HOT PROTOCOL

PETER VOLNOV, GEORGII KUKSA and ANDREY ZHEVLAKOV
HOT Labs

Abstract. HOT Protocol provides the infrastructure that allows smart contracts on EVM-compatible networks and Stellar Blockchain to securely own and manage private keys. The Multi-Party Computation (MPC) Network manages signing keys. By running an MPC node inside the Trusted Execution Environment (TEE), the protocol achieves stronger security guarantees while lowering economic requirements for participants. The NEAR Protocol provides a decentralized and efficient state layer. Any smart contract can start managing a dedicated private key by implementing a read-only method for signature authorization.

Contents

1. Introduction

Smart contracts are restricted to modifying and verifying state within their native blockchain, resulting in a fragmented Web3 ecosystem. Enabling smart contracts to cryptographically sign messages under their own identity would allow autonomous control over actions extending beyond the boundaries of a single blockchain domain.

This design introduces two challenges:

(1) Message signing must be externally triggered, since smart contracts act only upon incoming transactions

(2) A smart contract cannot securely store private keys, as its state is publicly accessible

To solve the first issue, a pull-based approach is used: any participant may request a signature from the key linked to a contract, but only when the contract explicitly authorizes signing a given message.

The second challenge can be solved with a Multi-Party Computation (MPC) network. It securely creates and uses a secret key for signing messages, and any operation with that key requires approval from a threshold number of participants.

Further reading describes implementation of the approaches and their interconnection

2. System Design

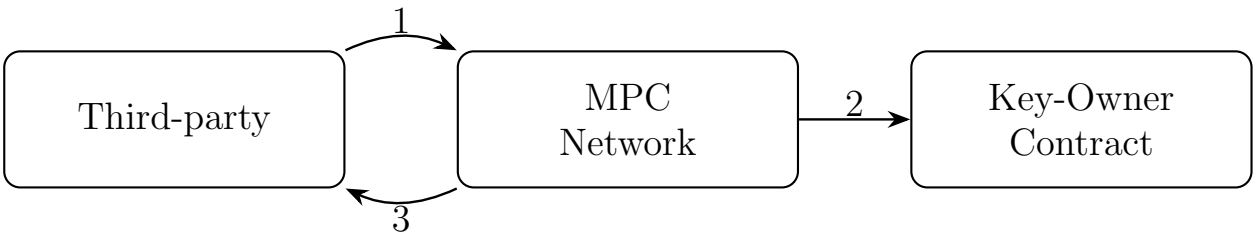

FIGURE 1. High-level flow for obtaining a signature from a smart contract.

(1) A third party triggers the pull mechanism, attempting to obtain a signature for a message from a private key linked to a specific smart contract. A smart contract may be linked to the multiple private keys.
(2) The MPC network performs a read-only verification to ensure that the smart contract authorizes the signing of a particular message.
(3) The MPC network securely signs the message and sends the result back to the third party.

2.1. Key-Owner Contract

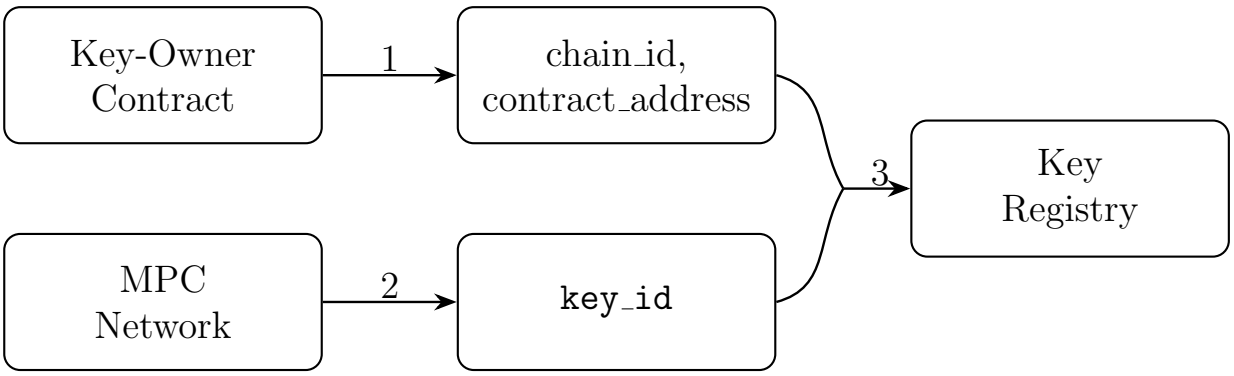

FIGURE 2. Key-Management Setup

(1) Key management requires the contract to define the following read-only method:

Listing 1. Authorization function signature

```
fn hot_verify(
       message: String,
       key_id: bytes32,
       metadata: bytes,
) -> bool;
```

Input arguments:

- `message` — the message being signed
- `key_id` — an identifier used to distinguish between multiple keys owned by the contract. One contract can hold multiple keys, removing the need to redeploy logic per user.
- `metadata` — auxiliary data used in the authorization process. For example, if `message` is a transaction hash, the transaction preimage is passed as `metadata` for invariant verification.

Return value: A boolean, indicating whether the message is authorized for signing.

Each contract is uniquely defined by the pair `(chain_id, contract_address)`.

(2) A key is reserved via the MPC Network (2.2.5), returning a `key_id`.
(3) The Key Registry contract stores the mapping between `key_id` and its authorization contract.

After setup, the MPC Network verifies a `key_id` by:

(1) fetching its authorization contract from the registry;
(2) calling `hot_verify` with the target message.

The Key-Owner contract should expose `key_id` to prove exclusive MPC control.

Practical example can be found in Appendix A.

2.2. MPC Network

The function of the MPC network is to manage a private key used for message signing.

2.2.1. *Design Principles*

The MPC network is designed around three main principles:

- Decentralization – no single entity controls the private key or computation; all nodes collaborate in a trust-minimized manner.
- Security – resistant to adversarial behavior, preserving computational correctness and data confidentiality.
- Scalability – designed to operate efficiently as the number of participating nodes or supported chains increases.

This leads to the idea that the private key used for message signing is divided into cryptographic shares and distributed among the participants of the MPC network.

2.2.2. *Security Guarantees*

The properties of this design are as follows:

- Threshold cooperation: a minimum number of nodes must collaborate to successfully generate a valid signature.
- Key secrecy: no single node ever possesses the complete private key; each holds only a share. An adversary would need to compromise at least the threshold number of nodes to reconstruct or misuse the key.
- Signature indistinguishability: MPC-generated signatures are identical to those from the master private key, unlike traditional multisignature outputs.

2.2.3. *Interface*

The MPC network exposes the following API:

- Distributed Key Generation (DKG): securely generates a private key and distributes its shares across participating nodes. Implementation follows the protocol by Gennaro et al. "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems"(2007)[1]
- Key Resharing – enables the addition or removal of participants, or modification of the threshold value. The key remains the same, but all shares are refreshed to reflect the new configuration. The implemented protocol is a specific instance of the general Distributed Key Generation (DKG) scheme.
- Message Signing — cooperatively generates a valid signature when a threshold of nodes participates, without ever reconstructing the private key. The implementation depends on the chosen signature scheme. In this work, we focus on the two most widely adopted ones:
  - ECDSA, used in Bitcoin, EVM-compatible chains, and Tron. The implementation the protocol by Damgård et al. *Fast Threshold ECDSA with Honest Majority*(2020)[2]
  - EdDSA, used in Solana, NEAR, Stellar, and TON. The implementation follows the protocol by Komlo and Goldberg "FROST: Flexible Round-Optimized Schnorr Threshold Signatures"(2021)[3]

2.2.4. *Diagram: Initialization flow*

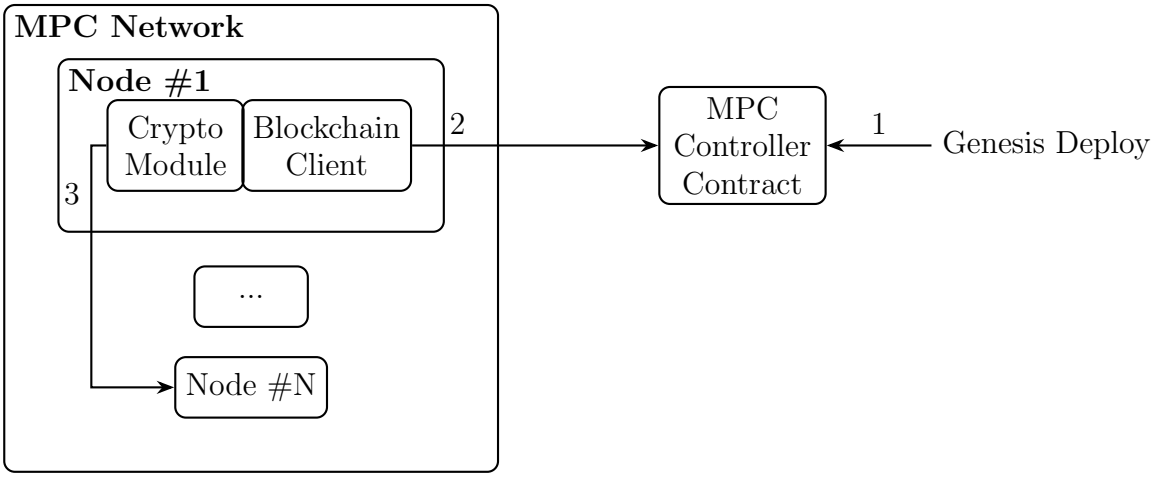

FIGURE 3. MPC Initialization flow

(1) The MPC Controller Contract is deployed in the genesis block to serve two main purposes:
   - State management: stores operational data of the MPC network, including participant details like IP addresses, encryption public keys, and identifiers
   - Network coordination: handles configuration changes through on-chain voting and triggers key resharing when thresholds or participant sets are updated.

   The contract only needs to be cost-efficient for state storage, making NEAR Protocol an appropriate choice. The same applies to the Key Registry contract (3).
(2) Nodes get their data from the Controller Contract. To keep the diagram simple, only one arrow is shown — but in practice, every node fetches the data
(3) Nodes run the DKG process to generate and receive their private key shares. For brevity reasons a single arrow is shown. In practice there is an arrow for each pair of participants.

2.2.5. *Key Derivation Function*

To ensure scalability, the network avoids repeating DKG and Key Resharing for every new configuration. Instead, a single root key is established through DKG, and all subsequent operational keys are obtained through deterministic derivation.

This derivation model follows a mechanism similar to idea by Wuille *BIP32: Hierarchical Deterministic Wallets*(2012)[4], where a child public key can be derived from a parent public key using a known tweak, without access to the private shares.

2.2.6. *Message Signing Authorization*

The MPC Network works with key-owner contracts, which decide whether a particular key is allowed to be used for signing.
Before running the signing protocol, each node:

a) Retrieves the contract address associated with the specified key ID from the Key Registry.
b) Executes the authorization methods of the key-owner contract.

The process for (a) is described in 2.1.
For (b), each MPC node must operate a light node for every supported network, or a full node where a light node implementation is not available.

2.2.7. *Trusted Execution Environment*

A Trusted Execution Environment (TEE) is a hardware-enforced isolated execution context that offers the following guarantees:

- Confidentiality: The TEE guarantees that both code and data executed within an enclave remain isolated and unreadable to any external software layer, preserving the secrecy of sensitive computations
- Integrity: The TEE ensures that enclave code executes as deployed and that neither its logic nor runtime state can be altered by external software.
- Attestation: The TEE can produce cryptographic proof of its software identity and state, allowing remote parties to verify that it runs trusted code.

The MPC node is executed within a trusted environment, providing isolation for high-risk operations:

- Computations with private key shares
- Verifying contract authorization method

Utilizing the integrity property, node logs can be treated as a source of truth, providing a reliable basis for the governance and economic model, as will be described in section 3.
To maintain correctness guarantees, each MPC node periodically provides an attestation to the Controller Contract, which performs verification of the reported state.
In practical terms, following technologies is being used to provide hardware-assisted isolation at the VM layer:

- TDX on Intel platforms (Cheng et al. *Intel TDX Demystified: A Top-Down Approach*(2023)[5])
- SEV on AMD platforms (Advanced Micro Devices inc. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*(2020)[6])

2.2.8. *Access Control Layer: Gatekeeper Network*

To target threat of overload and malicious use of the network, access to the MPC Network is restricted to the Gatekeeper Network.

Gatekeepers serve two primary functions:

- Perform load balancing and filtering of incoming requests to guarantee correctness and validity
- Ensures clear separation between the economic domains of MPC nodes and end users. Gatekeepers never handle or store private key material. Developers may customize and self-host Gatekeepers for their specific workflows, even without TEE support. This separation, along with the incentive model for this role, is described in section 3.2.2.

Offloading these two tasks to the Gatekeeper reduces the attack surface of the MPC nodes.

2.3. Full Signature generation flow

The following diagram illustrates the end-to-end signature generation flow for a key-owner contract.

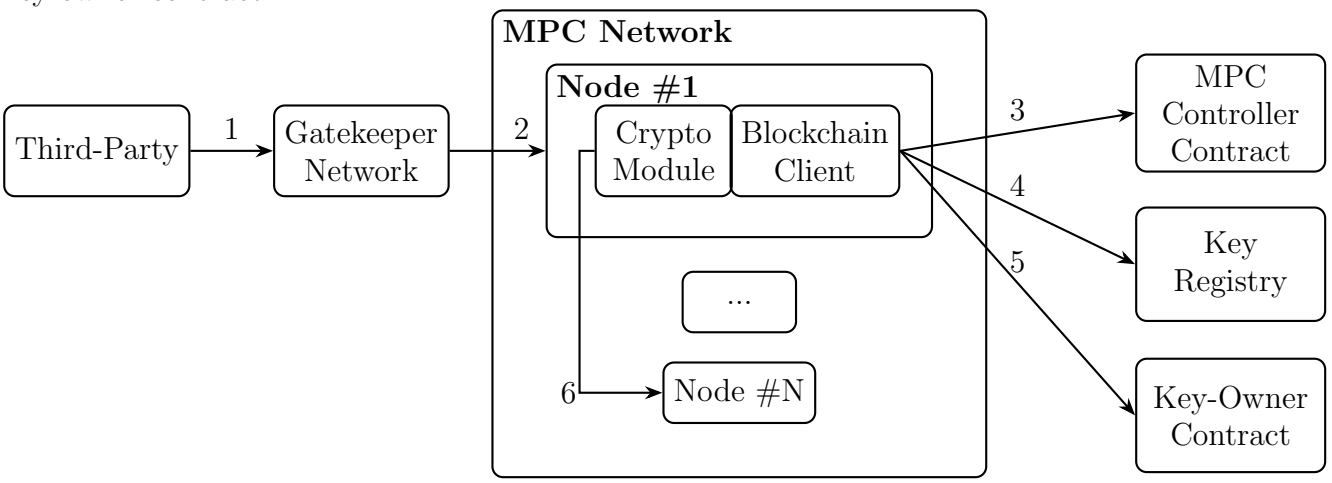

FIGURE 4. Signature generation flow

(1) A third party triggers the pull-based signing flow via the Gatekeeper Network, requesting a signature for a message tied to the public key of a given smart contract and submitting the necessary inputs to its authorization method(s).
(2) The Gatekeeper manages interactions with the third party and chooses which threshold nodes will take part in the signing.
(3) MPC nodes retrieve protocol-specific details from one another, such as IP addresses. In reality, this happens only once during initialization, with updates performed on each MPC configuration change. For simplicity, the diagram here and in the following steps shows a single arrow.
(4) MPC nodes obtain the authorization methods corresponding to the public key involved in the request.
(5) MPC nodes call the read-only authorization method to check whether the message is allowed to be signed.
(6) After successful authorization, the MPC nodes collaboratively generate the cryptographic signature.

3. Governance Model

The protocol combines cryptographic and hardware security with economic and governance mechanisms to guarantee accountability, liveness, and deterministic network costs. At the core of the architecture is the `gTOKEN`, a native asset that underpins staking, incentive alignment, and governance execution across all protocol layers.

3.1. Governance token

`gTOKEN` is a fungible token on NEAR Protocol (Kuzyakov and Zaremba *Fungible Token Standard*(2022)[7]).
Used to:

- rent MPC capacity via Gatekeepers
- lock collateral for nodes and Gatekeepers
- distribute rewards for honest behavior. Each NEAR Protocol epoch mints new `gTOKEN`s, distributed according to participants' contributions.

3.2. Roles

The system architecture defines five principal actor classes:

(1) MPC Nodes
(2) Decentralized Autonomous Organization (DAO)
(3) Gatekeepers
(4) Fishermen
(5) Smart Contract Users

Each component interacts through verifiable and economically secured interfaces, forming a cohesive framework for distributed key management, transaction validation, and protocol governance

3.2.1. *DAO*

The DAO safeguards protocol correctness by:

- manage core protocol parameters;
- vetoing destructive or risky proposals.

DAO members are independent organizations with a material `gTOKEN` stake. Joining requires member approval and a signed pledge to the DAO manifesto. Members of DAO must:

- Veto any decision that endangers security or stability of the MPC network
- Approve new DAO members that meet the requirements
- Approve new MPC network parameters change
- Approve new Gatekeepers
- Process reports against Gatekeepers and MPC nodes; verify TEE-signed logs; execute slashing when violations are proven

DAO Also controls:

- Inflation rate
- Rewards for MPC Nodes
- Minimal stake for Gatekeeper and MPC Nodes
- Stake size per compute unit to access the network

Members of DAO must:

- Veto any decision that endangers security or stability of the MPC network
- Approve new DAO members that meet the requirements
- Approve new MPC network parameters change
- Approve new Gatekeepers
- Process reports against Gatekeepers and MPC nodes; verify TEE-signed logs; execute slashing when violations are proven

3.2.2. *Gatekeeper*

Gatekeepers lease a portion of the protocol's capacity, defined by rate limits, and then allocate or resell that capacity to their users. They are free to determine their own business model: charging usage fees, accepting stablecoins, or provide it for free for some users depending on the other factors.
Each Gatekeeper co-signs a user's request `(gatekeeper_id, request, deadline)` and relays it to the MPC Network. These signed receipts are publicly available in MPC logs and can serve as evidence for slashing a Gatekeeper's stake in cases of malicious behavior or rate-limit violations.

3.2.3. *Fishermen*

Any third party may monitor public TEE-signed logs and open disputes in the controller contract.
Dispute resolution relies on TEE-signed MPC node logs, which include:

- timestamps;
- signing requests submitted by Gatekeepers;
- validation outcomes;
- protocol round statuses;
- node unavailability reports;
- protocol-level errors.

Submitting a dispute incurs a `gTOKEN` fee to prevent spam. Upon successful slashing, the Fisherman is rewarded with a share of the penalized stake.

3.2.4. *MPC Node provider*

Core hardware provider. Supports the continuous and stable operation of its MPC Node and gets rewarded for this from inflation.

- Stake `gTOKEN` to join the network
- earn `gTOKEN` after each epoch
- `gTOKEN` can be slashed for unstable operation

Role	Stake gTOKEN	Earn gTOKEN	Contribution
DAO Member	YES	YES	Governance
MPC Node	YES	YES	Hardware
Gatekeeper	YES	NO	Distribution
User	NO	NO	Smart contract user
Fishermen	NO	YES	Monitoring

Table 1

4. Security Model

This section outlines the scope of attacks. The analysis is limited to adversaries targeting:

(1) Availability of the system, and
(2) Confidentiality of private keys.

4.1. Denial of Service

4.1.1. *Uncooperative node*

Threat: a node stalls or refuses rounds to cause signing delays or failures.
Mitigations:

- governance: vote to exclude the node and run key resharing; slashing on proven misbehavior
- operational: Gatekeeper selects any threshold of responsive nodes; can blacklist slow nodes; group-testing (GBS-style) selection
- liveness: health checks maintain an active set; rolling updates avoid synchronized downtime

4.1.2. *Number of active nodes* $\leq$ `threshold` $- 1$

Mitigation: Increase n and diversify operators and hosting across independent clusters.

4.1.3. *Gatekeeper quota abuse*

Threat: Gatekeeper quota abuse against the MPC network — a malicious or compromised Gatekeeper floods nodes at maximum RPS, reducing throughput and availability Enforce per-node request rate limits and auto-eject policies; allow MPC nodes to collectively vote to remove the offending Gatekeeper, applying slashing penalties verified via TEE-signed logs.

4.1.4. *Gatekeeper downtime*

If a Gatekeeper fails, users switch to another Gatekeeper; no unique access is held by any single Gatekeeper.

4.2. Confidentiality

4.2.1. *Mimicry of TEE*

Threat: a node pretends to run attested code. Mitigation: time-bound remote attestation, periodic re-attestation, code-identity checks pinned in Coordinator; nodes failing re-attestation are disabled before protocol rounds.

4.2.2. *Key leakage via RAM/Persistence*

Threat: a coalition reconstructs user keys by reading shares from memory or disk. Preconditions:

- coalition size $\geq$ `threshold`
- bypass TEE memory and persistence protections

Mitigation: TEE enclaves (memory isolation), periodic re-attestation

4.2.3. *Controller Contract hijacking*

Threat: Attacker gains control over the Controller Contract, disables TEE attestation verification, triggers key resharing to an attacker-controlled participant set, and reconstructs the MPC key. Mitigations: The contract operates without access keys.

4.2.4. *Key Registry hijacking*

Threat: Attacker modifies authorization method mappings to redirect signing authority to malicious contracts. Mitigations: The contract operates without access keys. Any attempt to add or remove an authorization method must be accompanied by authorization proofs from the existing methods.

5. Conclusion

HOT Protocol enables smart contracts to securely control their own cryptographic keys using threshold Multi-Party Computation (MPC) within Trusted Execution Environments (TEE) and coordinated on-chain governance. It allows contracts to authorize and produce signatures for external actions, forming the basis for Chain-Abstracted Applications operating assets across multiple networks.
Possible use-cases include:

- Bitcoin-style multi-signature wallets without new on-chain deployments for each policy
- Two-factor (2FA) authorization
- Seed phrase rotation (if we use seed phrase to authorize for specific MPC key and this authorization seed phrase can be rotated)
- Wallet recovery flows
- Passkey-based wallets

- Other high-assurance authorization policies that are costly to implement with traditional smart wallets
- Chain Abstracted dApps, managing assets across multiple chains

References

[1] Rosario Gennaro et al. "Secure Distributed Key Generation for Discrete-Log Based Cryptosystems". In: *Journal of Cryptology* 20.1 (2007), pp. 51–83. DOI: 10.1007/s00145-006-0347-3. URL: https://doi.org/10.1007/s00145-006-0347-3.

[2] Ivan Damgård et al. *Fast Threshold ECDSA with Honest Majority*. Cryptology ePrint Archive, Paper 2020/501. 2020. URL: https://eprint.iacr.org/2020/501.

[3] Chelsea Komlo and Ian Goldberg. "FROST: Flexible Round-Optimized Schnorr Threshold Signatures". In: (2021). Ed. by Orr Dunkelman, Michael J. Jacobson Jr., and Colin O'Flynn, pp. 34–65. URL: https://eprint.iacr.org/2020/852.pdf.

[4] Pieter Wuille. *BIP32: Hierarchical Deterministic Wallets*. 2012. URL: https://github.com/bitcoin/bips/blob/651e273ee8512dd40af8a9e695acd97018e02e1c/bip-0032.mediawiki.

[5] Pau-Chen Cheng et al. *Intel TDX Demystified: A Top-Down Approach*. 2023. arXiv: 2303.15540 [cs.CR]. URL: https://arxiv.org/abs/2303.15540.

[6] Advanced Micro Devices inc. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. 2020. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[7] Evgeny Kuzyakov and Robert Zaremba. *Fungible Token Standard*. 2022. URL: https://github.com/near/NEPs/blob/9dc048d9a1da42f830bd44c4d5af3bb792fafb3d/neps/nep-0141.md.

Appendix A. Passkey Bitcoin Wallet

A passkey is a passwordless authentication method based on public-key cryptography, where a private key stays securely on your device and a public key is registered with the service. It lets you sign in by verifying a cryptographic signature instead of entering a password, making logins both simpler and more secure.

To demonstrate the use of HOT Protocol, we present an implemented application that enables passkey-based Bitcoin wallets — that is, wallets where transaction signing is controlled solely by the user's device passkey, itself secured by fingerprint or facial recognition.

Deployed on NEAR Protocol, `passkey.auth.hot.tg` maps each `key_id` (Bitcoin wallet) to its passkey `public_key`. It serves as the key-owner contract with the following `hot_verify` method:

Listing 2. Authorization function signature

```
fn hot_verify(
        message: String,
        key_id: bytes32,
        _metadata: bytes,
) -> bool {
        ecdsa_secp256r1.verify(message, passkey[key_id])
}
```

The initialization flow for a user is as follows:

(1) Obtain the passkey `public_key` from their device;
(2) Reserve a `key_id` via the MPC Network;
(3) Bind the `key_id` to the `public_key` in the registry contract.