Reactive Key-Loss Protection in Blockchains

Sam Blackshear[1], Konstantinos Chalkias[1], Panagiotis Chatzigiannis[2], Riyaz Faizullabhoy[1], Irakliy Khaburzaniya[1], Eleftherios Kokoris Kogias[1,3], Joshua Lind[1], David Wong[1], and Tim Zakian[1]

[1] Novi Financial / Facebook Research
[2] George Mason University
[3] IST Austria

Abstract. We present a novel approach for blockchain asset owners to reclaim their funds in case of accidental private-key loss or transfer to a mistyped address. Our solution can be deployed upon failure or absence of proactively implemented backup mechanisms, such as secret sharing and cold storage. The main advantages against previous proposals is it does not require any prior action from users and works with both single-key and multi-sig accounts. We achieve this by a 3-phase $Commit() \rightarrow Reveal() \rightarrow Claim() - or - Challenge()$ smart contract that enables accessing funds of addresses for which the spending key is not available. We provide an analysis of the threat and incentive models and formalize the concept of reactive KEy-Loss Protection (KELP).

Keywords: blockchain · smart contracts · key-loss protection · front-running · key management · commitment scheme.

1 Introduction

One of the main criticisms over the usability of cryptographic protocols is that of key-management. This problem is further aggravated in the context of blockchain systems and decentralized finance, where users need to frequently use their keys to sign transactions. To add insult to injury, even users that manage to carefully handle their keys are not fully protected since software bugs or simple human errors can result in sending funds to the wrong address which might not have a known associated key.

A side-effect of poor key-management is losing access to the signing key material, without necessarily this being compromised. That can happen for various reasons, the most common being accidentally deleting a key, forgetting a password and malfunctioned or even discarded hard disks [27] that make private keys unrecoverable. Several solutions exist towards alleviating this problem, however the vast majority, if not all, are *proactive*. The latter means that users should be educated enough to setup a backup mechanism [6] or to secret-share their key with trusted parties [19] or their social-circle [16]. Unfortunately, these solutions

Panagiotis Chatzigiannis did part of this work during an internship at Novi Financial / Facebook Research.

have yet to gain traction, due to the extra steps required, as well as the need to trust someone to hold custody of secret information without abusing them. Additionally, they do not address the second problem of mistyped addresses since such an error is by definition unexpected.

In this paper we address both problems with a single solution, the time-lock based *reactive* KEy-Loss Protection (KELP). On a first glance, reactive recovery looks impossible since by definition the recovering party does not hold a secret to convince the blockchain of being the owner of some locked account. In essence, the rightful owner of that account seems indistinguishable from the adversary.

In KELP, we circumvent this paradox by taking advantage of the fact that there exists information asymmetry, between the account owner and everyone else, on whether a key has actually been lost. Specifically, it is usually the account owner who knows this information first. Based on this assumption, users can claim ownership of any address, but the real owner of such an address has the right within some time to cancel any claim by showing that the secret-key of the address is still available via a proof-of-possession logic, i.e., by signing a new transaction. Thus, a claim can succeed only when the key has indeed been lost, which, the owner of the account is in unique position to know.

A naive protocol, however, is susceptible to two potential attack vectors that should be addressed:

- Front-running: Given the lack of atomicity in blockchains, an adversary can see the pending claim transaction and front-run it by invoking the claim first.
- Random testing: Since a claim transaction does not need approval from the account holder, the adversary can randomly try to claim funds from a large number of accounts.

To address these challenges, KELP employs multiple defensive mechanisms. To protect against front-running attacks, we (a) use a commit-then-reveal scheme and (b) let parties produce *cover traffic* on their accounts [15], by periodically transmitting "fake" lost-key claims, tricking the adversary to front-run them. The cover traffic is then coupled with our protection from random testing, where we automatically require a fee during the claim process which is given to the owner of the challenged address. As a result, an adversary that falsely tries to front-run will be penalized by paying these claim fees, which significantly reduces the incentives and potential rewards of distributed or targeted attacks.

1.1 Background and Related Works

Mistyped addresses. Due to the irreversible nature of blockchains, transactions can neither be cancelled nor reversed once put in a verified block. Addresses are long enough that many accidental typos have been reported in the past [17], where funds have been sent to accounts for which the private key is unknown. When this happens, it is impossible to reclaim these coins and they are essentially burnt forever. Until now, most of the proposed solutions can only offer proactive defenses and include the following:

- **append a checksum** to the address format. For instance, Bitcoin addresses have an embedded checksum code where mistyping a character would result to another valid address with a very slim probability of about 1 to 4.3 billion. However other blockchains, like Ethereum, do not officially apply checksum protection; but there exist related proposals [11]. A 2018 analysis reported that at least over 12,000 Ether have been lost forever due to typos up to block height 5 million [26];
- **use QR codes**, which have an embedded Reed Solomon error correction to demotivate unfriendly textual copy paste typos [20];
- **address creation with a different script**. Unlike other systems, the account-based Diem blockchain natively supports two different transaction types for (a) generating addresses and (b) sending funds to existing accounts only, similarly to the traditional bank account system [1]. This reduces the probability of accidentally sending coins to mistyped addresses, as the account should already exist to receive funds.

Key-loss. There exist several different ways to protect blockchain assets against accidental key loss or compromise attacks, but until now there was not a generic solution for users to regain access to accounts for which the key is unrecoverable. Obviously, if the signing key is compromised (or if there is a protocol bug), attackers can directly transfer assets to accounts they own. Pragmatically, there are not a lot of things we can reactively do for the above, apart from initiating a legal investigation or agreeing as a community on forking the blockchain [14]. However, just losing access to the key is a different scenario where although funds remain in the account, nobody can use them. Our KELP approach is probably the first generic reactive solution to cases where the key has not been compromised yet. Before we explain how it works, we need to enlist the current state of the art in key management and account recovering.

Usually, it is a "wallet" that provides the service for users to generate, store and manage private keys of blockchain addresses. These are mainly categorized as software, hardware, paper and website wallets and they provide different functionalities and security threat assumptions. More demanding custodial wallets have also implemented advanced key management and backup processes to a-priori minimize the risk of key loss. On top of that, modern cryptographic and blockchain protocols emerged, such as hierarchical deterministic key generation and efficient secret sharing protocols. Briefly, a list of current key management and recovery techniques is provided below:

- **cold storage**, where the key resides in a medium that is not connected to the internet, thereby protecting the secret from unauthorized access and other vulnerabilities. Examples include writing down the key on a paper and using a safe or offline Hardware Security Module (HSM). Even these however are susceptible to data or hardware degradation under extreme circumstances, and require a secure (and sometimes expensive) process;
- **custodial services**, where one can delegate key management or backup to a third party that safeguards a sealed copy of the signing key. Although,

this is similar to the traditional method of using safes or notaries, it requires an interactive (and sometimes slow) process to recover the key, while the maintenance cost might be significant [12];
- **distributed key**, where using Shamir's secret sharing [19], concrete threshold elliptic curve signature schemes or secure multiparty computations (MPC), the key material is distributed to multiple nodes [8]. Most of these solutions are interactive per signing or they are complex in terms of implementation, while some of them do not guarantee accountability on who (from these parties) signed. In [12], a few reasonable enhancements are provided to improve the practicality of these schemes;
- **multi-signatures *M-of-N***. Some Blockchains like Bitcoin and Diem natively support the so called *multi-sig* or *M-of-N* addresses, a type of signature that combines multiple unique key signatures into one concatenated statement. Although they are easier to implement and solve the accountability issues of other threshold schemes, these transactions are more expensive as they require at least M signatures to be submitted on-chain. Moreover, in blockchains where this functionality is not natively supported, such as Ethereum, problematic smart contract implementations that simulate this logic have caused loss of funds in the past [10];
- **deterministic key generation**, such as the BIP32 [21] protocol, which requires to store or encrypt a single master seed that derives all of the other account keys;
- **social recovery**, like EIP55 [16] where users select a list of Ethereum addresses, called "guardians", which can authorize the recovery of a private key. A similar approach is supported in Diem blockchain [2] via the rotation capability, where an account can delegate the power of its key rotation to another account or smart contract logic;
- **password-derived keys**, usually via the BIP39 protocol which uses a mnemonic phrase, a group of at least 12 easy to remember words, to serve as a back up to regenerate a private key or master seed. In practice though, many wallets recommend writing down and safely store the phrase, and similarly to common passwords, there have been reports of people forgetting their mnemonics [13];
- **biometrics-recovery**. As a solution to weak memorability of passwords, the work in [5] proposes recovery from secret loss by splitting a biometric-encrypted key to multiple nodes. Apart from the limitation of requiring a trusted third party, the recommendation of using fingerprints is questionable, since they are relatively easy to be reconstructed from high resolution images;
- **vault transactions**, a special type of transaction which enforces its output to be locked for a period of time [25]. During the time lock, the legitimate account owner has the option to abort the vault transaction using a secondary recovery key, typically stored offline, providing some form of private key theft protection. In case the recovery key also gets compromised, this effectively blocks the funds from being spent (as both the attacker and the legitimate owner would use the recovery key, aborting all transactions). Fraud proofs [28] also share a similar concept;

– **paralysis proofs** are based on SGX enclaves and smart contracts and focus on threshold or multi-sig keys only [32]. In short, they enable recovery of funds when enough signers become provably unavailable, which results to not being able to satisfy the threshold. To the best of our knowledge, this is the only existing reactive key recovery solution in the literature, however it focuses on a different problem and only works for *M-of-N* key-structures, but not single keys.

1.2 Our Contributions

We have designed a novel 3-phase time-lock based smart contract that enables key recovery in case of key loss or sending funds to unknown addresses. The major benefits of our approach is that in the best of our knowledge it is the first generic solution that requires *no* prior action from its legitimate holder. Our contract's basic parameters rely on time periods and fees, which need to be carefully selected to mitigate and discourage potential abuse by attackers.

We present our protocol and discuss these considerations in Section 2. Section 3 discusses several considerations towards the contract's practical deployment that need to be taken into account for balancing the contract's usability and its attack surface. Finally, in Section 4 we show potential contract extensions that are applicable to specific blockchains.

2 KELP Protocol

We provide a description of the time-lock KEy-Loss Protection (KELP) logic, a *three-phase smart contract* that allows reclaiming funds from a locked blockchain address after an account spending-key loss. KELP relies on on-chain time locked commitments, similarly to HTLC smart contracts [29] used in atomic swaps (or layer-2 channel opening/closing) to defend against front-running attacks.

We describe the protocol in two parts: first we present the protocol in terms of generic parameters, and then, in a separate section, we discuss potential appropriate choices for these parameters.

2.1 General protocol description

Assuming a hash function h and two time-lock periods t_1 and t_2, a key-loss protection smart contract $\mathsf{KELP} = (\mathsf{KELP.Commit}, \mathsf{KELP.Reveal}, \mathsf{KELP.Claim}, \mathsf{KELP.Challenge})$ is a four function logic defined as follows:

KELP.Commit

$\mathsf{KELP.Commit}(address_c, address_r, nonce) \rightarrow com, fee_1$ is a transaction which can be executed by any user who wants to claim ownership of $address_c$ for which they believe the spending key has been lost or forgotten. It outputs a

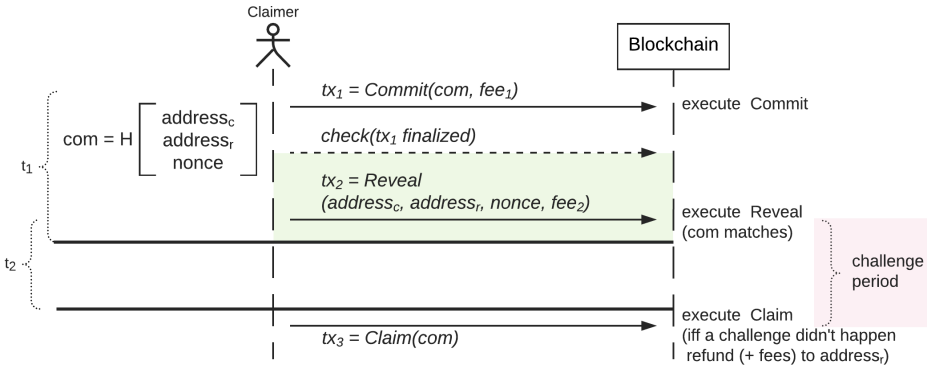

Fig. 1. KELP $Commit() \rightarrow Reveal() \rightarrow Claim()$ flow.

commitment value $com = h(address_c||address_r||nonce)$, which is recorded in the blockchain[4].

The commitment indicates that, in case of a successful claim, the ownership of $address_c$ will be transferred to the owner of $address_r$. The mechanism of this transfer is blockchain-specific. For example, in some blockchains, funds of $address_c$ can be simply moved to $address_r$. In other blockchains, $address_r$ may specify a new spending key to be associated with $address_c$. In cases when $address_r$ cannot be set to arbitrary values (e.g. when $address_r$ must already exist on chain), an extra *nonce* should be included in the commitment to make guessing of $address_c$ and $address_r$ impractical via brute-force attacks.

For the commitment to be recorded in the blockchain, an extra fee_1 must be included with the Commit transaction. Unlike a regular transaction fee, fee_1 is not paid to miners/validators but to the user who successfully executes one of the Claim or Challenge transactions. The purpose of this fee is to discourage random-testing attacks where malicious actors try to issue claims against a large number of accounts in hopes of randomly finding one for which the key has indeed been lost. The magnitude of fee_1 can be set using a variety of strategies, some of which we discuss in section 2.2.

KELP.Reveal

KELP.Reveal$(com) \rightarrow address_c, address_r, nonce, fee_2$ is a transaction executed by the user who previously executed a Commit transaction for the specified commitment. The purpose of this transaction is to reveal $address_c$ in the clear so that, in cases when the account's key has not been lost, the legitimate account owner has an opportunity to challenge the claim.

[4] Note that *com* can be implemented with any reasonable and efficient commitment scheme i.e., via HMACs, but typically a regular hash function is already available as part of the underlying blockchain's instruction set.

It is important that the user revealing the claim should wait until their Commit transaction gets finalized to avoid front-running. Specifically, if the Commit transaction has not been finalized, and since $address_c$ is now exposed, someone else (e.g. miner, block producer) could claim its ownership by executing Commit and Reveal transactions of their own ahead of the original Commit.

Before accepting a revealed claim, a KELP contract checks that:

- $h(address_c||address_r||nonce)$ is equal to the com value form the corresponding Commit transaction;
- $timeof(\mathsf{Reveal}) - timeof(\mathsf{Commit}) < t_1$, where $timeof()$ is a function which returns time (e.g. as block height) of when a transaction was included into the blockchain;
- Sufficiently large fee_2 was included with the transaction. The appropriate magnitude of fee_2 can be set using a variety of strategies, some of which we discuss in section 2.2.
- There are no active claims against $address_c$, or if there are, the new claim can override the currently active claim. We say that claim A can override claim B if $timeof(\mathsf{Commit}_A) < timeof(\mathsf{Commit}_B)$.

More informally, for a claim to be accepted by the contract, the Reveal transaction must be executed within t_1 period of the corresponding Commit, it must include a large enough fee_2, and it must be able to override other active claims against the account, if any. This way, only one revealed claim is possible against an account at a given point in time.

If a revealed claim is accepted by the contract, both fee_1 and fee_2 are immediately transferred to the account at $address_c$, however, account ownership is not transferred until successful execution of Claim transaction.

KELP.Claim

$\mathsf{KELP.Claim}(address_c, address_r, nonce) \rightarrow com$ is a transaction which transfers ownership of $address_c$ to the owner of $address_r$. This transaction is accepted by the contract only if:

- com commitment corresponds to a currently active claim against $address_c$;
- $timeof(\mathsf{Claim}) - timeof(\mathsf{Reveal}) > t_2$. This ensures that the period allotted for the account owner to challenge the claim has elapsed. For practical considerations of choosing the value for t_2 please refer to the following section, but in general it should be on the order of months or even years.

Upon successful execution of a Claim transaction, the ownership of $address_c$ is transferred to the owner of $address_r$. As mentioned earlier, the exact mechanism of ownership transfer is blockchain-specific. In some blockchains this may include transferring balances between the accounts involved, while in other blockchains, it might update spending keys associated with $address_c$.

It is important to note that the cost of a successful claim is negligible. This is because both fee_1 and fee_2 are added to the balance of the account at $address_c$

upon successful execution of the Reveal transaction. Thus, a successful Claim execution has the effect of refunding these fees to the owner of $address_r$.

KELP.Challenge

KELP.Challenge$(address_c, signature_c) \rightarrow cancel_{claim}$ is a transaction executed by the owner of $address_c$ to protect the account from malicious claims. This transaction must include a proof that the spending key associated with the account has not been lost - e.g. a $signature_c$ generated with the account's spending key. However, the exact implementation of such transaction is blockchain-specific. For example, in some blockchains, any regular transaction executed from $address_c$ may suffice, while in other blockchains, the Challenge may need to be a special type of transaction.

Executing a Challenge immediately cancels any potential claim (up to that moment) against the account, and, in effect, transfers fee_1 and fee_2 of the unsuccessful claim to the owner of $address_c$. Assuming the value $fee_1 + fee_2$ is significant, this mechanism makes unsuccessful claims very costly, and discourages users from submitting claims unless they are confident that an account's spending key has indeed been lost or forgotten.

All in all, one can compare KELP to legal challenges, where any body can mount a legal case against somebody else but a claim without merit will be quickly rejected and the costs of doing so should be prohibitive.

2.2 Protocol parameters

This section analyzes the four parameters involved in the KELP protocol:

1. t_1 - time period during which a committed claim must be revealed.
2. t_2 - time period after which a revealed claim can be executed.
3. fee_1 - extra fee included with Commit transaction.
4. fee_2 - extra fee included with Reveal transaction.

Optimal values for these parameters are highly blockchain-specific and depend on such properties as consensus algorithm (e.g. finality times), blockchain structure (e.g. UTXO vs. account-based), degree of centralization etc. We, therefore, leave in-context analysis of these parameters to future research, and provide only general guidelines and considerations as to how they can be chosen.

Parameter t_1 The purpose of this parameter is to set an upper bound on the time a committed claim can remain hidden. To prevent front-running attacks, t_1 should be no shorter than the time it takes for a transaction to be finalized. However, setting t_1 to higher values has an additional benefit as it provides more opportunity for Reveal transactions to be included into the blockchain. This can protect against censorship attacks where a powerful adversary can try to delay block inclusion of Reveal transactions in hopes that t_1 expires and they will be able to execute Commit and Reveal transactions of their own.

At the same time, setting t_1 to very large values (or even to infinity) is not desirable because it would enable malicious actors to issue a large number of Commit transactions against many (or even most) accounts. These commitments will linger in the blockchain, bloating the state and potentially imposing significant burden on node operators. More importantly, a large number of lingering claim commitments makes it risky for a legitimate users to initiate account recovery via the KELP protocol as it increases the probability that there exists a claim commitment against a user's account which predates the Commit transaction which the user could issue.

Given the above considerations, setting t_1 to a period between several hours and several days may be appropriate in most cases.

Parameter t_2 The purpose of this parameter is to provide sufficient time for a legitimate account owner to challenge an adversarial claim. This period should be long enough for the account owner to:

1. Detect a revealed claim against their account;
2. Execute Challenge transaction against the active claim.

Both of these may require significant time. Detecting a claim may be complicated by a number of factors including physical unavailability of the account owner for prolong periods of time. Similarly, executing Challenge transactions may be delayed by the spending key being stored in cold storage or protected by a complicated multi-sig recovery scheme. Moreover, a powerful adversary may try to censor the network and prevent inclusion of Challenge transaction into the blockchain until t_2 elapses. Long t_2 periods would reduce feasibility of such attacks as maintaining complete censorship control over a decentralized network for prolonged periods of time is increasingly difficult.

Thus, depending on the specifics of the underlying blockchain, it may be appropriate to set t_2 to a period of several months or even years.

Parameter fee_1 The purpose of this parameter is to increase the cost of opportunistic Commit transactions and discourage malicious actors from issuing a large number of claim commitments. In this way, this parameter is similar to relatively short t_1 periods. However, whereas short t_1 periods force such actors to periodically renew their opportunistic claims, relatively large fee_1 increases the cost of every such claim.

An important consideration for fee_1 parameter is that it should not reveal any information about the account against which Commit transaction is executed. Otherwise, malicious actors may de-anonymize the account for which the spending key has been lost and attempt to front-run the Commit transaction of the legitimate account owner. Thus, we are not making fee_1 proportional to the balance of the account in question.

However, it is desirable to make fee_1 large enough to make random-testing attacks impractical for the vast majority of accounts. One strategy to achieve this could be to make fee_1 proportional to the balance of an average account

on the network. For example, if the average account holds a balance of $1,000, fee_1 could be set to $100. This will make opportunistic Commit transactions against most accounts impractical. At the same time, since fee_1 is returned to the account owner upon successful execution of Claim transaction, such a high fee has a negligible impact on the cost of a legitimate claim.

Parameter fee_2 The purpose of this parameter is to increase the cost of opportunistic Reveal transactions and discourage malicious actors from making illegitimate claims. It is similar to fee_1, however, while the magnitude of fee_1 may provide sufficient protection for most accounts, it is disproportionately low as compared to balances of high-value accounts. Thus, a malicious actor may choose to periodically execute Commit transactions against a relatively small set of high-value accounts (and pay the associated fees), in hopes that if a spending key of one such account is lost, they will be able to override the legitimate Reveal transaction and recoup their "investment".

To mitigate this attack, we need to make the cost of an unsuccessful claim unbearably high. However, since imposing outsized fees at Commit time will make the protocol unsuitable for most users, we impose additional fees at Reveal time, when the account becomes publicly known. This allows us to make fee_2 proportional to the balance of the account in question. Such a proportion should be significant, but its exact value could vary based on the specifics of the underlying network, and thus, we leave it to future economic incentives research and analysis.

Assuming fee_2 represents a significant portion of the claimed account's balance (e.g. 10%), to reveal their claim, malicious actors would need to be very confident that their Reveal transaction cannot be overridden by someone else's Reveal with an earlier corresponding Commit. Avoiding such situations would require a very high degree of coordination between all potential adversaries, and may not be feasible in practice.

Moreover, a legitimate account owner can exacerbate this uncertainty by periodically, at random intervals, issuing "fake" Commit and Reveal transactions against their own account. To potential attackers, this would look like an initiation of KELP protocol, implying that the account's owner lost its spending key. Thus, if there is an attacker with a preemptive Commit against the account, they will execute a Reveal transaction of their own to override the Reveal transaction of the legitimate owner. However, since the key is not lost, the legitimate owner can immediately challenge the claims and receive $fee_1 + fee_2$ as a reward.

It is important to note that the cost of a "fake" claim is negligible to the legitimate account owner. This follows from the fact that both fee_1 and fee_2 are returned to the account regardless of whether the claim is overridden by the attacker's claim or not. However, in cases when the original claim is overridden by the attacker, the legitimate account owner will receive an additional reward in the form of fee_1 and fee_2 from the attacker's claim. This may provide a sufficient incentive for owners of high-value accounts to periodically engage in this "bluffing" behavior.

2.3 Considerations for practical deployments

Besides selecting values for the four parameters described above, a real-world deployment of KELP protocol will need to address a number of issues which we briefly discuss in this section.

Optionality and defaults. It is important to understand that the KELP protocol does modify the trust model somewhat (see Section 3 for additional discussion). Therefore, we do not recommend it as a mandatory feature for all accounts of a blockchain. Instead, it could be an optional feature which users can freely enable or disable on per-account basis. The exact mechanism of how to do so is blockchain-specific. In some blockchains this could be done via an account-level flag, in others, this would require a new address format. We, therefore, leave a more detailed discussion of the exact mechanism to future in-context research.

Assuming KELP is adapted as an optional account-level feature, the question arises as to whether it should be enabled by default. A conservative approach would be to have it disabled by default, and require users to explicitly opt-in to use the feature. However, this approach has two notable drawbacks:

1. Any funds sent to non-existent addresses become unrecoverable as non-existent addresses would have KELP disabled by default.
2. It could lead to a relatively small number of KELP-enabled accounts, thereby reducing the anonymity set of *Commit* transactions, and in general, making random-testing attacks more viable.

Both of these drawbacks can be mitigated by a number of blockchain-specific strategies. For example, in some blockchains, sending funds to a non-existent address is impossible by design [1]. Similarly, attractiveness of random-testing attacks can be reduced by higher fee_1 and more aggressive "bluffing" strategies as described in the previous section. We, therefore, make no recommendation on whether KELP should be enabled by default, and leave this question to future blockchain-specific analysis. Finally it is made clear that in the case where KELP is optionally enabled, this contradicts to our original statement of offering a completely reactive mechanism. However, even in such a scenario, the actions required are more straightforward than existing proactive practices, mainly because there is no requirement of complex cryptography protocols, storing and/or delegating secrets.

Wallet support Before KELP-enabled accounts can be supported by a blockchain, care must be taken to ensure that there is enough support in the entire ecosystem for this feature. Specifically, wallets which desire to support KELP-enabled accounts should be able to provide the following functionality:

1. Implement a proactive claim notification system and detect revealed claims submitted against accounts managed by the wallet;

2. Issue challenge transactions against detected claims;[5]
3. Issue *cover transactions* by faking KELP commits periodically;
4. Execute full key recovery protocol for a user-provided address.

While implementing these may not be overly difficult in software wallets, many custodial services implement their logic in specialized equipment such as HSMs or with multi-signature keys, which may prove non-trivial to update in practice. Thus, to enable KELP on a blockchain in a backward-compatible way, KELP feature should be either disabled by default, or it should be introduced in a way which would make it impossible for legacy wallets to generate KELP-enabled accounts.

KELP.Challenge transaction Rather than having the KELP.Challenge transaction be a distinct transaction type, it is desirable to have any regular transaction issued from an account to have the effect of KELP.Challenge transaction, in that it would cancel any revealed claim against the account. In blockchains where this can be implemented, legacy wallets would still be able to defend themselves against attackers attempting to falsely claim their keys, as long as they can monitor the chain and detect adversarial Reveal transactions.

2.4 Reactive Recovery and Synchrony Assumptions

KELP is a construction that can be seen as a simulation of a special layer-2 construction. The two abstract parties transacting are (a) the owner of the account who lost access to the key and (b) the claimer. The goal of the protocol is to securely communicate the knowledge that the *secret* key of (a) has been lost, similarly to opening and closing a layer-2 channel on-chain, which can be challenged. Obviously, in the happy scenario of KELP, the same user has the role of both parties in this hypothetical channel.

Modelling KELP as such, we can investigate the impossibility result introduced by Zyamatin et al. [31], which shows that recovery needs either a time-synchrony assumption (as we do in KELP via periods t_1 and t_2) or some abstract, potentially distributed, trusted third party (as done in proactive recovery mechanisms [19, 25]). To the best of our knowledge, the only channel construction that does not assume synchrony employs threshold security assumptions [4], a classic proactive recovery mechanism, which implies that KELP's reactive recovery approach and synchrony assumptions go hand-in-hand.

[5] More proactive ways to issue challenges include for example an *intelligent* wallet that learns its owner's behaviour (frequency of use), so that it can distinguish between the state in which a user simply hasn't logged in for a while, and the state in which they have misplaced their key, automatically issuing challenges in the former state.

3 Trust model and attack vectors

In this section we discuss the underlying trust model and potential attack vectors against KELP. It is important to separate potential threats into two broad categories:

1. Attacks against user accounts which are enabled by the KELP protocol. In such cases, we assume that a user has *not* lost their keys, but an attacker is trying to exploit some aspect of the KELP protocol to steal their funds.
2. Attacks against successful execution of the KELP protocol. For these attacks, we assume that a user has lost their key, and the attacker is trying to interfere with the user's ability to reclaim their funds via the KELP protocol.

3.1 New attacks against user accounts

Long-range censorship attack. An adversary which can censor arbitrary transactions on the network for prolonged periods may be able to steal funds from any KELP-enabled account. Such an adversary can issue Commit and Reveal transactions against a target account, and then censor any Challenge transactions from the legitimate account owner. Once t_2 period elapses, the adversary can issue a Claim transaction, thereby completing the attack and taking control of the user's funds.

Therefore, our trust assumption for KELP protocol is that the ability to censor transactions for prolonged periods of time is not feasible in the underlying blockchain. The exact duration of such a period is defined by the t_2 parameter, which can be made excessively large (e.g., months or years). We would argue that if there exists a party which can censor arbitrary transactions on the network for a period of several years, then the network is not secure in the first place.

Key destruction attack. An adversary which can destroy a user's key (or delay the user's ability to use their key for a long period of time), can attack the user's account as follows: the adversary destroys the key and immediately executes a Commit transaction against the account with a now destroyed key. Then, once t_1 period elapses, the attacker executes a Reveal transaction, and eventually, a Claim transaction. Since the key is lost, the legitimate account owner has no way of challenging the claim. A potential defence against this threat is presented in Section 4, via the so called *dead man's key*. Also, such adversaries might be disincentivised by the fact that they do not have knowledge on whether the key owner makes use of offline backups, possibly handwritten.

3.2 Attacks against fund recovery

Short-range censorship attack. An adversary which can slow down inclusion of transactions into the blockchain, may attempt to do the following: upon seeing a Reveal transaction, the adversary may delay its inclusion into the blockchain,

and instead execute their own Commit transaction against $address_c$ specified in the original Reveal transaction. If time t_1 expires before the original Reveal transaction can be included into the blockchain, the adversary will issue their own Reveal transaction followed by a Claim transaction, thereby stealing the funds.

Potential mitigating strategies against this attack are sufficiently long t_1 periods, and periodic *cover transaction* "faking" KELP commits and reveals.

Random testing attack. Malicious actors can proactively issue Commit transactions against a large number of accounts in hopes that owners of some of these accounts will eventually lose their keys and try to reclaim their funds via the KELP protocol. Then, upon seeing a Reveal transaction from one such account, the attacker can issue a Reveal transaction of their own against the same account. Assuming that the attacker's proactive Commit transaction predates a Commit transaction of the legitimate account owner, the attacker will override the original Reveal, and once t_2 period elapses, will execute a Claim transaction, thereby receiving control of the funds.

To further increase the effectiveness of this attack, an attacker might issue proactive commits for accounts using heuristic techinques, e.g. focus on high-value accounts which have been dormant for a while.

Potential mitigating strategies against this attack include:

- Relatively short t_1 periods, which would force the attacker to periodically renew their preemptive Commits thereby decreasing the probability that the attacker's Commit would predate the Commit of a legitimate user (or a commit of another malicious actor).
- Relatively high fee_1 and fee_2 to make issuing preemptive Commits against a large number of accounts costly, and to make the cost of an unsuccessful claim unbearably high.
- Periodic *cover transaction* from legitimate account owners "faking" KELP commits and reveals against their accounts, thereby increasing uncertainty of whether a Reveal transaction can be indicative of a key loss.

Side-channel attack. Revealing a committer's identity via spending account and/or transactor's IP disclosure will result in a reduced list of potential addresses that the Commit refers to. For instance, the account address that submitted a Commit transaction might leak information about the committed address, enabling bruteforce front-running attacks to be more targeted. Such information can be extracted by analyzing the transaction graph.

There exist tools for hiding sender's identity, here is a short list:

- ZCash [9] provides sender/receiver identity hiding, but unfortunately it does not support arbitrary smart contracts yet;
- Monero [30] does partial identity mixing by creating smaller anonymity sets, but again no custom smart contracts are supported;

- CoinJoin [22] type of mixers could work as intermediate services, but still one needs to trust the mixing third party entity;
- similarly to CoinJoin, third party services can offer such a functionality via non-disclosure agreements and business deals. For instance, associations might undertake this role in permissioned systems like Algorand [24], Corda [18], Hyperledger [3] and Diem [2];
- indistinguishable regular and commit transactions, as discussed in the Extensions section;
- using TOR and/or VPNs could help on obfuscating the committer's IP address;
- using pre-purchased anonymous tokens as shown in Section 4;
- using a brand new address, and ensuring this is totally independent from previous transactions made by the same user.

4 Extensions

We already mentioned *cover transactions* and using a regular (transfer coins) script to simulate a Challenge as potential features of KELP. Here we present several useful extensions of the generic KELP protocol, some of them being applicable to specific blockchains only.

Indistinguishable transactions. As already mentioned, by de-anonymizing a committer's address, the set of address candidates who lost the key can be reduced by transaction graph analysis [23]. One way to circumvent this issue is by making regular and commit transactions indistinguishable. This will make every transaction in the system looking like a potential commit. A simple approach is to expect all of the transactions to carry a 32-byte metadata value, which works as a commitment (i.e., HMAC) to a referred script. Transactions not requiring any commitment can just attach a random nonce, so that any transaction becomes a commit candidate and thus increasing the committers anonymity set. This is not only applicable to KELP, but a generic obfuscation pattern for other *commit - reveal* schemes where hiding committer's identity is important.

Anonymous commit tokens. Another option to avoid revealing committer's identity is by using pre-purchased anonymous tokens [7]. In such a model, selected entities can issue tokens which work as anonymous cashier checks. Then, instead of spending from a UTXO or account address, one just submits a purchased token which carries some value and an embedded gas/fee payment logic.

Dead man's key. As a defense against key destruction attacks, one could specify a secondary key for the account - a *dead man's key*. This key can have weaker storage restrictions and can be distributed more freely as its only role is to issue Challenge transactions. To execute a successful attack, the adversary would need to destroy both the signing key and all copies of this secondary key.

Key Rotation vs. Claiming Funds. Interestingly, Diem blockchain offers a feature where the address and spending key are decoupled. We can gain advantage of this property and upon a successful Claim, the committer gets a permission to rotate the account's key instead of transferring its funds to a different address. Among the others, this allows for fully controlling the account's

state and not just the funds, which is ideal in cases where this address continues to be advertised in QR codes or shared between friends and businesses, and thus might keep receiving assets after the Claim.

Committing to sequence-ID. Both Ethereum and Diem use a sequence-ID under an account state, which increments on each transaction as a defense against replay attacks. It also works as an indicator on how many transactions an account submitted until the most current block. If KELP was implemented for these blockchains, we should include the current sequence-ID of the account who lost the key as part of the commitment hash. That would allow for an easier implementation logic to check if any transaction occurred after Commit, which would imply that the key was not lost. Thus, regular transactions would invalidate any active Commit, which makes KELP compatible with already up and running custodial wallets that haven't implement the challenge logic yet.

Customizable KELP parameters. In Section 2.3 we discussed if KELP should be an optional feature or the default logic. Going a step further, we could allow custom values for all of the four KELP parameters (t_1, t_2, fee_1, fee_2) at account level. That offers extra flexibility to custodial or unhosted users to control how KELP works for their account in particular. For instance, custodial wallets who can monitor the chain at real-time, might set high fees for important accounts and smaller time periods for faster recovery.

Acknowledgements. The authors would like to thank all anonymous reviewers of FC21 WTSC workshop for comments and suggestions that greatly improved the quality of this paper.

References

1. Diem documentation - accounts (2020), https://developers.diem.com/docs/core/accounts/#creating-accounts
2. Amsden, Z., *et al*: The Libra blockchain. Calibra corp p. 29 (2019)
3. Androulaki, E., *et al*: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018. pp. 30:1–30:15 (2018)
4. Avarikioti, G., Kokoris-Kogias, E., Wattenhofer, R.: Brick: Asynchronous state channels. CoRR **abs/1905.11360** (2019), http://arxiv.org/abs/1905.11360
5. Aydar, M., Cetin, S.C., Ayvaz, S., Aygun, B.: Private key encryption and recovery in blockchain. arXiv preprint arXiv:1907.04156 (2019)
6. Baldimtsi, F., Camenisch, J., Hanzlik, L., Krenn, S., Lehmann, A., Neven, G.: Recovering lost device-bound credentials. In: Applied Cryptography and Network Security. pp. 307–327. Springer International Publishing, Cham (2015)
7. Baldimtsi, F., Lysyanskaya, A.: Anonymous credentials light. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 1087–1098 (2013)
8. Battagliola, M., Longo, R., Meneghetti, A., Sala, M.: A provably-unforgeable threshold eddsa with an offline recovery party (2020)
9. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE

Symposium on Security and Privacy. pp. 459–474. IEEE Computer Society Press (May 2014). https://doi.org/10.1109/SP.2014.36
10. Brenner, M.: How I snatched 153,037 ETH after a bad tinder date (2017), https://eprint.iacr.org/2019/1128
11. Buterin, V., Van de Sande, A.: EIP-55: Mixed-case checksum address encoding (2016), https://eips.ethereum.org/EIPS/eip-55
12. Di Nicola, V., Longo, R., Mazzone, F., Russo, G.: Resilient custody of crypto-assets, and threshold multisignatures. Mathematics **8**(10), 1773 (2020)
13. Duncan1949: Lost passphrase for extra account on trezor (2015), https://www.reddit.com/r/TREZOR/comments/33i03g/lost_passphrase_for_extra_account_on_trezor
14. Falkon, S.: The story of the DAO - its history and consequences (2017), https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee
15. Grube, T., Thummerer, M., Daubert, J., Mühlhäuser, M.: Cover traffic: A trade of anonymity and efficiency. In: International Workshop on Security and Trust Management. pp. 213–223. Springer (2017)
16. Guilherme Schmidt, R., Mota, M., Buterin, V., naxe: Secret multisig recovery (2019), https://gitlab.com/status-im/docs/EIPs/blob/secret-multisig-recovery/EIPS/eip-2429.md
17. Haig, S.: Eth community discuss DAO for reversing funds lost to wrong addresses (2020), https://cointelegraph.com/news/eth-community-discuss-dao-for-reversing-funds-lost-to-wrong-addresses
18. Hearn, M.: Corda: A distributed ledger https://www.r3.com/wp-content/uploads/2019/08/corda-technical-whitepaper-August-29-2019.pdf
19. Jarecki, S., Kiayias, A., Krawczyk, H., Xu, J.: Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In: 2016 IEEE European Symposium on Security and Privacy (EuroS P). pp. 276–291 (2016). https://doi.org/10.1109/EuroSP.2016.30
20. Khan, A.G., Zahid, A.H., Hussain, M., Riaz, U.: Security of cryptocurrency using hardware wallet and qr code. In: 2019 International Conference on Innovative Computing (ICIC). pp. 1–10. IEEE (2019)
21. Khovratovich, D., Law, J.: BIP32-Ed25519: Hierarchical deterministic keys over a non-linear keyspace. In: 2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 27–31. IEEE (2017)
22. Maxwell, G.: Coinjoin: Bitcoin privacy for the real world. bitcointalk. org (2013)
23. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of bitcoins: characterizing payments among men with no names. In: Proceedings of the 2013 conference on Internet measurement conference. pp. 127–140 (2013)
24. Micali, S.: ALGORAND: the efficient and democratic ledger. CoRR **abs/1607.01341** (2016), http://arxiv.org/abs/1607.01341
25. Möser, M., Eyal, I., Sirer, E.G.: Bitcoin covenants. In: Financial Cryptography and Data Security - FC 2016 International Workshops. LNCS, vol. 9604, pp. 126–141. Springer (2016)
26. Pfeffer, J.: Over 12,000 ether are lost forever due to typos (2018), https://media.consensys.net/over-12-000-ether-are-lost-forever-due-to-typos-f6ccc35432f8
27. Pollock, D.: Infamous discarded hard drive holding 7,500 bitcoins would be worth $80 million today (2017), https://cointelegraph.com/news/infamous-discarded-hard-drive-holding-7500-bitcoins-would-be-worth-80-million-today

28. Ruffing, T., Kate, A., Schröder, D.: Liar, liar, coins on fire! penalizing equivocation by loss of bitcoins. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 219–230. CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813686, https://doi.org/10.1145/2810103.2813686
29. TierNolan: Bitcoin wiki: Atomic cross-chain trading (2013), https://en.bitcoin.it/wiki/Atomic_swap
30. Van Saberhagen, N.: Cryptonote v 2.0 (2013), https://cryptonote.org/whitepaper.pdf
31. Zamyatin, A., Al-Bassam, M., Zindros, D., Kokoris-Kogias, E., Moreno-Sanchez, P., Kiayias, A., Knottenbelt, W.J.: Sok: Communication across distributed ledgers. Cryptology ePrint Archive, Report 2019/1128 (2019), https://eprint.iacr.org/2019/1128
32. Zhang, F., Daian, P., Kaptchuk, G., Bentov, I., Miers, I., Juels, A.: Paralysis proofs: Secure access-structure updates for cryptocurrencies and more. Cryptology ePrint Archive, Report 2018/096 (2018), https://eprint.iacr.org/2018/096

A KELP implementation in Diem blockchain

In this appendix, we present an implementation of the KELP protocol for the Diem Framework. The code is mostly a straightforward, but has a few Diem-specific features that we will explain here:

Automatic challenges using sequence numbers. Like many other blockchains, Diem accounts have sequence numbers that are incremented each time a transaction is sent from the account. Our implementation timestamps each reveal on $address_c$ with the sequence number of $address_c$. In the code for `claim`, we check that no transactions have been sent from $address_c$ since the reveal. This ensures that any transaction sent from $address_c$ is implicitly a challenge.

Reclaiming entire accounts with `KeyRotationCapability` Diem accounts support key rotation. Each account `a` has a unique `KeyRotationCapability { a: address }` resource whose holder has the permission to rotate the authentication key for `a`. An account that opts in to KELP recovery must give its `KeyRotationCapability` to the KELP resource. KELP then uses this resource to rotate the key for `a` in the logic for `claim`. This allows the claiming party to completely regain control of the account, not just its funds.

Using the `signer` type to avoid some uses of $address_r$ The Move language has a type called `signer`[6] that represents an authenticated user with a specific address. Our implementation leverages this type to omit some uses of $address_r$ from the protocol. For example: we don't need to include $address_r$ in the Commit message because we use `signer` to ensure that a commit and reveal transaction originate from the same address.

[6] https://developers.diem.com/docs/move/move-signer/

```
module KELP {
    use 0x1::BCS;
    use 0x1::Errors;
    use 0x1::Diem::{Self, Diem};
    use 0x1::DiemAccount::{Self, KeyRotationCapability};
    use 0x1::DiemTimestamp;
    use 0x1::Hash;
    use 0x1::Signer;
    use 0x1::Vector;
    use 0x1::XUS::XUS;

    /// Published under an account that supports KELP recovery
    resource struct KELP {
        /// Key rotation capability for the account that has enabled KELP recovery
        rotate_cap: KeyRotationCapability,
        /// Size of the commit fee
        fee1_amount: u64,
        /// Size of the reveal fee
        fee2_amount: u64,
        /// pooled fees from commit and reveal transactions
        fees: Diem<XUS>,
        /// Length of challenge period between commit and reveal
        t1: u64,
        /// Length of challenge period between reveal and claim
        t2: u64,
    }

    /// Published under an account that has performed a Commit operation to initiate recovery
    resource struct Commit {
        /// sha3(KELP address | nonce)
        commit: vector<u8>,
        /// Locked fee to be deposited upon reveal
        fee1: Diem<XUS>,
        /// Time when the commit occurred
        commit_time: u64,
    }

    /// Published under an account that has performed a successful Reveal operation
    resource struct Reveal {
        /// Time when the reveal occurred
        reveal_time: u64,
        /// Sequence number of the KELP account at the time of the reveal
        reveal_seq: u64,
    }

    const EBAD_REVEAL: u64 = 0;
    const EBAD_CHALLENGE: u64 = 1;
    const EBAD_CLAIM: u64 = 2;
    const EREVEAL_TOO_SOON: u64 = 3;
    const ECLAIM_TOO_SOON: u64 = 4;

    /// Enable KELP recovery for `account_r`
    public fun initialize(
        account_r: &signer, fee1_amount: u64, fee2_amount: u64, t1: u64, t2: u64
    ) {
        let rotate_cap = DiemAccount::extract_key_rotation_capability(account_r);
        let fees = Diem::zero<XUS>();
        move_to(
            account_r,
            KELP { rotate_cap, fee1_amount, fee2_amount, fees, t1, t2 }
        )
    }

    /// Commit to a future claim on a KELP account
    public fun commit(account_r: &signer, commit: vector<u8>, fee1: Diem<XUS>) {
        let commit_time = DiemTimestamp::now_seconds();
        move_to(account_r, Commit { commit, fee1, commit_time })
    }


    /// Reveal a previous claim on a KELP account
    public fun reveal(
        account_r: &signer, address_c: address, nonce: vector<u8>, fee2: Diem<XUS>
    ) acquires Commit, KELP {
```

```
        let address_r = Signer::address_of(account_r);
        let Commit { commit, fee1, commit_time } = move_from<Commit>(address_r);
        let message = BCS::to_bytes(&address_c);
        Vector::append<u8>(&mut message, nonce);
        assert(Hash::sha3_256(message) == commit, Errors::invalid_argument(EBAD_REVEAL));

        let kelp = borrow_global_mut<KELP>(address_c);
        let reveal_time = DiemTimestamp::now_seconds();
        assert(reveal_time - commit_time > kelp.t1, Errors::limit_exceeded(EREVEAL_TOO_SOON));

        let reveal_seq = DiemAccount::sequence_number(address_c);
        move_to(account_r, Reveal { reveal_time, reveal_seq });

        // sweep the commit and reveal fees into the KELP resource
        Diem::deposit(&mut kelp.fees, fee1);
        Diem::deposit(&mut kelp.fees, fee2)
    }

    /// Finalize a claim on a KELP account
    public fun claim(
        account_r: &signer, new_key: vector<u8>, address_c: address
    ): Diem<XUS> acquires Reveal, KELP {
        let address_r = Signer::address_of(account_r);
        let Reveal { reveal_time, reveal_seq } = move_from<Reveal>(address_r);
        let kelp = borrow_global_mut<KELP>(address_c);
        let claim_time = DiemTimestamp::now_seconds();
        // ensure the reveal was not invalidated by a subsequent "challenge" (i.e., a
             transaction sent
        // from address_c)
        assert(reveal_seq < DiemAccount::sequence_number(address_c),
             Errors::limit_exceeded(EBAD_CLAIM));
        // ensure the reveal happened afer the conclusion of the challenge period
        assert(claim_time - reveal_time > kelp.t2, Errors::limit_exceeded(ECLAIM_TOO_SOON));

        // successful claim. allower claimer to reclaim account by rotating key
        DiemAccount::rotate_authentication_key(&kelp.rotate_cap, new_key);

        // return fees to the claimer
        Diem::withdraw_all(&mut kelp.fees)
    }

    /// Collect all commit/reveal fees in the KELP resource under `account`. This can be called
    /// by the owner of the KELP resource at any time. Note: a transaction that calls
    /// `collect_fees` will also (implicitly) issue a challenge by incrementing `account_c`'s
    /// sequence number.
    public fun collect_fees(account_c: &signer): Diem<XUS> acquires KELP {
        let address_c = Signer::address_of(account_c);
        let kelp = borrow_global_mut<KELP>(address_c);
        // return fees to the challenger
        Diem::withdraw_all(&mut kelp.fees)
    }
}
```