B0B9393Dd767289E80f2: {
   1: {
    from: "0x07e7c8904f8b6cab9cb4b0b9393d...80f2",
    nonce: "0x1",
    to: "0x5d0d08f8061c31dc179882cf153c...69c1"
   },
   2: {
    from: "0x07e7c8904f8b6cab9cb4b0b9393d...80f2",
    nonce: "0x2",
    to: "0x884d0838de824345653c81b72ca5...1aa8"
   },
   3: {
    from: "0x07e7c8904f8b6cab9cb4b0b9393d...80f2",
    nonce: "0x3",
    to: "0xfcf1538ab751126e47641b4d36ad...c579",
    hash: "0x38ea75ac52dd619a513b93be26...6ad3",
    input: "0xf9fbd554000000000000000b...3b588"
    },
```

Listing 7. Replacement notification and smart contract invocation

We can see the event logs in the transaction after the transaction being mined. The value in topics can be used as a filter. The first parameter in topics field is this event's identifier. The second parameter in the topics field is the filter we set and we can use this filter to find all transactions that have the same event logs' filter. The value in the data field is general data and we save the replaced transactions' hash value.

```
> eth.getTransactionReceipt("0
    x38ea75ac52dd619a513b93be264ebdd07b9005c6f...6
    ad3")
{
  from: "0x07e7c8904f8b6cab9cb4b0b9393dd767289e80f2"
      ,
  logs: [{
      data: "0x0000000000000000000000000000000000000
          ...0000",
      topics: ["0
          x61aef8cd79541943c29722c77e582aa0e9b580c7
          ...3280", "0
          xa728aa3d202cfdae8bc312089ffc2e631127b1...
          b298"],
  }],
  to: "0xfcf1538ab751126e47641b4d36ad281be217c579",
  transactionHash: "0x38ea75ac52dd619a513b93be26
      ...56ad3",
}
```

Listing 8. Transaction event logs

V. Conclusion

Although Ethereum has a certain degree of protection against double spending attacks, it can still see the attack happens. The attacker uses the 51% attack or the replacement transaction to discard the original transaction to cause a double spending attack. Because of the PoW consensus mechanism, the 51% attacks can only be observed afterward and cannot be completely blocked beforehand. Therefore, the double spending attack generated by the 51% attack cannot be completely avoided. Since these attacks on Ethereum do not violate any internal rules, the attacks can only be detected after they have occurred, so we hope to notify potential victims as soon as possible when their transactions have been replaced. We use the database in the private chain environment to record all the replacement information. The model first discards transactions that may be problematic when the transaction enters the transaction pool. Then the model sends a transaction with a warning message to the to address of the replaced transaction so that the user can know the potential attack as early as possible without surrendering products and services.

References

[1] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] "Zcash counterfeiting vulnerability successfully remediated." https://z.cash/blog/zcash-counterfeiting-vulnerability-successfully-remediated/.

[3] "Deep chain reorganization detected on ethereum classic (etc)." https://blog.coinbase.com/ethereum-classic-etc-is-currently-being-51-attacked-33be13ce32de.

[4] "Pow 51% attack cost." https://www.crypto51.app/.

[5] "Etherscan api." https://etherscan.io/apis.

[6] G. Ateniese, B. Magri, D. Venturi, and E. Andrade, "Redactable blockchain–or–rewriting history in bitcoin and friends," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 111–126, IEEE, 2017.

[7] D. Deuber, B. Magri, and S. A. K. Thyagarajan, "Redactable blockchain in the permissionless setting," *arXiv preprint arXiv:1901.03206*, 2019.

[8] I. Puddu, A. Dmitrienko, and S. Capkun, "μchain: How to forget without hard forks.,"

[9] S. Anderson and B. Q. Nguyen, "Filtering and redacting blockchain transactions," 2018. US Patent App. 15/348,581.

[10] M. Florian, S. Beaucamp, S. A. Henningsen, and B. Scheuermann, "Erasing data from blockchain nodes," *CoRR*, vol. abs/1904.08901, 2019.

[11] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Generation Computer Systems*, 2017.

[12] I.-C. Lin and T.-C. Liao, "A survey of blockchain security issues and challenges.," *IJ Network Security*, vol. 19, no. 5, pp. 653–659, 2017.

[13] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, pp. 95–102, June 2018.

[14] G. O. Karame, "Two bitcoins at the price of one? double-spending attacks on fast payments in bitcoin," in *In Proc. of Conference on Computer and Communication Security*, 2012.

[15] G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Čapkun, "Misbehavior in bitcoin: A study of double-spending and accountability," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, p. 2, 2015.

[16] H. Mayer, "Ecdsa security in bitcoin and ethereum: a research survey," *CoinFaabrik*, 2016.

[17] "Wannacry ransomware attack." https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[18] N. Christin, "Traveling the silk road: A measurement analysis of a large anonymous online marketplace," in *Proceedings of the 22nd international conference on World Wide Web*, pp. 213–224, ACM, 2013.

[19] "Uk national risk assessment of money laundering and terrorist financing." https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/468210/UK_NRA_October_2015_final_web.pdf.

[20] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts.," *IACR Cryptology ePrint Archive*.

[21] "The dao (organization)." https://en.wikipedia.org/wiki/The_DAO_(organization).

[22] C. Pinzón and C. Rocha, "Double-spend attack models with time advantange for bitcoin," *Electronic Notes in Theoretical Computer Science*, vol. 329, pp. 79–103, 2016.

[23] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, pp. 22–23, 2013.

[24] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.