Received 20 October 2024, accepted 4 November 2024, date of publication 13 November 2024, date of current version 25 November 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3497302

APPLIED RESEARCH

A Practical Recovery Mechanism for Blockchain Hardware Wallets

VARUN DESHPANDE, HARISH J, AND ATHARVA VIJAY KHADE
Samsung Research, Bengaluru 560048, India
Corresponding author: Varun Deshpande (varun.d@samsung.com)

ABSTRACT Blockchain hardware wallets, through their security-by-design architecture, offer higher security assurances. They fundamentally differ from software wallets due to an important security property called Unicity. Unicity ensures that ownership is tied to a unique hardware entity, both physically and logically. This property is highly desirable if cryptocurrency assets under ownership are high in value. However, when such a hardware wallet is backed up, this unicity property is lost as the root seed or private key is cloned. The resulting security ramifications are numerous, ultimately leading to theft of funds in many cases. In this work, we introduce a practical recovery mechanism for hardware wallets that does not involve extraction or cloning of the private key or root seed for backup, thus preserving this unicity property. The proposed recovery mechanism ensures that the owner can access their cryptocurrency funds in case of malfunction/theft of the hardware wallet, even when it is not backed up. The novel mechanism is based on Symmetric Secret Sharing, a Key Revocation Certificate, a Smart Contract-based Registry, and Smart Accounts and can be practically implemented. We compare our mechanism with other solutions and show how it performs better on all security parameters. The paper solves the important problem of secure backup of hardware wallets without compromising the design paradigms associated with it.

INDEX TERMS Blockchain, hardware wallet, smart account, smart contract, cryptography, hash function, recovery, cryptocurrency.

I. INTRODUCTION

Blockchain wallets are becoming increasingly popular, as many people are adopting them to access their digital assets stored on the blockchain [1]. These wallets come in various forms, primarily categorized into hardware wallets and software wallets [2], [3]. Hardware wallets are physical devices specifically designed for the safe generation and storage of private keys offline. They come in various form factors, such as a USB stick or bank card, e.g., Ledger Nano S Plus [4], Trezor Model T [5], etc. Software wallets, on the other hand, are digital applications designed to manage private keys securely on computers, mobile devices, or servers, e.g., Metamask [6], Electrum [7], etc.

Hardware wallets store private keys offline (they are not connected to the internet), reducing the risk of hacking attempts compared to software wallets, which are constantly exposed to online threats and vulnerabilities [8]. Some instances of software wallets getting compromised include the downfall of FTX, one of the largest cryptocurrency exchanges, involving the illegal use of users' assets [9], [10]. Another attack on the popular Electrum wallet resulted in the loss of over 1,400 bitcoins [11]. These examples raise concerns about software wallets security, making hardware wallets the preferred choice for cryptocurrency users seeking reliability.

While hardware wallets offer security benefits, they require responsible handling from users as the loss or theft of the wallet can lead to the loss of digital assets. To address this concern, reliable backup and recovery systems are crucial, along with strict access controls in case of lost or theft. Following this, there have been various serious attempts to develop secure backup and recovery systems for hardware wallets; however, to the best of our knowledge, these attempts are not sufficient/compatible with existing **security-by-design** paradigm constraints. These constraints forms the

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

basis of inherent security assurances of hardware wallets. For example, the *seed unicity* property[1] is violated once a backup is created. Furthermore, the unique seed/private key is also exported (whether logically or programmatically) during this backup process. Such approaches are strongly discouraged for any device based on hardware *root-of-trust*. In fact, in other domains, these constraints are indeed followed. For example, in Trusted Platform Module (TPM) [12].

The hard requirement, namely that the private key should never leave the Hardware Trusted Environment in any form, is the key to ensuring hardware *root-of-trust*. This is applicable to both web3 and web2. Based on this principle, we propose a novel and practical mechanism that operates in a decentralized setting, enabling complete hardware wallet recovery without the need for a backup, all while preserving the *unicity* property, which the current solutions completely miss.

The mechanism draws inspiration from traditional Public Key Infrastructure (PKI) [13] and makes use of Revocation Certificates to effectively prevent potential attackers from accessing accounts associated with the stolen or compromised hardware wallet. This is accomplished when the affected user submits their revocation certificate to a blockchain-based smart contract registry. By doing so, the legitimate user can transfer ownership of their funds to a new account utilizing smart account[2] functionalities. The challenge of verifying the identity of the affected user without their original key is overcome through a carefully designed algorithm based on symmetric *secret* sharing. The *secret* is obfuscated through various stages using a *one-way trapdoor function* (hash) to ensure compatibility in a decentralized trust-less environment. Combining the techniques of i) *secret* sharing with registry, and ii) Smart Accounts, enables the complete recovery of funds and on-chain access without the need for any third-party involvement.

The main contributions of this paper are:

- Proposition of a novel recovery mechanism for blockchain hardware wallets that preserves secure-by-design paradigms like the unicity property.
- Comparison with the current available solutions, highlighting its uniqueness, clear advantages, and non-dependence/non-requirement of additional components or services.
- A reference implementation of the proposed mechanism demonstrating its practical feasibility along with its evaluation for security, performance, and overhead.
- A comprehensive security analysis of the proposed recovery mechanism, highlighting its robustness against various attacks.

The paper is outlined as follows: Section II discusses works related to various backup and recovery mechanisms. Section III elaborates on various technical components used in our mechanism. Section IV explains the problem statement in detail, while our proposed solution and designed algorithms are illustrated in Section V. The rationale of timing windows crucial for the protocol is discussed in Section VI. The practicality of our approach is demonstrated in Section VII, while Section VIII comments on various security aspects and attacks. Finally, we conclude our work in Section IX.

II. RELATED WORKS

Most hardware wallets use a Mnemonic Phrase (also called a mnemonic) to encode the root seed.[3] These mnemonics are typically 12-24 words long [14] and are used to recover the wallet if the device is lost or damaged. However, if the mnemonic is lost or forgotten, the user may lose access to their funds forever.

While it is recommended to record this mnemonic either on a recovery sheet provided by the wallet manufacturer or store it digitally, it remains critical to keep the mnemonic backup safe, as anyone who gains access to it can recover all the funds.

To address this inconvenience, several solutions have been proposed. A simple solution is to use another word, also known as a Mnemonic Passphrase (or simply passphrase), to encrypt the existing mnemonic [14], [15]. The passphrase acts like two-factor authentication (2FA) in wallets, where it is the final security check required for granting access. However, it suffers from obvious flaws, such as:

- A passphrase, in general, is just a single word, making brute-force attacks still possible if the mnemonic is very weak.
- There is a risk of forgetting the passphrase or leaking both the mnemonic and passphrase.

A good option involves creating a backup of the private key on another mirrored hardware wallet [16]. However, it has a few problems:

- Buying a new hardware wallet solely for the purpose of backup can be costly.
- Regular firmware updates are needed on both devices to keep up with security patches, thus doubling maintenance efforts.
- If the user loses the backup hardware wallet, it results in permanent loss of access to funds.
- The user is now responsible for the safekeeping of two devices. The loss/theft of the backup device is just as serious as in the case of the main device.

Another approach is to use a heavy metal locker which is relatively inexpensive, easy to set up and durable [17], [18], [19]. It can withstand extreme conditions, such as fire, water, and physical damage, ensuring the long-term preservation of your private keys. However, even this approach has similar issues:

- The user can still lose or misplace the locker.

[1]The guarantee that it is unique, and no other device has the same seed.
[2]Smart contract based account.

[3]Seed used to generate private key(s) in Hierarchical Deterministic (HD) wallet.

- A chosen *secret* (passphrase/pin) is needed to secure the locker. The user has to remember this *secret* to unlock the metal device.
- The user still has to ensure the security of the metal device from theft or extreme cases of damage.

An alternative approach involves utilizing a recovery service offered by the wallet provider, where the private key is encrypted and split into multiple fragments, which are then stored and protected separately [20]. This method requires associating the user's identity, such as biometrics, with the fragments, allowing reconstruction of the private key using a threshold of shares (e.g., 2-out-of-3) upon successful authentication by the service [21]. It has the following disadvantages:

- The private key leaves the hardware wallet, which compromises the wallet's core principle of keeping sensitive information within the device.
- The user's personal identity is linked to the private key shares, violating the anonymity principle of blockchain.
- Trust in third-party providers to store the key shares is necessary, introducing potential risks associated with relying on external entities

A further solution is social recovery, where users select a group of trusted individuals known as ''guardians'' who can authorize a recovery request in case the user's device is lost or stolen [22], [23]. It has its drawbacks too:

- Collusion among guardians is possible which either discards valid recovery requests or authorize invalid/illegal ones.
- The wait time to recover a wallet is longer since guardian approvals are required.
- It is susceptible to denial-of-service (DoS) attacks, where an attacker floods the system with numerous recovery requests.

Many other solutions exist [24] that follow an approach similar to one of the aforementioned schemes and therefore suffer from similar shortcomings. Further, apart from the social recovery mechanism, none of them follow seed unicity property. In our proposed scheme, we aim to solve most of these shortcomings effectively while maintaining the required level of security and strict adherence to the unicity property. In our proposed scheme, the user does not need to remember the mnemonics or passphrases as there is no concept of *backup*. Additionally, our scheme does not require users to depend on third-party providers for storing secrets, similar to Shamir Secret Sharing (SSS) which downgrades the usability of wallets. Furthermore, there is no dependence on any social entity for recovering the wallet.

III. TECHNICAL BACKGROUND

In this section, we provide a short technical summary of related general concepts and discuss the technical building blocks of our proposed solution, which will be elaborated in Section V. For ease of readability, Table 1 illustrates all the abbreviations used in this paper.

TABLE 1. Abbreviations.

Term	Definition
CA	Certificate Authority
CRL	Certificate Revocation List
DoS	Denial of Service
ECDSA	Elliptic Curve Digital Signature Algorithm
Gwei	GigaWei
ID	Identity Document
KRC	Key Revocation Certificate
MEV	Maximal Extractable Value
OCSP	Online Certificate Status Protocol
PKI	Public Key Certificate
RNG	Random Number Generator
s	Secret
SE	Secure Element
SIM	Subscriber Identity Module
SSS	Shamir Secret sharing
T	Timestamp
TEE	Trusted Execution Environment
2FA	Two-factor Authentication
TPM	Trusted Platform Module
TSE	Trusted Storage Environment

A. HARDWARE WALLETS

Hardware wallets are physical devices designed to securely store private keys offline, making them immune to cyber attacks, phishing sites, and malware. They utilize a Secure Element (SE) chip to safeguard the private keys [25]. SE-based hardware wallets employ specialized firmware [26] for handling transaction signing and other blockchain-related operations. These wallets offer user interfaces for managing cryptocurrency assets and sending signed transactions to the blockchain network, which can be accessed through mobile or laptop devices.

B. SMART ACCOUNTS

Smart accounts are smart contracts that employ the logic of a wallet to provide users with customizable and programmable functionalities. They extend beyond traditional wallets secured by private keys, offering features such as passkey-based signing, web-2-like session management, gas sponsorship, and more [27]. This enhancement of the user experience streamlines common tasks and reduces friction in day-to-day interactions. Smart accounts can also incorporate robust security measures, including the option for social recovery, ensuring the safety and integrity of cryptocurrency assets.

C. KEY REVOCATION CERTIFICATE (KRC)

In traditional PKI systems, certificate revocation involves invalidating a digital certificate before its intended expiration date. This process is crucial because, when a user's private key is lost or compromised, the legitimacy of the certificate owner's identification is questioned, or doubts arise about the certificate's credibility. To accomplish this, a revocation certificate is generated, informing other entities within the PKI system that the specific public key should no longer be trusted. Depending on the scenarios, the revocation certificate can be either generated by the Certificate Authority (CA) or the end-user themselves.

The process of revocation is typically accomplished by either adding the revoked certificate to a Certificate Revocation List (CRL) or querying it using the Online Certificate Status Protocol (OCSP). In our proposed architecture, we draw inspiration from PKI revocation certificates and introduce the concept of a Key Revocation Certificate (KRC). Conceptually, it is similar to a revocation certificate but is used for blocking transactions originating from the blockchain address mentioned in the KRC. This is achieved by including the blockchain address mentioned in the KRC in a block-list. The KRC is digitally signed using the private key associated with the user's account in their hardware wallet. The contents of the KRC include the blockchain account address to be revoked, the smart account contract address issued to the user, a hashed *secret*, and a digital signature over these contents.

D. KEY REVOCATION REGISTRY

In our proposed solution, the Key Revocation Registry (or simply Registry) is a smart contract deployed on the blockchain. Similar to the Certificate Revocation List (CRL) in PKI, it stores the block-list containing the list of revoked blockchain account addresses. With the help of this Registry, any transactions made through the user's lost hardware wallet account can be blocked. This is done by the user by publishing their KRC corresponding to the lost account to the Registry smart contract. After authentication, the Registry transfers the ownership of funds from the lost hardware wallet account to the user's new account. This transfer of ownership of funds is seamless, as we use smart accounts for funds management.

E. SECRET

A 256-bit long *secret* is generated by leveraging the Random Number Generator (RNG) embedded in the SE of hardware wallets. This *secret* serves as the key component for recovering account ownership. It is shared with the Registry in an obfuscated form initially while blocking the original account and later revealed in plain text for the final account ownership transfer. The logic-level implementation of this process is explained in Section V.

IV. PROBLEM STATEMENT

Trusted Execution Environment (TEE) and Trusted Storage Environment (TSE)-based devices follow certain design paradigms, such as the security-by-design approach, strict access control, seed unicity, etc. Naturally, when TEE and TSE-based devices were introduced as a solution to web3 problems, more specifically, as a solution for secure wallets, these design paradigms should have been offered by default.

Given the central theme of decentralization in web3 and some incompatibilities, certain design paradigms could not be fulfilled. For instance, an SE-based wallet that offers TEE and TSE, cannot offer the seed unicity property, given that there is a hard requirement of backing up the seed. Once hardware wallet is backed up, the unicity guarantee is lost. Subsequently, SE-based (hardware) wallets become no different from software wallets, apart from their security-by-design assurance.

Just as transmitting private keys is unacceptable in any traditional security architecture design, similarly, the loss of the unicity property offered by hardware wallets should not be acceptable just for enabling the backup of the seed. However, the problem of effectively backing up a hardware wallet without losing its unicity property and without transmitting its seed/private key outside the TEE/TSE, to the best of our knowledge, has not been solved effectively. On one hand, to preserve the unicity property, the hardware wallet should not replicate the seed/keys in any form; on the other hand, backup is necessary for recovery in case of malfunction, theft or loss of the hardware wallet.

This problem is not new and various domains face it. For e.g., chip-based bank credit/debit cards, biometric passports, SIM cards, ID cards, etc. They need to be unique and it is not generally possible to create a clone for backup purposes. Only a replacement can be requested in case of loss/theft/malfunction from the issuing authorities. This solution works for traditional applications where there is a centralized authority controlling everything. For web3 and blockchain, the equation changes entirely as no centralized authority has control.

To resolve this, in this paper, we propose a novel and practical solution that works in a decentralized setting for hardware wallet recovery without the need for a backup, all while preserving the unicity property which current solutions completely miss. We take cues from traditional PKI systems, smart contracts and smart accounts to effectively execute it without a need for change at the blockchain protocol level. In addition, this solution completely maintains the decentralization aspect without compromising on security as well as the mandated design paradigms of hardware wallets. The detailed solution is presented in the next section.

V. PROPOSED SOLUTION

Premise: A user owns a hardware wallet that is initialized. The hardware wallet is not backed up. For simplicity, this hardware wallet has one public-private key pair. The address derived from this public key inside hardware wallet is called "owner address". On the blockchain side, there is a smart contract based account i.e., smart account. This smart account can perform all operations such as transfer of funds, interaction with other smart contracts, etc. It has strict access control, i.e., only user's "owner address" can access it. This is achieved by instantiating the specific "owner address" in the "owner" variable in the smart account's code. The smart account, before performing any operation, checks if the sender of the instruction is the same as the value in the "owner" variable. Note that the "owner address" is controlled by the user via its original hardware wallet. Suppose, the user loses their hardware wallet and therefore, can no longer send any transactions using "owner address" to the smart account. Since the hardware wallet was not backed up (to preserve *unicity*), a new recovery mechanism is proposed, which will allow the user to transfer ownership of the associated smart account to their new "owner address". This new "owner address" is from a new hardware wallet, although this is not a necessary requirement.

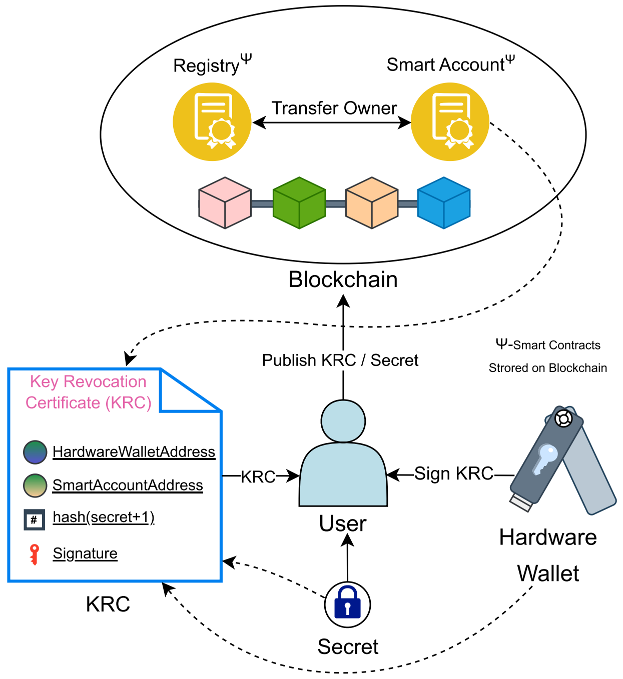

FIGURE 1. Entities interaction diagram.

The proposed mechanism can be implemented in two ways: i) via protocol-level changes or ii) via smart contracts. For simplicity, we will focus on the second approach, which does not require any changes to the underlying blockchain protocol. In this approach, the framework depends on three main components: 1. KRC, 2. Smart Contract-based Registry and 3. Smart Account.

To maintain the unicity and other security-by-design mandates of hardware wallets, the seed (or private key) should not come out of the hardware wallet in any form (whether logically or physically). Thus, any mechanism of backup of the seed/private key cannot be used. To allow the recovery of funds in case of an anomaly, a novel ownership transfer mechanism is proposed. This mechanism is based on symmetric secret sharing.

The proposed mechanism involves three steps: 1) Block Current Owner, 2) Register New Owner, and 3) Transfer Ownership. As these steps work in a decentralized and asynchronous setting, it is necessary to link actions from various steps to one user. This is done by sharing the obfuscation of the *secret* value s at the first two steps and linking them in the final step. These steps are elaborated in the subsequent subsections, respectively, with reference to the entities interaction diagram illustrated in Figure 1.

A. BLOCK CURRENT OWNER

Blocking of current owner is the first step of the proposed mechanism. The goal of this step is to register a new KRC. In the event that the physical hardware wallet has been lost/stolen or broken, the user can quickly start the recovery of their compromised "owner address" by publishing all the contents of the KRC (elaborated in Section III-C) on the blockchain. The registry smart contract is the target destination for the publication. On reception of the KRC, the registry is responsible for verifying the KRC authenticity by checking if it is signed by the same "owner address" which is to be added to the block-list. For this, the registry uses the public key of the user's original "owner address". Figure 2 highlights the main sequence of events for block current owner (Refer listing I in appendix for the implementation).

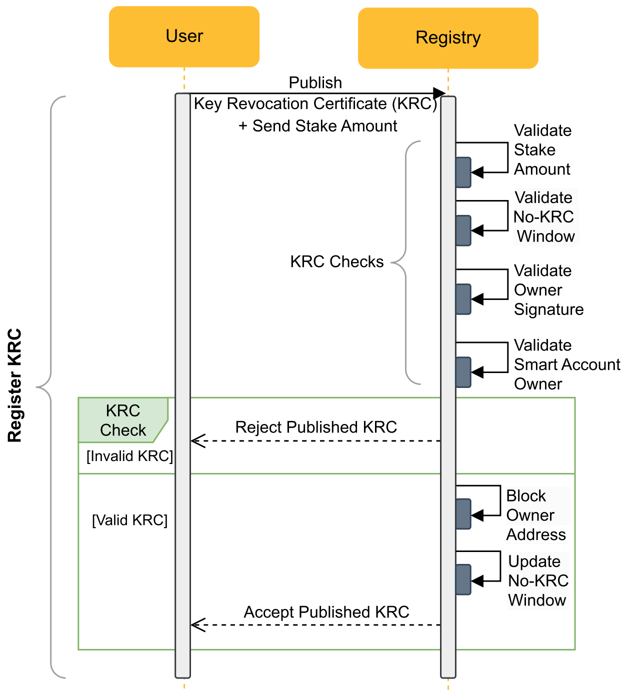

FIGURE 2. Sequence diagram highlighting blocking of current owner address.

On success, the registry will block the user's "owner address" by setting a flag in the registry database. This flag will prevent any further transactions from being executed by this address for its smart account.[4] Since the KRC was generated and signed by the owner address' private key when the hardware wallet was first initialized, ascertaining the identity is straightforward through signature verification.

Further, in the registry, the obfuscation of *secret* s is registered as *hash*($s+1$). This is used as the key in the registry mapping. In the value field, other values, i.e., the blocked owner account address, the smart account address, and the time of registration of the KRC (blocktime) are stored.

[4]Where this address is registered as owner.

Given that an attacker can generate a dummy KRC and block a particular *hash(s+1)*, even without having knowledge of the *secret* value *s*, a stake amount has to be sent along with the request, which subsequently gets refunded to the user after successful recovery. This prevents Denial of Service (DoS) attacks, as the attacker has to pay for each attempt. In the other scenario, when the KRC is leaked, the attacker can temporarily block access of the owner by publishing the KRC but again has to pay the stake amount for it, indirectly compensating the user for the loss of service.

Next, a timing window called "no-krc window" is enforced to prevent attackers from registering a new KRC during the ongoing recovery process. More information on various timing windows is elaborated in Section VI. Algorithm 1 illustrates the process of registering a KRC. The user should ensure their KRC is correctly registered with the right values before proceeding to the next step.

Algorithm 1 Register KRC

```
Data: Key Revocation Certificate (KRC)
Result: Address revoked
1  begin
2      initialization;
3      Get: Key Revocation Certificate (KRC);
4      if $txn.stake < stakeAmount$ then
5          revert transaction;
6      else
7          if no-KRC window is not expired then
8              revert transaction;
9          else if KRC signature is not valid then
10             revert transaction;
11         else if owner(KRC.smartAccAddress) is not
             valid then
12             revert transaction;
13         else
14             key ← KRC.hash(s+1);
15             block-list[KRC.address] ← True;
16             registry[key].address ← KRC.address;
17             registry[key].stake + = txn.stake;
18             Update no-KRC window;
19             sAdd ← KRC.smartAccAddress;
20             registry[key].smartAccAddress ← sAdd;
```

B. REGISTER NEW OWNER

Once the current "owner address" has been blocked, the next step is to register a new "owner address". This is done by sending a registration request to the registry containing the user's new "owner address" and the *secret s*. As this is carried out in a public smart contract setting, to prevent various attack vectors like MEV, this registration request is obfuscated. Owing to this deliberate obfuscation, no verification is done/possible. Figure 3 highlights the main sequence of events related to it.

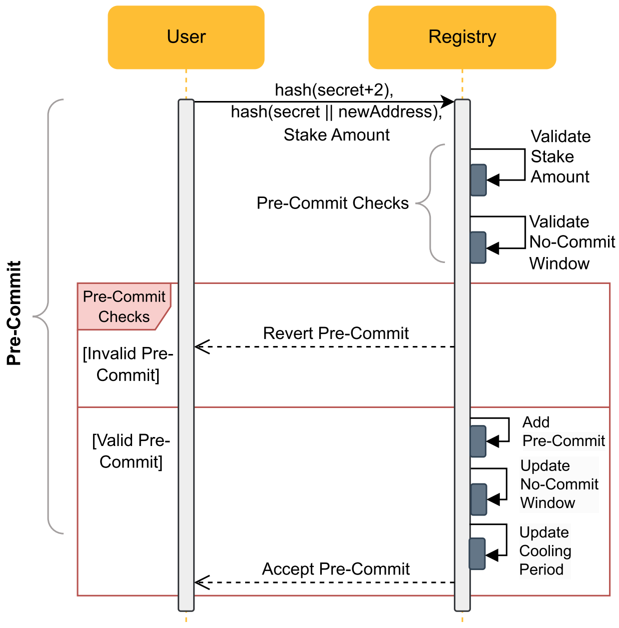

FIGURE 3. Sequence diagram highlighting registration of new owner address.

At the other end, where the pre-commit function of the registry receives this obfuscated registration request, it just receives two hashes as arguments, one key: *hash(s+2)* and one value: *hash(s||newAddress)*. The choice of *hash(s+2)* for obfuscation is deliberate since it removes any relation between step 1 and 2 (owing to different obfuscations of the *secret*) and prevents DoS attacks. Further, a stake amount is required, similar to step 1, that is returned to the user on successful recovery. This penalizes the attacker and compensates the user (see Section VIII). To further harden the system against attacks, two timing windows i.e., "no-commit" and "cooling" are proposed. During an ongoing "no-commit" timing window, no new pre-commits are allowed for same obfuscated *secret*. Similarly, while in "cooling period", executing the next steps of recovery is prohibited (see Section VI).

The two timing windows viz.,"no-commit", and "cooling" are initiated simultaneously once the pre-commit is registered. These ensure that an attacker cannot send a new pre-commit and a final account transfer request, respectively. This prevents any potential disruptions in the recovery routine. The rationale is discussed in Section VI. Algorithm 2 illustrates the process of pre-commit in detail. The user should ensure their pre-commit is correctly registered with the right values before proceeding to the final step.

C. TRANSFER OWNERSHIP

Once the first two steps are completed and the data is verified, the user can initiate the final account recovery process. This step transfers the ownership of the smart account to a new "owner address". This is a verification step that acts on the

Algorithm 2 Pre-Commit

```
   Data: hash(s+2), hash(s||newAddress)
   Result: Do pre-commit & update no-commit window
1  begin
2      initialization;
3      Get: hash(s+2) & hash(s||newAddress);
4      key ← hash(s+2);
5      value ← hash(s||newAddress);
6      if $txn.stake < stakeAmount$ then
7          revert transaction;
8      else if no-commit window is not expired then
9          revert transaction;
10     else
11         pre-commit[key].hash ← value;
12         Update no-commit window;
13         Update cooling period;
14         pre-commit[key].stake + = txn.stake;
```

data sent previously and stored on the registry. The process starts when the user sends its *secret* value *s* and the new designated "owner address" to the registry in plaintext. Note that, even if the details are sent in plaintext, due to various timing windows (see Section VI), this step is not vulnerable to MEV attacks. Figure 4 depicts the sequence of the ownership transfer process.

Next, on receiving the *secret* value and the new "owner address", the registry computes the various keys (*hash(s)*, *hash(s+1)*, and *hash(s+2)*) and gets corresponding values to cross-verify the following:

- If the *secret s* is valid and not consumed previously.
- If the old "owner address" is still the owner of the smart account in question.
- If the new "owner address" is the same as the one provided during pre-commit.
- If the cooling period is passed.

On success, the registry invokes the smart account's "transfer owner" function and assigns the new address as the owner. Further, it transfers all the staked amount to the smart account and marks the *secret s* as consumed by adding the obfuscated *hash(s)* to a key-value mapping. The reason for storing *secret s* in obfuscated form is to prevent leaking additional information about its generation and prevent dictionary attacks in the future. Algorithm 3 elaborates this process in detail (Refer Listing I–III in Appendix).

D. FURTHER DISCUSSION

It can be correctly estimated that the above mentioned first two steps of the mechanism, namely *Block Current Owner* and*Register New Owner*, can be combined into a single step. However, considering various practical scenarios, the end-user may not always have another hardware wallet or a new "owner address" ready to immediately replace the compromised one. Although, if such a choice is forced, the end-user may use less secure means to execute recovery. Therefore, we proposed it as a two-step approach wherein, as soon as the compromise occurs, access to funds is blocked.

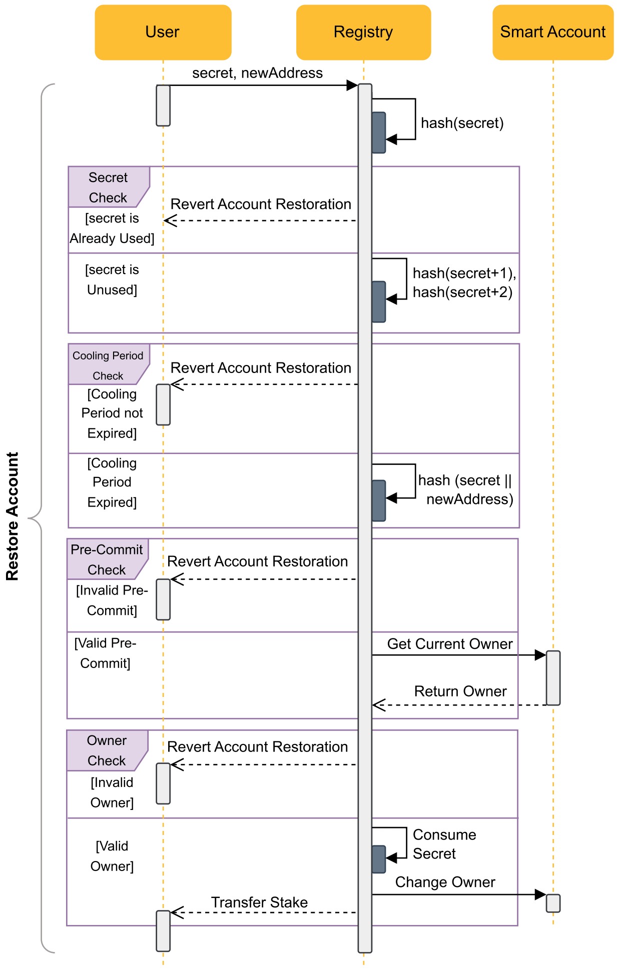

FIGURE 4. Sequence diagram highlighting ownership transfer process.

Next, it is evident that the registry's logic can also be part of the smart account. However, a dedicated registry helps to track the block-list and ensures coherency of the implementation logic. Further, since attackers can see who are interacting with any smart contract, an integrated registry is more prone to being surveilled and DoS attacks despite staking requirements. The reason being a link can easily be established between *hash(s+1)* and *hash(s+2)*. In a standalone public registry, where multiple users are interacting at a given time, only a probabilistic relationship can be established between them. It can be further reduced to near-zero by the user utilizing different addresses for interacting with the registry at each step.

In the last verification step, it can be remarked that the registry does not check if "no-krc" or "no-commit" windows are ongoing. This is a deliberate design choice since the idea of these windows is to prevent attacks, not recovery. However, for security, the user is strongly advised to ensure these windows are in-force before executing the third step.

Algorithm 3 Restore Account

```
Data: Secret s and newAddress
Result: Restore funds & stakes to newAddress
1  begin
2      initialization;
3      Get: secret s and newAddress;
4      key1 ← hash(s+1);
5      key2 ← hash(s+2);
6      sAdd ← registry[key1].smartAccAddress;
7      if secret s is consumed then
8          revert transaction;
9      else if cooling period is not expired then
10         revert transaction;
11     else if pre-commit tampered or invalid then
12         revert transaction;
13     else if owner(sAdd) ! = registry[key1].address
        then
14         revert transaction;
15     else
16         stk1 ← registry[key1].stake;
17         stk2 ← pre-commit[key2].stake;
18         totStake ← stk1 + stk2;
19         Delete registry[key1] & pre-commit[key2];
20         Mark secret s as consumed;
21         sAdd.transferOwner(newAddress);
22         Transfer totStake amount to newAddress;
```

Further, predictably, the mechanism can be well utilized for software wallets. However, the original/main proposition of guaranteeing seed unicity won't be maintainable as the keys are easily extractable for software wallets, depending on implementations. Barring this, the proposed mechanism is indeed a better option with added security assurances compared to other backup and recovery mechanisms for software wallets (see Section VIII).

VI. TIMING WINDOWS

The proposition of the timing windows is essential in our proposed mechanism to safeguard against MEV attacks. Since all the steps are carried out in a decentralized setting, there is no single authority to ascertain the identity of an incoming request, as the original wallet is already inaccessible for the user. In principle, we propose two timing windows, namely "no-krc" and "no-commit".

With reference to Figure 5, at step 1, when the user registers its KRC, the timestamp (current blocktime - T_2) is recorded. This is the starting reference point for the "no-krc" window. As the name indicates, this essentially enforces the condition that no new KRC can be registered against the same obfuscated s value (*hash(s+1)*) during this window (T_2 - T_7).

Similarly, at step 2, when the user pre-commits his new "owner address", the timestamp (current blocktime - T_4) is recorded. This becomes the starting reference point for the "no-commit" window. As the name suggests, this similarly enforces the condition that no new pre-commit can be registered against the same obfuscated s value (*hash(s+2)*) during this window (T_4 - T_9).

Both these conditions are necessary because they ensure the relevant rows in the mappings are not updated when the final transfer of ownership is initiated. This secures step 3 against possible MEV attacks. Another way to prevent changes to the mappings would be to ensure that they are written only once. However, this approach exposes the user to *irrecoverability* if the attacker blocks the mapping by inserting random values.

Next, to harden the proposed mechanism against fork attacks, we introduce the concept of "cooling period". The cooling period is defined as the *minimum* time interval between the end of the second step and the start of the third step. This period starts after the function for step 2 is invoked, i.e., T_4. The user has to wait until the "cooling period" expires (T_4 - T_5) before executing the final step. During this period, the system waits for the blockchain to finalize the transaction (i.e., the block containing the transaction is confirmed by the network). This ensures that the final step is executed only after the previous steps have been confirmed and part of the main chain, making it resistant to fork attacks. Ultimately, if they end up in the forked branch, the user can redo step 1 and 2 without waiting, as for the blockchain, the data on the forked branch will be eventually dropped.

Furthermore, to maintain security across various steps and the safety of funds, the user must execute step 3 only when the "no-krc" and "no-commit" windows are still in force while ensuring that the "cooling period" has passed. Additionally, the user should also ensure that the estimated time remaining after executing step 3 is at least equal to the "cooling period" before either of the timing windows expires. In the current example, Figure 5, this period is T_6 - T_7. This ensures that the mappings are not only secure before executing the final step but also after execution until the result becomes part of the main chain. Further, the user should also ensure that the relevant rows of the mappings contain the correct values. Failing this, the recovery may be disrupted. If any of these conditions are unmet, the user has to recommence the recovery process (after the expiry of any ongoing timing windows). To help the user, a support function *isRestorePossible()* is implemented in the registry (refer to listing 1 in Appendix), which informs if it is safe to reveal *secret s* for completing account recovery.

VII. TESTBED IMPLEMENTATION AND EVALUATION

The algorithms of the proposed recovery mechanism (see Section V) are implemented within a smart contract called Registry on the Ethereum testnet. In addition, a library with helper functions for the Registry is also implemented. This library assists the Registry with various functions like encoding, hashing, signature verification, etc. Furthermore, a simple implementation of a smart account is also carried out to simulate the final ownership transfer.

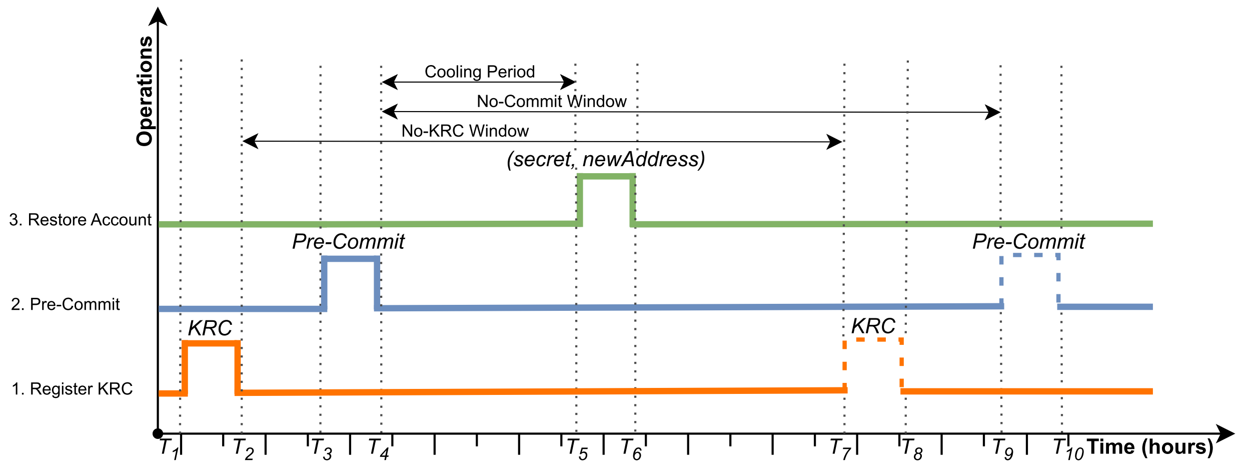

FIGURE 5. Timing diagram for pre-commit and restore account.

The smart contracts were written in the Solidity programming language [28] and compiled using Solidity compiler version 0.8.24. The smart contracts' code is available under Appendix. All the smart contracts were deployed using the Remix framework [29] and MetaMask [6] on the Sepolia Ethereum testnet [30], [31] for evaluating performance, feasibility, and gas consumption.

The amount of gas consumed in each operation - KRC registration, pre-commit, and account restoration - is measured and plotted in the form of a bar graph, as illustrated in Figure 6.

The deployment of the Registry contract requires 1,678,390 gas. The gas fee on the mainnet[5] at the time of our deployment (on 26 August 2024, 16:00 UTC) is 3 Gwei per unit of gas and 1 Ethereum costs 2710 USD. Considering 1 Ethereum equals 10^9 Gwei, the total dollar cost to deploy the Registry contract is 13.64 USD. The deployment of the Registry smart contract is a one-time cost; once deployed, its code is immutable, and no recurring cost is incurred to maintain the Registry contract on-chain.

Similarly, the cost of registering a KRC is 154,957 gas. Using the aforementioned conversion rates, this amounts to 1.25 USD in dollar terms. The main contribution to this gas component is the ECDSA signature verification performed to check the legitimacy of the KRC.

The transaction fee for registering a pre-commit is 93,271 gas or 0.75 USD. This includes the cost of storing the pre-commit data in the contract storage and updating the registry state for timing windows. The cost for the final transfer of ownership is 83,881 gas or 0.69 USD. This includes the cost of transferring the ownership of the smart account to the new "owner address" and also the cost of transferring the stake amount to the smart account.

The other smart contracts residing on-chain may also need to check the block-list in the Registry to prevent approving

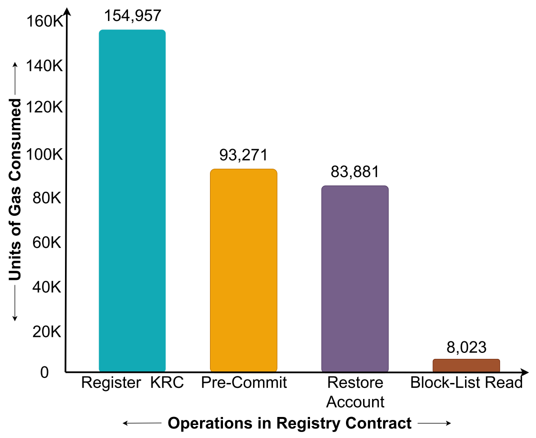

FIGURE 6. Gas consumption of various operations of the proposed recovery mechanism.

transactions from a blocked "owner address". This operation consumes 8,023 gas (0.07 USD) when the block-list read operation is followed by state mutation operations. In other cases, it's free as read-only operations do not incur costs for the user since they are executed locally.

From the above figures, it can be summarized that the cost of using our recovery mechanism is very low[6] with negligible overhead costs, even when implemented on heavily congested blockchains [32], [33] like ethereum. On less congested blockchains like layer-2s [34], the cost will be near zero, all while maintaining the unicity property and various other security advantages discussed in Section VIII.

VIII. SECURITY ANALYSIS

Given the sensitive nature of account ownership, the proposed mechanism is evaluated against various attacks. The algorithms have been hardened against the attacks

[5]Mainnet fee taken into account for estimating the real-world costs.

[6]Total recovery cost is ≈ 2.7 USD.

mentioned below. In the next sub-section, we elaborate on the assumptions followed by an individual analysis of various attacks.

A. ASSUMPTIONS

For the security analysis of our approach, we make the following assumptions:

- The KRC and the *secret* are stored securely and are accessible to the users.
- The user owns a smart account with a proper "transferOwner()" function.
- The user has sufficient funds required for staking and executing transactions when utilizing the proposed recovery mechanism.
- The user has access to a secure graphical interface to interact with the registry.
- The underlying one-way trap-door *hash* function is secure [35].
- The underlying signature algorithm used for KRC signing is secure and resistant to tampering.

B. MEV ATTACKS

The Maximal (formerly "miner" in the context of Proof of Work) Extractable Value (MEV) refers to the additional value profit that can be gained by censoring and/or changing the order of transactions in a block. When a transaction is sent to the blockchain, there is a delay between the time when it was broadcasted to the network and when it is finally mined into a block. During this delay period, the transaction is a part of a pool of pending transactions called the *mempool* where the contents are visible to everyone [36].

Arbitrageurs and miners can monitor the *mempool* and find opportunities to maximize their profits e.g., by front-running transactions.[7] If a front-runner is a miner, they can also reorder or even censor transactions. In an MEV attack, a miner, for e.g., will prioritize their own transactions or delay other transactions to maximize their profits.

In our mechanism, the data stored at various steps is obfuscated. The obfuscated values protecting the *secret s* are calculated using a one-way trap-door function (*hash*). At the *mempool* level, the miner can only see the hashed values. With an MEV attack intent, the miner can only change or tamper with those hash values. However, to have a valid full recovery, the original *secret s* obfuscation has to be included since this value is part of the original signed KRC (and there is no other access control possible). At the time of the last step of recovery, the miner, without the knowledge of *secret s*, will fail to complete the request and thus will lose the stake amount that was mandated in the algorithm.

Further, at the last step of the recovery, when the plaintext *secret s* is revealed, due to the enforcement of the various timing windows (see Section VI), MEV attacks are prevented as no change of mappings is allowed when the timing windows are active. Due to the hardening of the algorithms, both before and after of each step, MEV attacks remain infeasible and also financially taxing for the attacker.

[7]Sending a similar transaction with higher fees so that it gets confirmed earlier.

TABLE 2. Theft of KRC/secret.

Theft/lost scenarios	Recommended Actions
KRC stolen/lost, wallet still present.	Create a new KRC and *secret*. Safely dispose old *secret*.
Secret stolen/lost, wallet still present.	Create a new KRC and *secret*. Safely dispose old KRC.

TABLE 3. Comparison between backup/recovery schemes.

Case/ Scheme	Secure Backup Scheme [16]	Ledger Recover [21]	Social Recovery [22], [23] & Multi-signature methods [39]	Our scheme
Private key export	Yes	Yes	**No**	**No**
Guardian support	**No**	**No**	Yes	**No**
Collusion vulnerability	**No**	Yes	Yes	**No**
Instant account blocking	No	No	No	**Yes**
Third party dependence	**No**	Yes	**No**	**No**
Unicity property	No	No	**Yes**	**Yes**
Denial of Service	**No**	**No**	Yes	**No**

C. DOS ATTACK

In our proposed mechanism, an attacker possessing the user's KRC can launch a Denial of Service (DoS) attack at step 1 by publishing the same KRC to the Registry smart contract. Upon publication, the user associated with the KRC is unable to use their account. We mitigate the likelihood of such DoS attacks by requiring a stake amount when registering a KRC. This stake amount is then transferred to the associated smart account at the last step of recovery.

As the user is the only party possessing the knowledge of *secret s* linked to the published KRC, they are the only ones who can reclaim the stake amount during the final step of the recovery. Since the attacker lacks knowledge of the *secret s*, they are disincentivized from publishing the KRC as they will lose the associated stake amount. Similarly, an attacker can initiate a DoS attack at step 2 by adding random *hash* entries to the mappings. However, by enforcing the similar aforementioned staking requirement, the attacker is penalized for the attack, while the user is compensated for the DoS.

The practical idea of associating a staked amount with the obfuscated *secret s* and ensuring its transfer to the original smart account on successful recovery by the user

```
// Copyright(C) 2024, Samsung Research
// Authors: Varun Deshpande, Harish J, and Atharva Vijay Khade
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.24;

import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import "./SmartAccount.sol";
import "./KeyRevocationLib.sol";

contract KeyRevocationRegistry {
    // Cold wallet address block status tracker
    mapping(address => bool) public isAddressBlocked;
    // hash(s+1) -> cold wallet address
    mapping(bytes32 => KeyRevocationLib.RegisterKrcStruct)
        private registerKrcInfo;

    // hash(s+2) -> hash (s || new wallet address)
    mapping(bytes32 => KeyRevocationLib.PrecommitStruct) private preCommitInfo;

    // hash(s) -> boolean consumed or not
    mapping(bytes32 => bool) private isHashConsumed;

    KeyRevocationLib.SecurityParams private securityParams;

    // stakeAmount is in WEI
    constructor(
        uint256 coolingPeriod,
        uint256 noCommitWindow,
        uint256 noKrcWindow,
        uint256 stakeAmount
    ) {
        require(
            noCommitWindow > coolingPeriod && noKrcWindow > coolingPeriod,
            "Invalid timing windows"
        );
        securityParams.coolingPeriod = coolingPeriod;
        securityParams.noCommitWindow = noCommitWindow;
        securityParams.noKrcWindow = noKrcWindow;
        securityParams.stakeAmount = stakeAmount;
    }

    // User publishes KRC to block the account when hardware wallet is lost
    function registerKRC(
        address coldWalletAddress,
        address smartAccountAddress,
        bytes32 secretHash,
        bytes memory signature
    ) public payable {
        require(
            msg.value >= securityParams.stakeAmount,
            "Insufficient WEI sent"
        );
        require(
            block.timestamp >
                registerKrcInfo[secretHash].blockTimeStamp +
                    securityParams.noKrcWindow,
            "register KRC not allowed"
        );

        bytes memory encodedData = KeyRevocationLib.encode(
            coldWalletAddress,
            smartAccountAddress,
            secretHash
        );
        bytes32 hash = MessageHashUtils.toEthSignedMessageHash(
            KeyRevocationLib.getRevocationHash(encodedData)
        );

        require(
```

LISTING 1. Registry smart contract.

```
            coldWalletAddress == ECDSA.recover(hash, signature),
            "Invalid Signature"
        );
        require(
            SmartAccount(payable(smartAccountAddress)).owner() ==
                coldWalletAddress,
            "Invalid Owner"
        );
        isAddressBlocked[coldWalletAddress] = true;
        registerKrcInfo[secretHash].coldWalletAddress = coldWalletAddress;
        registerKrcInfo[secretHash].stakeAmount += msg.value;
        registerKrcInfo[secretHash].blockTimeStamp = block.timestamp;
        registerKrcInfo[secretHash].smartAccountAddress = smartAccountAddress;
    }

    // This function is called for doing pre-commit
    function preCommitForAccountRestore(
        bytes32 hashSp2,
        bytes32 hashSecretPlusNewAddr
    ) public payable returns (bool) {
        require(
            msg.value >= securityParams.stakeAmount,
            "Insufficient WEI sent"
        );
        // Check if caller can do the pre-commit or not
        require(
            block.timestamp >
                preCommitInfo[hashSp2].blockTimeStamp +
                    securityParams.noCommitWindow,
            "pre-commit not allowed"
        );
        // Do the pre-commit, update the blockTime and return true
        preCommitInfo[hashSp2].hashSecretPlusNewAddress = hashSecretPlusNewAddr;
        preCommitInfo[hashSp2].blockTimeStamp = block.timestamp;
        preCommitInfo[hashSp2].stakeAmount += msg.value;
        return true;
    }

    // User reveals secret and newAddress to transfer ownership
    function restoreAccount(bytes32 secret, address newAddress) public {
        bytes32 secretHash = keccak256(abi.encode(secret));
        // avoid re-use of s
        require(
            isHashConsumed[secretHash] == false,
            "Secret is already consumed"
        );

        bytes memory Sp1Bytes = abi.encode(uint256(secret) + 1);
        bytes memory Sp2Bytes = abi.encode(uint256(secret) + 2);

        bytes32 hashSp1 = keccak256(Sp1Bytes);
        bytes32 hashSp2 = keccak256(Sp2Bytes);

        require(
            (block.timestamp >
                preCommitInfo[hashSp2].blockTimeStamp +
                    securityParams.coolingPeriod),
            "RE: Activation time still not elapsed"
        );

        bytes32 hashSecretPlusNewAddr = KeyRevocationLib.calculateHash(
            secret,
            newAddress
        );

        // if attacker has tampered with pre-commit
        if (
            preCommitInfo[hashSp2].hashSecretPlusNewAddress !=
            hashSecretPlusNewAddr
        ) {
            revert("pre-commit tampered");
        }
```

LISTING 1. *(Continued.)* **Registry smart contract.**

```

        //transfer owner
        address smartAccountAddress = registerKrcInfo[hashSp1]
            .smartAccountAddress;
        require(
            SmartAccount(payable(smartAccountAddress)).owner() ==
                registerKrcInfo[hashSp1].coldWalletAddress,
            "Invalid Owner"
        );

        //invalidate mappings
        uint256 stakeAmt = registerKrcInfo[hashSp1].stakeAmount +
            preCommitInfo[hashSp2].stakeAmount;

        delete registerKrcInfo[hashSp1];
        delete preCommitInfo[hashSp2];

        // set secret as consumed
        isHashConsumed[secretHash] = true;
        SmartAccount(payable(smartAccountAddress)).transferOwner(newAddress);

        payable(smartAccountAddress).transfer(stakeAmt);
    }

    function isRestorePossible(
        address smartAccountAddress,
        bytes32 hashSp1,
        bytes32 hashSp2,
        bytes32 hashSecretPlusNewAddr
    ) public view returns (bool) {
        uint256 registerKrcTimeStamp = registerKrcInfo[hashSp1].blockTimeStamp;
        address coldWalletAddr = registerKrcInfo[hashSp1].coldWalletAddress;
        uint256 preCommitTimeStamp = preCommitInfo[hashSp2].blockTimeStamp;

        // cooling period is subtracted from the minimum to account for potential restore account txs not
             getting on-chain.
        uint256 upperBoundWindowLimit = KeyRevocationLib.min(
            registerKrcTimeStamp + securityParams.noKrcWindow,
            preCommitTimeStamp + securityParams.noCommitWindow
        ) - securityParams.coolingPeriod;

        uint256 lowerBoundWindowLimit = KeyRevocationLib.max(
            registerKrcTimeStamp,
            preCommitTimeStamp + securityParams.coolingPeriod
        );
        require(
            SmartAccount(payable(smartAccountAddress)).owner() ==
                coldWalletAddr,
            "Invalid Owner"
        );
        if (
            preCommitInfo[hashSp2].hashSecretPlusNewAddress !=
            hashSecretPlusNewAddr
        ) {
            revert("pre-commit tampered");
        }
        if (
            block.timestamp > lowerBoundWindowLimit &&
            block.timestamp <= upperBoundWindowLimit
        ) return true;
        return false;
    }
}
```

LISTING 1. ***(Continued.)*** **Registry smart contract.**

(on revealing the plaintext *secret s*) ensures that the user is compensated by the attacker for even attempting the attack.

For the last step, the DoS attack can happen if the mappings from the previous two steps are tampered, disrupting the whole recovery process. Since, in this step, only revealing takes place, the DoS attack is prevented by timing windows (see Section VI), which ensure that the mappings are immutable for a period of time. While the mappings are temporarily immutable,[8] the attacker can neither register new KRC nor issue a new pre-commit for the associated *secret s*.

[8]Only for the corresponding *secret s*.

```
// Copyright(C) 2024, Samsung Research
// Authors: Varun Deshpande, Harish J, and Atharva Vijay Khade
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.24;

import "./KeyRevocationRegistry.sol";

contract SmartAccount {
    address public owner;
    address public KEY_REGISTRY_ADDRESS;

    constructor(address keyRegistryAddress) {
        owner = msg.sender;
        KEY_REGISTRY_ADDRESS = keyRegistryAddress;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "only owner");
        _;
    }

    modifier onlyRegistry() {
        require(
            msg.sender == KEY_REGISTRY_ADDRESS,
            "only callable from key revocation registry"
        );
        _;
    }

    function call(
        address target,
        uint256 value,
        bytes memory data
    ) public onlyOwner {
        // check the block listing
        require(
            KeyRevocationRegistry(KEY_REGISTRY_ADDRESS).isAddressBlocked(
                msg.sender
            ) == false,
            "Sender's address is blocked"
        );

        (bool success, bytes memory result) = target.call{value: value}(data);
        if (!success) {
            assembly {
                revert(add(result, 32), mload(result))
            }
        }
    }

    function transferOwner(address _newAddress) public onlyRegistry {
        owner = _newAddress;
    }

    receive() external payable {}
}
```

LISTING 2. Smart account smart contract.

This ensures that the user can safely complete his recovery and transfer of ownership.

D. BRUTE FORCE ATTACKS

A brute force attack involves randomly trying out different possible values until the correct one is found. In our proposed mechanism, the *secret* s can be subjected to a brute force attack. The proposed length of *secret* s is 256-bits. Brute forcing such a long value is extremely time-consuming and demands a lot of computational power. Powerful supercomputers will take approximately 3×10^{51} years to correctly guess a 256-bit *secret* [37]. A quantum computer with Grover's algorithm will require executing 2^{128} combinations, making it currently infeasible to achieve [38].

E. REPLAY ATTACK

In our proposed mechanism, the user's *secret* becomes known to the entire network upon successful completion of the restore account operation. This allows the attacker to launch replay attacks by utilizing the user's *secret* to gain ownership of their smart account. To execute this attack, an attacker can repeat the Register KRC, pre-commit, and restore account operations, using the user's KRC and now-revealed *secret* along with its account address for ownership transfer.

```
// Copyright(C) 2024, Samsung Research
// Authors: Varun Deshpande, Harish J, and Atharva Vijay Khade
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.24;

library KeyRevocationLib {
    struct RegisterKrcStruct {
        address coldWalletAddress;
        address smartAccountAddress;
        uint256 blockTimeStamp;
        uint256 stakeAmount;
    }

    struct PrecommitStruct {
        bytes32 hashSecretPlusNewAddress;
        uint256 blockTimeStamp;
        uint256 stakeAmount;
    }

    struct SecurityParams {
        uint256 coolingPeriod;
        uint256 noCommitWindow;
        uint256 noKrcWindow;
        uint256 stakeAmount;
    }

    function encode(
        address coldWalletAddress,
        address smartAccountAddress,
        bytes32 secretHash
    ) public pure returns (bytes memory ret) {
        return abi.encode(coldWalletAddress, smartAccountAddress, secretHash);
    }

    function getRevocationHash(
        bytes memory encodedData
    ) public pure returns (bytes32) {
        return keccak256(encodedData);
    }

    function calculateHash(
        bytes32 secret,
        address newAddress
    ) public pure returns (bytes32) {
        return
            keccak256(
                abi.encode(secret, bytes32(uint256(uint160(newAddress))))
            );
    }

    function min(uint256 a, uint256 b) public pure returns (uint256) {
        return (a <= b ? a : b);
    }

    function max(uint256 a, uint256 b) public pure returns (uint256) {
        return (a >= b ? a : b);
    }
}
```

LISTING 3. Utility library smart contract.

The possibility of such a replay attack is effectively eliminated at the last step (Step 3) when the *secret s* is revealed. The registry checks if the *secret* is already consumed and stops recovery if true. Further, it also checks if the account owner address that signed the KRC is still the owner of the smart account for which the account transfer is requested. These checks harden the mechanism against replay attacks.

F. THEFT OF KRC/SECRET

Our proposed mechanism is anchored on two components i.e., *secret s* and KRC. These components together help with account recovery. Thus, for safety, it is recommended to keep them separate. The ramifications if they are lost or stolen together or separately are different. In either case of *secret s* or KRC being lost or stolen independently, the underlying smart account is still safe. The recommended actions to execute in each loss/theft scenario are elaborated in Table 2.

The two components in our mechanism effectively break the risk into two, where neither one of them is sufficient to cause financial loss. Given their ephemeral nature, a replacement can be quickly generated. This is in contrast to the current most widely used mnemonic paper backup solutions, where if an attacker finds the backup paper, it can recover the root seed quickly and steal all funds. To have the same ramification, both KRC and *secret* need to be stolen together by a single attacking party.

Through such an approach, a natural comparison is warranted, especially with approaches that work on a similar paradigm of dividing the risk into multiple components. Notable examples include Shamir Secret Sharing, Social Recovery (based on multi-signature schemes [39]), etc. Table 3 shows how our proposed mechanism is much better with little to no overhead cost (see Section VII). Ideal values are highlighted in bold. The point to highlight is that none of them supports the unicity property except social recovery/multi-signature schemes, which are very prone to DoS and collusion.

IX. CONCLUSION

In this work, we propose a novel and practical recovery mechanism for blockchain hardware wallets that effectively solves the problem of wallet recovery without a backup while preserving unicity and other security-by-design paradigms. The mechanism is inspired by PKI, wherein the KRC and *secret* are the fundamental components. The algorithms within the proposed mechanism are hardened and evaluated against various security attacks. We implemented a prototype of our proposed mechanism and estimated the gas overhead for each operation - KRC register (refer the Algorithm 1), pre-commit (refer the Algorithm 2), and account restoration (refer the Algorithm 3). The total cost of recovering back our account from our experiments is just 2.7 USD, showcasing the cost-effectiveness of our approach. We also compared our solution with the existing ones and highlighted its unique security advantages.

APPENDIX
SMART CONTRACTS SOURCE CODE

See Listing 1–3.

ACKNOWLEDGMENT

The authors would like to express their sincere gratitude to Srivatsan Deenadayalan and Gandhi Kishor Addanki at the Blockchain Center of Excellence, Samsung Research, Bengaluru, India, for their invaluable support and guidance in this research work. *(Harish J and Atharva Vijay Khade contributed equally to this work.)*

REFERENCES

[1] Y. Yu, T. Sharma, S. Das, and Y. Wang, "'Don't put all your eggs in one basket': How cryptocurrency users choose and secure their wallets," in *Proc. CHI Conf. Human Factors Comput. Syst.* New York, NY, USA: Association for Computing Machinery, May 2024, pp. 1–17.

[2] M. Guri, "BeatCoin: Leaking private keys from air-gapped cryptocurrency wallets," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Phys. Social Comput. (CPSCom) IEEE Smart Data (SmartData)*, Jul. 2018, pp. 1308–1316.

[3] TrustWallet. (2023). *Hardware Wallets Vs. Software Wallets: What's the Real Difference*. Accessed: Aug. 26, 2024. [Online]. Available: https://trustwallet.com/blog/hardware-wallets-vs-software-wallets-whats-the-real-difference

[4] Ledger. (2024). *Ledger Nano S Plus*. Accessed: Oct. 18, 2024. [Online]. Available: https://shop.ledger.com/pages/ledger-nano-s-plus

[5] Trezor. (2024). *Trezor Model T*. Accessed: Oct. 18, 2024. [Online]. Available: https://trezor.io/trezor-model-t?srsltid=AfmBOooxfsFo_h4iR9aDbQv7SzaGwhtOAxAO-cnHD8hBwPfIvua2pehe

[6] (2024). *Metamask*. Accessed: Aug. 26, 2024. [Online]. Available: https://metamask.io/

[7] (2024). *Electrum Bitcoin Wallet*. Accessed: Aug. 26, 2024. [Online]. Available: https://electrum.org/

[8] J. N'Gumah, "Evaluating security in cryptocurrency wallets," Master's thesis, St. Cloud State Univ., Herberger School Bus., St. Cloud, MI, USA, 2021. [Online]. Available: https://repository.stcloudstate.edu/msia_etds/115/

[9] E. Akyildirim, T. Conlon, S. Corbet, and J. W. Goodell, "Understanding the FTX exchange collapse: A dynamic connectedness approach," *Finance Res. Lett.*, vol. 53, May 2023, Art. no. 103643. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S154461232300017X

[10] A. Hetler. (2024). *Ftx Scam Explained: Everything You Need to Know*. Accessed: Aug. 26, 2024. [Online]. Available: https://www.techtarget.com/whatis/feature/FTX-scam-explained-Everything-you-need-to-know/

[11] P. Garg. (2022). *Almost $1 Million Stolen in Phishing Attack on Electrum Wallet*. Accessed: Aug. 26, 2024. [Online]. Available: https://crypto.news/almost-1-million-stolen-phishing-attack-electrum-wallet/

[12] V. Pamnani and P. Matarazzo. (Jul. 2024). *Trusted Platform Module Technology Overview | Microsoft Learn*. Accessed: Aug. 28, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/trusted-platform-module-overview

[13] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, document RFC 5280, May 2008. [Online]. Available: https://www.rfc-editor.org/info/rfc5280

[14] (2013). *Mnemonic Code for Generating Deterministic Keys*. Accessed: Jul. 18, 2024. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

[15] Y. Liu, R. Li, X. Liu, J. Wang, L. Zhang, C. Tang, and H. Kang, "An efficient method to enhance bitcoin wallet security," in *Proc. 11th IEEE Int. Conf. Anti-Counterfeiting, Secur., Identificat. (ASID)*, Oct. 2017, pp. 26–29.

[16] H. Rezaeighaleh and C. C. Zou, "New secure approach to backup cryptocurrency wallets," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2019, pp. 1–6.

[17] S. Suratkar, M. Shirole, and S. Bhirud, "Cryptocurrency wallet: A review," in *Proc. 4th Int. Conf. Comput., Commun. Signal Process. (ICCCSP)*, Sep. 2020, pp. 1–7.

[18] B. Mackay, "Evaluation of security in hardware and software cryptocurrency wallets," School Comput. Edinburgh, Napier Univ. Edinburgh, Edinburgh, Scotland, 2019, doi: 10.13140/RG.2.2.31686.29768.

[19] S. Jokić, A. Cvetković, S. Adamović, N. Ristić, and P. Spalević, "Comparative analysis of cryptocurrency wallets vs traditional wallets," *Ekonomika*, vol. 65, no. 3, pp. 65–75, 2019.

[20] S. He, Q. Wu, X. Luo, Z. Liang, D. Li, H. Feng, H. Zheng, and Y. Li, "A social-network-based cryptocurrency wallet-management scheme," *IEEE Access*, vol. 6, pp. 7654–7663, 2018.

[21] Y. Badiss and I. Castillo. (2023). *Ledger Recover*. Accessed: Jul. 18, 2024. [Online]. Available: https://github.com/LedgerHQ/recover-whitepaper

[22] A. B. Pedin, N. Siasi, and M. Sameni, "Smart contract-based social recovery wallet management scheme for digital assets," in *Proc. ACM Southeast Conf.*, Apr. 2023, pp. 177–181.

[23] (2008). *Argent White Paper*. Accessed: Jul. 18, 2024. [Online]. Available: https://github.com/argentlabs/argent-contracts/blob/develop/specifications/specifications.pdf

[24] A. V. Khade, H. R. Patel, and C. Modi, ''Mnemonic phrase management and SIM based two-factor authentication (2FA) for mobile wallets in blockchain,'' in *Proc. IEEE Int. Conf. Blockchain Distrib. Syst. Secur. (ICBDS)*, Oct. 2023, pp. 1–6.

[25] V. Deshpande, T. Das, H. Badis, and L. George, ''SEBS: A secure element and blockchain stratagem for securing IoT,'' in *Proc. Global Inf. Infrastructure Netw. Symp. (GIIS)*, Dec. 2019, pp. 1–7.

[26] V. Deshpande, L. George, and H. Badis, ''SaFe: A blockchain and secure element based framework for safeguarding smart vehicles,'' in *Proc. 12th IFIP Wireless Mobile Netw. Conf. (WMNC)*, Sep. 2019, pp. 181–188.

[27] Transak. (2024). *What are Smart Accounts? | Transak*. Accessed: Aug. 28, 2024. [Online]. Available: https://transak.com/blog/what-are-smart-accounts

[28] (2024). *Solidity*. Accessed: Aug. 26, 2024. [Online]. Available: https://docs.soliditylang.org/en/v0.8.26/

[29] (2024). *Remix*. Accessed: Aug. 26, 2024. [Online]. Available: https://remix.ethereum.org/

[30] (2024). *Sepolia Resources*. Accessed: Aug. 26, 2024. [Online]. Available: https://sepolia.dev/

[31] (2024). *Sepolia Testnet Explorer*. Accessed: Aug. 26, 2024. [Online]. Available: https://sepolia.etherscan.io/

[32] (2023). *Tackling Network Congestion in Blockchain: Strategies and Solutions*. Accessed: Aug. 26, 2024. [Online]. Available: https://zebpay.com/blog/tackling-network-congestion-blockchain

[33] A. Jain, C. Jain, and K. Krystyniak, ''Blockchain transaction fee and ethereum merge,'' *Finance Res. Lett.*, vol. 58, Dec. 2023, Art. no. 104507. https://www.sciencedirect.com/science/article/pii/S1544612323008796

[34] A. Gangwal, H. R. Gangavalli, and A. Thirupathi, ''A survey of layer-two blockchain protocols,'' *J. Netw. Comput. Appl.*, vol. 209, Jan. 2023, Art. no. 103539. https://www.sciencedirect.com/science/article/pii/S1084804522001801

[35] P. Gauravaram, ''Cryptographic hash functions: cryptanalysis, design and applications,'' Ph.D. dissertation, Cross-Faculty Collaboration, Queensland Univ. Technol., Brisbane, Australia, 2007.

[36] M. Wiki. (2022). *Introduction To Maximal Extractable Value (Mev)*. Accessed: Aug. 26, 2024. [Online]. Available: https://www.mev.wiki/

[37] B. Schneier, *Appl. Cryptography: Protocols, Algorithms, Source Code C*. Hoboken, NJ, USA: Wiley, 2007.

[38] C. Lavor, L. R. U. Manssur, and R. Portugal, ''Grover's algorithm: Quantum database search,'' 2003, *arXiv:quant-ph/0301079*.

[39] J. Han, M. Song, H. Eom, and Y. Son, ''An efficient multi-signature wallet in blockchain using Bloom filter,'' in *Proc. 36th Annu. ACM Symp. Appl. Comput.*, 2021, pp. 273–281.

VARUN DESHPANDE received the B.Tech. degree in electronics and communication engineering from the National Institute of Technology, Bhopal, India, in 2016, and the M.S. and Ph.D. degrees in computer science (blockchain and secure elements) from Université Paris-Est (ESIEE Paris), Champs-sur-Marne, France, in 2017 and 2020, respectively. Since then, he has been active as a Security Researcher and a Consultant for various startups and big companies/organizations. His research interests include blockchain technology, cryptography, and secure elements. He was an awardee of the First Prize at the Flagship Impact Challenge at the Harvard University (HPAIR), Cambridge, MA, USA, in 2020. He was a recipient of the Prestigious France Excellence Charpak Scholarship, in 2016.

HARISH J received the B.E. degree in computer science and engineering from the College of Engineering Guindy (CEG), Chennai, India, in 2023. Currently, he is with Samsung Research, Bengaluru, as a Blockchain Engineer. His research interests include blockchain, computer networks, and computer architecture. He secured a commendable rank of 110 out of half a million test takers in Graduate Aptitude Test in Engineering (GATE), a competitive exam for college graduates, in 2023.

ATHARVA VIJAY KHADE received the B.Tech. degree in computer science and engineering from the National Institute of Technology (NIT), Goa, India, in 2023. Currently, he is with Samsung Research, Bengaluru, India, as a Software Developer Engineer. His research publications include ''Mnemonic Phrase Management and SIM Based Two-Factor Authentication (2FA) for Mobile Wallets in Blockchain'' at the 2023 IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS). His research interests include applied blockchain, applied cryptography, computer networks, and artificial intelligence. He was awarded with the Gold Medal in computer science and engineering and the Director's Gold Medal for overall academic excellence at NIT. He was a recipient of the Second Best Paper Award at the IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS). He also has a passion for coding and has successfully qualified for the prestigious Google Code Jam Coding Competition.