Knox: A New Secure Coin To Protect Against Private Key Compromise

Nathan Ostrowski, Andres Montoya[1]

[1]*CPS 590, Duke University*

I. ABSTRACT

On current blockchain systems, a user's private key is their single point of failure. However, this private key is also necessary for every transaction a user wishes to make. For users prioritizing security, we believe this paradox of a private key providing the root of function and the root of security is a fundamental design flaw. Because the user must keep their private key available for every transaction, an attacker may more easily steal it by socially engineering the user or hacking the user's machine. In practice, this accounts for the majority of cryptocurrency stolen each year [1]. We propose a system called Knox to protect against this threat model. Our system balances safety with liquidity, sacrificing some transactional immediacy for a secure solution that does not rely on the integrity and confidentiality of any applications or servers. Furthermore, while our system primarily aims to protect against private key compromise, due to the addition of a delay and cancellable transactions, it also protects against account draining scams whereby a user inadvertantly allows a malicious application access to their wallet.

II. INTRODUCTION

Over the past decade, terms like blockchain, bitcoin, and NFT have come to occupy a frequent place in the public discourse. An increasing number of people around the world now hold some cryptocurrency, and big firms in investment and banking have joined the fray. As blockchain technology gains increasing adoption for regular consumers and corporations alike, securing blockchain assets is more critical than ever. Knox addresses one of the biggest issues regarding cryptocurrency today, loss of funds through private key compromise.

We view our primary contributions as (1) designing and specifying a private key compromise-resilient system that does not rely on a single source of trust, (2) implementing this system within the Ethereum ecosystem using Solidity. We detail our preliminary findings on the efficiency of this system in Evaluation.

It is important to note that implementing Knox on Solidity was our greatest challenge. Translating theory to code was quite difficult and the limitations of Solidity morphed Knox's design into what it is today. Nevertheless, Knox accomplished its goal to show that on-chain security of funds associated with addresses can be accomplished even in the event of private key compromise.

Despite our enthusiasm for the security of this design, we recognize a few significant limitations of our current implementation. As gas prices to execute code within the Ethereum ecosystem remain prohibitively expensive, our multi-step incurs a significant cost on the user. Though we have explored implementation on a few other L1 chains, we are not presently aware of whether or not other such implementations are possible. We also recognize that some actions on our system should not reasonably be completed by a user, so we delegate some responsibilities that ought to be implemented by a wallet application and backend serving our system. We discuss findings on cost, implementation difficulties, the infeasability of building on other L1 chains, and wallet-delegated actions for improving user experience in Limitations.

III. BACKGROUND AND RELATED WORK

Many cryptocurrency holders manage the funds in their accounts via a wallet—an interface that allows users to more easily transact on a given blockchain. Current cryptocurrency wallets sometimes employ certain security measures to keep a user's account secure and ensure that funds remain accessible to them. These wallet-based security measures include multi-signature wallets (i.e. ”vaults”), wallet-generated phrases, and multi-factor authentication. By using any of these techniques with a wallet, the user places trust in the wallet software—whether it be an app on their phone, an extension on their web browser, or software on their desktop at home—as well as the wallet software owner's networking infrastructure.

If a user wishes to keep their funds secure, a user may also use a personal hardware wallet (also known as a ”cold wallet”) which stores private keys offline. Arculus cold wallets are one example implementing a three-factor authentication scheme which includes some biometric data, a 6-digit pin, and a physical card. This physical card communicates with a wallet application, allowing one to use their private key to trade cryptocurrency [2]. This solution still trusts the Arculus wallet application and potentially the operating system to securely ferry sensitive information and avoid leaking keys to an observing third-party.

Coinbase Vault is a service from Coinbase that claims to store most of a user's assets in cold storage. A user can log into their Coinbase account to set up a ”Vault”, optionally specifying email addresses and phone numbers to work as multi-factor authentication, as well as an anti-phishing phrase to verify that a communication they receive actually originated from Coinbase [3]. Under this model, a user must place their trust in Coinbase to securely hold their private keys. A user must also trust Coinbase servers, the operating system of the device they use to communicate with Coinbase, and potentially their Coinbase wallet application depending on the permissions they have set up. In addition to their private keys, the user must trust Coinbase not to leak any of their personal identifiable information provided for multi-factor authentication. The authors of this proposal believe that the high level of trust a user must place in Coinbase and their managed Vaults is contrary to the spirit of Decentralized Finance for which users may seek out cryptocurrency.

Gnosis Safe is a multi-signature wallet software built on a smart contract model where users can secure their funds via a "Safe" for later retrieval, specifying some amount of "owner" keys. When wishing to move funds out of this safe for a transaction, a certain threshold of owners must approve the transaction. Thus, Gnosis Safe operates a sort of blockchain-enforced multi-factor authentication, targeted at individual users prioritizing security or business accounts where no single party should be able to move funds out of the wallet. Gnosis Safe's core smart contract component has the benefit of being formally verified according to the developers [4]. Unfortunately, Gnosis Safe's model is also expensive. Even with smart contract proxies, users can incur substantial fees to set up and use a Safe.

Dodis et al. consider a system that defends against private key compromise by continually reissuing new "secret keys" where each is only valid for some interval i. At initialization, a user registers some Public Key (PK) and some master Secret Key (SK_*) and stores the latter on a trusted, secure device. The PK used to encrypt messages does not change, but encodes the interval during which it was encrypted. When a user wishes to decrypt a message, the user must request an SK_i from the device containing SK_*. Therefore, if an attacker gains SK_i but not SK_*, the attacker cannot read messages from the next interval forward. The authors also consider a system where there is no such trusted, secure device—and the user must manually generate PK and SK, publish PK, and derive SK_* and SK_0. The user may send SK_* to an untrusted device, store SK_0 personally, then when reading, request a partial key SK'_i from the untrusted device and use SK_{i-1} and SK'_i to compute the actual key SK_i [5]. This personal generation approach appealingly reduces the parties a user must trust, but remains arguably impractical if a user is expected to store this master key and perform this computation without assistance from any machine. Furthermore, this timed-key approach still relies on a single master Secret Key SK_* which an attacker may steal or socially engineer through common methods.

Xu et al. propose and formally evaluate a system formed around Key Compromise-Resilient Signatures (KCRS). Similar to the system proposed by Dodis et al., there exists a master public/private key pair (pk_M, sk_M) which, when forming a block to place on the chain, one supplies to *SPKGen* and *SSKGen* functions with a random number to receive a randomly selected signing key pair (pk_S, sk_S). The user signs their message using the signing key pair and a function *SigGen*, which a miner can verify matches the signing public key and message using a function *SigVerify*. Importantly, the miner may also verify using the *SKDetect* function that the signing public key pk_S was generated using the master key pair (pk_M, sk_M). Because the signing key pair is only used for a single message, and the random number supplied to *SSKGen* is kept secret by the user, even if an attacker finds sk_S, further messages signed using this key will be rejected by miners on the chain [6]. The authors of this proposal do not see how this method reduces the use of the master private key, as the user must supply their master private key sk_M every time they wish to generate a new key pair. Rather, this system seems to defend against only the situation where the attacker compromises a signing private key sk_S, which the user uses only once and then discards.

Pal et al. proposed an innovative key management solution to protect against private key compromise. Specifically, they describe the security vulnerabilities relating to using block chain within IoT systems and proposed a Group Key Management (GKM) system to alleviate the burden. This ensures the security of payloads containing crypto keys within network communications for blockchain. GKM architecture divides a blockchain network into several groups and levels in a tree-like structure. Each group has its own group key (GK) and only nodes within that group have permission to access data within it. In code the framework is represented by (i, j, k) where i is the level, j is the position of parent group in the upper layer, and k is the position of the group in the current layer under the parent group [7]. This system of grouping many nodes into smaller branching subgroups with distinct keys relating to their position has appealing properties for security. Any node in a parent subgroup group may compute the keys associated with a child subgroup through a one-way function, so the parent subgroup keys remain securely hidden from the child subgroup. Like the systems proposed by Dodis et al. and Xu et al., this system similarly relies on certain "master" points of failure. In this case, the security of the system is governed by the confidentiality of keys held by nodes a level higher than any particular node subgroup.

IV. KNOX ARCHITECTURE

A. Overview

At a high level, Knox is a fungible token (i.e. coin) that maintains security for a user's funds even when the user's private key is stolen. Knox implements three high-level, fundamental changes to conventional blockchain systems to achieve this goal:

- Users specify security codes to associate with their account upon initialization
- Transactions are delayed and cancellable
- A user may exchange their public/private key pair using their security codes

To walk through Knox, consider a user, Jane, who has invested her savings in ether and wants to keep them protected. In today's world, Jane might use a wallet on her phone or computer to store her private key and provide an abstraction for actions using ether. The problem is: if Jane inadvertently allows access via a malicious source, or if that wallet app is hacked, or Jane's phone OS is hacked, or an attacker steals Jane's device, the attacker can use the private key to generate a signed transaction, broadcast it to miners, and have those miners transfer all of Jane's savings to their malicious account.

The following subsections detail key pieces of the Knox architecture and how they protect Jane from this threat.

B. Security Codes and Private Keys

Knox requires Jane to specify n security codes and a transaction delay time to associate with her public key. These security codes are just private keys, with their public keys appended to the delay time, signed, and broadcast to miners to place on the chain—where they are immutable and easily retrievable in the future. Jane can store these security codes in a hardware wallet, in a QR code, under her couch, with a parent, or wherever else she pleases. She can even lose some of these security codes as time goes by. As long as she retains 1 more valid security code than any attacker, she will remain in control of the funds in her account.

C. Delayed Transactions

In Knox, similar to how trusted bank transactions move from pending to processed, all transactions reference an aforementioned delay specified by the user in the initial, immutable security settings stored on the blockchain. When Jane wants to transfer funds to someone else, she can initiate a transfer request which is sent to miners and stored on the chain. After the delay period has passed, Jane can initiate an execute request to fully transfer funds to the receiving account. Of course, this delay model depends on a shared, trusted notion of time. As each block carries a timestamp, and each block is mined at roughly equal intervals, this allows time based computations to be executed. Every time a transaction is attempted for the first time, the block time of the transaction is recorded. Then, once the initialized delay has passed, the next transaction call between the two addresses can execute the transaction if the difference between the current block time and the previous block time is greater than the delay.

This delay is a crucial design limitation of Knox for everyday transactions, but it is also a crucial source of security. If an attacker were to compromise Jane's wallet and steal her private key (or otherwise trick her into granting access to a malicious application), the attacker may initiate a transfer request, but may not execute the transaction until the delay period passes. Meanwhile, Jane has time to see that a transaction has been initiated from her account and allows her the ability to broadcast a cancel request to miners on the network (signed with her private key). If the attacker requests to execute the transaction, verification will fail, as a cancellation block for that transaction with a valid signature has been added to the chain. We discuss optimal methods for ensuring that Jane sees malicious activity in Limitations.

D. Transferring to a Secure Address

Seeing that an attacker has compromised her private key and initiated a malicious transfer (that she has cancelled), Jane needs to transfer ownership from the compromised private key to one of her retained security codes. To do this, Jane initiates a "ReKey" request, which executes a Knox ReKey function using the secure address (Pkn) signed by the security code (Skn) she chooses. This trusted function (1) verifies that the signature is valid, (2) verifies that the Pkn is associated with Pk0 and stored on the blockchain (described in initialization), (3) verifies that the message is signed by the Skn associated with Pkn, (4) verifies that the user has not executed a previous ReKey request to Pkn. If all these criteria are met, the code immediately transfers all funds to Pkn. This renders the previous private key that the attacker has stolen completely useless.

V. THREAT MODEL

Knox's primary goal is to protect users from the most common blockchain attack vector: private key compromise. As a consequence of our delayed transaction model, our system also protects against common "account drain" scams whereby a user grants a malicious application access to their wallet. We consider an adversary A_S whose goal is to compromise the user's private key and extract the funds associated with the user. We also consider an adversary A_E whose goal is to extort the user, seeking complete control over a user's account (even without the ability to extract funds) and demanding some ransom to relinquish control.

We consider Routing, Sybil, and 51 percent attacks which an adversary may also use to extract the funds associated with a user to be out of scope. We assume that the system is secure at the point of setting secure codes; cases where A_S or A_E has compromised a user's computer prior to setup are out of scope. Our threat model assumes that the user will monitor their account at regular intervals, noticing (and cancelling) transactions within a set period of time, then optionally transferring ownership to a new private key (i.e. one of their security codes) via a ReKey request. Methods to reduce these assumptions and improve the user experience via features delegated to the wallet are explored in Limitations.

The strength of the system depends on the number of security codes a user creates at initialization and the length of the delay period specified. This flexibility allows for a wide variety of risk tolerances and liquidity preferences. As long as the user retains at least one more unused security code than any attacker, the user may initiate a ReKey request to transfer ownership to this safe state. At this point, if the user feels too many of their security codes have been compromised, the user may even transfer ownership to a new account with new associated security codes.

In our specification (see: Specification), we've set minimum and maximum values for some of these inputs. While these minimum and maximum values are somewhat arbitrary, we believe they represent best practice for the security guarantees our system provides.

Our proposed system acknowledges the following attack vectors:

Fee-siphoning: where a threat actor steals a user's private key and continually initiates transactions, draining a user's account via excessive gas fees. To prevent such a scenario, future research may consider implementing rate-limiting for transfer code or implementing additional protections enabled by the limited-use keys described by Xu et al. and Dodis et al. Under our currently proposed system, a user may transfer ownership quickly to a new secure private key to prevent fee-siphoning from continuing.

Total compromise: this represents the worst-case scenario when A_S successfully obtains access to the private key and *all* security codes before the user issues a call to rekey. In this case, A_S obtains the same privileges as the user. Unless A_S is able to hide these codes from the user, both A_S and the user will have the same privileges, and may engage in a competing cycle of transfer requests and cancellations, draining the user's account.

VI. SPECIFICATION

In Table 1, we specify the Knox API. Further in this section, we formally identify the meaning and validity of inputs to each of these functions. An implementation adhering to this specification must take care to check each of these input conditions before proceeding with function logic.

Universal

Input	**Meaning**
I_{Pk}	Invoker public key
I_A	Invoker address

Input	**Validity**
I_{Pk}	Let L_{Pk} be a set of valid digits or characters for public keys in a given implementation. I_{Pk} is valid if $I_{Pk} \in L_{Pk}^n$ where I_{Pk} is of n characters s.t. $\lvert I_{Pk} \rvert = 128$.
I_A	Let L_A be a set of valid digits or characters for addresses in a given implementation. I_A is valid if $I_A \in L_A^n$ where I_A is of n characters s.t. $\lvert I_A \rvert = 40$.

transfer(R, T)

Input	**Meaning**
R	Intended receiver address
T	Intended number of fungible tokens to send

Input	**Validity**
R	$R \in L_A^n$, $\lvert R \rvert = 40$
T	$T \in \mathbb{N}$ s.t. $T \leq$ total currency supply

cancel(R, T)

Input	**Meaning**
R	Intended receiver address of staged transaction to cancel
T	Intended number of fungible tokens of staged transaction to cancel

Input	**Validity**
R	$R \in L_A^n$, $\lvert R \rvert = 40$
T	$T \in \mathbb{N}$ s.t. $T \leq$ total currency supply

setSecurityCodesAndDelay$(P_k$***[], ***$D)$

Input	**Meaning**
$(P_{k0}, P_{k1}, ...P_{kn})$	Sequence of public keys
D	Delay (in milliseconds)

Input	**Validity**
$(P_{k0}, P_{k1}, ...P_{kn})$	If $(P_{k0}, P_{k1}, ...P_{kn})$ forms the set $\{P_{k0}, P_{k1}, ...P_{kn}\}$ s.t. $5 < \lvert\{P_{k0}, P_{k1}, ...P_{kn}\}\rvert \leq 50$ where for each P_{kn}, $P_{kn} \in L_{Pk}^n$ and $\lvert P_{kn} \rvert = 128$
D	$6 * 10^4 < D \leq 1.21 * 10^9$.

transfer(R, T ***)***	First invocation ”stages” the transfer. The second invocation attempts to execute the staged transfer matching the same parameters. If the delay has not passed since the transfer call was staged, or if ***cancel(*** R, T ***)*** was invoked, the transfer fails.
cancel(R, T ***)***	Cancels a transfer matching the specified receiver address and specified quantity.
setSecurityCodesAndDelay(P_k ***[], D)***	Initializes the public keys corresponding to security codes and the delay before a transfer may successfully execute. This function may only be called once.
reKey(F ***)***	Must be invoked from the secure address to which the user wishes to reKey, signed by the security code. This function verifies the mapping between the supplied address and the invoking secure address, immediately transferring funds if related.

TABLE I. Knox API. For Input breakdown, see Specification

reKey(F ***)***

Input	**Meaning**
F	Address from which to transfer ownership of funds

Input	**Validity**
F	$F \in L_A^n$, $\lvert F \rvert = 40$

VII. IMPLEMENTATION

When starting on our implementation, we naturally had to decide on a cryptocurrency system to fork. Gas fees were a particular point of interest for us, as our initial system relied heavily on pairs of smart contract (though we had planned to decrease the costliness through transaction proxies). We strongly considered Ethereum and Solana, and went with Ethereum for its robust development community and flexibility in adjusting core functionality. Most importantly, Ethereum allowed us to directly modify its low level functions, . Furthermore, it prevents any direct transactions from occurring even if an attacker has a private key. This also ensures that Knox adheres to the ERC-20 Standard.

In order to make the Knox integratible with existing infrastructure, we needed to follow ERC-20 standards. ERC-20 is a technical standard that all Ethereum-based smart contracts and tokens must follow. Our token implementation keeps with the Knox API specified in Table 1. To enable these functions, we store four additional arrays in the blockchain state:

- staged - mapping of addresses to receiving addresses containing the time they were staged.
- delays - mapping of addresses to their specified delays.
- secureAddresses - mapping of addresses to an array of secure addresses. These exhaust the values that may be supplied in a reKey call.
- hasSetSecurityCodes - mapping of addresses to a boolean specifying whether they have set up security codes

VIII. EVALUATION

Changes in gas fees are used to analyze the efficiency of the Knox system in comparison to the cryptocurrency it was built off, Ethereum. Thus, the evaluation of Knox was done through Remix, an online IDE for Solidity. Its debug editor recorded the cost for each computation as it was executed. The costs for Ethereum and Knox were then compared to derive the appropriate benchmarks.

A. Benchmarks

The computational cost displayed in Remix does not show the market cost of executing a computation. Instead it shows the cost incurred to the miner to execute the computation. From these base costs, a relative estimate is for cost of Knox's transactions using the current market cost for Ethereum. The table below shows how computationally expensive each action is within Ethereum and Knox.

Costs of Usage (In Ether)

Computation	**Ethereum**	**Knox**
Transaction	510^{-14}	110^{-13}
Initialization	0	910^{-14}

As seen in the table, each individual transfer call is **3%** more expensive in comparison to Ethereum. As each transaction requires 2 transfer calls (one for staging and one for executing), transactions between accounts are **206%** more expensive as a whole. As of time of writing, Ethereum transactions cost $18.45 per transaction. This means that transactions under Knox would theoretically cost $38.01 per transaction. It is important to note that transaction prices are determined by market forces generated by miners. Thus, this only provides an estimate for the cost to transact funds based on the current market forces affecting Ethereum. Additionally, there is a small, one-time transaction fee associated with initializing the secret codes and delay for each individual account. These additional costs, however, are fixed as they does not scale with the size of the currency transacted. This presents an appropriate trade off for the increased security provided by Knox as the cryptocurrency is meant to facilitate large, safe transfers of funds rather than day-to-day transactions.

IX. LIMITATIONS

A. Practical Implementation Goals

An ideal user of our system would generate keys offline in a secure manner, store those keys as long strings of text by hand, and run a node on a trusted server environment to continuously check for changes to the chain state where a transaction has been initiated. However, we recognize that these goals are not practical. While traditional wallets retain security risks, they also abstract away complexity from the user. We believe for our system to be practically implemented, some of this complexity ought to be abstracted away by a wallet application and a dedicated server.

First and foremost, a wallet application ought to provide an abstraction for security code generation. We recognize that one reason for existing wallet popularity is that users do not want to manually write down a long address—or indeed many long addresses—and therefore delegate responsibility to a wallet that they access with a password or physical key. A more robust wallet application may allow users to use multiple physical keys instead of randomly generating security codes. A more basic wallet implementation may instead translate those security codes to digestible seed phrases that a user may more easily copy. In both cases, the wallet application ought to automatically format the setup request and send it to miners, storing only the public keys for later convenience.

Our threat model assumes that the system is secure at security code setup. As this is the only occasion where a wallet application abstracting complexity would need to interact with security codes, this does not change our threat model. Nevertheless, we recognize that it provides a larger attack surface for a would-be attacker at initialization.

Another appealing abstraction to provide for the user is state update. In theory, the user should either (1) open the application and monitor state changes to the chain at least once in every delay interval, or (2) continually running a trusted server acting as a node to process state changes and update the user. Neither of these situations may be ideal for every user, so we believe that in practice, a user should have the option to offload some of this responsibility to a trusted backend. This trusted backend should store a pairing of users who have opted-in between some device-specific identifier and their public key. Whenever the backend sees a state change involving that public key, the backend ought to notify the application using the device-specific identifier. Of course, this option requires the user to trust that the backend will function properly at least for one full delay period, but represents a more practical approach for everyday consumers.

B. Other L1 Chain Considerations

While we chose Ethereum to initially implement Knox for its wide adoption and robust developer support, it is not without its challenges. Chief among these is transaction cost. Ethereum's current consensus model incurs high gas fees from miners who must put in an inordinate amount of effort to verify each block. Because our system requires users to both initiate and then execute transactions, and requires potentially many state changes when initiating security codes, our implementation further increases these gas fees. Further discussion of these fees can be found in Evaluation.

Thus, we considered other L1 chains with high adoption yet lower transaction fees where we might be able to add complexity to the transaction mechanism without adversely increasing gas fees and affecting user experience. We had initially considered and researched implementation on the Solana ecosystem, but hit a wall during implementation. At present, after speaking with developers on the Solana team, we believe there is no way to change the transaction logic for a particular token on Solana. This means that there is no way to enforce the Knox system for verifiers on a Solana token. A potentially creative solution to this problem may include creating a trusted app to act as a proxy, but at present, we are unaware of practices to restrict transaction and minting to this trusted application alone.

Future research directions ought to consider whether implementation is possible on other popular L1 chains with low transaction fees, such as Algorand and Cardano.

X. CONCLUSION

Knox is a fungible token that protects against private key compromise and wallet access scams. Knox achieves this security guarantee by requiring the user to set up security codes and a delay period for transactions. In the event that the user's original private key is compromised, or a malicious application initiates an unintended transaction, the user may cancel this unintended transaction and optionally transfer their funds to a secure address corresponding to one of their initial security codes. Knox is currently implemented within the Ethereum ecosystem as an ERC20-compliant token. Unfortunately, because Knox requires users to both stage and execute transactions, Knox incurs a 206% overhead on transactions. While this overhead not a limiting factor on some L1 chain currencies, on Ethereum, it restricts the use-case of Knox to operations where security is prioritized. The delay provided by Knox further cements this use-case. However, we believe that Knox strongly fits this niche, providing a novel, decentralized way to securely transfer and store funds. As detailed in Practical Implementation Goals, a consumer-ready implementation of Knox may also include a wallet to abstract the complexity of the system and a backend system to communicate activity anonymously with users. Future work towards Knox should be directed towards converting its functionality into a smart contract to make its security practices more accessible. For instance, if the Knox standard is enforced upon addresses within Ethereum through a smart contract, addresses in these ecosystems would obtain the same protection mechanisms given to those in Knox. Nevertheless, regardless of how far Knox spreads, it will ideally bring more attention towards creating decentralized, on-chain cryptocurrency security solutions for users.

XI. CODE

Our current implementation can be found at: `https://github.com/KnoxCoin/KnoxCoin/`

[1] J. Su, “Hackers stole over 4 billion from crypto crimes in 2019 so far, up from 1.7 billion in all of 2018.,” *Forbes Magazine*, 2019.

[2] A. by CompoSecure, “Arculus: Secure your crypto, secure your future,” 2021.

[3] Coinbase, “Vaults — coinbase help,” 2021.

[4] Gnosis, “Gnosis safe documentation,” 2021.

[5] Y. Dodis, J. Katz, S. Xu, and M. Yung, “Key-insulated public key cryptosystems,” in *Advances in Cryptology — EUROCRYPT 2002* (L. R. Knudsen, ed.), (Berlin, Heidelberg), pp. 65–82, Springer Berlin Heidelberg, 2002.

[6] L. Xu, L. Chen, Z. Gao, X. Fan, K. Doan, S. Xu, and W. Shi, “Kcrs: A blockchain-based key compromise resilient signature system,” in *Blockchain and Trustworthy Systems* (Z. Zheng, H.-N. Dai, M. Tang, and X. Chen, eds.), (Singapore), pp. 226–239, Springer Singapore, 2020.

[7] O. Pal, B. Alam, V. Thakur, and S. Singh, “Key management for blockchain technology,” *ICT Express*, vol. 7, no. 1, pp. 76–80, 2021.