Octopus Contract and its Applications

Sora Suegami, Leona Hioki

November 2023

Abstract

We propose the concept of Octopus contracts, smart contracts that operate ciphertexts outside the blockchain. Octopus contracts can control the decryption of the ciphertexts based on on-chain conditions by requesting validators to sign the specified messages. Signature-based witness encryption (SWE) enables users to decrypt them with the signatures. Moreover, Octopus contracts can evaluate arbitrary functions on encrypted inputs with one-time programs built from SWE and garbled circuits. These features extend the functionality of smart contracts beyond the blockchain, providing practical solutions for unresolved problems such as a trustless bridge for a two-way peg between Bitcoin and Ethereum, private AMM, minimal anti-collusion infrastructure without a centralized operator, and achieving new applications such as private and unique human ID with proof of attribution, private computation with web data, and more.

1 Introduction

Smart contracts cannot natively hold, reveal, and manipulate secrets. If they can do this, they can remove trusted third parties from some applications, such as the centralized bitcoin bridgee [6], minimal anti-collusion infrastructure (MACI) [8], and TLS verification [28, 30, 39].

Recently, solutions have been proposed to allow smart contracts to control secrets using new cryptographic schemes [35], e.g., Lit protocol with threshold encryption [29], smart contracts with secret sharing multi-party computation (MPC) [40], and smartFHE with fully homomorphic encryption (FHE) [31]. However, they require an honest majority of custodians to perform new cryptographic operations or intercommunications. In other words, using validators on the existing consensus layer as custodians of these schemes increases their

Sora Suegami invented the idea of using OTPs for private function evaluation and its applications in Section 4. Leona Hioki invented a trustless bitcoin bridge and private AMM. The main writer of Section 1, Subsections 2.1 and 2.2, and Section 4 is Sora Suegami, and that of Abstract, Subsections 2.3, and Section 5 is Leona Hioki. The other sections are written together.

computational and communication costs and requires significant modifications to the node implementation of the validators.

To minimize the additional costs and modifications, we propose Octopus contracts, smart contracts that operate ciphertexts outside the blockchain using signature-based witness encryption (SWE) [14]. They can control the decryption of ciphertexts, that is, decrypting ciphertexts based on on-chain conditions or specifying who can decrypt them after seeing them. However, the validators only have to sign the message specified by the Octopus contract and encrypt this signature with public-key encryption (PKE). The ciphertexts under SWE are secure assuming the honest majority of the validators. These features allow the Octopus contract to provide practical solutions for unresolved problems such as a trustless bridge for a two-way peg between Bitcoin and Ethereum, private Uniswap, and so on.

Beyond the controlled decryption of ciphertexts, by employing one-time programs (OTPs) [18], the Octopus Contract can evaluate arbitrary functions on encrypted inputs **without increasing the validators' works**. We estimate that our scheme is more efficient than existing schemes based on the secret sharing MPC and multi-key FHE as follows:

- The validators' computational and communication costs in MPC-based schemes increase by the evaluated circuit size. In contrast, those of our scheme depend only on the input size because the number of signatures for the OTP is equal to the input size.

- Both FHE-based schemes and our scheme can delegate heavy evaluation of the circuit to an untrusted third party. However, the former requires the heavy computation of lattice cryptography and the proof generation of the evaluation result [31], whereas the latter just evaluates a garbled circuit [37, 5, 9], which only employs a hash function and bit operations in the optimal construction [38]. For example, according to Table 2 in [3], a garbled circuit optimized for private inference of neural networks is approximately seventy times faster than the inference with FHE.

The validators' work in vetKeys [11] is similar to ours: the validators generate BLS signatures for an ID and encrypt the signatures under the user's public key to pass the user a private key of that ID in ID-based encryption. However, it is our novel point to use the signatures for the private evaluation of functions on encrypted inputs.

Despite the above advantages, the Octopus contract using OTPs further relies on rabble MPC, n-of-n MPC among randomly selected people **who are different from the validators**. It is used to generate an OTP while embedding a private key unknown to humans in the evaluated circuit. The rabble MPC is more secure and feasible than existing MPC-based schemes for the following reasons:

- **If at least one participant is honest**, the MPC is secure, i.e., revealing no information other than the OTP.

	Octopus Contract	OTP by Octopus	Multi-key FHE	ADP based Witness Encryption of Blockchain	Intel SGX
Security	Majority of N	Majority of N +1/M honesty	Majority of fixed N	1/N honesty	Hardware
Liveness	Majority of N	Majority of N	Majority of fixed N	No requiement	Hardware
Significant Computing Cost	Only Pairings	Pairings + Garbled Circuit	FHE	3x for each gate (exponential)	within 100M bytes memory
Use Cases	Trustless Bitcoin Bridge, Private AMM, etc..	Generalized Privacy same as Intel SGX	Generalized Privacy same as Intel SGX	Same as Octopus Contract	Generalized Privacy
Language/ Library	Solidity (etc..)	Solidity + Garbled Circuit	Domain Specific Language	ADP	SGX library

N is the number of the validators, and M is the number of the Rabble MPC participants

Figure 1: Comparison of security, liveness, computing cost, use cases, and development environments between our scheme and the existing schemes.

- Even if the MPC fails because some participants leave the MPC, different participants can start new MPCs **any number of times**. Also, many MPCs can be performed in parallel.
- Once the OTP is generated, there is nothing more for the MPC participant to do.

Using the embedded private key, the Octopus contract not only solves unresolved problems, e.g., MACI without a centralized operator [8] and TLS verification without a trusted Notary node [28], but also achieve new applications, e.g., verification of computation by private conditions, private and unique human ID with proof of attribution, and private computation with web data.

Figure 1 summarizes the comparison between our scheme and the existing schemes.

2 Signature-based Witness Encryption

2.1 Definition of SWE

We adopt a SWE scheme defined for t-out-of-n BLS signatures in [14]. It provides the following algorithms. While they are based on Definition 1 of [14], some inputs are omitted or modified.

- $\mathsf{ct} \leftarrow \mathsf{SWE.Enc}(V = (\mathsf{vk}_1, \ldots, \mathsf{vk}_n), h, m)$: it takes as input a set V of n BLS verification keys, a hash h of a signing target T, and a message to be encrypted m. It outputs a ciphertext ct.
- $m \leftarrow \mathsf{SWE.Dec}(\mathsf{ct}, \sigma, U, V)$: it takes as input a ciphertext ct, an aggregate signature σ, two sets U, V of BLS verification keys. It outputs a decrypted message m or the symbol $\perp$.

If more than or equal to t validators of which verification keys are in V, i.e., $|U| \geq t$ and $U \subseteq V$, generate a valid aggregated signature σ for the hash h, the correctness holds, i.e., $\mathsf{SWE.Dec}(\mathsf{SWE.Enc}(V = (\mathsf{vk}_1, \ldots, \mathsf{vk}_n), h, m), \sigma, U, V) = m$. Otherwise, the $\mathsf{SWE.Dec}$ algorithm returns the symbol $\perp$. Therefore, once the validators release the signature σ, anyone can decrypt the ciphertext ct.

2.2 Access-Control of the Signature

We use the same technique as vetKeys [11] to control who will be able to decrypt the ciphertext by encrypting the validators' signatures. Specifically, when a legitimate decryptor provides a public key pubKey of the PKE scheme, each validator publishes an encryption ct_{σ_i} of the signature σ_i under pubKey, i.e., $\mathsf{ct}_{\sigma_i} \leftarrow \mathsf{PKE.Enc}(\mathsf{pubKey}, \sigma_i)$. That decryptor can recover the message by decrypting each ct_{σ_i} with the private key $\mathsf{privKey}$ corresponding to the pubKey, aggregating the recovered signatures into σ_Σ, and decrypting the ciphertext under SWE by σ_Σ. However, the other users cannot do that because they cannot obtain σ_Σ from ct_σ without $\mathsf{privKey}$. Besides, even the validators cannot decrypt them as long as their honest majority does not reveal each signature σ_i.

As proposed in [11], if the PKE scheme is additive-homomorphic, e.g., EC-ElGamal encryption, the decryptor can first compute the encryption of the aggregated signature by computing the weighted sum of the encrypted signatures and then decrypt only the aggregated one. By outsourcing this computation, the decryptor can reduce the computation cost.

2.3 Estimated Benchmark of SWE

We estimate a benchmark of the SWE scheme based on [14] assuming a threshold $\frac{t}{n} = \frac{2}{3}$. For $n = 500$ and $n = 2000$, encryption takes approximately 10 and 60 seconds, and decryption takes around 20 and 350 seconds, respectively. These results suggest the appropriate number of allocated validators for each use case. They also imply that more improvement in the SWE scheme will enhance the security of ciphertexts, i.e., increasing the number of allocated validators, without sacrificing the performance.

3 Octopus Contract

In our scheme, the Octopus contract helps users request the Ethereum validators to sign a specific message for the SWE decryption. Specifically, they work as follows.

1. Firstly, some validators register with and watch the Octopus contract made by an application developer.

2. An encryptor, the user willing to encrypt a message using SWE, calls the Octopus contract to register a signing target T.

3. The Octopus contract records the hash $h := \mathsf{Hash}(\mathsf{PREFIX}, T)$ derived from T. Note that PREFIX is a fixed unique string, which prevents the validators from signing messages for the Ethereum consensus algorithm.

4. The encryptor generates an encryption ct of the messages m under the hash h and n validators' verification keys $V = (\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$ allocated by the Octopus contract.

5. A decryptor, a user willing to decrypt ct, has a PKE key pair $(\mathsf{privKey}, \mathsf{pubKey})$ and calls the Octopus contract, passing the h and the PKE public key pubKey to request validators' signatures.

6. The Octopus contract checks if the decryptor is legitimate based on the required on-chain conditions. If the decryptor does not pass the conditions, the contract rejects the decryptor's request.

7. More than or equal to t validators generate the encryption ct_σ of the aggregated signature σ for the hash h in a way described in Subsection 2.2.

8. The validators provide the Octopus contract with the encryptions ct_σ along with a proof π to prove that they are valid encryptions of the aggregated signatures.

9. The decryptor first decrypts ct_σ with $\mathsf{privKey}$ to obtain the signature σ, and then decrypts ct with σ to recover the messages m.

In this way, the encryptor can encrypt messages under some on-chain conditions without knowing who satisfies the conditions in the future. As long as more than or equal to the threshold of the validators behave honestly, i.e., sign only the message confirmed by the Octopus contract, only the legitimate decryptor can decrypt the ciphertext.

When implementing our scheme, we can prepare a shared smart contract for common management of the registered validators and requests for signatures. Each application contract specifies the signing messages and checks if the decryptor is legitimate.

3.1 Applications

The abstract mechanism of using blockchain states and conditions to decrypt ciphertext outside the blockchain is expected to lead to the creation of many unforeseeable types of applications. However, it has been confirmed that the Octopus Contract can provide solutions to some practical unresolved problems that have already been discussed in the blockchain research space. Below, we present the details of the Trustless Bitcoin Bridge and the Private AMM, as their construction is not self-evident.

3.1.1 Trustless Bitcoin Bridge

1. A depositor, the user willing to deposit bitcoins, calls the Octopus contract.

2. The Octopus contract allocates a unique signing target T and records $h := \mathsf{Hash}(\mathsf{PREFIX}, T)$.

3. Each trusted-setup node NODE_i for $i \in \{1, \ldots, N\}$ generates its deposit partial address dep_i.

4. All dep_i are collected to form a N-of-N multisig address, referred to as the deposit address $\mathsf{depAddr}$.

5. NODE_i encrypts a private key $\mathsf{depPriv}_i$ of its deposit partial address dep_i using SWE and creates a proof with Zero-Knowledge Proof (ZKP) to claim the validity of the encryption. The proof is verified by the contract when the deposit address is registered. In summary,
   $ct_i \leftarrow \mathsf{SWE.Enc}(V, h, \mathsf{depPriv}_i)$
   $pi_i \leftarrow \mathsf{ZKP.Proof}(V, h, ct_i, \mathsf{depPriv}_i)$
   $\mathsf{ctSet} \leftarrow (ct_1, \ldots, ct_N)$ The trusted setup ends here.

6. The depositor deposits bitcoins into the deposit address $\mathsf{depAddr}$ and creates a proof of the deposit. The depositor passes the proof to the tokenized BTC smart contract to receive tokenized BTC (tBTC).

7. A withdrawer, a future holder of tBTC, burns the tBTC to withdraw the corresponding bitcoins. The Octopus contract then emits an event to request the validators to sign h.

8. Following the mechanism of the Octopus contract described above, the withdrawer decrypts all $\{ct_i\}_{i\in[N]}$ to obtain $\{\mathsf{depPriv}_i\}_{i\in[N]}$ and reclaims bitcoins from $\mathsf{depAddr}$.

This mechanism primarily enhances the performance of the previous research result, Trustless Bitcoin Bridge with Witness Encryption (TBBWE) [19], to a practical level. However, unlike TBBWE, it is difficult to perform slashing against attackers if restaking is not used. Also, if restaking is conducted, there is a difference in the size of the validator set compared to TBBWE. In both cases, the amount of each deposit and withdrawal is fixed.

3.1.2 Private AMM

We demonstrate that the Octopus contract can practically create something close to a Private AMM, which was considered difficult to construct based on Barry Whitehat's research [36], though it is not practical in terms of UX. Specifically, to observe the status of the pool, a certain amount of swap is required, and unless the swap is paid every time to observe, the history of the AMM

cannot be traced. While it does not provide complete privacy, it makes tracking activities extremely difficult. Although the construction of more practical versions is expected, this section merely demonstrates feasibility.

1. A liquidity provider (LP) calls the Octopus contract.
2. The Octopus contract allocates a unique signing target T and records $h := \mathsf{Hash}(\mathsf{PREFIX}, T)$.
3. Each trusted-setup node NODE_i for $i \in \{1, \ldots, N\}$ generates its deposit partial address, dep_i.
4. All dep_i are collected to form a N-of-N multisig address, referred to as the deposit address $\mathsf{depAddr}$.
5. NODE_i encrypts a private key $\mathsf{depPriv}_i$ of its deposit partial address dep_i using SWE and creates a proof with ZKP to claim the validity of the encryption. The proof is verified by the contract when the deposit address is registered. In summary,
   $ct_i \leftarrow \mathsf{SWE.Enc}(V, h, \mathsf{depPriv}_i)$
   $pi_i \leftarrow \mathsf{ZKP.Proof}(V, h, ct_i, \mathsf{depPriv}_i)$
   $\mathsf{ctSet} \leftarrow (ct_1, \ldots, ct_N)$ The trusted setup ends here.
6. For asset A, let $\mathsf{UtxoPoolA}$ be a list of UTXOs of that asset in the pool and the encryptions of its ownership keys under SWE, i.e, $(\mathsf{depAddr}, \mathsf{ctSet}) \in \mathsf{UtxoPoolA}$. We define $\mathsf{UtxoPoolB}$ in the same way for asset B. When the LP deposits tokens A and B, it creates a proof of the deposits to the deposit address $\mathsf{depAddr}$ for each token.
7. If the proof is valid, the Octopus contract adds each $(\mathsf{depAddr}, \mathsf{ctSet})$ for tokens A and B to $\mathsf{UtxoPoolA}$ and $\mathsf{UtxoPoolB}$, respectively. The $\mathsf{UtxoPoolA}$ and $\mathsf{UtxoPoolB}$ themselves are also encrypted with SWE, i.e., $ct_{\mathsf{pool}} \leftarrow \mathsf{SWE.Enc}(V, (\mathsf{UtxoPoolA}, \mathsf{UtxoPoolB}), h)$.
8. When trader X sells asset A to get asset B, X deposits asset A to the deposit address $\mathsf{depAddr}$ and creates a proof of this deposit. If the proof is valid, the Octopus contract requests the validators to sign h. Their signature gives X access to $(\mathsf{UtxoPoolA}, \mathsf{UtxoPoolB})$.
9. X calculates the amount of the swapped tokens using the total amount of $\mathsf{UtxoPoolA}$ and $\mathsf{UtxoPoolB}$.
10. Based on the UTXOs in $\mathsf{UtxoPoolB}$, X creates a combination of outgoing UTXOs of which the total amount slightly exceeds the amount of the exact amount of token B swapped from token A. X then sends a transaction to remove these UTXOs from $\mathsf{UtxoPoolB}$ and deposits a certain amount of margin to the contract. If the margin is asset B, the margin is the same amount of the difference between the total amount of the UTXOs and the exact swapped amount. The deposited token B is added to $\mathsf{UtxoPoolB}$.

11. The above transaction makes the contract emit an event to request the validators to sign the hash for each outgoing UTXO. X then decrypts all ciphertexts for those UTXOs with SWE, which gives X access to asset B. X sends the obtained asset B to X's wallet in a completely private manner.

4 Extending Octopus Contract with OTP

4.1 Basic Ideas

The Octopus contract with SWE described above has the following limitations.

- The ciphertext must be decrypted in a rather short time because the validators that can generate signatures for the decryption are fixed at the time of encryption.
- It is impossible to apply some functions to the encrypted message m without revealing it to the decryptor.

We solve them by introducing OTPs. The OTP is an encoded circuit that can be evaluated on at most one input. Goyal et al., [18] constructs a blockchain-based one-time program (BOTP) from witness encryption (WE) and garbled circuits. A generator of BOTP makes a garbled circuit of the circuit and encrypts its garbled inputs under WE. Its evaluator can decrypt each encryption of the garbled input for the bit $b \in \{0, 1\}$ of the i-th input bit by committing b as the i-th input bit on-chain. Subsequently, the decryptor evaluates the garbled circuit with the recovered garbled inputs. The decryptor can input only one bit b for each input bit to the circuit because the decryption condition of WE requires the decryptor to prove that b is committed first to the blockchain finalized by the honest majority of validators. In other words, the decryptor cannot input $1 - b$ without tampering with the finalized block containing the commitment of b.

While the OTP has the limitation of one-time input, it has a useful security feature that the evaluator cannot learn non-trivial information about the circuit. Therefore, the generator can embed secret data and algorithms in the circuit of the OTP. Moreover, if multiple generators use n-of-n MPC, which we call rabble MPC, to generate a private key unknown to humans, embed it in the circuit, and output its OTP, the OTP can hold a private key that no human knows as long as at least one MPC participant and the honest majority of the validators are honest. It can be used to decrypt the encryption of the circuit input and sign the circuit output inside the circuit. For example, the OTP with the embedded private key allows us to bootstrap a SWE ciphertext, i.e., encrypting the same message under a different set of verifying keys. The OTP for the SWE bootstrap decrypts the encrypted signature with the private key, uses the signature to recover the message from the SWE ciphertext, and encrypts the same message under new verifying keys. We can generalize this approach to evaluate arbitrary functions on encrypted inputs.

4.2 One-time Program with SWE

Instead of existing WE constructions supporting general decryption conditions, which are impractical or depend on heuristics cryptographic assumptions [1, 4, 12, 13, 15, 16, 17, 21, 22, 25, 27, 34], we adopt SWE to build OTPs. Let $k_{i,b}$ and $\widetilde{C}$ be a garbled input for the bit b of the i-th input bit and a garbled circuit of the input size $|x|$, respectively. The generator, the evaluator, and the Octopus contract managing $\widetilde{C}$ collaborate as below:

1. The generator registers $2|x|$ signing targets $\{(i,b)\}_{i\in[|x|],b\in\{0,1\}} = \{(1,0),(1,1),\ldots,(|x|,0),(|x|,1)\}$ with the Octopus contract.
2. The Octopus contract records $2|x|$ hashes $\{h_{i,b} = \mathsf{Hash}(\mathsf{PREFIX},(i,b))\}_{i\in[|x|],b\in\{0,1\}}$ and allocates n validators of which the verification keys are $V = (\mathsf{vk}_1,\ldots,\mathsf{vk}_n)$.
3. The generator generates a garbled circuit $\widetilde{C}$ and its garbled inputs $\{k_{i,b}\}_{i\in[|x|],b\in\{0,1\}}$.
4. For each $i \in [|x|], b \in \{0,1\}$, the generator encrypts $k_{i,b}$ under V and $h_{i,b}$, i.e., $ct_{i,b} \leftarrow \mathsf{SWE.Enc}(V, h_{i,b}, k_{i,b})$.
5. The evaluator registers the input x with the Octopus contract.
6. The Octopus contract checks if the other inputs have not been registered before. If so, it requests the allocated validators to sign the $|x|$ hashes $\{h_{i,x_i}\}_{i\in[|x|]}$ without specifying a public key to encrypt the signatures.
7. The evaluator obtains aggregated signatures $\{\sigma_i\}_{i\in[|x|]}$ and uses them to decrypt $\{ct_{i,x_i}\}_{i\in[|x|]}$, i.e., $k_{i,x_i} \leftarrow \mathsf{SWE.Dec}(ct_{i,x_i}, \sigma_i, U, V)$.
8. The evaluator evaluates $\widetilde{C}$ on $\{k_{i,x_i}\}_{i\in[|x|]}$.

Notably, in formal security proof, the garbled circuit is secure only against a selective adversary that chooses the input x before seeing the garbled circuit $\widetilde{C}$. However, as far as our knowledge, it does not mean that there is a practical attack on the garbled circuit scheme when x is chosen adaptively. Besides, Yao's garbled circuit [37] without modification is proven to be adaptively secure if the circuit is an NC1 circuit, i.e., a low-depth circuit [20]. To bootstrap it to a polynomial-sized circuit, we may be able to use a similar technique in [10] that bootstraps an indistinguishability obfuscation of NC1 circuits using a randomized encoding such as Yao's garbled circuit [2].

4.3 Rabble MPC for Key-Embedded OTPs

OTPs of key-embedded circuits are generated through the rabble MPC, n-of-n MPC among randomly selected people. The Octopus contract manages the participants of the rabble MPC and randomly assigns their subset to each generation of the OTP. These participants are different from the validators, and the Octopus contract can require a lower stake to participate in the rabble MPC than that of validators.

After registering the signing targets with the Octopus contract as described above, the selected n participants perform the n-of-n MPC to privately generate a new OTP for a circuit C taking s inputs as follows:

1. Each participant provides the randomness r_i as input.
2. They derive private and public keys $(\mathsf{privKey}, \mathsf{pubKey})$ from the XOR of all randomnesses $\bigoplus_{i=1}^{n} r_i$. These keys are assumed to be usable for both PKE and digital signature schemes.
3. They construct a key-embedded circuit $C[\mathsf{privKey}]$ that takes s encryptions of inputs $(ct_{x_1}, \ldots, ct_{x_s})$ under pubKey, decrypts them with $\mathsf{privKey}$, provides the s inputs $(x_1, \ldots, x_s)$ for C, signs the output $y = C(x_1, \ldots, x_s)$ with $\mathsf{privKey}$, and outputs y and the signature σ_{otp}. Let u be the input bits size of $C[\mathsf{privKey}]$.
4. They generate a garbled circuit of $C[\mathsf{privKey}]$ denoted by $\widetilde{C[\mathsf{privKey}]}$ and its garbled inputs $\{k_{i,b}\}_{i\in[u],b\in\{0,1\}}$.
5. They encrypt each garbled input $k_{i,b}$ under the allocated validators' verification keys V and the hash $h_{i,b} = \mathsf{Hash}(\mathsf{PREFIX}, (i, b))$, i.e., $ct_{i,b} \leftarrow \mathsf{SWE.Enc}(V, h_{i,b}, k_{i,b})$.
6. They outputs the OTP $(\mathsf{pubKey}, \widetilde{C[\mathsf{privKey}]}, \{ct_{i,b}\}_{i\in[u],b\in\{0,1\}})$.

In the following, we show how to define a circuit for each use case. Some encryptions of the circuit inputs can be specified by the Octopus contract.

4.4 SWE bootstrapping

To bootstrap a SWE ciphertext ct_V under a set V of verifying keys to a new ciphertext $ct_{V'}$ under a new set $V^{'}$, we employ an OTP of the following circuit:

1. Take as input the ciphertext ct_V, two sets of verifying keys U, V such that $|U| \geq t$ and $U \subseteq V$, a signature σ generated by the validators of U, the new set $V^{'}$, and a hash h.
2. Decrypt ct_V with σ, i.e., $m \leftarrow \mathsf{SWE.Dec}(ct_V, \sigma, U, V)$.
3. Encrypt m under $V^{'}$ and h, i.e., $ct_{V'} \leftarrow \mathsf{SWE.Enc}(V^{'}, h, m)$.
4. Output $ct_{V'}$.

A user willing to bootstrap ct_V calls the Octopus contract to decrypt ct_V along with h and a public key of the private key embedded in the OTP. When the validators of $U \subseteq V$ provide the contract with the encryption of the aggregated signature ct_σ under the public key, the contract registers ct_σ and encryptions of $ct_V, U, V, V^{'}, h$ under the public key as the inputs to the OTP. The evaluation of the OTP outputs $ct_{V'}$ and a signature for $ct_{V'}$ verifiable by the public key.

4.5 Applications

4.5.1 Verification of Computation by Private Conditions

One limitation of ZKP is that the verification conditions are public. In some applications, it enables a malicious prover to cheat the verifier. For example, to demonstrate a programming skill, the prover wants to prove the knowledge of a program that passes some test cases. However, if the test cases are public, the malicious prover can use a dummy program that passes only the specific test cases in the verification conditions. To prevent it, the verifier needs to obtain the test results without telling the prover the test cases. However, if the verifier receives the prover's program and runs the test cases by itself, the verifier consumes much computation costs, which also causes a DoS attack vector.

To solve this problem, we can use an OTP with a circuit that evaluates the provided program on the test cases as follows.

1. Take as input the test cases TEST and the program P.
2. Run a virtual machine (VM) with TEST and P.
3. If the VM causes no error, output $b = 1$. Otherwise, output $b = 0$.

When a public key in the OTP is pubKey, the prover and the verifier register the encryption of P and TEST under pubKey, respectively. Since the OTP returns a signature for the output of the circuit, the verifier can check the test results of P by verifying the signature with the expected message $b = 1$ and pubKey.

We can easily generalize the above solution for various kinds of verification. For instance, it is also possible to verify EVM bytecodes to prevent malicious contract updates or mitigate the risk of adversarial examples [32] against deep-learning-based authentications by hiding model parameters.

4.5.2 Private and Unique Human ID with Proof of Attribution

We consider how to build a human ID, an ID unique to each human, from some kinds of human attributions as follows.

1. The human ID is derived from some attributions unique to each human, e.g., an iris code.
2. Each human can obtain only one human ID for the same type of attributions at the same time by proving that the human is a correct holder of the attributions, e.g., having a photo of eyes that corresponds to the claimed iris code and simultaneously passes some detections of spoofing attacks [23, 24, 26].
3. **Even if one knows the attributions themselves, no one can derive another person's human ID from the attributions without proving the possession of them.**

Notably, the above scheme focuses on how to derive the human ID privately and treats the algorithm to prove the possession of the attributions as a black box.

Such IDs are useful for distinguishing between humans and bots, online voting where one person has one vote, and on-chain death games where once you die in the game, you cannot be resurrected by creating a new account.

If we assume a trusted issuer of the human ID, holding a private salt salt, it can issue the human ID for the attributions attr by computing $\mathsf{humanID} = \mathsf{Hash}(\mathsf{salt}, \mathsf{attr})$ if and only if a requesting user provides a valid ZKP proof of the possession of attr. Since the human ID depends on salt, no one can derive $\mathsf{humanID}$ from attr without presenting the valid proof to the issuer.

We can replace the trusted issuer with the Octopus contract by implementing it in a circuit of the OTP. Specifically, the circuit works as follows.

1. Take as input the private salt salt, the human's attributions attr, a proof π, the user's public key $\mathsf{pubKey}_{\mathsf{user}}$ and a public key $\mathsf{pubKey}'_{\mathsf{otp}}$ of the next OTP.
2. Encrypt salt under $\mathsf{pubKey}'_{\mathsf{otp}}$, i.e., $\mathsf{ct}_{\mathsf{salt}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pubKey}'_{\mathsf{otp}}, \mathsf{salt})$.
3. Verify π to check if the user is a correct holder of attr. If not, output $(\mathsf{ct}_{\mathsf{salt}}, \perp)$.
4. Compute $\mathsf{humanID} \leftarrow \mathsf{Hash}(\mathsf{salt}, \mathsf{attr})$.
5. Generate a signature σ for $(\mathsf{attr}, \mathsf{humanID})$ with the private key embedded into this OTP.
6. Encrypt $\mathsf{humanID}$ and σ under $\mathsf{pubKey}_{\mathsf{user}}$, i.e., $\mathsf{ct}_{\mathsf{user}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pubKey}_{\mathsf{user}}, (\mathsf{humanID}, \sigma))$.
7. Output $(\mathsf{ct}_{\mathsf{salt}}, \mathsf{ct}_{\mathsf{user}})$.

When the user provides the Octopus contract with encryptions of attr, π, and $\mathsf{pubKey}_{\mathsf{user}}$ under the public key of the current OTP, the contract first prepares the next OTP having the public key $\mathsf{pubKey}'_{\mathsf{otp}}$ to inherit the same salt. It then registers the provided encryptions, the encryption of $\mathsf{pubKey}'_{\mathsf{otp}}$, and the encryption of salt returned by the previous OTP as the inputs to the current OTP. The OTP returns $(\mathsf{ct}_{\mathsf{salt}}, \mathsf{ctuser})$, in which the first output $\mathsf{ct}_{\mathsf{salt}}$ will be used by the next OTP. The user decrypts $\mathsf{ct}_{\mathsf{user}}$ with the private key of $\mathsf{pubKey}_{\mathsf{user}}$ to obtain $\mathsf{humanID}$ and σ. It allows the user to prove the possession of $\mathsf{humanID}$ with ZKP.

You may think that ZKP and a sparse Merkle tree are enough to build a unique human ID while keeping anonymity as follows.

1. Each user first registers the user's attr and a hash of the randomness with the contract. If attr has been already registered, the contract returns an error.
2. The contract constructs a sparse Merkle tree in which each user's key and value are attr and the hash, respectively.

3. The user then proves that the hash is included in the tree at the key of attr, the user correctly holds attr, and the randomness of the registered hash derives an exposed nullifier.

4. If the proof passes the ZKP verification and the nullifier has not been used, the nullifier is registered as a valid $\mathsf{humanID}$.

The above scheme, however, cannot hide which attr is registered on-chain. If it allows the user to introduce randomness to hide attr at the registration, a malicious user can obtain multiple human IDs using the same attr and different randomnesses, which breaks the uniqueness property. Therefore, we can conclude that our human ID with OTPs is a novel construction establishing the privacy of attr at the registration and the uniqueness property.

4.5.3 MACI without a Centralized Operator

Buterin [8] proposes MACI to build on-chain applications requiring collusion resistance such as voting. It relies on a centralized operator holding a key pair $(\mathsf{privKey}_{\mathsf{op}}, \mathsf{pubKey}_{\mathsf{op}})$. Its user holding private and public keys $(\mathsf{privKey}_{\mathsf{user}}, \mathsf{pubKey}_{\mathsf{user}})$ signs a message for an action in the application, e.g., a vote in the voting, by $\mathsf{privKey}_{\mathsf{user}}$ and sends the operator an encryption of the message and the signature under $\mathsf{pubKey}_{\mathsf{op}}$). Besides, the user can revoke the current key and change it to a new one $\mathsf{pubKey}'_{\mathsf{user}}$ by signing $\mathsf{pubKey}'_{\mathsf{user}}$ and sends its encryption. Even if the user proves to the other users that the user sent an encrypted message for a specific action, e.g., a vote for a specific person, the other users cannot ensure that the used private key has not been revoked. Therefore, the MACI can reduce the incentive to force the other to take specific actions.

As Buterin [7] suggests using code obfuscation to remove the centralized operator from MACI, we can implement the functionality of the operator in the circuit of OTP. Specifically, the circuit takes as input the users' messages and their signatures, a public key of the next OTP, and the state containing the users' current public keys. It then verifies the signatures, updates some data based on the users' messages, and returns the outputs of the application such as voting results and the encryption of the new state under the next public key. Each user provides the Octopus contract with the encryption of the message and its signature, and the contract itself registers the encrypted next public key and the encrypted state returned by the previous OTP as the other inputs.

4.5.4 Private Computation with Web Data

TLS verification protocols, e.g., DECO [39], PECO [30], TLSNotary [28], allow smart contracts to verify data sent by existing web servers through TLS protocol. However, they depend on a trusted server, which we call Notary, interacting with a prover working as a web client to confidentially generate the client's responses in the TLS protocol through two-party computation. The Notary is necessary because the TLS protocol authenticates data with symmetric secrets derived from the server and client secrets. In more detail, the Notary prevents the prover

from directly learning the symmetric secrets and forging the authenticated data. The smart contracts accept the claimed data if and only if the Notary signs it.

We can replace the Notary with OTPs of circuits each of which takes as input the data from the server and outputs the client's data. Specifically, when the TLS protocol has k round communications, we assume the rabble MPC simultaneously generates k OTPs for circuits $(C_1, \ldots, C_k)$ with the same embedded private key. The circuit C_i corresponds to the i-th round of the communication, which takes the server's response and returns the client's response at that round. However, the first circuit C_1 takes the prover's public key, and the last circuit C_k returns an encryption of the claimed data from the server under the prover's public key. The client-side randomness is derived from the embedded private key.

The prover first calls the Octopus contract to register the prover's public key. At the i-th round, the prover registers with the contract the encryption of the server's latest response under the public key of the OTPs, obtains the client's response from the OTP of C_i, and sends it to the server. After getting the output from the last OTP, i.e., the encryption of the claimed data ct_{data} and the signature σ for ct_{data}, the prover finally generates a ZKP proof π to claim that σ is valid for ct_{data} and the public key of the registered OTPs and the decryption of ct_{data} results the claimed data data. The verifier, which can be another smart contract, verifies π with data and the public key of the registered OTPs.

In addition to verifying the web data, the Octopus contract can evaluate some functions on the authenticated web data without revealing them. For example, we can consider an on-chain scholarship for students with the following specifications.

- Each student submits a GPA on a university's website and an account balance on a bank's website to the Octopus contract for the scholarship.
- To provide the scholarship for the most diligent but poorest student, the contract privately computes $\mathsf{score} = \frac{\mathsf{gpa}}{\mathsf{balance}}$ and chooses the student of the largest score.
- The contract sends the chosen student the scholarship.

To achieve the above specifications, the Octopus contract prepares two kinds of OTPs: one is for the TLS verification of the GPA and the bank account balance, and the other is for computing each student's score and choosing the student of the largest score. Firstly, each student calls the Octopus contract to use the first OTP. At that time, the student passes an encryption of a public key of the second OTP as the input to the first OTP, which returns an encryption of $(\mathsf{gpa}, \mathsf{balance})$ under the public key of the second OTP and a signature σ for that output. The contract accepts the encryption if the student provides a valid signature. After most students submit the encryptions, the contract registers those encryptions as the input to the second OTP. It finally sends the scholarship to the student chosen by the second OTP.

5 Selecting a Subset of Validators

There are several methods to select a validator set from the consensus layer of Ethereum as follows: 1. Hard fork Ethereum to force all validators to sign messages from Octopus contracts. 2. Soft fork Ethereum, allowing any validators to sign messages from Octopus contracts. 3. Use a re-staking mechanism such as Eigen Layer [33], enabling validators to have dual roles.

Even in the first case, which imposes the greatest burden on the Ethereum network, the validators' signatures for the same messages can be aggregated, so that the additional cost of pairing is at most for each ciphertext. However, in that case, we should note that security is not completely inherited because the validators cannot be penalized in the same way as in the case of double voting when they sign messages not specified by the Octopus contracts.

In the second and third cases, we can maintain the existing protocol of the consensus layer as the modification to the node implementation for our scheme is optimal in similar to MEV-related protocols. While the restaking in the third case is easy to introduce, the soft fork supported by many validators will improve the security of our scheme more significantly.

References

[1] H. Abusalah, G. Fuchsbauer, and K. Pietrzak. Offline witness encryption. In *International Conference on Applied Cryptography and Network Security*, pages 285–303. Springer, 2016.

[2] B. Applebaum. Garbled circuits as randomized encodings of functions: a primer. In *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 1–44. Springer, 2017.

[3] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski. Garbled neural networks are practical. *Cryptology ePrint Archive*, 2019.

[4] J. Bartusek, Y. Ishai, A. Jain, F. Ma, A. Sahai, and M. Zhandry. Affine determinant programs: a framework for obfuscation and witness encryption. In *11th Innovations in Theoretical Computer Science Conference*, 2020.

[5] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796, 2012.

[6] W. Bitcoin. Wrapped bitcoin (wbtc) an erc20 token backed 1:1 with bitcoin. https://wbtc.network/. (Accessed on 12/14/2023).

[7] V. Buterin. Hard problems in cryptocurrency: Five years later. https://vitalik.eth.limo/general/2019/11/22/progress.htmlnumberfour, 2019. (Accessed on 12/08/2023).

[8] V. Buterin. Minimal anti-collusion infrastructure - applications - ethereum research. https://ethresear.ch/t/minimal-anti-collusion-infrastructure/5413, 2019. (Accessed on 12/07/2023).

[9] V. Buterin. A quick garbled circuits primer. https://vitalik.eth.limo/general/2020/03/21/garbled.html, 2020. (Accessed on 12/13/2023).

[10] R. Canetti, H. Lin, S. Tessaro, and V. Vaikuntanathan. Obfuscation of probabilistic circuits and applications. In *Theory of Cryptography: 12th Theory of Cryptography Conference, TCC 2015, Warsaw, Poland, March 23-25, 2015, Proceedings, Part II 12*, pages 468–497. Springer, 2015.

[11] A. Cerulli, A. Connolly, G. Neven, F.-S. Preiss, and V. Shoup. vetkeys: How a blockchain can keep many secrets. Cryptology ePrint Archive, Paper 2023/616, 2023. https://eprint.iacr.org/2023/616.

[12] P. Chvojka. *Time Reveals The Truth-More Efficient Constructions of Timed Cryptographic Primitives*. PhD thesis, Universität Wuppertal, Fakultät für Elektrotechnik, Informationstechnik und . . . , 2021.

[13] P. Chvojka, T. Jager, and S. A. Kakvi. Offline witness encryption with semi-adaptive security. In *International Conference on Applied Cryptography and Network Security*, pages 231–250. Springer, 2020.

[14] N. Döttling, L. Hanzlik, B. Magri, and S. Wohnig. Mcfly: verifiable encryption to the future made practical. *Cryptology ePrint Archive*, 2022.

[15] S. Garg, C. Gentry, A. Sahai, and B. Waters. Witness encryption and its applications. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 467–476, 2013.

[16] C. Gentry, A. Lewko, and B. Waters. Witness encryption from instance independent assumptions. In *Annual Cryptology Conference*, pages 426–443. Springer, 2014.

[17] S. Goldwasser, Y. T. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. How to run turing machines on encrypted data. In *Annual Cryptology Conference*, pages 536–553. Springer, 2013.

[18] R. Goyal and V. Goyal. Overcoming cryptographic impossibility results using blockchains. In *Theory of Cryptography Conference*, pages 529–561. Springer, 2017.

[19] L. Hioki. Trustless bitcoin bridge creation with witness encryption - cryptography - ethereum research. https://ethresear.ch/t/trustless-bitcoin-bridge-creation-with-witness-encryption/11953, February 2022. (Accessed on 09/19/2022).

[20] Z. Jafargholi and D. Wichs. Adaptive security of yao's garbled circuits. In *Theory of Cryptography Conference*, pages 433–458. Springer, 2016.

[21] A. Jain, H. Lin, and A. Sahai. Indistinguishability obfuscation from lpn over f_p, dlin, and prgs in nc^ 0. *Cryptology ePrint Archive*, 2021.

[22] A. Jain, H. Lin, and A. Sahai. Indistinguishability obfuscation from well-founded assumptions. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 60–73, 2021.

[23] S. Khade, S. Ahirrao, S. Phansalkar, K. Kotecha, S. Gite, and S. D. Thepade. Iris liveness detection for biometric authentication: A systematic literature review and future directions. *Inventions*, 6(4):65, 2021.

[24] J. S. Kim, Y. W. Lee, J. S. Hong, S. G. Kim, G. Batchuluun, and K. R. Park. Lrfid-net: A local-region-based fake-iris detection network for fake iris images synthesized by a generative adversarial network. *Mathematics*, 11(19):4160, 2023.

[25] K. Kluczniak. Witness encryption from garbled circuit and multikey fully homomorphic encryption techniques. *IACR Cryptol. ePrint Arch.*, 2020:1502, 2020.

[26] N. Kohli, D. Yadav, M. Vatsa, R. Singh, and A. Noore. Detecting medley of iris spoofing attacks using desist. In *2016 IEEE 8th International Conference on Biometrics Theory, Applications and Systems (BTAS)*, pages 1–6. IEEE, 2016.

[27] D. Pan, B. Liang, H. Li, and P. Ni. Witness encryption with (weak) unique decryption and message indistinguishability: constructions and applications. In *Australasian Conference on Information Security and Privacy*, pages 609–619. Springer, 2019.

[28] Privacy and S. Exploration. Tlsnotary, proof of data authenticity. https://tlsnotary.org/. (Accessed on 12/08/2023).

[29] L. Protocol. Lit protocol. https://www.litprotocol.com/. (Accessed on 12/13/2023).

[30] M. B. Santos. Peco: methods to enhance the privacy of deco protocol. *Cryptology ePrint Archive*, 2022.

[31] R. Solomon, R. Weber, and G. Almashaqbeh. smartfhe: Privacy-preserving smart contracts from fully homomorphic encryption. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 309–331. IEEE, 2023.

[32] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[33] E. L. Team. Eigenlayer: The restaking collective. https://docs.eigenlayer.xyz/overview/readme/whitepaper. ().

[34] V. Vaikuntanathan, H. Wee, and D. Wichs. Witness encryption and null-io from evasive lwe. *IACR Cryptol. ePrint Arch.*, 2022:1140, 2022.

[35] D. C. G. Valadares, A. Perkusich, A. F. Martins, M. B. Kamel, and C. Seline. Privacy-preserving blockchain technologies. *Sensors*, 23(16):7172, 2023.

[36] B. Whitehat. Why you can't build a private uniswap with zkps. https://ethresear.ch/t/why-you-cant-build-a-private-uniswap-with-zkps/7754. ().

[37] A. C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

[38] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.

[39] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels. Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1919–1938, 2020.

[40] Y. Zhu, X. Song, S. Yang, Y. Qin, and Q. Zhou. Secure smart contract system built on smpc over blockchain. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1539–1544. IEEE, 2018.