On the Consideration of Vanity Address Generation via Identity-Based Signatures

Shogo Murasaki
Kanazawa University, Japan

Kazumasa Omote
University of Tsukuba, Japan

Keita Emura
Kanazawa University, Japan
AIST, Japan

***Abstract*—An address is indicated as an identifier of the user on the blockchain, and is defined by a hash value of the ECDSA verification key. A vanity address is an address that embeds custom characters such as a name. To generate a vanity address, a classical try-and-error method is employed, and thus the number of characters to be embedded is limited. In this paper, we focus on the functionality of identity-based signatures (IBS) where any strings can be employed as a verification key, and explore whether IBS can be used for generating a vanity address. We attach importance to the fact that it is not realistic to replace ECDSA with key recovery, which is currently employed for issuing transactions in Ethereum, to an IBS scheme. Even if this replacement is possible, it is not a reasonable price for the ease of the vanity address generation. Thus, we pay attention to a generic construction of IBS from signatures, and construct an IBS scheme from ECDSA with key recovery. Though we cannot directly generate a vanity address due to the key recovery functionality of the underlying ECDSA, we can connect any string with an address due to the functionality of IBS that can give additional meaning to the address. We implement our system by Solidity, and demonstrate that the gas cost is almost same as that of the ECDSA signature verification.**

***Index Terms*—Blockchain, Vanity address, Identity-based signatures, ECDSA with key recovery**

I. Introduction

A User signs a transaction using own secret ECDSA signing key when the user issues the transaction in Ethereum, where ECDSA stands for elliptic curve digital signature algorithm. An address is indicated as an identifier of the user on the blockchain, and is defined by a hash value of the ECDSA verification key. Precisely, Ethereum Yellow Paper [13] writes that "A Ethereum address is defined as the rightmost 160-bits of the Keccak-256 hash of the corresponding ECDSA public key". Here, an address is looked as a random value because the underlying verification key is generated by a secret signing key that is chosen at random. It is not trivial to check whether an address is valid (meaning that the address is managed by an expected user).

Vanity Address. A vanity address is an address that embeds custom characters such as a name. It is not Ethereum-specific. For example, Edward Snowden publishes a public key of Nostr "npub1sn0wdenkukaku0d9df....".[1] Here, sn0wden (o is replaced to 0) is embedded, and it can be seen as a vanity address. An article (Top 5 Bitcoin Vanity Addresses [5]) reports an address embedding the currently longest character "Embarassable" (actual value is EMBARraSS) and a palindrome address "1234m....U4321". From these examples, it seems that the motivation to embed some meaningful string/character in an address, that is essentially a random value, is relatively popular. We also introduce vanity URLs. The article [10] introduces vanity URLs as follows:

> "*custom-made, easy-to-remember, and they pack a punch when it comes to branding*".

Moreover, it insists that

> "*But why should you care about vanity URLs? Well, they're essential for your brand. They make your web address more user-friendly, more memorable, and they can significantly boost your SEO. They target your audience (buyers) specifically, enhancing the user experience all around. So, if you're serious about taking your branding to the next level, it's time to get familiar with vanity URLs.*

Though this is an article about vanity URLs, it also well explains the effectiveness of vanity address as well .

How to Generate a Vanity Address at Present. A vanity address is generated by a classical try-and-error method as follows.

1) Choose a secret signing key at random.
2) Generate the verification key.
3) Check whether its hash value contains the expected character, and repeat this cycle until the desired result is obtained.

Issues in the Current Generation Method of Vanity Addresses. Here, we discuss issues of the above classical try-and-error method in terms of computational costs and the number of characters to be embedded. To generate a verification key, algebraic operations (additions over an elliptic curve in the case of ECDSA) are required. These operations are quite inefficient compared to computations of a hash function. Moreover, a secret signing key needs to be chosen at random (to prevent the guessing attack) and thus generating a vanity address will not be completed in a realistic amount of time when a relatively long character is considered to be embedded. An article [9] introduces that a Bitcoin vanity address can

This work was done when he was in Kanazawa University. He is currently a master student at Institute of Science Tokyo.
Corresponding Author: k-emura@se.kanazawa-u.ac.jp

[1]https://x.com/Snowden/status/1620790688886718466

be generated by one hour when 5 characters are considered to be embedded, but it requires more than three months if 7 characters are considered to be embedded. Concretely, we tried to generate a Ethereum vanity address using VANITY-ETH [2]. For 7 characters, it displayed that "50% probability: 5 hours, 11 minutes", for 8 characters, it displayed that "50% probability: 3 days, 7 hours", and for 9 characters, it displayed that "50% probability: 1 month, 3 weeks". On the other hands, we can generate a vanity address in seconds when 3 characters are indicated. We remark that, in addition to the number of characters to be embedded, it is restricted that only 0-9 and A-F are possible in Ethereum since each address is expressed by hexadecimal numbers.

Real Incident. To reduce the computational cost, we should not use a relatively small random number to efficiently generate a vanity address. Actually, a vulnerability of a tool for vanity address generation, Profanity, has been reported where a 32-bit seed was set for generating a random number (See CVE-2022-40769 [1]), and a cryptocurrency market maker, Wintermute, was hacked for around $160 million in September 2022.

Our Motivation. In view of the above situation, it seems a natural motivation to embed a character with some meaning to an address, but the number of character is limited as a few words and vanity address generation takes an enormous amount of time. Though the above incident seems an implementation vulnerability, it is highly desirable to propose a method to safety and efficiently generate a vanity address.

As the first attempt, we focus on the functionality of identity-based signatures (IBS) [11] where any strings, say identity ID in the IBS context, can be employed as a verification key, and explore whether IBS can be used for generating a vanity address as follows (we will explain that the attempt fails later).

1) Indicate a character to be embedded to a vanity address.
2) Choose a 128-bit random number.
3) Compute its hash value, and repeat this cycle until the desired result is obtained.
4) Set the random value as ID when a desired address is generated.

Then, it is expected that the number of algebraic operations can be drastically reduced compared to the classical try-and error method. We remark that a key generation center (KGC) is defined that issues a secret signing key for ID using the own master secret key. Then, a key escrow problem happens where the KGC also can generate signatures. To solve the problem, TEE (Trusted Execution Environment) could be employed (See Section IV).

Limitation of the First Attempt. Though IBS seems a promising tool to safety and efficiently generate a vanity address, we attach importance to the fact that it is not realistic to replace ECDSA with key recovery, which is currently employed for issuing transactions in Ethereum, to an IBS scheme. Even if this replacement is possible, it is not a reasonable price for the ease of the vanity address generation.

Our Contribution. In this paper, we pay attention to the fact that IBS can be generically constructed from any signature scheme [4], [6], [7], and construct an IBS scheme from ECDSA with key recovery. Then, we consider whether a vanity address can be generated by using the ECDSA-based IBS scheme via the above attempt. Counterintuitively, we cannot directly generate a vanity address. We stress that, although it can be seen as a negative result, clarifying this counterintuitive fact is also our important contribution.

On the other hands, we can connect any string with an address due to the functionality of IBS that can give additional meaning to the address. For example, a name can be written together with an address, and the unforgeability of IBS *cryptographically* guarantees that the name is connected to the address. This is a crucial difference from the case that a name is just written together with an address.

We implement our system by Solidity, and demonstrate that the gas cost is twice compared to that of the ECDSA signature verification. We further pointed out that one of two ECDSA signature verification is independent to the message (transaction) in the IBS signature verification procedure, and is for verification of a certification. Since the certification needs to be verified only once, the actual gas cost for verifying a transaction is essentially same as that of the ECDSA signature verification.

Related Work. Baldimtsi et al. [2] showed that the security is preserved even if a part of the output of a hash function is previously indicated in terms of the bit security framework defined by Watanabe and Yasunaga [12]. Though the main purpose of Baldimtsi et al. is to reduce the storage cost by fixing a part of hash value, they mentioned about vanity addresses. To the best of our knowledge, this is the only work that considers vanity addresses from the cryptographic point of view, and they did not consider IBS in their paper.

II. ECDSA with Verification Key Recovery

In the conventional signature scheme, the verification algorithm takes a verification key in addition to a signature and a message. Concretely, let $\mathsf{SIG} = (\mathsf{Sig.KeyGen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$ be a signature scheme, $(\mathsf{vk_{sig}}, \mathsf{sigk_{sig}}) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$ be a pair of a verification key and a signing key (here, $\lambda \in \mathbb{N}$ is a security parameter), $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sigk_{sig}}, M)$ be a signature on a message M. Then, the verification algorithm is run such that $\mathsf{Sig.Verify}(\mathsf{vk_{sig}}, \sigma, M)$. However, Ethereum employs ECDSA with key recovery where a verification key is recovered from $(sigma, M)$ and the verification algorithm does not take a verification key as input. Concretely, check whether the hash value of the recovered verification key is equal to an address. Precisely, an address is described as $\mathsf{addr} = \mathcal{B}_{96..255}(H(\mathsf{vk_{sig}}))$ where $\mathcal{B}_{96..255}$ is the rightmost 160-bits of the Keccak-256 hash (See Ethereum Yellow Paper [13]). This ECDSA with

[2]ETH vanity address generator: https://vanity-eth.tk/

key recovery is employed as the underlying signature scheme to construct an IBS scheme via the generic construction [4], [6], [7]. The selection is reasonable when the proposed system is considered to be implemented in the actual blockchain environment.

Next, we introduce ECDSA with key recovery as follows. Let p and q be prime numbers, $H : \{0,1\} \to \mathbb{Z}_q$ be a hash function, $E/\mathbb{F}_p$ be an elliptic curve with order q defined over $\mathbb{F}_p$, and $G \in E(\mathbb{F}_p)$ be a base point. Assume that each algorithm implicitly takes (E, G, p, q) as input. We describe a point on E as $R = (R_x, R_y)$. Here, if $R = (R_x, R_y)$ is a point on E, then $-R = (R_x, -R_y)$ is a point on E. To determine R from R_x, a flag v is introduced that indicates whether R_y is greater than $q/2$ or not. That is, $R = (R_x, R_y)$ is uniquely determined by (R_x, v). Let $x \xleftarrow{\$} S$ denote that an element x is chosen at random from a set S.

ECDSA with key recovery

- $\mathsf{ECDSA.Sig.KeyGen}(1^\lambda)$: Choose $d \xleftarrow{\$} \mathbb{Z}_q$ and compute $P = dG$. Output $\mathsf{sigk_{sig}} = d$, $\mathsf{vk_{sig}} = P$, and $\mathsf{addr} = \mathcal{B}_{96..255}(H(P))$.
- $\mathsf{ECDSA.Sig.Sign}(\mathsf{sigk_{sig}}, M)$: Choose Choose $r \xleftarrow{\$} \mathbb{Z}_q$, and compute $h = H(M)$ and $R = rG$. Let $R = (R_x, R_y)$. Compute $s = \frac{h + dR_x}{r} \bmod q$ and output $\sigma = (s, R_x, v)$.
- $\mathsf{ECDSA.Sig.Verify}(\mathsf{addr}, \sigma, M)$: Parse $\sigma = (s, R_x, v)$. Recover $R = (R_x, R_y)$ from (R_x, v), and compute $P = \frac{s}{R_x}(R - \frac{h}{s}G)$. Output 1 if $\mathsf{addr} = \mathcal{B}_{96..255}(H(P))$, and 0 otherwise.

From the original verification equation $\frac{h}{s}G + \frac{R_x}{s}P = \frac{h}{s}G + \frac{R_x d}{s}G = \frac{h + R_x d}{s}G = rG = R$, $P = \frac{s}{R_x}(R - \frac{h}{s}G)$ holds.

Ethereum Yellow Paper [13] stipulates that a signature $\sigma = (s, R_x, v)$ is invalid if $0 < s < q/2 + 1$ does not hold. This is because $(-s, R_x, \bar{v})$ is a valid ECDSA signature when (s, R_x, v) is a valid signature where $\bar{v}$ is the opposite frag of v. More precisely, if $\frac{h}{s}G + \frac{R_x}{s}P = R$ holds, then $\frac{h}{-s}G + \frac{R_x}{-s}P = -R$ holds. The x-coordinate of R and $-R$ are the same and the verification of the original ECDSA checks the x-coordinate only. Thus, in the $\mathsf{ECDSA.Sig.Sign}$, $-s$ is used for generating a signature if $s > q/2 + 1$. We omit this procedure from the description of the $\mathsf{ECDSA.Sig.Sign}$ algorithm above, for the ease of understanding. We do not explicitly consider the range of s anymore in this paper.

III. Generic Construction of IBS from Signatures

In this section, we introduce a generic construction of IBS $\mathsf{IBS} = (\mathsf{IBS.Setup}, \mathsf{IBS.KeyDer}, \mathsf{IBS.Sign}, \mathsf{IBS.Verify})$ from a signature scheme $\mathsf{SIG} = (\mathsf{Sig.KeyGen}, \mathsf{Sig.Sign}, \mathsf{Sig.Verify})$ [4], [6], [7]. This construction is so-called "Certificate-based Construction" where the KGC generates a certificate cert for ID using the master secret key msk. In the verification algorithm $\mathsf{IBS.Verify}$, the validity of cert is also checked in addition to the usual signature verification.

- $\mathsf{IBS.Setup}(1^\lambda)$: The setup algorithm takes a security parameter $\lambda \in \mathbb{N}$. Run $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$ and output a pair of a master public key and a master secret key $(\mathsf{mpk}, \mathsf{msk})$.
- $\mathsf{IBS.KeyDer}(\mathsf{msk}, ID)$: The key derivation algorithm takes msk and ID. Run $(\mathsf{vk_{sig}}, \mathsf{sigk_{sig}}) \leftarrow \mathsf{Sig.KeyGen}(1^\lambda)$, compute $\mathsf{cert} \leftarrow \mathsf{Sig.Sign}(\mathsf{msk}, \mathsf{vk_{sig}} || ID)$, and output a secret signing key for ID $\mathsf{sigk_{ibs}} = (\mathsf{cert}, \mathsf{vk_{sig}}, \mathsf{sigk_{sig}})$.
- $\mathsf{IBS.Sign}(\mathsf{sigk_{ibs}}, M)$: The signing algorithm takes $\mathsf{sigk_{ibs}}$ and M to be signed. Parse $\mathsf{sigk_{ibs}} = (\mathsf{cert}, \mathsf{vk_{sig}}, \mathsf{sigk_{sig}})$. Compute $\sigma \leftarrow \mathsf{Sig.Sign}(\mathsf{sigk_{sig}}, M)$ and output a signature $\sigma_{\mathsf{ibs}} = (\sigma, \mathsf{vk_{sig}}, \mathsf{cert})$.
- $\mathsf{IBS.Verify}(\mathsf{mpk}, ID, \sigma_{\mathsf{ibs}}, M)$: The verification algorithm takes mpk, ID, σ_{ibs}, and M. Parse $\sigma_{\mathsf{ibs}} = (\sigma, \mathsf{vk_{sig}}, \mathsf{cert})$. Output 1 if both $\mathsf{Sig.Verify}(\mathsf{mpk}, \mathsf{cert}, \mathsf{vk_{sig}} || ID) = 1$ and $\mathsf{Sig.Verify}(\mathsf{vk_{sig}}, \sigma, M) = 1$ hold, and 0, otherwise.

If the underlying signature scheme is unforgeable (i.e., EUF-CMA secure where EUF-CMA stands for existential unforgeability under chosen message attack), then the IBE scheme is also unforgeable (See [4], [6], [7]). Basically, no adversary can produce a valid signature under some ID even the adversary obtains signing keys of other identities.

Next, to clarify the case that the above generic construction is instantiated by ECDSA with key recovery, we introduce the IBS scheme as follows. Due to the key recovery functionality, we replace $\mathsf{vk_{sig}}$ contained in $\sigma_{\mathsf{ibs}} = (\sigma, \mathsf{vk_{sig}}, \mathsf{cert})$ to addr. Moreover, we replace mpk that is an input of the $\mathsf{IBS.Verify}$ algorithm to $\mathsf{addr_{KGC}}$.

- $\mathsf{IBS.Setup}(1^\lambda)$: Choose $d_{\mathsf{KGC}} \xleftarrow{\$} \mathbb{Z}_q$ and compute $P_{\mathsf{KGC}} = d_{\mathsf{KGC}}G$. Output $\mathsf{msk} = d_{\mathsf{KGC}}$, $\mathsf{mpk} = P_{\mathsf{KGC}}$, and $\mathsf{addr_{KGC}} = \mathcal{B}_{96..255}(H(P_{\mathsf{KGC}}))$.
- $\mathsf{IBS.KeyDer}(\mathsf{msk}, ID)$: Parse $\mathsf{msk} = d_{\mathsf{KGC}}$. Choose $d \xleftarrow{\$} \mathbb{Z}_q$ and compute $P = dG$. Choose $r \xleftarrow{\$} \mathbb{Z}_q$ and compute $h_{ID} = H(P||ID)$ and $R = rG$. Let $R = (R_x, R_y)$. Compute $s = \frac{h_{ID} + dR_x}{r} \bmod q$ and set $\mathsf{cert} = (s, R_x, v)$. Output $\mathsf{sigk_{ibs}} = (\mathsf{cert}, P, d)$.
- $\mathsf{IBS.Sign}(\mathsf{sigk_{ibs}}, M)$: Parse $\mathsf{sigk_{ibs}} = (\mathsf{cert}, P, d)$ and $\mathsf{cert} = (s, R_x, v)$. Let $\mathsf{addr} = \mathcal{B}_{96..255}(H(P))$. Choose $r' \xleftarrow{\$} \mathbb{Z}_q$ and compute $h = H(M)$ and $R' = r'G$. Let $R' = (R'_x, R'_y)$. Compute $s' = \frac{h + dR'_x}{r} \bmod q$ and set $\sigma = (s', R'_x, v')$. Output $\sigma_{\mathsf{ibs}} = (\sigma, \mathsf{addr}, \mathsf{cert})$.
- $\mathsf{IBS.Verify}(\mathsf{addr_{KGC}}, ID, \sigma_{\mathsf{ibs}}, M)$: Parse $\sigma_{\mathsf{ibs}} = (\sigma, \mathsf{addr}, \mathsf{cert})$, $\sigma = (s', R'_x, v')$, and $\mathsf{cert} = (s, R_x, v)$. Compute $h = H(M)$. Compute $R' = (R'_x, R'_y)$ from (R'_x, v'), and compute $P = \frac{s'}{R'_x}(R' - \frac{h}{s'}G)$. Compute $h_{ID} = H(P||ID)$. Compute $R = (R_x, R_y)$ from (R_x, v). Compute $P_{\mathsf{KGC}} = \frac{s}{R_x}(R - \frac{h_{ID}}{s}G)$. Output 1 if both $\mathsf{addr_{KGC}} = \mathcal{B}_{96..255}(H(P_{\mathsf{KGC}}))$ and $\mathsf{addr} = \mathcal{B}_{96..255}(H(P))$, and 0, otherwise.

Evaluation on the IBS scheme based on ECDSA with key recovery. Here, we consider whether a vanity address can be generated by using the ECDSA-based IBS scheme via the above attempt. First, let us check the impact of introducing

$\mathsf{addr}_{\mathsf{KGC}}$. Now, all users are required to manage cert which is a valid ECDSA signature under $\mathsf{addr}_{\mathsf{KGC}}$. Here, the corresponding message is $P||ID$, and does not a transaction. We may consider a case that a user sets a transaction as ID. However, no cryptocurrency is stored on $\mathsf{addr}_{\mathsf{KGC}}$ and the corresponding message contains P in addition to ID. Thus, we conclude that introducing $\mathsf{addr}_{\mathsf{KGC}}$ does not affect the security.

Second, let us check the impact of introducing ID. Since the underlying ECDSA provides the key recovery functionality, ID needs to be recovered if ID is required to be a verification key. That is, even if we can suitably set ID to produce a vanity address via the procedure (introduced in Our Motivation part), ID additionally needs to satisfy

$$ID = \frac{s'}{R'_x}(R' - \frac{h}{s'}G)$$

This indicates that the KGC needs to find d satisfying $ID = dG$, and it requires the same procedure of the classical try-and-error method. Moreover, mpk is also required for signature verification (i.e., even if the hash value of ID can be set as a desired vanity address, signatures are not verified by ID only).

To sum up, the IBS scheme is not directly employed to generate a vanity address because:

- Due to the verification process of ECDSA with key recovery, ID needs to be recovered from a ECDSA signature and a message in the ECDSA-based IBS scheme that requires the same procedure of the classical try-and-error method.
- Due to the syntax of IBS where, in addition to ID, the master public key is required for running the verification algorithm. Here, the master public key is an ECDSA verification key in the ECDSA-based IBS scheme.

IV. Proposed System

Due to the evaluation in the previous section, we assign a desired value to ID directly (this is not the same as the procedure introduced in Our Motivation part). Let $\mathsf{addr} = \mathcal{B}_{96..255}(H(P))$ be an address of a user where P is contained in $\mathsf{sigk}_{\mathsf{ibs}} = (\mathsf{cert}, P, d)$. If P is connected to ID (i,e, cert is a valid ECDSA signature on $P||ID$), we say that the user of $\mathsf{addr} = \mathcal{B}_{96..255}(H(P))$ is ID.

- A user selects ID (as a desired value such as the user's name), and obtains $\sigma_{\mathsf{ibs}} = (\sigma, \mathsf{addr}, \mathsf{cert})$ from the KGC. The user sets addr as own address and opens addr together with ID.
- When the user issues a transaction M, the user generates $\sigma_{\mathsf{ibs}} = (\sigma, \mathsf{addr}, \mathsf{cert})$ using the $\mathsf{IBS.Sign}$ algorithm. Set $(ID, \sigma, \mathsf{cert})$ be a signature on M.
- A transaction verifier recovers P from $\sigma = (s', R'_x, v')$ and also recovers P_{KGC} from P, ID, and $\mathsf{cert} = (s, R_x, v)$. The verifier accepts that the transaction issuer is ID if both $\mathsf{addr}_{\mathsf{KGC}} = \mathcal{B}_{96..255}(H(P_{\mathsf{KGC}}))$ and $\mathsf{addr} = \mathcal{B}_{96..255}(H(P))$.

Since two signatures are verified by the transaction verifier, introducing the IBE scheme does not affect security compared to the case that ECDSA with key recovery is employed. More concretely, the procedure that the transaction verifier recovers P from $\sigma = (s', R'_x, v')$ and checks $\mathsf{addr} = \mathcal{B}_{96..255}(H(P))$ is the totally the same as the transaction verification procedure in Ethereum. Additionally, the transaction verifier recovers P_{KGC} from P, ID, and $\mathsf{cert} = (s, R_x, v)$ and checks $\mathsf{addr}_{\mathsf{KGC}} = \mathcal{B}_{96..255}(H(P_{\mathsf{KGC}}))$. We remark that the verification of cert (i.e., verification of whether the user of addr is ID) is independent to the underlying transaction M. Thus, cert only needs to be verified once, and introducing the IBE scheme does not affect security in terms of transaction verification.

Note that we need to consider the key escrow problem where the KGC also can generate signatures. To solve the problem, TEE (Trusted Execution Environment) could be employed: Assume that an enclave stores the master secret key. A user sends ID and a public key pk of a public key encryption scheme to a TEE via a remote attestation. Then, the TEE generates a secret singing key for ID on the enclave, encrypts the secret signing key by pk, and returns the ciphertext to the user. Then, the user can obtain the secret signing key by decrypting the ciphertext. Our work is regarded as the first stepping stone to employ IBS in the blockchain environment and further evaluation of the key generation procedure is left as a future work of this paper.

V. Efficiency Evaluation

In this section, we implement the IBS scheme based on ECDSA with key recovery by Solidity, and check the gas cost for the signature verification. To the best of our knowledge, no other scheme employing IBS for generating vanity address has been considered, as mentioned in Related Work section. Moreover, no attempt to construct an IBE scheme from ECDSA with key recovery also exists. In this perspective, we compare the performance of our system with ECDSA with key recovery. Intuitively, the gas cost is twice compared to that of the ECDSA signature verification since two ECDSA verification procedures are run in the IBS signature verification procedure, for cert and σ. As mentioned above, the verification of cert is independent to the underlying transaction M and cert only needs to be verified once. That is, the gas cost is almost same as that of the ECDSA signature verification after cert has been verified.

Our implementation environment is described as follows: Precision Tower3431 (Processor: 3.10 Ghz octa-core Intel Core i9, Memory: 16 GB). For signature generation, we employed libraries, ethereumjs-util and ethereumjs-wallet, which are run on node.js (v20.17.0). For signature verification, we implement a smart contract using Solidity (we indicates the version as >=0.7.0<0.9.0). We set ID as 128-bit value since it can express 13 characters by ASCII codes and seems sufficient to express a name, an e-mail address, and so on.

Our Solidity code is described in Listing 1 as follows. Here, let MSG be a message to be sent, SIGNER_ADDRESS be the address of the transaction issuer, (s, Rx, v) be a signature on MSG, SIG_PBK_ID be a strong that contains the issuer's public key and arbitrary string ($P||ID$), KGC_ADDRESS be

the address of the KGC ($\mathsf{addr}_{\mathsf{KGC}}$), and (CERT_s, CERT_Rx, CERT_v) be a signature on SIG_PBK_ID. Let the function ECDSA.Sig.Verify() be a verification algorithm that checks whether MSG is sent from the issuer and the function Cert.Verify() be a verification algorithm that checks whether the public key of the transaction issuer is associated to ID.

```
string MSG;
address SIGNER_ADDRESS;
bytes32 s;
bytes32 Rx;
uint8 v;

string SIG_PBK_ID;
address KGC_ADDRESS;
bytes32 CERT_s;
bytes32 CERT_Rx;
uint8 CERT_v;

function ECDSA.Sig.Verify() public view returns (bool) {
  bytes32 msgHash = keccak256(bytes(MSG));
  address signer = ecrecover(msgHash, s, Rx, v);
  if (signer == SIGNER_ADDRESS) {
      return true;
  } else {
      return false;
  }
}

function Cert.Verify() public  view returns (bool){
 bytes32 msgHash = keccak256(bytes(SIG_PBK_ID));
 address signer = ecrecover(msgHash, CERT_s, CERT_Rx, CERT_v);
 if (signer == KGC_ADDRESS) {
      return true;
 } else {
     return false;
 }
}
```

Listing 1. Example of basic Solidity Code of IBS Signature Verification

We denote gas costs for running each function in Table I. We used USD price on October 22, 2024. Remark that $\mathsf{IBS.Verify}(\mathsf{addr}_{\mathsf{KGC}}, ID, \sigma_{\mathsf{ibs}}, M)$ runs both ECDSA.Sig.Verify() and Cert.Verify().

TABLE I
GAS COSTS

	ECDSA.Sig.Verify()	Cert.Verify()
Gas	21,849	26,251
USD	0.77	0.92

We expected that the verification costs of IBS signature constructed by ECDSA with key recovery is twice as those of ECDSA with key recovery. However, the actual gas cost is approximately 2.2 times higher. The reason behind is that the cost of Cert.Verify() is 1.2 times higher than that of ECDSA.Sig.Verify(). Here, the message to be signed is $P||ID$ in Cert.Verify() and the size of P is 64 bytes (512 bits) (See Ethereum Yellow Paper [13]) and the size of the message is 640 bits. On the other hands, the message M is a transaction encoded by RLP (Recursive Length Prefix) and is represented by 256 bits. Since the hash value of the message is computed in the ECDSA signature verification procedure, the difference of the underlying message size is the main reason why Cert.Verify() requires a higher gas cost than that of ECDSA.Sig.Verify(). As above mentioned, however, the verification of cert is independent to the underlying transaction M and cert only needs to be verified once. That is, the gas cost for verifying a transaction is almost same as that of the ECDSA signature verification after cert has been verified.

VI. CONCLUSION

In this paper, we consider whether IBS can be employed to generate a vanity address and demonstrate that the IBS scheme (constructed by ECDSA with key recovery) is not directly employed to generate a vanity address. As the next attempt, we propose a method to connect any value to an address using IBS. The actual gas cost for verifying a transaction is almost same as that of the ECDSA signature verification after cert has been verified. Since the experimental simulation is overly simple, considering comprehensively blockchain performance is left as a future work. Moreover, implementation evaluation of the key generation procedure using TEE is also an important future work.

We remark that we do not deny any possibility to generate a vanity address safety and efficiently via an IBS scheme. Even if we turn blind eye the key recovery functionality of ECDSA, we need to consider how to treat mpk when an IBS signature is verified. We may be able to employ the Barreto et al. IBS scheme [3] because Liu et al. [8] proposed a signature scheme (for enhancing the security of stealth address) using an IBE scheme that does not require mpk for the signature verification procedure, and introduced that the Barreto et al. IBS scheme as such an IBS scheme. Further considering vanity address generation methods via IBS is left as a future work.

Acknowledgment: The authors thank Mr. Kota Chin for his invaluable comment against Ethereum vanity address. This work was supported by JSPS KAKENHI Grant Numbers JP21K11897, JP23K24844, and JP25H01106.

REFERENCES

[1] CVE-2022-40769, 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-40769.
[2] F. Baldimtsi, K. Chalkias, P. Chatzigiannis, and M. Kelkar. Truncator: Time-space tradeoff of cryptographic primitives. Financial Cryptography and Data Security 2024, to appear. Available at https://eprint.iacr.org/2022/1581.
[3] P. S. L. M. Barreto, B. Libert, N. McCullagh, and J. Quisquater. Efficient and provably-secure identity-based signatures and signcryption from bilinear maps. In *ASIACRYPT*, pages 515–532, 2005.
[4] M. Bellare, C. Namprempre, and G. Neven. Security proofs for identity-based identification and signature schemes. In *EUROCRYPT*, pages 268–286, 2004.
[5] J. Buntinx. Top 5 Bitcoin Vanity Addresses (2017-01-16), 2017. https://themerkle.com/top-5-bitcoin-vanity-addresses/.
[6] Y. Dodis, J. Katz, S. Xu, and M. Yung. Strong key-insulated signature schemes. In *Public Key Cryptography*, pages 130–144, 2003.
[7] C. Gentry and A. Silverberg. Hierarchical ID-based cryptography. In *ASIACRYPT*, pages 548–566, 2002.
[8] Z. Liu, G. Yang, D. S. Wong, K. Nguyen, and H. Wang. Key-insulated and privacy-preserving signature scheme with publicly derived public key. In *IEEE EuroS&P*, pages 215–230, 2019.

[9] H. Partz. How to put words into a bitcoin address? Here's how vanity addresses work (2023-08-23), 2023. https://cointelegraph.com/news/how-vanity-addresses-work.
[10] K. Pratt. Unlocking brand power: A comprehensive guide on vanity urls and why they matter, May 14, 2024.
[11] A. Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO*, pages 47–53, 1984.
[12] S. Watanabe and K. Yasunaga. Bit security as computational cost for winning games with high probability. In *ASIACRYPT*, pages 161–188, 2021.
[13] G. Wood. Ethereum yellow paper (Shanghai version 47e97f5-2024-08-26), 2024. https://ethereum.github.io/yellowpaper/paper.pdf.