Received January 5, 2021, accepted January 15, 2021, date of publication February 2, 2021, date of current version February 11, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3056623

A Protocol for Preventing Transaction Commitment Without Recipient's Authorization on Blockchain and It's Implementation

YOKO KAMIDOI[1], (Member, IEEE), RYOUSUKE YAMAUCHI[2], AND SHIN'ICHI WAKABAYASHI[1], (Member, IEEE)
[1]Graduate School of Information Sciences, Hiroshima City University, Hiroshima 731-3194, Japan
[2]Faculty of Information Sciences, Hiroshima City University, Hiroshima 731-3194, Japan

Corresponding author: Yoko Kamidoi (yoko@hiroshima-cu.ac.jp)

ABSTRACT In recent years, blockchain is utilized practically as a distributed secure digital ledger of some sorts of transactions. Blockchain is regarded as one of the most important next generation infrastructure technologies of the financial industry, as well as artificial intelligence and big data. In 2020, cryptocurrencies based on blockchain, such as Bitcoin, Ethereum, or XRP, have a value of more than $450 billion in the market capitalization. Furthermore, on blockchains such as Ethereum, transactions can also represent automatic executions of programs, which are called smart contracts. Thus, many institutes in various categories show their positive attitude toward processing financial transactions or non-financial contracts on blockchain. Although many researchers have studied for various types of issues on blockchain, there always exist security and privacy concerns for blockchain. In this paper, we point out a new concern for abusing the publicity of blockchain and also show the possibility of suspicions aroused by the concern. Then we propose a selective mechanism for self-protecting against the approach from crimes or computer viruses on blockchain, whether the disclosure of user's privacy occurs or not. Next, we also propose a concrete implementation of our proposed selective mechanism with two new address types. We aim to incorporate the mechanism in Bitcoin Core, which is the official Bitcoin client software, and using libbitcoin library functions for Bitcoin software development. We show experimental results to estimate overhead costs for processing our proposed address types toward processing the current standard address type in nodes on the peer-to-peer network.

INDEX TERMS Blockchain, abuse of publicity, recipient's authorization, security, implementation.

I. INTRODUCTION

Emergence of the blockchain protocol proposed by Nakamoto [13] had introduced successful cryptocurrencies such as Bitcoin, Ethereum, XRP etc. Blockchain technologies have developed new applications, which have never been previously practical [20].

Before emergence of blockchain, although payments by digital cash can be authenticated by using digital signature, the double spending problem of easily replicable digital cash could not be solved without a trusted third party. On the one hand, the blockchain technology proposed in the article [13] by Nakamoto, achieves a break-through by solving the double spending problem without the trusted third party.

Blockchain keeps consistency of a decentralized digital ledger by utilizing the special consensus algorithm, called Proof-of-Work. The concept of Proof-of-Work was developed, based on Hashcash as a denial-of-service counter measure [3]. The idea of introducing the cryptographic based measure as a consensus algorithm on full distributed systems, is the key to get trust for the ability to make payment non-reversible.

By the emergence of blockchain, the reliability of digital cash has rapidly increased, and digital cash on blockchain has been called cryptocurrency. The market capitalization of successful cryptocurrencies is over $450 billion in 2020 [6]. On Ethereum blockchain, the executions of smart contracts

The associate editor coordinating the review of this manuscript and approving it for publication was Patrick Hung.

have attracted considerable attention. The concept of smart contracts was developed by Szabo in 1997 [16], before the emergence of blockchain. Although Bitcoin blockchain supported smart contracts before the start of Ethereum, Ethereum's smart contract are major because of the ability to describe contracts as the high level Turing-Complete Language, called *Solidity* [4]. Smart contract is a transaction so that conditions for the transfer of value are defined as a program and when the conditions are satisfied, the transition of value will be executed automatically.

There exist two benefits of processing smart contracts on blockchain. One benefit is that contracts between untrusted parties can be processed safely whether each party is trust-worthy or not. This is because the peer-to-peer network justifies the validities of the contracts clearly from the third party point of view. The other benefit is to be able to reduce the cost because of the automatic executions of contracts without a trusted third party. Financial firms, consulting firms, IT venders and governments have attracted an application area of smart contracts on blockchain, i.e., willings, an electrical vote, transfer of property rights and proof of a production area. But, there exist several well-known issues of blockchain, such as the majority occupation attacks called the 51% attacks, the harm problem for decentralization by mining pools [14], etc. In addition, it was pointed out that crypocurrencies have been transferred on illicit online sites called "darknet" marketplaces [8]. We should understand the risks of involvements in criminal cryptocurrency transactions.

Blockchains of Bitcoin or Ethereum, where anyone can participate, are called public blockchains. On the one hand, blockchains where only permitted members, but each other may be competitors, can participate, are called permitted blockchains. Additionally, blockchains where members in one organization can only participate, are called private blockchains. Whereas permitted blockchains or private blockchains can protect privacy in transactions strongly and renew data easily, attacks for consistency of digital ledger, i.e. 51% attack, might success easier than attacks in public blockchains. Thus, their blockchain could give a weaker proof that the records are consistent.

Although blockchains have been implemented and become practical, privacy and security of those have been concerned continuously [19]. Thus, issues, which are not critical in transactions with the trusted third party, might induce significant problems.

In this paper, we explain a concern pointed out in our preliminary version [18] of this paper on blockchain caused by abusing public information. We especially focus on the recipient's information of a transaction, while most previous works focus on sender's security, because of resulting in loss of value directly. We show possibilities of frauds for recipients caused by the pointed out concern. Next, we present a new mechanism for recipient's self-protection from these frauds. This mechanism consists of a new protocol for transaction publication and a new structure of transactions. Furthermore, we present a concrete implementation of our selective mechanism for Bitcoin Core client software and analyze the security of our implementation. In order to estimate overhead costs of introducing our mechanism, we show experimental results finally.

The remainder of the paper is structured as follows. In Section II, we show known concerns for security of blockchain. In Section III, we introduce two major previous models of address on transactions and our focused blockchain model in this paper. Section IV points out new concern and the possibility of frauds. We propose a new mechanism for solving the concern and extend it to a more flexible mechanism in Section V. Section VI shows a concrete implementation of our mechanism for Bitcoin client software. We explain our experimental environment and show experimental results in Section VII. The paper is concluded in Section VIII.

II. KNOWN SECURITY AND PRIVACY ISSUES OF BLOCKCHAIN

A. DOUBLE-SPENDING ATTACKS

Since digital information can be duplicated easily, the reliability of systems might be lost significantly, if cryptocurrencies or smart contracts could be used multiple times. Two major factors of this double-spending problem are counterfeit for senders by malicious and intentional duplication, which a sender spends a value in multiple times on purpose. To prohibit counterfeit for senders, sender's digital signature on a transaction must be necessary as sender's authentication. Additionally, to protect intentional duplication by senders, blockchains introduce publicity for all transaction information, the eventual consensus algorithm and an incentive system to make reasonable peers verify a validity of those sender's inputs.

If a majority of nodes on the peer-to-peer network justifies the invalidity of the sender's setting for the transaction, double-spending attacks will be prohibited.

B. MAJORITY OCCUPATION ATTACKS

If a majority of nodes on the peer-to-peer network were malicious, results of consensus algorithms would be arbitrary. Therefore, the above double-spending attack would be successful. To protect from the majority occupation attacks or 51% attacks, blockchain utilizes cryptographic techniques for malicious nodes not to be able to control the majority of computing powers of the peer-to-peer network, and set difficulties of consensus properly. In addition, each blockchain system gives miners incentive and makes miners keep soundness of digital ledgers. By carefully turning of difficulties of Proof-of-Work, they can also absorb gradational changes of computing powers on the peer-to-peer network.

C. PRIVACY DISCLOSE

As opposed to transactions with the trusted third party, for keeping soundness of transactions and consistency of the digital ledger, it is necessary to publish transaction information on the peer-to-peer network. Accordingly, the relation between intrinsic security of blockchain and privacy protection is very complex. In blockchains, a way of maintaining privacy is to use public key anonymous techniques. An address as a pseudo-identifier has one to one and one-way correspondence to a public key. Although individuals can use multiple addresses as pseudo-identifiers without revealing their names, link-ability, i.e., the ability to infer some relations from combination of background knowledge and public information on the digital ledger, could reveal true identities of the users. Zhang *et al.* [19] pointed out that blockchain implementations only achieve pseudonymity but do not protect from link-ability. Thus, privacy disclosure might occur.

To prohibit privacy disclose, the following techniques are introduced in blockchain.

- Mixing: Mixing is a random exchange of user's cryptocurrency with other user's cryptocurrency. Mixing service is designed for user's privacy protection and similar to traditional communication mixes by cryptographic techniques.
- Fungibility: The value of cryptocurrencies should be independent on the history of exchanges, which are the same as paper currencies. The desired property is called fungibility. In order to achieve fungibility on blockchain, the Pay-to-EndPoint protocol [10] was proposed. The protocol sets not only sender's address, but also recipient's address as input addresses of a transaction. In the protocol, the receiver, instead of the sender, broadcasts the final information of the transaction publicly.

D. FLOWING TO AND FROM DARKNET MARKETPLACES

Online marketplaces connecting buyers with sellers of illicit goods and services, are called *darknet marketplaces* [8]. It has been known that many darknet marketplace participants have used cryptocurrencies to buy illicit goods [9]. Whereas some identified darknet marketplaces was closed by cyber crime law enforcement, [7] reported that darknet marketplace activity using cryptocurrencies had been remarkably resilient, despite continued efforts of law enforcement and total volume of Bitcoin sent to darknet markets in 2018, was over $600 million. Moreover, [8] has reported that there existed the transactions flowing directly from addresses associated with known illicit actors to conversion services, and over half a million bitcoin moved directly from illicit sources to conversion services in the period 2013-2016.

Thus, general users as recipients might be involved in such illicit transactions from addresses associated with illicit actors as the senders, unconsciously. To make matter worse, illicit senders could involve ordinal users in their transactions by abusing published information on the blockchain digital ledger and make the ledger record their transactions, while users were unaware of existences of their transactions. This is why we focus on the recipient's information of a transaction in this research.

III. PRELIMINARIES AND PREVIOUS ADDRESS MODELS

A. PUBLICITY OF BLOCKCHAIN

In blockchain, the applied solution for the double-spending problem without the trusted party is to publish all transaction information for the peer-to-peer network and record those in blocks on a distributed large digital ledger. When majority of nodes of the network decides to agree with the validity of the published transaction, a block with the transaction record and the mining result called ''nonce'' is put on the digital ledger and the transaction is seemed as a committed transaction.

B. ADDRESS MODELS

Bitcoin blockchain uses addresses based on UTXO (Unspent Transaction Output) model. On the one hand, Ethereum uses addresses based on the account model. We show pros and cons of each model, as follows.

1) UTXO MODEL

UTXO describes an unspent transaction output. Each transaction has a transaction ID, a sender's address and his public key, and a recipient's address and sending value for the recipient. In Bitcoin, there exist some address types called P2PK (Pay to Public Key), P2PKH (Pay to Public Key Hash), MultiSig and P2SH (Pay to Script Hash) [15], etc. We show a more detail explanation for Bitcoin address types in Appendix A.

Here, we explain the P2PKH type of addresses, since it is used as the standard type in Bitcoin protocol when one public key is used.

If a transaction which contains N-id as a transaction ID, a recipient's address add_B, and v as the sending value for add_B, has been committed and the recipient of the transaction wants to use the sending value, the recipient will make a transaction with the previous transaction N-id and a digital signature by the recipient's secret key corresponds to the public key used on making the address add_B. Then, the new transaction means a transfer of the value v from the address add_B to another recipient's address.

The merit of UTXO model is to be able to execute multiple transactions of the same senders concurrently. The demerit of UTXO is for each user to need complex managements of lists of own pairs of UTXO and the corresponding private key.

2) ACCOUNT MODEL

In contrast to UTXO model, addresses are not connected to transactions in the account model. In the account model, each account's balance is updated on the timing of chaining a block, if the block contains transactions with those related accounts. Thus, by accessing the last block containing a transaction with the account as a sender's address or a recipient's address, the balance of the account can be confirmed. If an owner of the account wants to use a part of his balance, he can set any value, which is at most within his balance, as sending value of a new transaction. Conversely, multiple transactions related with one account should be executed serially for keeping atomicity.

3) COMMON PROPERTIES

In both of UTXO model and the account model, each address is made with owner's public key and the public hash function. If the owner would like to prove the property of the owner's address, the owner must show the information attached to his digital signature by his public key corresponding to his address for verifiers.

C. STRUCTURE OF BLOCKS AND TRANSACTIONS

Blockchain consists of blocks and directed links representing relations between two blocks. A block includes multiple transaction informations, a hash value of the previous block called a previous hash value, and a value called nonce as the proof-of-work. A relation from a block B_x to a block B_y holds, if the hash value of the block B_x has been included as the previous hash value in the next block B_y. Information of transactions on a block is also saved as a tree called "Merkel tree" and is aggregated as a hash value called "Merkel root." By using the value "Merkel root" of a block, we can check efficiently whether a given transaction is included in the block or not [15].

Each transaction consists of one or multiple sender's addresses as input cryptocurrencies owned by the payer and one or multiple recipients' addresses as the payee's account with values of transferred cryptocurrencies. In the following, on the assumption that each transaction has one sender's address and one recipient's address for simplicity, we explain the structure of transaction information in more detail.

Each transaction involves a sender A and a recipient B who are respectively identified by the sender's address add_A as the payer's account and the recipient's address add_B as the payee's account. The sender A and the recipient B have pairs of a private key and the corresponding public key (SK_A, PK_A) and (SK_B, PK_B), respectively. Since each address is made based on the owner's public key, the owner's public key is used as the proof of ownerships for each address, and the owner's pair of the public key and the corresponding private key is used for the authorization for sending owner's cryptocurrencies. The recipient's address has an additional information "*value*" shown a quantity of cryptocurrencies sent from the sender A to the recipient B. If the transaction has multiple payers' accounts or multiple payees' accounts, each account is distinguished by the index of the corresponding element in the input array for payers' accounts or the output array for payees' accounts. Each transaction is identified by the transaction ID which is the hash value of the transaction information.

In the above arguments, add_A might be represented as a pair of the previous transaction ID and the output index in the transaction, if a type of add_A is UTXO model. Else, if a type of add_A is the account model, add_A might be the corresponding account.

Let Sig_{SK_A}(input sequence) be the output of the function Sig_{SK_A}(input sequence) as the digital signature function for a hash value of the input sequence by the sender's secret key SK_A. Now, we consider that the transaction processes sending the value v from the address add_A of the sender A to the address add_B of the recipient B. Then, the transaction includes the sender's digital signature $Sig_{SK_A}(add_A, v, add_B)$ and the public key PK_A. Then, nodes on the peer-to-peer networks can verify the correctness of the sender's address add_A by checking whether the double-spending problem might occur or not, and also verify the sender's authorization for the transaction by using the sender's signature $Sig_{SK_A}(add_A, v, add_B)$ and the sender's public key PK_A.

D. CURRENT TYPICAL PROTOCOL FOR PUBLISHING TRANSACTIONS

We introduce a current typical protocol [11] between the sender A and the recipient B for creating a transaction information. The sender A and the recipient B are hereinafter referred to as he and she, respectively. First, the recipient B sends her address add_B to the sender A in order to specify her account. Second, the sender A selects his address add_A and the corresponding pair of his secret key SK_A and his public key PK_A. The sender A gets a digital signature $Sig_{SK_A}(add_A, v, add_B)$ which shows that he authorizes the sending of the value v from the sender's address add_A to the recipient's address add_B by using his secret key SK_A. Next, the sender A makes a new transaction information $Tx = (add_A, PK_A, Sig_{SK_A}(add_A, v, add_B), v, add_B)$ by setting his digital signature $Sig_{SK_A}(add_A, v, add_B)$ and his public key PK_A. Finally, the sender A publishes the transaction Tx to the peer-to-peer network, and requests nodes to commit the transaction Tx.

IV. NEW CONCERN

In [18], we pointed out a new concern for the above current protocol. After the notice of the recipient's address, the sender A makes a transaction and publishes it for the peer-to-peer network without interaction with the recipient, in the current protocol for creating a transaction information. We wonder if a malicious sender M could generate and publish a malicious transaction freely by abusing information of the published transaction. We explain our concerned situation by using Fig.1.

The malicious sender M could contrive abusing information of the published transaction "Transaction1," as shown in Fig.1 (a). M could make freely a new and fake transaction called "Transaction4" and set the recipient's address "Address2" on the transaction "Transaction4" without any notification for the recipient (Fig.1 (b)). If the transaction "Transaction4" was published by the malicious sender M, the transaction could be committed by the peer-to-peer network, without the recipient's awareness (Fig.1 (c)). The reason is that nodes on the peer-to-peer network could not distinguish the transaction "Transaction1" by the regular

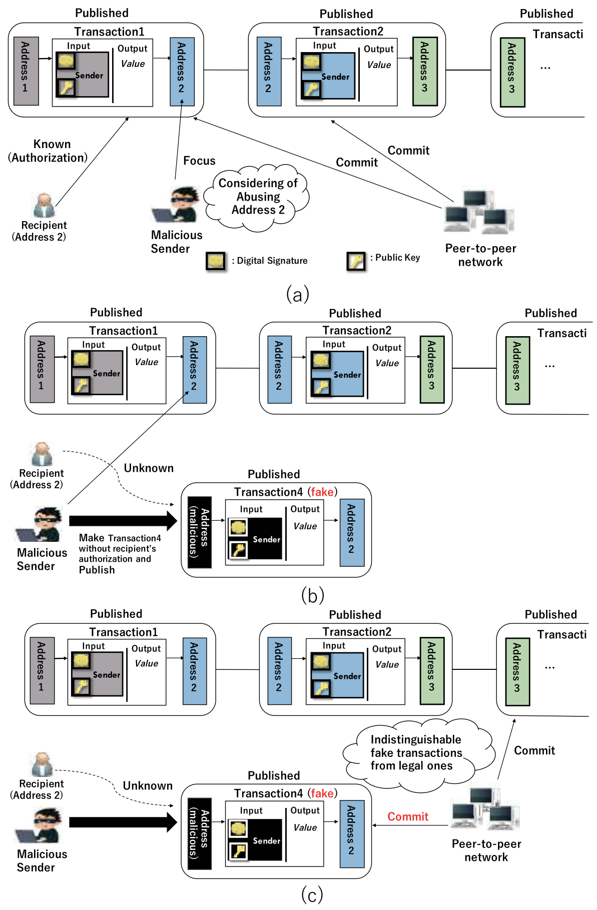

FIGURE 1. **Example of abusing published information by malicious sender's with the new concern.**

sender and the fake transaction "Transaction4" published by the malicious sender M.

The transaction published by the malicious sender M might become the loan of cryptocurrencies or money transfer with relation to crimes for pools. Also, when the transaction represents the sending process of digital contents instead of cryptocurrencies, the malicious sender might scatter malware for many unspecified users through blockchain.

In addition to the above concern, the *orphan* address problem [2] exists. Orphan addresses are not associated to any user or account. If some cryptocurrencies are sent to an orphan address, they are lost forever. The orphan address problem could occur by a lack of notices for the recipient.

V. A PROTOCOL FOR NEW CONCERN

A. OUR IDEA

From our investigation in the above section, we noticed that it is important to set some limitation for reusing the published transaction information. Our idea [18] is to make it impossible to set a recipient's address of a published transaction as a recipient's address of the different transaction again. In our new protocol, the new structure of a transaction does not only enable for nodes on the peer-to-peer network to verify recipient's authorization for it, but also prohibits malicious senders from abusing the recipient's address. Furthermore, it also prohibits the recipient from modifying the transaction information and publishing it freely.

In the following, first, we introduce the new structure of a transaction, then, we propose a new protocol for publishing transactions based on the new structure of a transaction.

B. OUR STRUCTURE OF A TRANSACTION

We define our structure of a transaction as follows. Let a transaction Tx be a seven-tuple $(add_A, PK_A, SIG_{SK_A}, v, add_B, PK_B, SIG_{SK_B})$, where add_A, PK_A and SIG_{SK_A} are the sender A's address, his public key and his digital signature by his secret key SK_A, v is a value of transferring cryptocurrencies from the sender A to the recipient B, and add_B, PK_B and SIG_{SK_B} are the recipient B's address, her public key and her digital signature by her secret key SK_B. If some term in the transaction information is not fixed, the term will be replaced by the empty string ϵ.

C. OUR PROTOCOL

We propose a new protocol for publishing a transaction Tx under the constraint that the recipient's address is protected from resetting it by malicious senders in different transactions as an output address but the recipient cannot change the published transaction information without the sender's authorization. We show our protocol as follows.

Step 1: Recipient B sends her public key PK_B as her tentative address.

Step 2: Sender A selects his address add_A for paying a value v of cryptocurrencies. Sender A makes an initial transaction information $Tx_{initail} = (add_A, \epsilon, \epsilon, v, \epsilon, PK_B, \epsilon)$, and sends the transaction information $Tx_{initial}$ to Recipient B.

Step 3: Recipient B verifies the correctness of the initial transaction information $Tx_{initial}$. If she can confirm the correctness, she gets a digital signature $Sig_{SK_B}(Tx_{initial}) = SIG_{SK_B}$ for the sender's address add_A, the value v and PK_B by using her secret key SK_B. Then, she makes a new address $onetime.ADD_B$, generated by the concatenation of an prefix "*onetime*." and a hash value $ADD_B = hash(SIG_{SK_B}, PK_B)$ with her digital signature SIG_{SK_B} and her public key PK_B. We call this type of address S-address, where "S" means "Signature". Recipient B sends her digital signature SIG_{SK_B}, and the S-address $onetime.ADD_B$ as an output address.

Step 4: Sender A verifies the correctness of the recipient's S-address $onetime.ADD_B$ by checking recipient's signature SIG_{SK_B} and using the hash value of the concatenation of her signature and her public key PK_B. If he confirms it, he updates the next transaction information $Tx_{update} = (add_A, \epsilon, \epsilon, v, onetime.ADD_B, PK_B, SIG_{SK_B})$ by setting recipient's S-address $onetime.ADD_B$ as an output address, recipient's public key PK_B and her digital signature SIG_{SK_B} as a proof of her authorization to set her address in this transaction. Sender A gets his digital signature $Sig_{SK_A}(Tx_{update}) = SIG_{SK_A}$ for the transaction information Tx_{update} by using his secret key

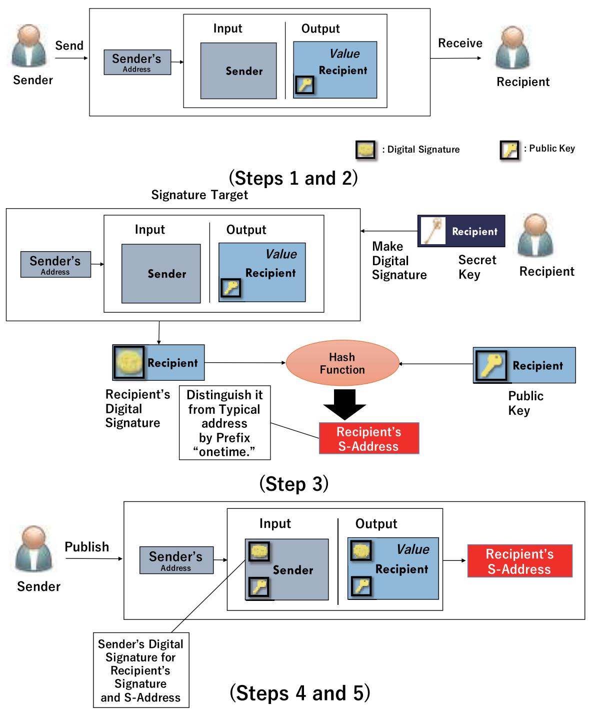

FIGURE 2. Our proposed *S*-address and the method of making *S*-address.

SK_A. Sender A makes the final transaction information $Tx = (add_A, PK_A, SIG_{SK_A}, v, onetime.ADD_B, PK_B, SIG_{SK_B})$ by setting his digital signature SIG_{SK_A}, as his authorization for this transaction.

Step 5: Sender A publishes the final transaction information Tx for the peer-to-peer network.

We also show a flow of the proposed protocol in Fig. 2.

D. COMMITMENT BY PEER-TO-PEER NETWORK

Given a published transaction $Tx = (add_A, PK_A, SIG_{SK_A}, onetime.ADD_B, PK_B, SIG_{SK_B})$ generated through our protocol, to construct a new block or verify the correctness of previous blocks, nodes on the peer-to-peer network should verify the correctness of each transaction in the block. We assume that nodes can recognize the type of the recipient's address as the S-address by the prefix "*onetime*." of the address. Then, for the transaction including the S-address as the recipient's address, nodes have to check not only the sender's authorization, but also the correctness of setting the S-address as an output address. In the following, we explain the verification steps in nodes on the peer-to-peer network, for the published transaction Tx.

First, nodes verify the correctness of the input address add_A and the digital signature SIG_{SK_A} for the mid transaction information $(add_A, \epsilon, \epsilon, onetime.ADD_B, PK_B, SIG_{SK_B})$ using the sender's public key PK_A to confirm that the sender has a value v in the input address add_A and admits to give the value v to the recipient's address $onetime.ADD_B$ through the transaction. Next, nodes also verify the digital signature SIG_{SK_B} for the initial transaction information $(add_A, \epsilon, \epsilon, v, \epsilon, PK_B, \epsilon)$ and the correctness of setting the output address using the recipient's public key PK_B, to confirm for the recipient to accept the sending of the value from the sender's address add_A to the recipient's address $onetime.ADD_B$ through the transaction.

We assume that a malicious sender M publishes a transaction $Tx_M = (add_M, PK_M, SIG_{SK_M}, v', onetime.ADD_B, \epsilon, \epsilon)$ without the recipient's authorization by abusing the recipient's S-address as an output address. Then, nodes will find out that the malicious sender abused the recipient's address of some previous transaction. Because of the prefix of the address, "*onetime*." shows that the transaction information must include the recipient's digital signature for the initial transaction.

Even if a malicious sender M publishes a transaction $Tx'_M = (add_M, PK_M, SIG_{SK_M}, v', onetime.ADD_B, PK_B, SIG_{SK_B})$ using the recipient's S-address as an output address with the recipient's authorization for the different transaction, nodes will also find out that the malicious user's abused the recipient's address through checking whether SIG_{SK_B} is the recipient's signature for the initial transaction information $(add_M, \epsilon, \epsilon, v', \epsilon, PK_B, \epsilon)$. The reason is that the signature SIG_{SK_B} satisfies that $SIG_{SK_B} = Sig_{SK_B}(add_A, \epsilon, \epsilon, v, \epsilon, PK_B, \epsilon)$ and the checking results in failure.

There exists a difference of structures between published transactions by our protocol and published transactions by the current protocols. The type of the S-address has to be distinguished from other types of addresses by nodes in the peer-to-peer network. Thus, we should advertise that a different type of addresses has been introduced by our new protocol, toward nodes on the peer-to-peer network.

E. RELATIONS BETWEEN *S*-ADDRESS AND PREVIOUS ADDRESS MODELS

We consider relations between S-address, UTXO address model and the account model.

S-address can be seemed as an address of strict UTXO model, because of the strict request for S-address so that different output address must be set for each transaction. Moreover, S-address prohibits a malicious sender from abusing the output addresses of published transactions.

On the other hand, multiple different S-addresses can be constructed from one recipient's public key. If we manage records and cryptocurrencies of transactions for each public key, instead of each address, the management of multiple S-address from one public key can be seemed as the account management. When multiple S-addresses from the same public key are set as input addresses of a transaction, only one digital signature by the public key will be needed as the sender's digital signature for sender's authorization. In addition, multiple transactions, having S-addresses from one public key as sender's address or recipient's address, can be executed as concurrently and safely as ones of UTXO model.

Thus, our S-address improves the UTXO model and the account model, by maintaining the merits of both models and overcoming disadvantages of those models.

F. EXTENSION TO FLEXIBLE PERMITTED ADDRESS

In our protocol, a recipient implicitly makes a one-time address having a limitation on the transaction where the address can be set as the recipient's address. However, the limitation might be too strict and inflexible on transactions from partially-trusted senders. Thus, we relax the limitation on transactions for setting the S-address as recipient's address to a limitation on a sender's address of transactions, or to a limitation on a group of senders' addresses, etc. In our protocol, we can set a recipient's digital signature for the specific sender's address instead of the recipient's digital signature for the transaction, and make an S-address by the digital signature and recipient's public key. We can also set a recipient's digital signature for the specific group of senders' address.

In order to give users such flexibility of setting limitations when he makes an S-address, we prepare three kinds of extensions of S-address, "onetime.", "sender." and "group.". If a prefix of the S-address is "sender.", then verifiers will check whether the attached digital signature is recipient's digital signature for the sender's address or not. Otherwise, if a prefix of the S-address is "group.", then verifiers will check whether the attached digital signature is recipient's digital signature for a list of senders' addresses and the sender's address is included in the list, or not.

VI. CONCRETE IMPLEMENTATION OF S-ADDRESS ON BITCOIN PROTOCOL

We focus on implementation of the S-address targeted on Bitcoin Core protocol. Bitcoin Core protocol is the official client software of Bitcoin and the C++ source codes of Bitcoin Core protocol are published on the Web service GitHub (https://github.com/bitcoin/bitcoin). We will combine our S-address with Bitcoin Core protocol. In the following, we introduce general elements of transactions on Bitcoin protocol, and propose our implementation targeted on Bitcoin protocol.

A. GENERAL ELEMENTS OF TRANSACTIONS ON BITCOIN PROTOCOL

1) TRANSACTION DESCRIPTION

In Bitcoin Core protocol, a description of each transaction has two arrays, i.e., one array "*Tx_In*" of the data structure "*TxIn*" for descriptions of inputs of the transaction and the other array "*Tx_Out*" of the data structure "*TxOut*" for descriptions of outputs of one. We show the outline of the standard transaction structure in Fig. 3.

The data structure "*TxIn*" has a transaction identification "*TxID*" and an integer "*index*." For each sender's input as UTXO, an element of the data structure "*TxIn*" in the array "*TX_In*" will be allocated in the description of transaction. The sender's UTXO corresponds to the "*index*"-th output in the transaction having "*TxID*" as the identification. Thus, a pair of the transaction identification "*TxID*" and the number "*index*" represents implicitly the sender's address of his UTXO.

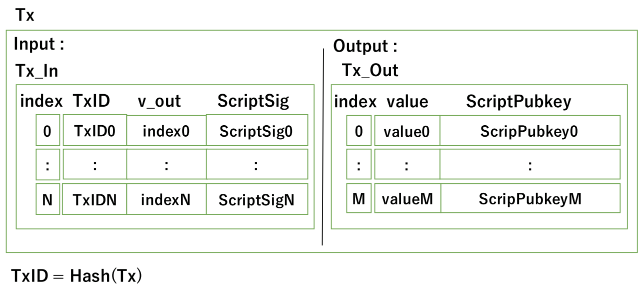

FIGURE 3. The outline of the standard transaction structure of the bitcoin.

The data structure "*TxIn*" also has a sequence called "*ScriptSig*" and an integer called "*Scriptbytes*," which represents the length of "*ScriptSig*." In general, the sequence "*ScriptSig*" includes the sender's public key used for locking the UTXO and the sender's digital signature by the corresponding secret key to the public key. In Fig. 3, we omit descriptions of the lengths of data elements, i.e., "*Scriptbytes*."

The data structure "*TxOut*" consists of an integer "*value*," an sequence "*ScriptPubkey*" and an integer "*ScriptPubkeybytes*." We explain the above three elements as follows.

The sequence "*ScriptPubkey*" is a part of a simple program of the scripting language by Reverse Polish Notation. When nodes on the peer-to-peer network want to verify whether the owner of the UTXO is a sender and the sender admits that a transaction will be executed, a full program is constructed by concatenating the sequence "*ScriptSig*" on the transaction in which the sender wants to utilize his UTXO, and the sequence "*ScriptPubkey*." In the next subsection, we will explain script programs by using an example in more detail.

The sequence "*ScriptPubkey*" includes a hash value of the recipient's public key generally. The mapping value of the hash value by the certain one-to-one function (called base58check encoding [17]) is referred to as the recipient's address.

The integer "*ScriptPubkeybytes*" represents the length of the sequence "*ScriptPubkey*." And the integer "*value*" represents a quantity of UTXO sent to the recipient's address.

Although we defined a structure of a transaction on Section V, we redefine a structure of a transaction for the Bitcoin as follows. Let a transaction Tx be a quadruple $((Prev\_TxID, index), ScriptSig(SIG_{SK_A}, PK_A), v, ScriptPubkey(add_B))$, where $(Prev\_TxID, index)$ and $ScriptSig(SIG_{SK_A}, PK_A)$ are the sender A's UTXO and his $ScriptSig$, v is a value of transferring cryptocurrecies from the sender A to the recipient B, and $ScriptPubkey(add_B)$ is the recipient's $ScriptPubkey$. In the above definition, we also define the descriptions $ScriptSig(\ )$ and $ScriptPubkey(\ )$ as the formats of $ScriptSig$ and $ScriptPubkey$, respectively. We assume that we can get the sender A's $ScriptSig$ by setting

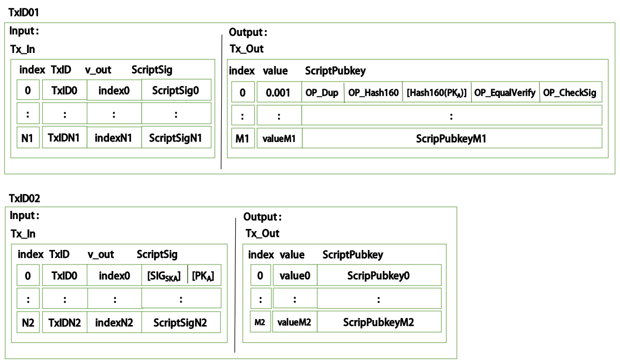
TxID01
Input:
Tx_In
index TxID v_out ScriptSig
0 TxID0 index0 ScriptSig0
N1 TxIDN1 indexN1 ScriptSigN1
Output:
Tx_Out
index value ScriptPubkey
0 0.001 OP_Dup OP_Hash160 [Hash160(PK_A)] OP_EqualVerify OP_CheckSig
M1 valueM1 ScripPubkeyM1
TxID02
Input:
Tx_In
index TxID v_out ScriptSig
0 TxID0 index0 [SIG_SKA] [PK_A]
N2 TxIDN2 indexN2 ScriptSigN2
Output:
Tx_Out
index value ScriptPubkey
0 value0 ScripPubkey0
M2 valueM2 ScripPubkeyM2

FIGURE 4. **The standard scriptpubkey structure and scriptpubkey structure.**

the sender A's signature SIG_{SK_A} and her public key PK_A on the format *ScriptSig*( ), and the recipient B's *ScriptPubkey* by setting the recipient B's address add_B on the format *ScriptPubkey*( ).

2) BITCOIN SCRIPTS

Bitcoin uses a simple Reverse Polish Notation system called "*Script*," in order to describe clearly how to lock and unlock each UTXO with respect to input elements or output elements of transactions. A script is a list of data elements and operations [1]. As mentioned above, each transaction has two types of scripts, "*ScriptPubkey*" and "*ScriptSig*." The script "*ScriptPubkey*" is called as a locking script, by describing how to unlock the received UTXO. Thus, a content of the script "*ScriptPubkey*" must be decided in each output by the recipient on each transaction. The other type of scripts, "*ScriptSig*" is called an unlocking script, and it provides data elements to satisfy the unlocking condition decided in "*ScriptPubkey*" of the previous transaction in advance.

The scripting system gives senders and recipients flexibility to describe rules of checking. In Fig. 4, we show examples of "*ScriptPubkey*" and "*ScriptSig*," and explain how the third party checks the validity of transferring the UTXO, using these examples. In the following, we assume that the script "*ScriptPubkey*" is in the first output element $Tx\_Out[0]$ of the transaction whose ID is "$TxID01$," and the script "*ScriptSig*" is in the first input element $Tx\_In[0]$ of the transaction whose ID is "$TxID02$". And, we also assume that data elements $Tx\_In[0].TxID$ and $Tx\_In[0].index$ of the input array element $Tx\_In[0]$ of the transaction "$TxID02$" are "$TxID01$" and "0."

In the following, we assume that scripts *ScriptSig* of general type transactions can be constructed by setting the sender A's address type, his digital signature SIG_{SK_A}, and his public key PK_A on a format *ScriptSig*( ). If $type(add_A)$ is P2PKH, $ScriptSig(type(add_A), SIG_{SK_A}, PK_A)$ returns the following script as *ScriptSig* for the sender A.

ScriptSig(P2PKH): "$[SIG_{SK_A}][PK_A]$"

And, we also assume that scripts *ScriptPubkey* of general type transactions can be constructed by setting the recipient B's address type and her public key PK_B on a format *ScriptPubkey*( ). If $type(add_B)$ is P2PKH, $ScriptPubkey(P2PKH, PK_B)$ returns the following script as *ScriptPubkey* for the recipient B.

ScriptPubkey$_{P2PKH}$**:** "OP_Dup OP_Hash160 [Hash160 (PK_B)] OP_EqualVerify OP_CheckSig"

If nodes on the peer-to-peer network or the third parties want to check the validity of transferring the UTXO corresponding to the input $Tx\_In[0]$ of the transaction "$TxID02$," nodes make a script by concatenating the script "*ScriptSig*" of the transaction "$TxID02$" and the script "*ScriptPubkey*" of the transaction "$TxID01$". Next, nodes give the stack-based programming system of Bitcoin the concatenated script as an input. More strictly, for the security, the stack based programming system processes the script "*ScriptSig*" with an empty initial stack first. Then it saves the stack, and processes the script "*ScriptPubkey*" with the saved stack as an initial stack [15]. If the system returns the true, the nodes will see that the verification is success. Otherwise, the verification ends in failure and the transaction "$TxID02$" is excluded from candidates of transactions in a new block. We will show a more detail explanation for the verification procedure for the scripts in Appendix B.

B. OUR IMPLEMENTATION

We propose a concrete implementation of our selective mechanism targeting to incorporate into Bitcoin Core protocol. Our implementation has a characteristic property such that a recipient's signature for the S-address is put on the sender's script "*ScriptSig*" on one transaction. This is because nodes on the peer-to-peer network target only inputs of each transaction for verification of the validity of signatures. Thus, if we put the recipient's signature on the recipient's script "*ScriptPubkey*," nodes on the peer-to-peer network might miss the necessity of verification of the recipient's signature for the S-address. In the following, we show an outline of our implementation of our selective mechanism for the Bitcoin protocol.

We define our new type of a transaction for the Bitcoin as follows. Let a transaction Tx_S be a quadruple $((Prev\_TxID, index)$, $ScriptSigS(ScriptSig(type(add_A)$, SIG_{SK_A}, $PK_A)$, $SIG_{SK_B})$, v, $ScriptPubkeyS(add_B$, $PK_B))$, where $ScriptSigS(ScriptSig(type(add_A), SIG_{SK_A}, PK_A)$, $SIG_{SK_B})$ and $ScriptPubkeyS(add_B, PK_B)$ are the sender A's *ScriptSig* and the recipient B's *ScriptPubkey* for the recipient B's S-address, respectively, while $(Prev\_TxID, index)$ and v are the same ones of the general transaction Tx. Thus, in our implementation, we introduce new formats *ScriptSigS*( ) and *ScriptPubkeyS*( ). And, we extend the format *ScriptSig*( ) so that we can apply our S-address to the format. In the following, we assume that we can get the sender A's new type *ScriptSig* by setting the sender A's general *ScriptSig* as $ScriptSig(type(add_A), SIG_{SK_A}, PK_A)$ and the recipient B's signature SIG_{SK_B} on the format *ScriptSigS*( ), and the recipient B's new type *ScriptPubkey* by setting the recipient B's address add_B and her public key PK_B on the format

$ScriptPubkeyS(\ )$. If some term in the formats is not fixed, we assume that the term can be replaced by the empty string ϵ.

Although we imply that a pair $(Prev\_TxID, index)$ represents the sender A's address add_A, strictly speaking, we need the script "*ScriptPubkey*" of the *index*-th output on the transaction $Prev\_TxID$ to verify whether the sender A owns the UTXO. Thus, let $Prev\_TxID.out[index].ScriptPubkey$ be the *ScriptPubkey* of the *index*-th output on the transaction $Prev\_TxID$. The sender A's public key PK_A has to correspond to the address information add_A in the script $Prev\_TxID.out[index].ScriptPubkey$. In other words, the hash value of his public key PK_A has to be included in the script $Prev\_TxID.out[index].ScriptPubkey$. In the following, we refer to $Prev\_TxID.out[index].ScriptPubkey$ as $Prev\_ScriptPubkey$.

Now, we show the outline of our protocol for constructing the transaction information for the Bitcoin, based on the above definitions.

Our protocol for the Bitcoin

Step 1: Recipient B sends her public key PK_B as her tentative address.

Step 2: Sender A selects a pair $(Prev\_TxID, index)$ of a transaction index of a previous transaction and his output's index as his address add_A for paying a value v of cryptocurrencies. Sender A sets Recipient B's public key PK_B on the S-address type format $ScriptPubkeyS(\ )$ for the recipient's *ScriptPubkey* and constructs the initial script $ScriptPubkeyS(\epsilon, PK_B)$. Sender A makes an initial transaction information $Tx_{initail} = ((Prev\_TxID, index), \epsilon, v, ScriptPubkeyS(\epsilon, PK_B))$, and sends the transaction information $Tx_{initial}$ to Recipient B.

Step 3: Recipient B verifies the correctness of the initial transaction information $Tx_{initial}$. When she can confirm the correctness, she gets a digital signature $Sig_{SK_B}(Tx_{initial} \cdot ScriptPubkeyS(\epsilon, PK_B)) = SIG_{SK_B}$ for the sender's address add_A, the value v and PK_B by using her secret key SK_B. Then, she makes a new address $add_B = onetime.ADD_B$, generated by the concatenation of an prefix "*onetime.*," and a hash value $ADD_B = Hash160(SIG_{SK_B})$ with her digital signature SIG_{SK_B}.

Next, she constructs her script $ScriptPubkey_B = ScriptPubkeyS(add_B, PK_B)$ by setting her address add_B and her public key PK_B. Recipient B sends her digital signature SIG_{SK_B}, the S-address $onetime.ADD_B$ and her script $ScriptPubkey_B$ as an output script.

Step 4: Sender A checks Recipient B's signature SIG_{SK_B}. Next, he verifies the correctness of the recipient's S-address $onetime.ADD_B$ by using the hash value of the concatenation of Recipient's signature and her public key PK_B. When he confirms it, he updates the next transaction information $Tx_{update} = ((Prev\_TxID, index), \epsilon, v, ScriptPubkey_B) = ((Prev\_TxID, index), \epsilon, v, ScriptPubkeyS(add_B, PK_B))$ by setting Recipient B's $ScriptPubkey_B$ including Recipient B's public key PK_B and her output address $onetime.ADD_B$ made from her digital signature SIG_{SK_B} as a proof of her authorization to set her address in this transaction. Sender A gets his digital signature $Sig_{SK_A}(Tx_{update} \cdot Prev\_ScriptPubkey) = SIG_{SK_A}$ for the transaction information Tx_{update} by using his secret key SK_A.

Sender A constructs his general $ScriptSig(type(add_A), \ldots) = ScriptSIG_A$ by setting his address type and adding his public key PK_A, his signature SIG_{SK_A}, etc. to the input parameters depending on the address type. Next, he constructs his script $ScriptSigS(ScriptSIG_A, SIG_{SK_B})$ by setting his general $ScriptSIG_A$ and Recipient B's signature SIG_{SK_B}.

Sender A makes the final transaction information $Tx_S = ((Prev\_TxID, index), ScriptSigS(ScriptSIG_A, SIG_{SK_B}), v, ScriptPubkeyS(add_B, PK_B))$ by setting his script $ScriptSIG_A$ as his authorization and Recipient B's signature SIG_{SK_B} as her authorization for this transaction.

Step 5: Sender A publishes the final transaction information Tx_S for the peer-to-peer network.

On general transactions of the Bitcoin, the recipient's address is not described explicitly. However, it is given implicitly by the type of the recipient's script "*ScriptPubkey*" and the hash value on the script. Thus, we decide that the hash value of the recipient's signature is also put on the recipient's script "*ScriptPubkey*" in our implementation, and the script also represents the recipient's S-address implicitly.

In our implementation, we propose 2 types of concrete implementations of the S-address, called "Pay to Simple Signature (P2SSIG)" and "Pay to Signature (P2SIG)". The P2SSIG is a strict implementation of our S-address in Section V. The second type "P2SIG" might have a higher security level than the P2SSIG type of the S-address.

Next, we explain the two types of S-address implementations and show their properties, respectively. In the following subsections, we will explain the three script formats for $ScriptPubkeyS(\ )$, $ScriptSigS(\ )$, and $ScriptSig(S\text{-address}, \ldots)$ in more details.

$ScriptPubkeyS(\ )$ is the format of *ScriptPubkey* set on the output corresponding to the S-address as *ScriptPubkey*. $ScriptSig(\ )$ is the format of *ScriptSig* set on the first input as *ScriptSig* on a transaction which has one output corresponding to an S-address. $ScriptSig(S\text{-address}, \ldots)$ is the format of *ScriptSig* or a part of *ScriptSig*, set on the input whose address is an S-address.

1) P2SSIG: THE FIRST TYPE OF *S*-ADDRESS

In this implementation type, we utilize only one recipient's public key "PK_B" for a preparation of the transaction. We refer to this address type as Pay to Simple Signature (P2SSIG) type.

In the following, we define the three formats $ScriptPubkeyS_{P2SSIG}(SIG_{SK_B}, PK_B)$, $ScriptSigS_{P2SSIG}(ScriptSig(type(add_A), \ldots), SIG_{SK_B})$, and $ScriptSig(P2SSIG, SIG_{SK_A}, Prev\_SIG_{SK_A})$, where $ScriptSIG_A$ represents $ScriptSig(type(add_A), \ldots)$ and $Prev\_SIG_{SK_A}$ is the signature from which the S-address corresponding to the sender A's UTXO was made.

- $ScriptPubkeyS_{P2SSIG}(SIG_{SK_B}, PK_B)$
  Given the recipient B's signature SIG_{SK_B} and her public key PK_B, the hash value $Hash160(SIG_{SK_B})$ is computed and the hash value and the public key are set on the format, and the following script is constructed.
  ScriptPubkeyS$_{P2SSIG}$**:** "OP_Hash160 [Hash160 (SIG_{SK_B})] OP_EqualVerify [PK_B] OP_CheckSig"
- $ScriptSigS_{P2SSIG}(ScriptSIG_A, SIG_{SK_B})$
  Given the recipient B's signature SIG_{SK_B} and the sender A's script $ScriptSIG_A$, the signature and the script are set on the format, and the following script is constructed, where the opcode "OP_P2SSIG" shows our introduced new operation. The opcode "OP_P2SSIG" is to verify the correctness of the signature SIG_{SK_B} and check the equality of a hash value for the signature SK_{SK_B} and the hash value in the script $ScriptPubkeyS_{P2SSIG}(SIG_{SK_B}, PK_B)$ in the output of the transaction $TxID01$, atomically.
  ScriptSigS$_{P2SSIG}$**:** "[SIG_{SK_B}] OP_P2SSIG [$Script\text{-}SIG_A$]"
- $ScriptSig(P2SSIG, SIG_{SK_A}, Prev\_SIG_{SK_A})$
  Given the sender A's signature SIG_{SK_A} and the previous signature $Prev\_SIG_{SK_A}$, the two signature SIG_{SK_A} and $Pre\_SIG_{SK_A}$ are set on the format, and the following script is constructed.
  ScriptSig(P2SSIG): "[SIG_{SK_A}][$Prev\_SIG_{SK_A}$]"

The reason for introducing the new script operation "OP_P2SSIG" is that the recipient B can declare usage of her public key PK_B in her *ScriptPubkey* which is included in the digital signature targets, and can deny a counterfeit signature given by any others. Moreover, by checking the equality of two hash values in the operation "OP_P2SSIG", the sender A can prevent the recipient B from publishing remade transactions by replacing her digital signature SIG_{SK_B} of the targeted transaction with her another digital signature SIG'_{SK_B}, which is computable by using a different random parameter in the Elliptic Curve Digital Signature Algorithm (ECDSA).

We will show an example for construction of scripts for P2SSIG type of an S-address in Appendix C.

2) P2SIG: THE SECOND TYPE OF *S*-ADDRESS

In the second type of implementations, we apply the recipient B's two public keys "PK_B" and "PK'_B" for a preparation of transaction publication. We call this address type Pay to Signature (P2SIG) type.

If the recipient B wants to receive the transferred value by commitment of the transaction $TxID01$, she will make her digital signature by using her secret key "SK_B" as well as the first type of our implementation. However, she will make her *ScriptPubkey* by using her two public keys "PK_B" and "PK'_B."

The recipient B sets the concatenation result of "PK_B" and "PK'_B" as the input value of the Hash160 function and puts "PK_B" and the result of the hash function on her *ScriptPubkey* as the locking script. We introduce the new Script operation "OP_Concatenate" for the concatenation result of two input values.

In the following, we define the three formats $ScriptPubkey\text{-}S_{P2SIG}(SIG_{SK_B}, PK_B, PK'_B)$, $ScriptSigS_{P2SIG}(Script\text{-}SIG_A, SIG_{SK_B})$, and $ScriptSig(P2SIG, SIG_{SK'_A}, PK'_A, Prev\_SIG_{SK_A})$.

- $ScriptPubkeyS_{P2SIG}(SIG_{SK_B}, PK_B, PK'_B)$
  Given the recipient B's signature SIG_{SK_B}, her public key PK_B and her second public key PK'_B, the first hash value $Hash160(SIG_{SK_B})$ and the second hash value $Hash160(PK_B \cdot PK'_B)$ of the concatenation of PK_B and PK'_B, are computed. Then, the two hash values and the public key PK_B are set on the format, and the following script is constructed. We introduce the second new script operation "OP_Concatenate" for the concatenation result of two input values. The recipient B can make nodes set the concatenation result of "PK_B" and "PK'_B" as the input value of the Hash160 function by the new introduced opcode, when nodes verify the correctness of transferring UTXOs.
  ScriptPubkeyS$_{P2SIG}$**:** "OP_Hash160 [Hash160 (SIG_{SK_B})] OP_EqualVerify OP_Dup [PK_B] OP_Concatenate OP_Hash160 [Hash160($PK_B \cdot PK'_B$)] OP_EqualVerify OP_CheckSig"
- $ScriptSigS_{P2SIG}(ScriptSIG_A, SIG_{SK_B})$
  This format is the same as the $ScriptSigS_{P2SIG}(\ )$, except for our new opcode "OP_P2SIG" instead of "OP_P2SSIG," as follows.
  ScriptSigS$_{P2SIG}$**:** "[SIG_{SK_B}] OP_P2SIG [$ScriptSIG_A$]"
  We also introduce the third new script operation "OP_P2SIG" for atomically executing the following two processes, as the above proposed command "OP_P2SSIG." The one process is to verify the correctness of the recipient B's signature with her public key "PK_B" in her *ScriptPubkey*. The other process is to check the equality of the hash value of the recipient B's signature and the first hash value of the recipient B's script "*ScriptPubkey*."
  The sender A publishes the transaction information on the peer-to-peer network. After committing of the transaction, if the receiver B wants to transfer the value to a next receiver, she can unlock the UTXO by making her digital signature for the new transaction and constructing her *ScriptSig* with her digital signature by her second secret key SK'_B and her second public key "PK'_B."
- $ScriptSig(P2SIG, SIG_{SK'_A}, PK'_A, Prev\_SIG_{SK_A})$
  Given the sender A's second public key PK'_A, his signature $SIG_{SK'_A}$ by the second secret key SK'_A and the previous signature $Prev\_SIG_{SK_A}$, the two signature $SIG_{SK'_A}$ and $Pre\_SIG_{SK_A}$ and the public key PK'_A are set on the format, and the following script is constructed.
  ScriptSig(P2SIG): "[$SIG_{SK'_A}$][PK'_A][$Prev\_SIG_{SK_A}$]"

C. SECURITY ANALYSIS

We show our results of security analysis for the first type P2SSIG of the S-address of our selective mechanism for the

Bitcoin client software as follows. The same arguments for the second type P2SIG of the S-address also hold.

Theorem 1 (Recipient's Security): if, on the transaction $TxID01$, the recipient's address type is P2SSIG and her *ScriptPubkey* is "$ScriptPubkeyS_{P2SSIG}$" and security hash functions, the public key algorithm and the digital signature algorithm used for construction of scripts are secure, no one who does not know the recipient's secret key, can set the script $ScriptPubkeyS_{P2SSIG}$ or parts of the script including the first hash value and the recipient B's public key on any other committable transaction except for $TxID01$.

Proof: In the first type P2SSIG, the recipient public key PK_B in outputs' scripts "$ScriptPubkeyS_{P2SSIG}$" are included in the target message signed by the recipient B. And, the recipient B's script "$ScriptPubkeyS_{P2SSIG}$" is duplicated and occurs twice in the target message. If signatures without the recipient B's secret key SK_B was included in the sender A's script "$ScriptSigS_{P2SSIG}$" as the recipient B's signature, the execution of an operation "OP_P2SSIG" of the script "$ScriptSigS_{P2SSIG}$" would fail.

If the malicious sender M's script "$ScriptSigS_{P2SSIG}$" does not include the operation "OP_P2SSIG," the transaction will fail. This is because the recipient B's address type is "P2SSIG" and a transactions including a P2SSIG type address as output's address should have the sender M's script "$ScriptSigS_{P2SSIG}$" including "OP_P2SSIG". Thus, the recipient B's script "$ScriptPubkeyS_{P2SSIG}$" cannot be abused in any other transaction except for the transaction $TxID01$. □

Theorem 2 (Sender's Security): If, on the transaction $TxID01$, the recipient's address type is P2SSIG and the sender's *ScriptSig* is "$ScriptSigS_{P2SSIG}$" and security hash functions, the public key algorithm and the digital signature algorithm used for construction of scripts are secure, no one who does not know the sender A's security key SK_A, can set the script $ScriptSigS_{P2SSIG}$ or parts of the script including the sender's signature for the transaction $TxID01$ and his public key PK_A on any other committable transaction except for $TxID01$.

Proof: In our address type, all outputs' scripts "*ScriptPubkey*" are included on the signature targeted message by the sender A. The signed signature $Sig_{SK_A}(TxID01) = SIG_{SK_A}$ and the sender's public key PK_A cannot be used in any other committable transaction with outputs which are difference from outputs of the transaction $TxID01$.

On the other hand, the sender A's script "$ScriptSigS_{P2SSIG}$" is not included in the sender A's signed target message. There might exist possibility that the recipient B constructs her signature $Sig_{SK_B}(\text{TxID01}) = SIG'_{SK_B}$ different from SIG_{SK_B}, replaces the recipient B's signature SIG_{SK_B} of the sender A's script "$ScriptSigS_{P2SSIG}$" by her reconstructed signature SIG'_{SK_B}, and publishes the rewrote transaction. However, when our script commands "OP_P2SSIG" is executed for checking of the verification of the recipient's signature, it is also checked whether the hash value of the recipient B's signature SIG_{SK_B} matches with the first hash value of the recipient's script "$ScriptPubkeyS_{P2SSIG}$" or not. Since the recipient B's script "$ScriptPubkeyS_{P2SSIG}$" is included in the message targeted for signature by the sender A, the equality checking in the rewrote transaction published by the recipient B will fail. This is why the commands "OP_P2SSIG" should consist of the verification of the recipient's signature SIG_{SK_B} and the checking of equality of the hash value of SIG_{SK_B} and the first hash value of the recipient B's script "$ScriptPubkeyS_{P2SSIG}$." And, the two checks should be executed atomically in order to protect the sender A's security from the recipient B as well as the recipient B's security from the sender A. □

VII. EXPERIMENTAL RESULTS

We implemented our protocol for constructing transactions and a verification procedure for our scripts on a laptop computer with Quad-Core Intel Core i5 processor (6MB Cache, up to 1.4GHz) using the libbitcoin library v3.6.0 (https://github.com/libbitcoin/libbitcoin-server/releases) which is Bitcoin cross-platform C++ development toolkit and C++ (clang version 11.0.3) compiler. The OS of the laptop is MacOS Catalina (version 10.15.5). By using the libbitcoin library, we can simulate protocols with Elliptic Curves public key cryptography packages, the current key lengths and the Elliptic Curve Digital Signature Algorithm (ECDSA), etc., which are the same as ones used in the Bitcoin official client software Bitcoin Core. We also referred to C++ source files of the Bitcoin Core v0.20.0 rc1 in order to analyze some procedures.

First, we compared lengths of the sender's scripts "*ScriptSig*" and the recipient's scripts "*ScriptPubkey*" for our address type "P2SIG" with ones for the previous standard address type "P2PKH." Next, we estimate execution overhead of verification for our scripts toward the standard script verification.

In this section, we assume that the targeted current transaction consists of one input (the sender's address) and one output (the recipient's address), unless specific notices. And, we also assume that the transaction ID of the targeted current transaction is $TxID02$ and the transaction ID of the previous transaction for the input is $TxID01$.

A. OVERHEAD FOR THE LENGTHS OF SCRIPTS

We show our proposed "*ScriptSig*" and "*ScriptPubkey*" structures based on the current libbitcoin library, with the standard ones in Fig. 5 and Fig. 6. In Fig. 5 and Fig. 6, each description as "*Length*" shows the byte length of the succeeding data element.

In our selective mechanism, if the recipient's address type is "P2SIG," the sender's script "*ScriptSig*" should include the recipient's digital signature, such as $ScriptSigS_{P2SIG}(\ )$ which we defined in Section VI. Thus, the structure of the sender's script "*ScriptSig*" is changed depending on the recipient's address type of the current targeted transaction $TxID02$. In Fig. 5, if the sender's address type and the recipient's address type are the standard

ScriptSigX
(The case that type of recipient's address is standard type P2PKH and type of sender's address is standard P2PKH)

Length (1byte)	Signature (71-72 bytes)	Length (1byte)	Public key (33 bytes)

Total (106-107bytes)

ScriptSigX′
(The case that type of recipient's address is our proposed type P2SIG and type of sender's address is standard type P2PKH)

Length (1byte)	Signature (71-72 bytes)	OP_P2SIG (1byte)	Length (1byte)	Signature (71-72 bytes)	Length (1byte)	Public key (33 bytes)

Total (179-181bytes)

ScriptSigY
(The case that type of recipient's address is standard type P2PKH and type of sender's address is our proposed type P2SIG)

Length (1byte)	Signature (71-72 bytes)	Length (1byte)	Public key (33 bytes)	Length (1byte)	Signature (71-72 bytes)

Total (178-180bytes)

ScriptSigY′
(The case that type of recipient's address is our proposed type P2SIG and type of sender's address is our proposed type P2SIG)

Length (1byte)	Signature (71-72 bytes)	OP_P2SIG (1byte)	Length (1byte)	Signature (71-72 bytes)	Length (1byte)	Public key (33 bytes)	Length (1byte)	Signature (71-72 bytes)

Total (251-254bytes)

FIGURE 5. **The primary standard scriptsig structure and our scriptsig structure.**

ScriptPubkeyX
(The case that type of recipient's address is standard type P2PKH)

OP_Dup (1byte)	OP_Hash 160(1byte)	Length (1byte)	Hash value (20bytes)	OP_Equal-Verify(1byte)	OP_CheckSig (1 byte)

Total (25bytes)

ScriptPubkeyY
(The case that type of recipient's address is our proposed type P2SIG)

OP_Hash 160(1byte)	Length (1byte)	Hash value (20 bytes)	OP_Equal-Verify(1byte)	OP_Dup (1byte)	Length (1byte)	Public key (33 bytes)	OP_Concate-nate(1byte)	OP_Hash 160(1byte)	Length (1byte)	Hash value (20 bytes)	OP_Equal-Verify(1byte)	OP_Check Sig(1 byte)

Total (83bytes)

FIGURE 6. **The primary standard scriptpubkey structure and our scriptpubkey structure.**

type "P2PKH," we refer to the structure of the sender's script as "*ScriptSigX*," i.e., $ScriptSig(P2PKH, SIG_{SK_A}, PK_A)$. If the sender's address type is the standard type "P2PKH" and the recipient's address type is our type "P2SIG," we refer to the one as "*ScriptSigX*′," such as $ScriptSigS_{P2SIG}(ScriptSig(P2PKH, PK_A), SIG_{SK_B})$. If the sender's address type is our type "P2SIG" and the recipient's address type is the type "P2PKH," we refer to the one as "*ScriptSigY*," i.e., $ScriptSig(P2SIG, SIG_{SK'_A}, PK'_A, Prev\_SIG_{SK_A})$. If the sender's address type and the recipient's address type are our type "P2SIG," we refer to the one as "*ScriptSigY*′," i.e., $ScriptSigS_{P2SSIG}(ScriptSig(P2SIG, SIG_{SK'_A}, PK'_A, Prev\_SIG_{SK_A}), SK_B)$.

In Fig. 6, when the recipient's address type is standard, we refer to the structure of the recipient's script "*ScriptPubkey*" as "*ScriptPubkeyX*." And, when the recipient's address type is our type "P2SIG," we refer to the structure of the recipient's script "*ScriptPubkey*" as "*ScriptPubkeyY*." Note that the structures of the scripts "*ScriptPubkey*" do not depend on types of other addresses.

From Fig. 5 and Fig. 6, the overheads of our address type "P2SIG" toward the primary standard address type "P2PKH" is 69% on the lengths of "*ScriptSigs*," when either the sender's address or the recipient's address is our type "P2SIG," is 140% on the length of "*ScriptSig*," when both of the sender's address and the recipient's address are our type "P2SIG." On the other hand, the overhead of our address type is 140% on the lengths of the scripts "*ScriptPubkey*."

A main reason for the overheads is the addition of descriptions of the recipient's signature and her public key to the primary standard scripts. If only one public key is used in the scripts "*ScriptSig*" and "*ScriptPubkey*" such as our address type "P2SSIG," we can eliminate the description of the second public key (33 bytes) from "$ScriptSig(Y$ or $Y')$" and the description of the hash value and related operations (25 bytes) from "*ScriptPubkeyY*." However, security of this eliminated version "P2SSIG" type address is the same level as ones of the official but optional P2PK or MultSig type addresses, of which the recipient's script "*ScriptPubkey*" includes descriptions of the recipients' public keys. We adopted the use of 2 public keys such that one is in the scripts "*ScriptPubkeyY*" and the other is in "$ScriptSig(X'$ or $Y')$." Thus, the proposed address type "P2SIG" can have the security level for locking UTXO which is the same as one of the primary standard address type "P2PKH."

B. OVERHEAD FOR NODES' EXECUTION TIME

We estimate execution overheads of our address type P2SIG. We focus on the process of nodes on the peer-to-peer network. Each node on the peer-to-peer network should verify validity of setting of each input address for each transaction, if the node wants to construct a new block.

The verifications are executed by using an input's (the sender A's) "*ScriptSig*" of a targeted current transaction "$TxID02$" and the output's (A's, where A was the recipient) "*ScriptPubkey*" of the previous transaction "$TxID01$," in

$ScriptSigY\_f$ **of** $Tx01$**:** "[$Sig_{SK_B}(Tx01)$] OP_P2SIG"

$ScriptPubkeyY$ **of** $Tx01$**:** "OP_Hash160 [Hash160(SIG_{SK_B})] OP_EqualVerify OP_Dup [PK_B] OP_Concatenate OP_Hash160 [Hash160($PK_B \cdot PK_{B1}$)] OP_EqualVerify OP_CheckSig"

$ScriptSigY\_f\_add$ **of** $Tx01$**:** "[$Sig_{SK_B}(Tx01)$] OP_Dup [PK_B] OP_CkeckSigVerify OP_Hash160 [Hash160(SIG_{SK_B})] OP_EqualVerify"

FIGURE 7. Transformation from *ScriptSigY_f* to *ScriptSigY_f_add* by the append procedure.

which the sender A received the UTXO as a recipient. In the verification, the stack-based programming system executes a script given by connecting the sender's script "*ScriptSig*" of the transaction "*TxID*01" to the recipient's script "*ScriptPubkey*" of the transaction "*TxID*02." Since we wanted to estimate the overheads for node's execution time due to introducing our address type "P2SIG," we decided to compare execution times of variations of verifications which can occur. There exist the following verification procedures for 4 combinations as pairs of a type of the script "*ScriptSig*" of the transaction "*TxID*02" and a type of the script "*ScriptPubkey*" of the transaction "*TxID*01."

The first combination is the pair of types of "*ScriptSig*" and "*ScriptPubkey*," (*ScriptSigX*, *ScriptPubkeyX*). This combination occurs when the type of the sender's address and the type of the recipient's address on the current targeted transaction are the standard type "P2PKH."

The second combination is the pair of types of "*SciptSig*" and "*ScriptPubkey*," (*ScriptSigX*$'$, *ScriptPubkeyY*). This combination occurs when the type of sender's address is the standard type "P2PKH" and the type of recipient's address is our type "P2SIG."

The third combination is the pair of types of "*ScriptSig*" and "*ScriptPubkey*," (*ScriptSigY*, *ScriptPubkeyX*). This combination occurs when the type of sender's address is our type "P2SIG" and the type of recipient's address is the standard type "P2PKH."

The fourth combination is the pair of types of "*ScriptSig*" and "*ScriptPubkey*," (*ScriptSigY*$'$, *ScriptPubkeyY*). This combination occurs when the type of sender's address and the type of recipient's address are our type "P2SIG."

In order to estimate execution overhead of inducing our address type accurately, we also used official operations and script codes of the libbitcoin library as ones used in Bitcoin Core, as much as possible. In Fig. 8, we show the outline of our implemented procedure for executing the script, and in Fig. 9, we also show the outline of our implemented sub-procedure for executing the script "*ScriptPubkey*."

We created instances corresponding to scripts by applying constructors for the "program" abstract data type of the libbitcoin library, and executed instances by the "run" method of the libbitcoin library, in order to run the Reverse Polish interpreter. The "run" method returns the updated instance of the "program" abstract data type saving the stack state, the transaction, input_index, etc. We can apply some operations individually for the updated instance, and also continue the "run" method by giving the re-updated instance as the input.

```
bool Execute_Script( $Tx$, $index$, $PrevScript$, $ADD_B$ )
{
  Script $ScriptSig$ = $Tx\_In[index].ScriptSig$;
  if ( Type($ScriptSig$) is $ScriptSigX$ or $ScriptSigX'$ )
    { program $pg\_sigX$( $ScriptSig$, $Tx$, $index$, $PrevScript$ );
      if ( run ($pg\_sigX$ ) $\neq 0$ )
        return False;
      return Execute_ScriptPubkey( $Tx$, $index$, $PrevScript$,
                                   $ADD_B$, $pg\_sigX$ );
    }
  else if  (Type($ScriptSig$) is $ScriptSigY$ or $ScriptSigY'$ )
    { Script $SSigY\_f$, $SSigY\_b$;
      Divide_ScriptSigY( $SSigY\_f$, $SSigY\_b$, $ScriptSig$ );
      Script $SSigY\_f\_add$
        = Append( $SSigY\_f$, $Tx\_Out[0].ScriptPubkey$ );
      Transaction $TxS$ = RemoveHash( $Tx$ );
      program $pg\_foward$( $SSigY\_f\_add$, $TxS$, $index$,
                             $TxS\_Out[0].ScriptPubkey$ );
      if ( run( $pg\_foward$ ) $\neq 0$ ) return False;
      program $pg\_sigY$( $SSigY\_b$, $Tx$, $index$,
                         $PrevScript$, $pg\_forward$ );
      if ( run( $pq\_sigY$ ) $\neq 0$ )
        return False;
      return Execute_ScriptPubkey( $Tx$, $index$, $PrevScript$,
            $ADD_B$, $pg\_sigY$ );
    } return False;
}
```

FIGURE 8. C++ like Pseudo codes of the our implement of scripts verification procedure.

For processing of our introduced new commands "OP_P2SIG" in the script "*ScriptSigY*," we divide the script "*ScriptSigY*" into two scripts called "*ScriptSigY_f*" and "*ScriptSigY_b*." The script "*Script-SigY_f*" includes recipient's signature and the new code "OP_P2SIG." The script "*ScriptSig_b*" includes the sender's signature and his public key. Thus, the script "*ScriptSig_b*" becomes the same type as "*ScriptSigX*" or "*ScriptSigY*." We executed the new code by constructing an alternative script *ScriptSigY_f_add* by adding an "OP_Dup" code, a copy of the description of the recipient's public key on her script "*ScriptPubkeyY*," an "OP_CheckSigVerify" code, an "OP_Hash160" code, a copy of the first hash value on the recipient's script "*ScriptPubkeyY*," and an "OP_EqualVerify" code to the script "*ScriptSigY_f*" as shown in Fig.7 and running a program based on the alternative script as the Reverse Polish interpreter. For verification of the first signature

```
bool Execute_ScriptPubkey( $Tx$, $index$, $PrevScript$, $ADD_B$,
        $pg\_sig$ )
{
  if ( Type($PrevScript$) = $ScriptPubkeyX$ )
    { program $pg\_pkX$( $PrevScript$, $Tx$, $index$, $pg\_sig$ );
      if ( run( $pg\_pkX$ ) $\neq$ 0 )
        return False;
      return True;
    }
  else if (Type( $PrevScript$ ) = $ScriptPubkeyY$ )
    { Script $SPkY\_f$, $SPkY\_b$;
      Divide_ScriptPubkeyY( $SPkY\_f$, $SPkY\_b$, $PrevScript$ );
      program $pg\_foward$( $SPkY\_f$, $Tx$,
                                $index$, $pg\_sig$ );
      if ( run( $pg\_foward$ ) $\neq$ 0 )
        return False;
        data $dataB$ = $pg\_forward$.pop( ) ;
        data $dataA$ = $pg\_forward$.pop( );
        data $result$ = Concatenate( $dataA$, $dataB$ );
        $pg\_forward$.push( $result$ );
      program $pg\_PkY$( $SPkY\_back$, $Tx$, $index$, $pg\_forward$ );
      if ( run( $pq\_PkY$ ) $\neq$ 0 )
        return False;
      return True;
    } return False;
}
```

FIGURE 9. C++ like Pseudo codes of the our implement of *ScriptPubkey* verification procedure.

of "*ScriptSigY*" or "*ScriptSigY'*," we should give the transaction information *TxS* constructed by deleting the first hash value from the recipient's *ScriptPubkey* of *Tx* to the construction method of the "program." We assume that the procedure RemoveHash(*Tx*) can output the transaction information *TxS*.

For processing of our introduced second new command "OP_Concatenate," we also divide the script "*ScriptPubkeyY*" into two scripts, called "*ScriptPubkeyY_f*" and "*ScriptPubkeyY_b*." The script "*ScreptPubkeyY_f*" includes an "OP_Hash160" code, the first hash value, an "OP_EqualVerity" code, an "OP_Dup" code, the recipient's public key and the new code "OP_Concatenate." "*ScriptPubkeyY_b*" includes the residual sequence of the script "*ScriptPubkeyY*," i.e., an "OP_Hash160" code, the second hash value, "OP_EqualVerify" and "OP_ChecksSig" codes. For the new code execution, we deleted the new code "OP_Concatenate" from the script "*ScriptPubkeyY_f*." We created instances corresponding to the result "*ScriptPubkeyY_f*" by using the "program" constructor and executed an instance of the script "*ScriptSig*" and the script "*ScriptPubkeyY_f*" instance by using the "run" command. Next, we popped two public keys, PK_B and PK'_B from the top of the run stack, and pushed data constructed by the string concatenation of PK_B and PK'_B on the top of the run stack.

The run stack was passed to the interpreter before running the instance corresponding to the script "*ScriptPubkeyY_b*." We regarded CPU time for the above series of processing for the script "*ScriptSigY*" as processing time of the script "*ScriptSigY*." We show average execution times of 10 time

TABLE 1. Execution times of 4 combinations.

Combination	CPU time [μsec.]	Overhead ratio to standard combination
Standard type ($ScriptSigX$, $ScriptPubkeyX$)	178.1	-
($ScriptSigX'$, $ScriptPubkeyY$)	360.9 (221.5)	2.03 (1.24)
($ScriptSigY$, $ScriptPubkeyX$)	259.6 (259.6)	1.46 (1.46)
($ScriptSigY'$, $ScriptPubkeyY$)	398.8 (311.4)	2.24 (1.75)

executions for verification of each combination on Tab. 1. We also show average run times except times for modifying input scripts, in parentheses on the Table.

From Tab. 1, we can see that, in case of partial application of our address, the execution time of verification of signature was at most twice of one in the standard case. If the targeted transaction has multi-inputs and only one or two our type address exists in the inputs' addresses and outputs' addresses, the overhead will become lighter.

From the result of the combination (*ScriptSigY'*, *ScriptPubkeyY*), when the number of outputs is no more than the number of inputs, even if all inputs' address and all outputs' addresses are our type P2SIG, the verification execution time will be at most 2.3 times.

As mentioned above, in this experiment, we executed new commands by interruptions of verification running and insertions of some operations. If we can add the new commands to the libbitcoin library methods, the overhead will be enable to be reduced by smooth transfer of information used by two program commands and two run commands. In this case, we estimate that the overheads will be no more than ones shown in parentheses on Tab. 1.

VIII. CONCLUSION

We pointed out the new concern for blockchain by abusing recipient's address of previous published transaction. Then, we proposed a protocol for publishing transactions, and introduced a new structure of transaction and a new type of addresses. We implemented our proposed selective mechanism as Bitcoin protocol. Experimental results showed that our proposed mechanism has reasonable overheads.

As future work, we will propose and implement a wallet mechanism for managements of the *S*-addresses and the UTXO as candidates of a part of Bitcoin client software. Next, we will also investigate criticality of our pointed out concern for smart contract on blockchain, and try to incorporate our proposed mechanism into the Ethereum software.

APPENDIX A
BITCOIN ADDRESS TYPES

In Section III, we have described that there exist some address types called P2PK (Pay to Public Key), P2PKH (Pay to Public

Key Hash), MultiSig and P2SH (Pay to Script Hash), etc., in Bitcoin. In the following, we explain the P2PKH type and the P2SH type of addresses in more detail, since those are standard address types in Bitcoin.

In the P2PKH address type, the address add_B consists of the description of P2PKH (Pay to Public Key Hash) type, as the prefix of add_B, the hash value of the recipient's public key, and the checksome as the suffix of the address add_B. The checksome also is the 4 highest bytes of the hash value of add_B except for the checksome. It is used for simple checking the correctness of the address add_B.

In the P2SH address type, the recipient can set indication of multiple signatures by multiple public keys or more complex descriptions as a condition for transferring the UTXO. The description of the conditions for transferring the UTXO is called "redeem script." In the P2SH type address, the hash value of the condition "redeem script" is included in the address add_B, instead of the hash value of the recipient's public key in the P2PKH type address.

APPENDIX B
EXAMPLE OF VERIFICATION FOR BITCOIN SCRIPTS

In the following, we show the picked out descriptions of "*ScriptPubkey*" of the transaction *TxID*01 and "*ScriptSig*" of the transaction *TxID*02 from Fig. 3 and Fig. 4.

ScriptPubkey: "OP_Dup OP_Hash160 [Hash160(PK_A)] OP_EqualVerify OP_CheckSig"

ScriptSig: "[SIG_{SK_A}][PK_A]"

In more detail, the system pushes two data elements as [SIG_A] and [PK_A] of the sequence "*ScriptSig*" on *TXID*02, on the stack. Next, it applies the operation OP_Dup for the stack top data element as [PK_A], which results in three data elements as a duplicate of [PK_A], [PK_A], and [SIG_A] on the stack. Second, the stack top data element will be popped and be applied the operation Hash160 function for the elements and the return value [ResultA] will be pushed on the stack. Third, [Hash160(PK_A)]" will be pushed in the stack and for the OP_EqualVerify operation, two values [Hash160(PK_A)] and [ResultA] will be popped from the stack, and if the two values are equal, the process will continue, otherwise the verification process will stop and return False value.

Finally, [SIG_A] and [PK_A] will be popped from the stack and be checked whether the owner of [PK_A] appropriately admits the transfer of *TxID*02 or not. If validity of the digital signature is verified, "True" value will be pushed in the stack and the process will return with "True" value. Otherwise, "False" value will be pushed in the stack and the process will return with "False" value.

APPENDIX C
EXAMPLE FOR CONSTRUCTION OF SCRIPTS FOR P2SSIG TYPE OF *S*-ADDRESS

We show an example of the sender's *ScriptSig* and the recipient's *ScriptPubkey* on the transaction $Tx01$ and her next unlocking script *ScriptSign* on the transaction $Tx02$, as follows. In the following, we assume that $Sig_{SK_B}(Tx01) = SIG_{SK_B}$.

If the recipient B wants to receive the transferring value by commitment of the transaction *TxID*01, she will make her digital signature by using her public key "PK_B." First, she makes a "ScriptPubkey" by using her public key "PK_B" and only one operation "OP_CheckSig." Next, she makes a digital signature with her public key "PK_B" for the transaction information with her "ScriptPubkey" in one of output data elements of the type "*Tx_Out*."

Next, the recipient B sends her digital signature and an output transaction information with her "ScriptPubkey" to the sender. In case of the address type P2SSIG, the recipient B makes her digital signature by inputting a targeted message as the description of the current transaction *TxID*01 except for inputs' ScriptSigs, to which the description of the recipient's ScriptPubkey is appended. Thus, the recipient's ScriptPubkey occurs two times in the targeted message. One is as the ScriptPubkey of the output in the transaction *TxID*01 and the other is as the appended ScriptPubkey. Because of this construction method, the signature cannot be used as a counterfeit signature for any other recipient's P2SSIG address. Finally, the recipient B also sends her address made from a hash value of her digital signature "SIG_{SK_B}" and the checksome, to the sender.

The sender A should check the validity of the address sent from the recipient B using the checksome. If the checking is success, the recipient's signature will be checked, otherwise the sender should require the recipient B to send the address again. If the next checking is also success, the third checking will examine whether the hash value of the recipient's signature is corresponding to the recipient's address, otherwise the sender A should require the recipient B to send the recipient's signature again.

If the third checking is also success, the sender A makes his "ScriptSig" with the sender's digital signature and his public key "PK_A" in addition to recipient's digital signature and our proposed new operation "OP_P2SSIG." The operation "OP_P2SSIG" is to continue if the checking equality of the hash value of the recipient's digital signature and the first data element of the recipient's script "ScriptPubkey" is success and the verification of the recipient's digital signature with "PK_B," which is the second data element in the recipient's "ScriptPubkey," is success, stop otherwise. The sender A publishes the transaction information on the peer-to-peer network. After committing of the transaction, if the recipient B wants to transfer the value to a next recipient, she can unlock the UTXO by making her digital signature for a new transaction having the transaction identification *TxID*01 and her "ScriptSig" as one information of senders and constructing her ScriptSig with her digital signature.

ACKNOWLEDGMENT
This article was presented at the Proceedings of the COMPSAC 2019, Milwaukee, USA, 2019, pp.934-935.

REFERENCES

[1] A. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Crypto-Currencies*. Newton, MA, USA: O'relly Media, Dec. 2014.

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts SoK," in *Proc. Int. Conf. Principles Secur. Trust*, Apr. 2017, pp. 164–186.

[3] A. Back. (2002). *Hashcash—A Denial of Service Counter-Measure*. [Online]. Available: http://www.hashcash.org/papers/hashcash.pdf

[4] V. Buterin. (Jan. 2014). *Ethereum's White Paper: A Next-Generation Smart Contract and Decentralized Application Platform*. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[5] G. Chander, P. Deshpande, and S. Chakraborty, "A fault resilient consensus protocol for large permissioned blockchain networks," in *Proc. IEEE Int. Conf. Blockchain Cryptocurrency (ICBC)*, May 2019, pp. 33–37.

[6] CoinMarketCap. (Nov. 2020). *Cryptocurrency Market Capitalizations*. [Online]. Available: https://coinmarketcap.com/

[7] C. Team. (Jan. 2019). *Crypto Crime Report—Decoding Increasingly Sophisticated Hacks, Darknet Markets, and Scams*. [Online]. Available: https://blog.chainalysis.com/2019-cryptocrime-review

[8] Y. J. Fanusie and T. Robinson. (Jan. 2018).*Bitcoin Laundering: An Analysis of Illicit Flows Into Digital Currency Services*. [Online]. Available: https://www.fdd.org/analysis/2018/01/10/bitcoin-laundering-an-analysis-of-illicit-flows-into-diigital-currency-services/

[9] Financial Services Agency, Japan and Social ICT Innovation Division of Mitsubishi Research Institute. (Mar. 2019). *Research on Privacy and Traceability of Emerging Blockchain Based on Financial Transactions*. [Online]. Available: https://www.fsa.go.jp/policy/bgin/ResearchPaper_MRI_en.pdf

[10] M. Haywood. (Aug. 2018). *Improving Privacy Using Pay-to-End Point*. [Online]. Available: https://blockstream.com/2018/08/08/improving-privacy-using-pay-to-endpoint/

[11] Y. He, H. Li, X. Cheng, Y. Liu, C. Yang, and L. Sun, "A blockchain based truthful incentive mechanism for distributed P2P applications," *IEEE Access*, vol. 6, pp. 27324–27335, Apr. 2018.

[12] S. K. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey, "Measuring ethereum network peers," in *Proc. Internet Meas. Conf.*, Oct. 2018, pp. 91–104.

[13] S. Nakamoto. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[14] R. Pass and E. Shi, "FruitChains: A fair blockchain," in *Proc. ACM Symp. Princ. Distrib. Comput.*, Jul. 2017, pp. 315–324.

[15] J. Song, *Programming Bitcoin: Learn How to Program Bitcoin from Scratch*. Newton, MA, USA: O'reilly Media, Mar. 2019.

[16] N. Szabo. (1997). *Formalizing and Securing Relationships on Public Networks*. [Online]. Available: https://nakamotoinstitute.org/formalizing-securing-relationships/

[17] K. Rosenbaum, *Grokking Bitcoin*. Shelter Island, NY, USA: Manning Publications, May 2019.

[18] R. Yamauchi, Y. Kamidoi, and S. Wakabayasi, "A protocol for preventing transaction commitment without Recipient's authorization on blockchain," in *Proc. IEEE 43rd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, Jul. 2019, pp. 934–935.

[19] R. Zhang, R. Xue, and L. Liu, "Security and privacy on blockchain," *ACM Comput. Surveys*, vol. 52, no. 3, Jul. 2019, Art. no. 51.

[20] S. Underwood, "Blockchain beyond bitcoin," *Commun. ACM*, vol. 59, no. 11, pp. 15–17, Oct. 2016.

[21] H. Watanabe, S. Ohashi, S. Fujimura, A. Nakadaira, K. Hidaka, and J. Kishigami, "Niji: Bitcoin bridge utilizing payment channels," *Proc. IEEE Int. Conf. Blockchain*, Jul./Aug. 2018, pp. 1448–1455.

YOKO KAMIDOI (Member, IEEE) received the M.E. and Ph.D. degrees in systems engineering from Hiroshima University, Hiroshima, Japan, in 1991 and 1994, respectively. She is currently an Assistant Professor with the Graduate School of Information Sciences, Hiroshima City University, Hiroshima. Her research interests include information sharing on distributed systems, secure computing protocols, and privacy preserving data publishing.

RYOUSUKE YAMAUCHI received the B.S. degree in information sciences from Hiroshima City University, Hiroshima, Japan, in 2019. His research interest includes secure protocols for blockchain.

SHIN'ICHI WAKABAYASHI (Member, IEEE) received the the B.E. degree in electrical engineering and the M.E. and Ph.D. degrees in systems engineering from Hiroshima University, in 1979, 1981, and 1984, respectively. He was a Researcher with the Tokyo Research Laboratory, IBM Japan Ltd., from 1984 to 1988. From 1988 to 2003, he was an Associate Professor with the Faculty of Engineering, Hiroshima University. Since 2003, he has been a Professor with the Graduate School of Information Sciences, Hiroshima City University. His research interests include VLSI design, VLSI CAD, and combinatorial optimization.