Paralysis Proofs: Secure Dynamic Access Structures for Cryptocurrency Custody and More

Fan Zhang, Cornell University
Philip Daian, Cornell Tech
Iddo Bentov, Cornell University
Ian Miers, Cornell Tech
Ari Juels, Cornell Tech

ABSTRACT

The growing adoption of digital assets—including but not limited to cryptocurrencies, tokens, and even identities—calls for secure and robust digital assets custody. A common way to distribute the ownership of a digital asset is (M, N)-threshold access structures. However, traditional access structures leave users with a painful choice. Setting $M = N$ seems attractive as it offers maximum resistance to share compromise, but it also causes maximum brittleness: A single lost share renders the asset permanently frozen, inducing *paralysis*. Lowering M improves availability, but degrades security.

In this paper, we introduce techniques that address this impasse by making general cryptographic access structures *dynamic*. The core idea is what we call Paralysis Proofs, evidence that players or shares are *provably unavailable*. Using Paralysis Proofs, we show how to construct a *Dynamic Access Structure System* (DASS), which can securely and flexibly update target access structures *without a trusted third party*. We present DASS constructions that combine a trust anchor (a trusted execution environment or smart contract) with a censorship-resistant channel in the form of a blockchain. We offer a formal framework for specifying DASS policies, and show how to achieve critical security and usability properties (safety, liveness, and paralysis-freeness) in a DASS.

To illustrate the wide range of applications, we present three use cases of DASSes for improving digital asset custody: a multi-signature scheme that can "downgrade" the threshold should players become unavailable; a hybrid scheme where the centralized custodian can't refuse service; and a smart-contract-based scheme that supports recovery from unexpected bugs.

CCS CONCEPTS

• **Security and privacy → Key management**; **Access control**.

KEYWORDS

Blockchain, Trusted hardware, Access control, Digital Asset Custody, Wallet

1 INTRODUCTION

Nearly all key management and access control systems have an "always / never" requirement [54]: they should always provide availability when validly authorized, but never when not. Balancing availability against the potential for misuse is a pervasive challenge though. Consider the task of securing a single private key. Replicate the key broadly across geography, machine architectures, and custodians, and the risk of key compromise increases. Store the key in a single location and the probability of loss increases, as do barriers to timely access.

Generally, this challenging trade-off is navigated by distributing control across a small number of parties by means of an *access structure* [33]. An access structure is a policy determining which players can control a resource. (M, N)-*access structures*, as in, e.g., Shamir secret sharing [55], are a popular choice. They allow any M out of N players to access a target resource, e.g., to sign a cryptocurrency transaction. Varying M and N provides flexibility on the defensibility vs. availability spectrum. Unfortunately this flexibility is often still inadequate.

For example, in the setting of a cryptocurrency custodian, despite the availability of (M, N)-multisig wallets (which require M of N players to sign), loss and theft of cryptocurrency have been rampant for years [13, 23, 47, 48, 61]. Over 980,000 Bitcoin (currently worth about $6.4 billion) [52] have been stolen from exchanges alone. An estimated 4,000,000 Bitcoin (currently worth around $26 billion), have vanished forever due to lost keys [53].

One of the primary concerns looming over the use of (M, N)-multisig schemes is what we call a "shareholders' dilemma": setting $M = N$ is not only desirable for its maximum resilience to key compromise and collusion, but it could also be the only option to meet certain business requirements, e.g., when N people want a joint-account with equal ownership. However, setting $M = N$ also causes maximum brittleness: a single lost key share renders the asset permanently frozen.

Fundamentally, the problem with traditional (M, N)-access structures is that they are *static*. M cannot be lowered or N raised when funds become unavailable, so insecure choices are often made at system setup.

1.1 Our work

In this paper, we show how to achieve *both* good availability *and* good security, and to do so *without* trusted third parties (TTPs). Specifically, we show how to achieve a better trade-off than any static access structure alone can do. We accomplish this by means of access structures that are *dynamic*.

We introduce the idea of a *Dynamic Access Structure System* (DASS), which allows for secure *conditional downgrading*, e.g., changing $(3, 3)$-multisig to a $(2, 3)$-multisig if a player becomes unavailable. Consequently, the full security of a $(3, 3)$-multisig is available in the general case. A $(2, 3)$-multisig is instantiated in the critical case that funds would otherwise be lost. (If Alice has lost her key, it's better to run the risk of Bob and Carol absconding with funds than to lose funds with certainty forever.) In other words, a DASS

AFT '19, October 21–23, 2019, Zurich, Switzerland
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6732-5/19/10…$15.00
https://doi.org/10.1145/3318041.3355459

implements a policy of access-structure migrations (e.g., downgrades). Fig. 1 illustrates the idea by showing a three-player DASS that can withstand two key losses.

DASSes do not completely avoid security/availability trade-offs. They *do* however introduce new and desirable points on the security/availability spectrum, and thus new security management options for cryptocurrencies and a wide range of other cryptographic systems.

One major technical challenge in our work is ensuring that the downgrading of access structures only happens when a player is truly unavailable. Otherwise, players can cheat by simulating the disappearance of a live player. The tool we introduce for this purpose are strong proofs of player (or key-share) unavailability called *Paralysis Proofs*.

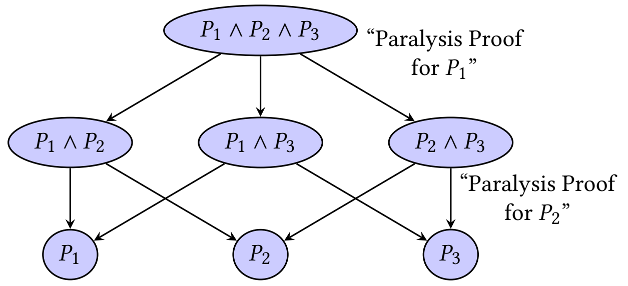

$P_1 \wedge P_2 \wedge P_3$
"Paralysis Proof for P_1"
$P_1 \wedge P_2$
$P_1 \wedge P_3$
$P_2 \wedge P_3$
"Paralysis Proof for P_2"
P_1
P_2
P_3

Figure 1: An example DASS with three players. Nodes represent access structures, edges migration conditions.

1.2 Approach overview and challenges

Proving that a player is available is easy: just have her sign a fresh message. Secure Paralysis Proofs, though, require that a player be able to signal liveness when other players collude and potentially try to block her network access. This is challenging even with a trusted third party (TTP).

Our constructions rely on a *censorship-resistant stateful channel* to detect and record the fact of an unavailable player. Such a channel has two properties: *censorship-resistance*, meaning players can freely send and receive messages with a bounded delay, and *statefulness*, meaning messages sent through the channel are recorded and can be retrieved later. These properties are strictly stronger than those of anonymous channels (e.g., Tor [59]), which lack statefulness.

We leverage public blockchains as censorship-resistant stateful channels in our work. The basic idea is that if a player, e.g., Alice, disappears, other players can post a challenge to her *on chain*. If Alice tries to post a response within Δ blocks (for some suitable Δ), she can do so with high probability even in the face of powerful network adversaries and her response will be world readable. Thanks to these properties, lack of response from Alice within Δ blocks of a challenge constitutes a Paralysis Proof.

Building a DASS from Paralysis Proofs is easy given a TTP, which can simply manage all users' keys. Again, we aim critically to avoid TTPs. Our DASS schemes thus reply on two technologies that server as trust anchors by emulating TTPs: trusted execution environment (TEEs) and smart contracts.

However, naïve combination of Paralysis Proofs with TEEs or smart contracts doesn't yield a DASS. Verifying a Paralysis Proof requires the verifier to have an up-to-date view of the blockchain in order to determine whether a given Δ is appropriate. Keeping TEEs in sync with a blockchain is undesirable and problematic, though, as TEEs are fundamentally stateless and unaware of accurate time. Attempts to work around these limitations result in larger attack surface and/or extra assumptions. As we will show shortly, our protocol does not require the TEE to have an up-to-date view of the blockchain. In fact, it does not require any view of the blockchain.

Smart contracts, on the other hand, doesn't suffer from the same limitation. Indeed, for assets controllable by smart contracts, implementing DASSes for them is straightforward. But for other assets smart contracts can't be employed in general because they can't manage secret keys. Moreover, smart contracts are known to suffer unavailability induced by accidental or unexpected bugs. We explore a novel scheme that allows secure recovery of smart-contract-controlled assets in face of software-induced paralysis.

To analyze and prove the security of DASSes, we provide a formal framework for specifying dynamic access structure policies (DASPs) and introduce new security definitions. Additionally, we also show how to overcome technical obstacles in securing DASSes in practice. One is to avoid placing a TEE on the critical path for ordinary transactions and risking service failures by, e.g., storing players' keys only in the TEE. We also consider minimizing the impact of side-channel attacks on TEEs, e.g., [62].

1.3 Improved digital asset custody with DASS

We consider three use cases of Paralysis Proofs that illustrate our techniques' wide range of application to improving digital asset custody and beyond:

Tolerate cryptocurrency key loss. We present a (M, N)-multisig wallet that can downgrade the threshold if players become unavailable. We report on implementations for both script-based cryptocurrencies (e.g., Bitcoin) and smart-contract-based ones (e.g., Ethereum).

Tolerate custodian failures. Centralized custodians, e.g., Coinbase [46], help users manage keys, but are single points of failure. We present a DASS that migrates control of funds to their owners if and only if a custodian fails or refuses service. The "only if" part guarantees that as long as the centralized custodian is operational, leaking a user's key won't breach the user's account, which is desirable.

Tolerate software-induced paralysis. Smart contract bugs can lead not just to theft, but paralyzed funds, as in the \$150 million lost to the infamous (second) Parity Multisig Wallet bug [49]. We propose a continuous-integration framework that regularly applies a test suite to a smart contract to validate its correct functioning, including liveness of funds. Critical error conditions such as paralysis trigger an "escape hatch" [39], failover logic that refunds or moves a smart contract's assets.

1.4 Contributions

In summary, our main contributions are as follows:

Paralysis Proofs. We introduce Paralysis Proofs, and show how to achieve them using censorship-resistant stateful channels such as blockchains. We also introduce Dynamic Access Structure Systems (DASSes), which offer security vs. availability trade-offs unachievable with conventional, static access structures.

Formal definitions and framework. We formally define key properties for a DASS (*liveness*, *safety*) and its underlying migration policy (*privilege-preserving*, *paralysis-free*) (Section 3). We present an ideal functionality that formally specifies security properties required for a broad range of applications (Section 4).

Improved digital asset custody with DASS. We present three example applications of DASS to protect digital asset custody from: cryptocurrency key loss (Section 4), cryptocurrency custody failures (Section 5), and smart contract failures (Section 6).

TEE compromise. We explore alternative DASS designs that provide resilience to TEE compromise, such as through side-channel attacks demonstrated against SGX (Section 4.4).

Implementation. We present implementations in Ethereum and Bitcoin for the first application, using smart contracts and TEEs (Intel SGX, in particular) respectively. To illustrate the limitations of pure blockchain approaches for Bitcoin, we also explore script-based schemes in Appendix B.

2 BACKGROUND

In this section we provide some basic background on Trusted Execution Environments, Bitcoin, and smart contracts.

2.1 Trusted Execution Environments and SGX

A Trusted Execution Environment (TEE) is an execution environment that provides confidentiality and integrity for applications running on potentially malicious hosts.

Intel Software Guard Extensions (SGX) [10, 31, 41] is a CPU-based TEE implementation available in recent Intel CPUs. SGX allows processes to execute in an *enclave*, an environment that enforces application confidentiality and integrity against even a malicious operating system and some classes of hardware attacks. SGX also enables applications to emit third-party verifiable *attestations* to their origin and outputs.

Limitations of TEEs. Although powerful, TEEs have fundamental security limitations. TEEs generally don't guarantee availability. Moreover, TEEs depend upon a potentially malicious operating system for I/O. A consequence is that TEEs cannot provide trusted sources of time. In the case of SGX, although a trusted relative timer is available in an off-CPU component, the communication between enclaves and the timer can be delayed by the malicious OS [1]. Thus SGX enclaves can only ascertain a lower bound on the elapsed time. Server-grade Intel CPUs offer no support for timers at the time of writing. Moreover, unless with additional protection mechanisms (e.g., [40]), SGX doesn't have trustworthy monotonic counters and therefore is susceptible to state rollback attack. These security limitations make Paralysis Proofs challenging even with TEE.

2.2 Bitcoin

Bitcoin is a decentralized electronic cash scheme in which transactions moving funds are recorded in an append-only log, a *blockchain*. Rather than storing funds in accounts whose balance is altered by transactions, Bitcoin uses transactions themselves to record both ownership and balance. Transactions consist of *inputs* and *outputs*. An output consists of an amount and a `script_pubkey` that specifies how that amount can be spent. Inputs specify the transaction output which is the source of the funds and include a `script_sig` showing authorization to use the funds. Thus transactions spend the outputs of previous transactions. Unconsumed outputs are known as Unspent Transaction Outputs (UTXOs). One can check if an output has been spent by seeing if it is in the set of UTXOs. By requiring that outputs can only be spent once and that the amount of money included in a transaction's inputs is at least as much as its outputs, Bitcoin enforces the invariants of a monetary system and prevents forgery.

Bitcoin Script. When creating an output, users can specify an access control policy by embedding a *script*—called `script_pubkey`—in the output. To spend an output, one must provide a witness—called `script_sig`—such that running the script with the provided witness outputs `true`. For example, a typical `script_pubkey` specifies the keys that must sign any transaction spending that output. This may be a single key or an arbitrary combination of keys, e.g., $(\mathsf{pk}_1 \wedge \mathsf{pk}_2) \vee \mathsf{pk}_3$. An input consuming an output with such a `script_pubkey` would then need a signature that satisfied that requirement, e.g., it would need to contain signatures under both pk_1 and pk_2. While in principle Bitcoin Script can represent complex logic, in practice limitations on supported instructions and the length of a script mean it is mainly used for simple authorization.

We use (V, ϕ) to denote an UTXO of V coins with a `script_pubkey` ϕ. A Bitcoin transition (with the exception of coinbase transitions) consumes a set of UTXOs and creates one or more new ones. We use $\langle \{\mathsf{In}_i\}_{i=1}^{n} \xrightarrow{w_1,\ldots,w_n} \{\mathsf{Out}_j\}_{j=1}^{m} \rangle$ to denote a Bitcoin transaction with n inputs, m outputs, and n witnesses, one for each input, such that w_i satisfies the script of In_i.

Time-based opcodes. An essential ingredient of Π_{SGX} is Bitcoin's relative timeout script opcode, also known as CheckSequenceVerify or `CSV` [17]. By putting the CSV instruction with parameter τ in the script ϕ of a UTXO u, we assert that the transaction that spends u must reside in a block whose height (or timestamp) is more than τ relative to u.

2.3 Smart contracts

Smart contracts are executable objects stored in a blockchain. Users can send transactions with input data to the blockchain to trigger the execution of a smart contract. To process a transaction, the blockchain executes the code with the provided input data and potentially alters the blockchain state.

A key differentiator between such smart contracts and script-based systems (e.g. Bitcoin) *rich-statefulness* [64]. In a smart contract system, all executing transactions have native access to persistent state stored across transactions, blocks, and time. Rich statefulness is particularly relevant to our system. It is Bitcoin's lack of such that renders TEEs the only practical solution.

2.4 Censorship-resistant stateful channels

We use a blockchain to realize a censorship-resistant stateful channel, i.e., messages sent to the blockchain will be delivered within Δ time, and items added to the blockchain cannot be removed. In practice, a central goal of public blockchains such as Bitcoin is to prevent a malicious miner from dropping a user's transactions [28]. Also thanks to the high redundancy of the underlying peer-to-peer network, as long as a user can connect to *some* network nodes, her transactions will be propagated and included in a block with high probability, thus offering censorship-resistant access.

Bitcoin and other public blockchains have been proposed as a means to facilitate an anonymous channel (e.g., in [4, 9]). We note, however, that public blockchains are more powerful than anonymous channels (such as Tor), because blockchain-based channels are also stateful, i.e., messages are persisted on-chain with strong availability guarantees. As statefulness is critical in Paralysis Proofs, anonymous channels such as Tor insufficient. The statefulness of blockchains is also used in [22] to enforce fairness in MPC, and in [35] to build stateful TEE execution.

3 DYNAMIC ACCESS STRUCTURE SYSTEMS

In this section, we develop formal definitions and a framework for reasoning about the security of dynamic access structures.

A *Dynamic Access Structure Policy* (DASP) consists of a set of access structures and rules dictating migration conditions among them. For example, "this Bitcoin fund requires signatures from Alice, Bob and Carol to spend; if any of them disappears, signatures from the remaining two suffice to spend the fund" informally specifies a DASP. The access structure is the set of holders authorized to spend Bitcoin, and migration entails removing unresponsive signers.

We use the term *Dynamic Access Structure System* (DASS) to denote a system that enforces a DASP. Essential to our DASS constructions is the use of Paralysis Proofs to demonstrate conditions, e.g., party incapacitation, that justify migration from one access structure to another.

Motivating example. Getting dynamic access control policies right is hard and intuitive correctness often is not enough. There are subtle vulnerabilities that are missed without the formalism. For example, consider s_1 = "at least a majority of $\{1, 2, 3, 4\}$ is required to access", s_2 = "at least a majority of $\{1, 2, 3\}$ is required to access". If the current access structure is s_1 and player 4 goes missing, it's intuitive to permit migration to s_2. However this is *not* secure as it would deprive the privilege of one of the three players.

Looking forward, Definition 2 correctly prevents such a migration because $s_2 \notin S_{\text{LP}}(\{1, 2, 3\})$. (The only secure migration is to stay at s_1.) Capturing subtle vulnerabilities like this is critical and challenging, and motivates our exploration of the formal framework presented below.

3.1 Policy specification

3.1.1 Basic definitions. A Dynamic Access Structure Policy (DASP) comprises a tuple $(\mathcal{R}, \mathcal{S}, \mathcal{M})$ that specifies the **resources** ($\mathcal{R}$) being access-controlled, a set of **access structures** ($\mathcal{S}$), and a set of **migration rules** ($\mathcal{M}$) dictating conditions under which access-structure migrations are permitted.

Let $\{P_i\} = \{P_i\}_{i=1}^N$ denote the set of N parties at beginning of the protocol, and L_t the set of *live* (i.e. not incapacitated) parties at time t. As we shall see shortly, correctly determining L_t, i.e. which parties are actually live, is the main technical challenge in enforcing a DASP. We use L_t to denote the ground truth. When generally referring to a single player, we drop the subscript and simply denote the player as P. We assume that if a party becomes incapacitated, it remains incapacitated throughout the protocol, i.e. $P \notin L_t$ implies $P \notin L_{t'}$ for all $t' > t$.

In this paper, an access structure s is a function $s(L) \to \{\text{true}, \text{false}\}$ that determines whether a set of live parties $L \subseteq \{P_i\}$ is allowed to access the managed resource. Access structures are monotonic, i.e., $s(L) = \text{true}$ and $L \subseteq L'$ together imply that $s(L') = \text{true}$.

A migration rule $m_{s_i,s_j} \in \mathcal{M}$ is a function $m_{s_i,s_j}(\omega) \to \{\text{true}, \text{false}\}$ that determines whether migrating from s_i to s_j is permitted given witness ω. We use $s_i \overset{\omega}{\to} s_j$ to denote $m_{s_i,s_j}(\omega) = \text{true}$. The exact form of ω depends on the migration rule. An example of ω is L_t, the set of live players.

For a given DASP, the set of access structures $\mathcal{S}$ and the associated migration rules $\mathcal{M}$ may be represented as a directed graph $G = (\mathcal{S}, \mathcal{M})$. Here we overload $\mathcal{S}$ and $\mathcal{M}$ to denote respectively the sets of nodes and edges. A node $s_i \in \mathcal{S}$ is an access structure and an enhanced edge $(s_i, s_j) \in \mathcal{M}$ represents the migration rule m_{s_i,s_j}, which specifies the condition to migrate from access structure s_i to s_j. Access structure s_n is said reachable from s_1 by ω, denoted $s_1 \overset{\omega}{\rightsquigarrow} s_n$, if there exists a path $(s_1, s_2, \ldots, s_n)$ in G such that $s_i \overset{\omega}{\to} s_{i+1}$ for all $i \in [1, n-1]$.

3.1.2 Security goals. A fundamental correctness requirement for any access control is that migration between access structures does not eliminate the privilege of live parties. We capture this notion by stipulating that a DASP be **privilege-preserving**. To define this property, we first require two technical definitions.

Definition 1. *The set of* least permissive *access structures for $L \subset \{P_i\}$, denoted by $S_{LP}(L)$, is defined as:* $S_{LP}(L) = \{s \in \mathcal{S} : s(L) = \text{true} \wedge (\forall L' \subsetneq L, s(L') = \text{false})\}$.

Intuitively, $S_{\text{LP}}(L)$ is the set of all access structures such that if the only possible live parties are in L, then *all* such parties must be live to access the resource. Given this, we define privilege-preserving:

Definition 2. (Privilege-preserving) *Let L_t be the set of live parties at time t. A DASP $(\mathcal{R}, \mathcal{S}, \mathcal{M})$ is* privilege-preserving *if L_t can never migrate to an access structure that can be satisfied with a set L' of parties such that $L_t \nsubseteq L'$ at any time t. Formally, $\forall s \in \mathcal{S}$ such that $s(L_t) = \text{true}$, if there exists a witness ω such that $s \overset{\omega}{\to} s'$, then $s' \in \bigcup_{L_t \subseteq L} S_{LP}(L)$.*

A DASP is **paralysis-free** if, when the current access structure cannot be satisfied, switching to another satisfiable access structure is permitted provided that the migration will not deprive the privilege of any live party.

Definition 3. (Paralysis-freeness) *Let L_t be the set of live parties at time t. A DASP $(\mathcal{R}, \mathcal{S}, \mathcal{M})$ is* paralysis-free *if at any time t, $\forall s \in \mathcal{S}$ such that $s(L_t) = \text{false}$, it holds that:*

$$S_{LP}(L_t) \neq \emptyset \implies \exists \omega, s' \in S_{LP}(L_t) \text{ s.t. } s \overset{\omega}{\to} s'.$$

Note that a paralysis-free DASP doesn't imply the availability of the resource. What a paralysis-free policy can guarantee is the *best possible availability*: if there is a access structure s' that can get the system out of paralysis, then the DASP should permit a transition to s'. However, if the set of live parties is too sparse to satisfy any of the prescribed access structures (i.e. $S_{\text{LP}}(L_t) = \emptyset$), then the desired availability cannot be achieved.

Example 1. *Let's take the example of N shareholders who wish to retain access to the resource $\mathcal{R}$ should one party disappear. Let $\mathcal{P} = \{P_i\}_{i=1}^N$ denote the set of N parties, and $\mathcal{P}_{-i} = \mathcal{P} \setminus \{P_i\}$ denote the set of $N-1$ parties that excludes P_i. Let $\mathbb{I}(\cdot)$ denote an indicator function. A DASP $(\mathcal{R}, \mathcal{S}, \mathcal{M})$ that realizes the aforementioned access control can be specified by $\mathcal{S} = \{s_i\}_{i=0}^N$ where $s_0 = \mathbb{I}_{\mathcal{P}}$ and $s_i = \mathbb{I}_{\mathcal{P}_{-i}}$ for $i \in [N]$, and the condition $m_{s_0, s_i}(\omega) \in \mathcal{M}$ outputs* true *if ω is a witness to $L_t = \mathcal{P}_{-i}$.*

One can examine that the DASP in Example 1 is privilege-preserving and paralysis-free.

3.2 Security definitions for a DASS

We use the term *Dynamic Access Structure System* (DASS) to denote a system that enforces a DASP. In this section, we define the security of a DASS with a Universal Composability (UC) [20] ideal functionality $\mathcal{F}_{\text{DASS}}$. Later in Section 4 we present a protocol Π_{SGX} that UC-realizes $\mathcal{F}_{\text{DASS}}$.

3.2.1 *Adversarial model.*

We assume an adversary that may corrupt an arbitrary number of parties. An honest party always follows the protocol, while a corrupted party controlled by the adversary may deviate arbitrarily (i.e., perform Byzantine corruption). In the realization of $\mathcal{F}_{\text{DASS}}$, we assume that the adversary has complete control of the network, with the exception that a censorship-resistant stateful channel is available to all parties, i.e., anyone can send and receive messages through the channel subject to bounded latency, and messages sent through the channel are persisted and can be retrieved later.

3.2.2 *Ideal functionality.*

We specify security goals of a DASS in the ideal functionality $\mathcal{F}_{\text{DASS}}$ as defined in Fig. 2.

To reduce clutter, we omit the handling of session IDs (SIDs) in $\mathcal{F}_{\text{DASS}}$ but readers are advised that messages received and sent by $\mathcal{F}_{\text{DASS}}$ are implicitly associated with an SID. When $\mathcal{F}_{\text{DASS}}$ sends subroutine output to parties, we use the *delayed output* terminology from [20] to reflect the network adversary. Specifically, when $\mathcal{F}_{\text{DASS}}$ sends a public delayed output to party P_i, the output is first sent to $\mathcal{A}$ and then forwarded to P_i after $\mathcal{A}$'s acknowledgement or Δ time has past, whichever happens first.

$\mathcal{F}_{\text{DASS}}$ maintains internal states (L_t, s) for the set of live parties and currently enforced access structure respectively. To capture the paralysis explicitly, we extend the standard corruption model [20] with special "paralysis" corruption. Upon receipt of a paralysis message from $\mathcal{A}$, a party immediately announces its paralysis and halts until the end of the protocol. In the ideal protocol, $\mathcal{A}$ sends (paralysis, P_i) to $\mathcal{F}_{\text{DASS}}$, who removes P_i from the set of live parties.

To access the resource, a set of parties P send (access, inp), in which inp specifies the parameter of access, to $\mathcal{F}_{\text{DASS}}$. If P is permitted to access by the current access structure, i.e. $s(P) = \text{true}$, $\mathcal{F}_{\text{DASS}}$ returns the result of accessing $\mathcal{R}$. A set of parties can initiate a migration to another access structure s' by sending (migrate, s') to $\mathcal{F}_{\text{DASS}}$. If the transition to s' is permitted by the enforced DASP, $\mathcal{F}_{\text{DASS}}$ sets the current enforced access structure to s'.

$\mathcal{F}_{\text{DASS}}[s_0, \mathcal{R}, \mathcal{S}, \mathcal{M}]$ with parties $\{P_i\}_{i=1}^N$

```
On receiving* (init) from any $P_i$:
    $L_t := \{P_i\}_{i=1}^n$, $s := s_0$
On receiving (paralysis, $P_i$) from $\mathcal{A}$:
    $L_t = L_t \setminus \{P_i\}$
On receiving (access, inp) from $P_i \in L_t$:
    let current time be $t$
    // if there is an unexpired access request for inp
    if find a stored (inp, $\mathcal{P}$, $T_0$) and $t < T_0 + \Delta_a$ then:
        add $P_i$ to $\mathcal{P}$
    // if no access request for inp or it has expired, create a new one
    else : store (inp, $\{P_i\}$, $t$), overwriting (inp, _, _) if exists
    if $s(\mathcal{P})$ = true then:
        send a public delayed output $\mathcal{R}(\mathcal{P}$, inp) to all parties in $\mathcal{P}$
On receiving (migrate, $s'$) from $P_i \in L_t$:
    assert $(s' \in \mathcal{S} \wedge m_{s,s'} \in \mathcal{M})$
    $L_{\text{fake-death}} = \emptyset$
    for all corrupted parties $P_c \in L_t$:
        ask $\mathcal{A}$ whether $P_c$ should be added to $L_{\text{fake-death}}$
    if $m_{s,s'}(L_t \setminus L_{\text{fake-death}})$ = true then:
        send a public delayed output ($s$, $s'$, $P_i$, ok) to all; set $s = s'$
```

Figure 2: The ideal functionality of a Dynamic Access Structure System. The entry point marked with $*$ is only executed once.

3.2.3 *Security properties.*

$\mathcal{F}_{\text{DASS}}[s_0, \mathcal{R}, \mathcal{S}, \mathcal{M}]$ encapsulates the following security properties of a Dynamic Access Structure System. Let s denote the effective access structure of $\mathcal{F}_{\text{DASS}}$, and L_t the set of live parties at time t. $\mathcal{F}_{\text{DASS}}$ guarantees both *safety* and *liveness* in all states $s \in \mathcal{S}$ at any time t:

Safety: A set of parties $L \subseteq L_t$ can access $\mathcal{R}$ only if $s(L) = \text{true}$. A transition to $s' \neq s$ occurs only if $m_{s,s'}(L_t) = \text{true}$.

Liveness: If $s(L) = \text{true}$ for some $L \subseteq L_t$, then L can access $\mathcal{R}$ within Δ time after interacting with the DASS honestly. If $m_{s,s'}(L_t) = \text{true}$, then a transition to $s' \neq s$ occurs within Δ after L_t interacts with the DASS honestly.

Examples. Consider a DASS enforcing the DASP in Example 1, the **Safety** property ensures that access is enforced by the current access structure at any time, and that the access structure can be downgraded to allow access by $N-1$ shareholders only if $|L_t| < N$, i.e., a collusion of $N-1$ shareholders cannot maliciously accuse the N^{th} shareholder of being incapacitated and thereby steal her share. The **Liveness** property ensures that access is granted if the structure is satisfied by a set of cooperating parties. Moreover, if allowed by the policy, the Liveness property ensures that the access structure will be downgraded within a bounded time should parties submit legitimate requests. Note that the Liveness property does not stipulate that access structure s_i is automatically instantiated if $|L_t| < N$. This is because parties may not immediately activate an access-structure migration; in fact, if all parties are incapacitated, such migration cannot happen.

4 PARALYSIS PROOFS FOR CRYPTOCURRENCIES

In this section, we expand on the use of Paralysis Proofs to recover from cryptocurrency key loss (and related failures, e.g., player disappearance).

Bitcoin, the most popular cryptocurrency with a market cap of \$110B at the time of writing, is an important application target for Paralysis Proofs. Implementing secure Paralysis Proofs for the current Bitcoin protocol, however, is challenging because of the limited expressiveness of Bitcoin scripts. Were a proposed enhancement called "covenants" available [44], we show in the full version [69] that Paralysis Proofs can be constructed, though with significantly higher complexity. Moreover, it is unclear when or whether covenants will be adopted in Bitcoin. Thus we explore an alternative approach involves the use of TEEs—specifically, Intel SGX—in this section. In Appendix B we give a similar Paralysis Proof construction compatible with the current Bitcoin protocol without using SGX, but it only provides a weaker security guarantee and incurs more than exponential overhead, making it impractical.

Realizing Paralysis Proofs is challenging even with TEEs, due to their inherent security limitations: they can't maintain persistent states, their I/O can be censored by the malicious OS and they don't have an accurate sense of time. We use the following strawman protocol to illustrate why these limitations lead to design challenges.

A strawman protocol. Suppose there are N players. On initialization, the enclave generates a key pair (sk, pk) and outputs pk. Players moves the fund to the address of pk. The enclave also initializes the access structure as an (N, N)-multisig scheme, i.e., signatures from all N players are required to spend the fund controlled by sk. To attempt to remove a player P, the enclave sends a challenge to P and starts the timer. If P doesn't respond within Δ time, the enclave removes P from the access structure.

This strawman protocol is insecure for three reasons. First, unless additional mechanisms (e.g., [12, 40]) are employed, an SGX enclave is susceptible to state rollback. A malicious player could restore the enclave to a state before she's removed from the access structure, defeating the basic security requirement. Second, since the enclave's network is censored by the network adversary, she can cause a live player to be removed by delaying the response.

Note that these two problems can be solved by using a blockchain as a censorship-resistant stateful channel between players and the enclave. Nonetheless, the strawman protocol is still insecure. The issue is that the enclave—lacking a trusted timer—cannot establish an update-to-date view of the blockchain. For example, the adversary could present the enclave with a fork without P's response. Thanks to the lack of a trusted timer—in the case of SGX, an enclave can only ascertain an *lower bound* on the elapsed time—even an adversary with a low computational power can mount such attacks.

Solutions. Instead of patching TEEs to make the above strawman protocol work, we took a different path that *does not rely on TEE for keeping track of time or maintaining the current access structure.* It is important to emphasize that the security of our protocol doesn't require the enclave to have an update-to-date view of the blockchain—in fact, the enclave need not any view of the blockchain state. Moreover, a challenged player does not communicate with the enclave to signal her liveness. Finally, the enclave doesn't maintain any state, except for a secret key. Since the state is static, our protocol is immune from state rollback.

The high-level idea behind our solution is to use Bitcoin's time-based opcodes to track time, and to condition the output of the enclave on the state of the blockchain that is presented to the enclave. For example, if the TEE is presented with a forked blockchain, then the enclave output would only be valid on that fork. We achieve this using "life signal" transactions with time-based opcodes.

4.1 System model and trust assumptions

In this section we setup the formal model in which we specify the protocol, and discuss trust assumptions.

SGX and attested execution. Throughout the paper, we use SGX as a building block, although the protocol can be realized by any TEE capable of attested execution.

In our formal specification, we adopt the (local) ideal functionality $\mathcal{F}_{\mathrm{SGX}}$ based on Pass et al [50] to model TEEs with attested execution. Informally, a party first loads a program $\mathsf{prog}_{\mathsf{encl}}$ into an SGX enclave with a install message. On a resume call, the program is run on the given input inp, generating an output along with an attestation $\sigma_{\mathrm{SGX}} = \Sigma_{\mathrm{SGX}}.\mathsf{Sig}(\mathsf{sk}_{\mathsf{att}}, (\mathsf{prog}_{\mathsf{encl}}, \mathsf{outp}))$, a signature under the hardware key $\mathsf{sk}_{\mathsf{att}}$. The public key $\mathsf{pk}_{\mathsf{att}}$ is can be obtained from $\mathcal{F}_{\mathrm{SGX}}.\mathsf{getpk}()$. We refer readers to [50] for details.

An ideal blockchain. Our protocol uses an ideal blockchain defined in Fig. 7 as a censorship-resistant stateful channel. $\mathcal{F}_{\mathrm{chain}}[\mathsf{succ}]$ generalizes the bulletin board model because it also captures the notion of item validity. $\mathsf{succ}(\mathsf{history}, \mathsf{item}) \rightarrow \{0, 1\}$ is a function that specifies the criteria for a new item to be appended to history. We retain the append-only property of blockchains but abstract away the inclusion of items in blocks. As reflected in $\mathcal{F}_{\mathrm{chain}}$, we assume items are timestamped when added. In practice, block numbers can serve as such timestamps.

Trust assumptions. Our protocol relies on TEE with attestation that protects the confidentiality and integrity of computation, and an censorship-resistant stateful channel. Concretely, we assume SGX is correctly implemented and that the Bitcoin blockchain is secure and available to all parties. Let us stress that the trust assumptions in SGX is *local*, i.e., only the parties in the protocol will be affected should SGX properties be broken. We discuss ways to minimize trust in SGX in Section 4.4.

4.2 Protocol details

We denote our DASS for Bitcoin by Π_{SGX}. Π_{SGX} is formally specified in Fig. 8 and Fig. 9. We give text descriptions below of the steps involved in Π_{SGX}, so formal protocol or ideal-functionality specifications are not required for understanding.

Recall that we use N to denote the number of players at the start of Π_{SGX}, and P_i for $i \in \{1, 2, \ldots, N\}$ to denote each player. Each P_i is associated with a Bitcoin public key pk_i, whose corresponding secret key is only known to P_i. For simplicity, $\{P_i\}$ is used to refer to the complete set of all players.

Initialization. To start the protocol, some honest party needs to load an SGX instance with $\text{prog}_{\text{encl}}$ (Fig. 9) and invoke the init procedure. For now we assume a single SGX available to all honest parties; thus any honest party can initiate the enclave (once initialized, sequential initialization will be ignored). In Section 4.4 we present an expanded distributed setup procedure that avoids this assumption and provides stronger guarantees.

After the setup procedure is completed, the parties send a small fund of δ₿ (e.g. $\delta = 0.00001$) to a new output that can be spent by pk_{SGX}. Then the parties launch the protocol by sending their unspent output of V coins (denoted $\text{UTXO}_{\text{fund}}$) to a new output of V coins with a script spendable by $\{\text{pk}_i\}_{i=1}^N$ or pk_{SGX}.

Spending funds. There are two ways to spend the funds that are managed in Π_{SGX}. At any time, the players can spend the money via a Bitcoin transaction that embeds their N signatures (per ϕ_{all} in Fig. 8). Hence, even in the case that all N SGX CPUs are destroyed, the players are still able to spend the funds just as they could before the execution of Π_{SGX}. However, a better way to spend the funds is by sending N requests to an enclave, letting the enclave create a Bitcoin transaction with a single signature (signed by sk_{SGX}). This reduces the on-chain complexity and the transaction fee.

Migrating to another access structure. The migrate procedure of Π_{SGX} resolves system paralysis by letting the live shareholders spend the money if one or more shareholders is incapacitated. Intuitively, the role of SGX is to be an arbitrator: when any shareholder alleges that the money is stuck due to an unresponsive party, SGX first gives the accused party Δ time to appeal. The set of shareholders that controls the fund will be reduced only if no appeal is observed on the blockchain within this sufficiently large Δ (meaning that such an appeal did not occur, assuming censorship resistance [28] holds on the underlying blockchain).

The core idea of implementing an "appeal" in Bitcoin is to use what we call *life signals*. A life signal for party P_k is a UTXO of negligible Bitcoin amount ε₿, that can be spent either by P_k—thereby signaling her liveness—or by pk_{SGX}, but only after a delay. Π_{SGX} makes use of life signals to securely migrate to remove a party from the current access structure. Specifically, suppose the current set of shareholders is $P = \{P_i\}_{i=1}^N$, to (propose to) remove party P_k from P, any live players can send a message (migrate, $\text{UTXO}_{\text{fund}}, P \setminus \{P_k\}$) to $\text{prog}_{\text{encl}}$. Then $\text{prog}_{\text{encl}}$ will generate two signed transactions, t_1 and t_2 (defined in Fig. 9 and illustrated in Fig. 3), as follows:

- t_1: a life signal for P_k.
- t_2: spends both the life signal $\text{UTXO}_{\text{lifesignal}}$ in t_1 and the escrowed fund $\text{UTXO}_{\text{fund}}$ to a script that is spendable without P_k (i.e., by $(\{\text{pk}_i\}_{i=1}^N \setminus \{\text{pk}_k\}) \vee \text{pk}_{\text{SGX}}$).

The SGX enclave gives both t_1 and t_2 together as output. If t_1 is sent to the Bitcoin blockchain, P_k can cancel her removal by spending t_1. Otherwise, t_2 will become valid after the Δ delay and can be sent to the blockchain, thereby removing P_k's control over the fund. Fig. 3 demonstrates an example with three players.

Notice that $\text{prog}_{\text{encl}}$ parses $\text{UTXO}_{\text{fund}}$ and obtains the list of current shareholders, so that $\text{prog}_{\text{encl}}$ does not have to keep track of current live shareholders locally, nor does it need to have an up-to-date view of the blockchain. As we will discuss shortly, this is an important security feature.

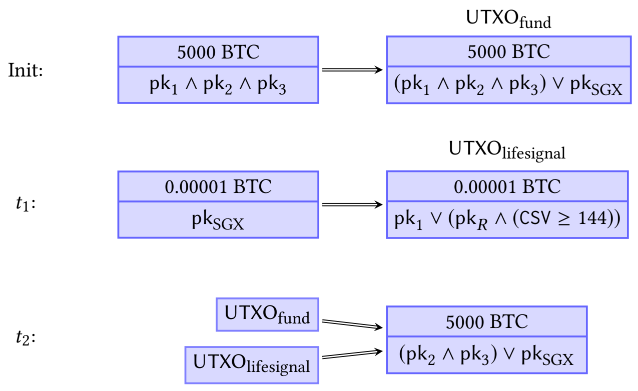


Figure 3: Example of Π_{SGX} with three players and P_1 accused of being incapacitated. Note that pk_R is a fresh ephemeral key generated for each life signal.

4.3 Security of Π_{SGX}

Intuitively, the security of Π_{SGX} stems from the use of SGX and the relative timeout feature of Bitcoin. We discuss the security of Π_{SGX} informally, then present a formal proof.

4.3.1 Security arguments. The core security property of Π_{SGX} is that a live party cannot be falsely removed from the access structure, no matter how many of parties are malicious. This is achieved by the use of the relative timeout feature of Bitcoin [17] in the fresh t_1, and the atomicity of the signed transaction t_2.

To elaborate, t_2 will be valid only if the witness of both inputs ($\text{UTXO}_{\text{fund}}$ and $\text{UTXO}_{\text{lifesignal}}$) is correct. The witness that the SGX enclave produced for spending the $\text{UTXO}_{\text{fund}}$ is immediately valid, but the witness for spending $\text{UTXO}_{\text{lifesignal}}$ becomes valid only after t_1 has been incorporated into a Bitcoin block that has been extended by Δ additional blocks (due to the `CSV` condition). The shareholder P_i that accused P_k of being incapacitated should therefore broadcast t_1 to the Bitcoin network, wait until t_1 is added to the blockchain, then wait for the next Δ blocks, and then broadcast t_2 to the Bitcoin network. However, while these Δ blocks are being generated, P_k has the opportunity to appeal by spending t_1 with the secret key sk_k that is known only to her (the script of t_1 does not require the `CSV` condition for spending with sk_k). Δ is set to a large enough value for two purposes: (1) to give P_k enough time to respond, and (2) to ensure that it is infeasible for an attacker to create a secretive chain of Δ blocks faster than the Bitcoin miners, and then broadcast this chain (in which t_2 is valid) to overtake the public blockchain.

A fresh, ephemeral key pair $(\text{sk}_R, \text{pk}_R)$ is generated for each life signal to ensure that t_1 is unique and hence does not already reside on the blockchain (e.g., P_k may have failed to respond to an earlier life signal but luckily another shareholder P_j was removed at that time). The SGX enclave does not need to store ephemeral keys, as they are consumed right after generation.

4.3.2 Strawman attacks. To illustrate the security properties of Π_{SGX}, we discuss a few strawman attacks in which $N-1$ malicious players (collectively referred to as the attacker) attempt to remove the Nth player (i.e., P_N) from the access structure and why they are prevented by Π_{SGX}. The attacker's goal is to somehow get t_2—the transaction that changes the access structure—included in the blockchain. Recall that we assume players can reliable connect to the blockchain (while the enclave may not.)

Withholding transactions. Clearly, withholding t_2 is only to the disadvantage of the attacker. On the other hand, withholding t_1 doesn't gain the attacker any advantage either. Due to the relative timeout in t_1, P_N always gets the Δ blocks worth of time to respond regardless when t_1 is included in the blockchain.

Manipulating the I/O of the enclave. In Π_{SGX}, the enclave receives as input (migrate, $\text{UTXO}_{\text{fund}}, P \setminus \{P_N\}$) and outputs signed transactions (t_1, t_2) as specified. The enclave doesn't (and generally can't) check the validity of the input UTXO, nor does it check that the output transactions are sent to the blockchain correctly. This is, however, not a security problem.

The attacker may feed the enclave with an invalid (e.g., spent or non-existent) UTXO. However, since the output transaction t_2 spends $\text{UTXO}_{\text{fund}}$, providing the enclave with an invalid $\text{UTXO}_{\text{fund}}$ would result in an invalid t_2 that can't change the access structure, making the attack a no-op.

The attacker may also send the output transactions to secretive forks as opposed to the main chain. Note that since t_2 spends t_1, in order for t_2 to be valid, both must be in the same fork. If both are sent to the main chain, then P_N would notice and respond normally. However, if both are sent to a secretive fork, the access structure would indeed be changed *on that fork*. Nonetheless, unless the attacker can overtake the main chain with that fork, which is at least as difficult as double spending attack assuming a large enough Δ, the fund on the main chain remains intact.

4.3.3 Security proofs. The security of Π_{SGX} is proven using the framework developed in Section 3, as summarized in Theorem 1. See the full version [69] for a proof sketch.

Theorem 1 (The Security of Π_{SGX}). *Assume $\mathcal{F}_{\text{SGX}}$'s attestation scheme and the digital signature used in Π_{SGX} are existentially unforgeable under chosen message attacks (EU-CMA). Then Π_{SGX} UC-realizes $\mathcal{F}_{\text{DASS}}[s_{P_0}, \mathcal{R}, \mathcal{S}, \mathcal{M}]$ in the $(\mathcal{F}_{\text{SGX}}, \mathcal{F}_{\textit{chain}})$-hybrid model, for static adversaries.*

4.4 Minimizing trust in TEEs

We now briefly consider some ways to minimize the trust placed in the TEE employed in our protocol.

Avoiding a single point of failure. Trusted hardware in general cannot ensure availability. In the case of SGX, a malicious host can terminate enclaves, and even an honest host could lose enclaves to outages. To avoid reliance on a centralized SGX server, each party in Π_{SGX} can run her own SGX enclave with an identical program. This way, any individual party (or set of parties) can always use all the capabilities of the protocol without being dependent on others.

Specifically, the initialization procedure of Π_{SGX} can be replaced with the following procedure that distributes the master key sk_{SGX} across multiple hosts. First, each enclave first generates a fresh key pair $(\text{pk}_{\text{SGX}_i}, \text{sk}_{\text{SGX}_i})$ and outputs pk_{SGX_i} while keeping sk_{SGX_i} secret. Then, each player uses her identity P_i to endorse pk_{SGX_i}, and all the players reach agreement on the list of SGX identities $\{\text{pk}_{\text{SGX}\,i}\}_{i=1}^{N}$. Finally, the enclaves then use $\{\text{pk}_{\text{SGX}\,i}\}_{i=1}^{N}$ to establish secure channels (TLS) with each other, and create a fresh shared secret key sk_{SGX} that is associated with $\{\text{pk}_{\text{SGX}\,i}\}_{i=1}^{N}$ (i.e., another invocation of the setup procedure will generate a different shared key). Given use of the secure hardware random number generator (RDRAND), secret keys generated by SGX are known only to the enclaves, not to any of the players. From now on, no inter-enclave communication is needed in the course of the protocol. Each enclave then seals its state (which mainly consists of sk_{SGX}) by encrypting it using the hardware key (unique to each CPU) and storing the ciphertext to persistent storage. Hence, the enclave program does not have to run persistently, and each players can run the backup on-demand when needed.

Side-channel resistance. Although SGX aims to provide confidentiality, recent work has uncovered data leakage via side-channel attacks (e.g. [62, 66]). Admittedly Π_{SGX} is not side-channel-free, but it has a relatively small and controlled attack surface. The only secret in SGX is sk_{SGX} and only operation involving sk_{SGX} is signature generation (besides key generation)—this makes Π_{SGX} amenable to software-level side-channel mitigations, such as constant-time ECDSA implementation (e.g. [25]).

A more powerful and somewhat more interesting approach is to design side-channel-free Paralysis Proofs. We claim that no side-channel-free construction of Paralysis Proofs exists given the current trust assumptions. However, if we relax the assumptions slightly, for example, by assuming a trusted relative clock in SGX (which is not available now [1]), or assuming certain stationarity properties of the blockchain (e.g. difficulty), a side-channel-free Paralysis Proofs can be constructed by establishing an up-to-date view of blockchain in SGX (e.g., using techniques in [21]). Specifically, SGX will only be activated when paralysis happens (which requires an up-to-date view of the blockchain to detect), and will generate a new key sk_{SGX} for every new $\text{UTXO}_{\text{fund}}$. Since the enclave secret is used only once, such a construction is side-channel-free.

Least-privileged SGX. In Π_{SGX} and the examples above the fund can be spent by pk_{SGX} alone, but it's important to note that is not the only option. In fact, one can tune the knob between security and paralysis-tolerance to the best fit their needs. Specifically, for a desired level of paralysis-tolerance, one can design a DASP such that the SGX is *least-privileged*. For example, if the three shareholders only desire to tolerate up to one missing key share, what they can do is to move the funds into 3-out-of-4 multisig wallet where the 4th share is only known to the SGX enclave. If all of the parties are alive, then they can spend without use of the SGX node. If one of them is incapacitated, the enclave will release its share upon presentation of a Paralysis Proof. Therefore, even if the secret state of the SGX node (i.e., the fourth share) is leaked via a successful side-channel attack, the attacker cannot spend the fund unless two malicious parties collude. It can be shown that the SGX in the above DASP is least-privileged, in the sense that compromise of its secret state imparts minimal capabilities to an adversary. Intuitively, since we want to retain access even one player is incapacitated, the enclave must store a credential equivalent to that of the lost player. We leave formal specification of least-privileged SGXs for future work.

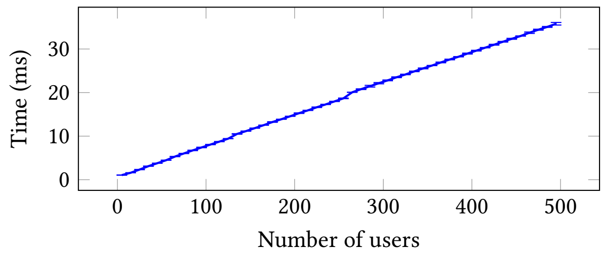


Figure 4: Execution time of P^{2}W when accusing one user.

4.5 Implementation

4.5.1 Paralysis Proofs for Bitcoin. We implemented Π_{SGX} as a Paralysis-Proof Wallet (called P^{2}W) for Bitcoin based on Intel SGX SDK and Bitcoin Core. The source code of P^{2}W is published at [2]. The same code works for Bitcoin and compatible systems, e.g., Litecoin.

To minimize the Trusted Computing Base (TCB), we ported only the necessary part of Bitcoin Core v0.14.0 to SGX, resulting in only using ~9.5% of it. The entire TCB includes 734 lines of C++ code written by us, ~9.5% of the Bitcoin Core and its Elliptic Curve library (libsecp256k1).

In the normal case, P^{2}W is essentially a (N, N)-multisig wallet. Users can spend the coins by either send N signatures to the blockchain, or to P^{2}W which then generates a single signature. The latter method reduces the transaction size and saves transaction fees, while the former ensures the availability of the fund. When a user is suspected to be unavailable, others can accuse her using P^{2}W, which sends transactions t_1 and t_2 (c.f. Section 4.2) to the blockchain. The accused user can appeal by spending the life signal UTXO in t_1 (a standard P2PKH output) using P^{2}W or any standard Bitcoin wallet.

We evaluated P^{2}W's performance by simulating N users and measuring the execution time of accusing one of them . As summarized in Fig. 4, P^{2}W is very fast. For example, for 100 users, it only takes about 10 ms to generate accusation transactions. t_1 is of constant size (224 bytes) which means the cost of a malicious accusation is about as high as dismissing it. t_2 is roughly $(443+34N)$ bytes. With a suggested fee rate of 20 Satoshi/byte[1] at the time of writing, the transaction fee of t_2 would be about \$0.02N, which is a very reasonable cost for such an infrequent and security-critical operation.

We validated P^{2}W's compatibility with Bitcoin by deploying it on the testnet. We set $\Delta = 144$ blocks (roughly 24 hours in Bitcoin) and simulated the following two scenarios: a) a user is falsely accused, but she appeals within Δ and maintains the ownership of the wallet, and b) a disappeared user is challenged and removed from the wallet after Δ. Accepted Bitcoin transactions can be found at [5, 6] (for scenario A) and [7, 8] (for scenario B). Note that in the latter scenario, t_2 is not accepted until 24 hours after t_1 is included, enforced by the `CSV` opcode.

[1]See, e.g., https://statoshi.info/dashboard/db/fee-estimates

4.5.2 Paralysis Proofs for Ethereum. For assets that can be controlled by smart contracts, an implementation of the ideal functionality $\mathcal{F}_{\text{DASS}}$ is straightforward. Our reference implementation of a paralysis-free multisig wallet for Ethereum [64] consists of 156 lines of commented Solidity code. Fully tested contract code, including logic for pruning incapacitated signers and updating the signature threshold is at [3].

This implementation differs from the ideal functionality only in minor engineering changes and optimizations. There is no way to asynchronously prune keyholders that fail to respond to a challenge in time in Ethereum, where all contract calls must be initiated by some user. We instead check and prune any signers that did not respond to a challenge at the beginning of each on-chain operation that requires checking or manipulating only valid signers. This ensures that the state of unparalyzed signers is correct, reflecting L_t in $\mathcal{F}_{\text{DASS}}$, before any contract action is processed.

A final caveat is that block timestamps are used to measure time; while this can be trivially replaced with block numbers, which are less susceptible to miner manipulation (timestamps are miner set), the bounded degree of manipulation and monotonically increasing timestamp constraints on Ethereum provide some assurance that the timestamps are reasonably accurate for our purposes.

One useful property of the Ethereum-based realization is that the multisignature key holders need not necessarily run archival nodes: because a log is emitted whenever a user is accused, users can simply watch transaction receipts for an accusation against them, using any Ethereum full or lite client to respond by calling the `respond` function (guaranteed to work as long as an adversary cannot censor a user's connection to the blockchain, given that the user accepts the relevant trust assumptions surrounding their choice of node software, hardware, and connectivity).

5 EXTENDED PARALYSIS PROOFS: THE CASE OF CUSTODIAN PARALYSIS

5.1 Motivation

A (digital asset) custodian is a *centralized* service that secures or helps secure a user's digital asset (or other key-controlled resource). Use of custodians is common as they offer better usability and security, compared to having end users managing secrets on their own. Coinbase recently indicated that it was holding 10% of all Bitcoin in circulation [11].

An obvious concern with centralized custodians is theft, in which a custodian (or hacker that has compromised a custodian) steals funds from users' accounts. In addition to theft, Denial-of-Service (DoS) is another major concern. If a custodian refuses to respond to a user's transaction requests, the user effectively loses her funds.

In this section, we show how DASSes enable an interesting new architecture for digital assets custodians—a *hybrid* custodian—to address both concerns.

To achieve these properties, we refine the definition of paralysis. In addition to unavailability, a party (i.e., a custodian here) is considered paralyzed if it exhibits a certain behavior (e.g., maliciously rejects users' withdrawal requests). This definition *refines* the previous ones, as it such cases, the custodian may be "available" in the sense of responding to clients, but exhibits paralysis by failing to process certain legitimate transaction requests.

In addition to realizing a new and appealing custody architecture, this example also illustrates how Paralysis Proofs can embody rich and precise conditions beyond mere unavailability. Specifically, we show that a publicly verifiable proof can be constructed if a custodian rejects valid authentication attempts. Generally, Paralysis Proofs can be a condition predicated on user and/or system secrets (passwords and/or keys for two-factor authentication), in addition to mere absence of response.

5.2 Protocol

Setup. In our hybrid custodian scheme, a user stores her funds in a $(2, 2)$-paralysis-proof-multisig wallet (using techniques in Section 4), of which one key is held by a TEE operated by the centralized custodian. The custodian is responsible for authenticating the user (e.g., via a password, SMS code, etc.) before authorizing a transaction. Both signatures from the user and the custodian are required to spend funds.

Such a setup has two security benefits. First, it prevents the custodian from stealing the funds, as the custodian only possesses one of the secret keys. Second, it offers the user stronger security than if she managed her own keys, given the general weakness of endpoint security and the fact that an institutional custodian is usually much better at managing and securing keys than an ordinary user is.

Dealing with malicious DoS. As mentioned, we add additional rules to basic Paralysis Proofs so that a party (i.e., the custodian) is considered paralyzed unless a predicate ϕ is met. In other words, when the custodian is accused (see Section 4 for terminology), it can only appeal if ϕ is true.

Specifically, ϕ will ensure an "appeal" is issued if and only if the custodian's TEE faithfully attempted and failed to authenticate the user (e.g., the user inputs a wrong password). In other words, the custodian is honest but the accusation is invalid. Without this predicate, a malicious custodian can always appeal to dismiss honest users' accusations. With the predicate, however, the custodian can only appeal against invalid accusations. Thus, we reduce the case of a malicious custodian actively refusing to accept authenticated transactions to that of a simple paralyzed party who cannot appeal.

Fig. 5 illustrates the idea of conditioning the issuance of response on the predicate of "the challenge contains valid authentication information." The exact details of this predicate can vary depending on the authentication mechanism, as explored by example below.

Example: Password-based authentication. Consider a simple scenario where the user authenticates to the custodian with a password. The user submits her password (encrypted under the TEE's public key) along with a partially signed transaction, and a signature over her full request. The custodian's TEE verifies the signature on the request and checks if the password is correct. If so, it signs the transaction with its key. Otherwise, the request is discarded. If the signature is correct but the password is wrong, then the custodian may advance a counter which can be used to trigger rate limiting or lock the account. Here the custodian's role would be to store a hashed version of the password securely, check it against the submitted password, and rate limit password attempts. Assuming that rollback attacks against TEEs are prevented using, e.g., distributed state [40], authenticating the password request with a signature prevents the custodian itself from submitting false authentication attempts in an effort to deny service by locking a user's account.

If the custodian is available but refuses to accept the password, we can now enter into an on-chain dispute resolution process, as shown in Fig. 5. The user posts a challenge containing the encryption of her password under a key known to the custodian's TEE. If the custodian doesn't respond within Δ, the standard Paralysis Proofs mechanism can kick in and remove the custodian from the access structure. To appeal with a response, the custodian's TEE must decrypt the submitted password and test if it is correct. If the password is correct, the TEE refuses to issue a response, which is equivalent to the first case where no response is issued.

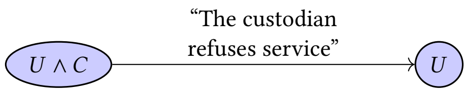
5.3 Extensions

For the ease of exposition the above protocol description uses simple password-based authentication. Various extensions are discussed below, including authentication via a third party (e.g., 2FA), user account recovery, and a general approach to use any interactive authentication protocol.

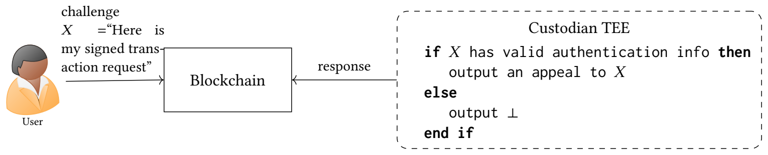
Third-party authentication. Another option is for the TEE to contact third parties directly to provide a primary or secondary authentication factor. Town Crier [68] demonstrated that it is possible to make a TLS request from an enclave to a third-party service and condition behavior on the response. Combined with input from the user via the blockchain, this can be used to directly authenticate a user, to provide a second factor via services such as Authy or Twillo, or to ensure that a user still has an account with some service. Many of these mechanisms depend on trusting a third party, but in the case of established and widely used services this may be more palatable than trusting any specific custodian. Multiple services can of course be combined to further distribute trust—with a trade-off against availability.

User account recovery. Using the same mechanisms for interactive protocols or authenticating via third parties, the custodian can provide a mechanism for account recovery in the case of lost credentials. Given a Paralysis Proof for the user, control can be transferred to a fresh multisig address. Indeed, if the custodian relies on third parties for authentication, then it inherits the account recovery mechanism automatically. This is, of course, a double-edged sword: the same mechanisms that are used for account recovery can be used to hijack the account. We are not limited, however, to simple password recovery mechanisms. If the custodian only controls one of two keys necessary to spend the funds, we can realize account recovery by requiring a Paralysis Proof against the user to migrate access control to another key.

More complex protocols. Following the techniques of [35], any interactive authentication protocol can be realized by posting encrypted messages between the client and the custodian to the blockchain. A Paralysis Proof then is simply an on-chain execution of the protocol where messages are delivered and logged via the blockchain. Indeed, the password example is a simple one-round version of this. The same idea can be applied to n-round protocols through repetition and extended to, for example, integrating federated authentication protocols or challenge-response-based two-factor authentication.



(a) The DASP in the custodian paralysis example. “U” denotes the user, and “C” denotes the custodian.



challenge X =“Here is my signed transaction request”
User
Blockchain
response
Custodian TEE
if X has valid authentication info then
output an appeal to X
else
output ⊥
end if

(b) Extended Paralysis Proofs: the accused party can appeal only if a predicate is met. Read Section 5.2 for details.

Figure 5: An example of custodian paralysis and migration.

6 PARALYSIS WITHIN SMART CONTRACTS

6.1 Motivation

We've explored the use of smart contracts in Paralysis Proofs in Section 4. In the broader smart contract community, it is well known that paralysis can occur within smart contracts themselves. A classic example is the second of two well-publicized and related Parity multisignature vulnerabilities [15] [56] which permanently and irrevocably froze hundreds of millions of US dollars in smart contracts. The Parity multisignature vulnerabilities are far from the only high-profile failures that resulted in paralysis; early analysis of Ethereum smart contract vulnerabilities [19] enumerated a number of vulnerable contracts with stuck funds and DoS vulnerabilities. Some vulnerabilities are subtle, involving low-level platform details like the “gas" model for pricing computation [64].

Most smart contract vulnerabilities, despite their different manifestations, have a fundamental commonality: in each case, a smart contract was operating as intended until some unexpected change to the state of the contract, the network, or the computation model under which the contract was operating. These changes then caused subsequent executions of previously working functionality to fail, leaving the funds in a contract potentially “paralyzed," and stuck indefinitely. This problem is so widespread and severe on Ethereum, that hard-fork-based manual remediation of affected contracts has been suggested as a major governance issue and debate [18] [30].

We find a natural way to capture to this class of failure in Paralysis Proofs using software engineering tradition. When integrating various system components which may potentially be faulty, developers often create and run integration tests [34], often continuously as software is developed, in a process known as “continuous integration" [26, 58]. We can naturally extend Paralysis Proofs using continuous integration: if a full integration test suite is available, a *provable* test failure constitutes a Paralysis Proof and triggers a “downgrading of access control", authorizing transfer of funds to a simpler recovery contract.

6.2 Protocols

Smart-contract based implementation. Two variant implementations of this idea are possible. The first is smart-contract based, with a protocol specification for the smart contract given in Fig. 6. (We omit user entry points from this specification.) In this protocol, there is a static, hardcoded set of tests T, with each test τ_i containing some storage locations to initialize that are required for correct operation of the test. (These locations are not necessarily comprehensive; tests can and should can pull storage variables from the environment.) For example, static state σ_i might contain an initialization of a special testing account with some balance available to be transferred that is not present in the real token contract, with the remainder of state required for the test sourced from the global stateful environment.

Program for Smart Contract Paralysis Proofs (prog_{ci})

```
Hardcoded: $T = \{(\pi_1, \sigma_1, \omega_1), \ldots, (\pi_n, \sigma_n, \omega_n)\}$ // test set
  // test $\pi_i$ should run in state $\sigma_i$ for expected output $\omega_i$
  where $\sigma_i = \{(s_1, v_1), \ldots, (s_j, v_j)\}$ // test setup
  $h_{\text{now}}$ // the latest height,
  $H$, // mapping from block hash to height,
  $\Delta$, // maximum time to construct a proof, in blocks
  $C = [C_P, C_R]$ // addresses of main and recovery contract
On input (access, $d$):
  Return $C[0](d, \text{msg.sender})$
On input (migrate, $(i, p = \{(s_1, v_1, p_1), \ldots, (s_k, v_k, p_k)\})$):
  Assert $\{p_i\}$ are all from the block (i.e. have the same Merkle root)
  Assert $h_{\text{now}} - H[p_1.\text{merkle\_root}] \leq \Delta$ // Ensure proof is fresh
  $\Sigma := \emptyset$ // Initialize state (location to value map) for test to run in
  For each $(s_j, v_j, p_j) \in p$:
    Assert is_valid($(s_j, v_j, p_j)$) // Ensure Merkle proofs are valid
    $\Sigma[s_j] = v_j$ // Initialize storage from environment
  For each $(s_j, v_j) \in \sigma_i$:
    $\Sigma[s_j] = v_j$ // Initialize static per-test storage
  If $\pi_i(\Sigma) \neq \omega_i$: // if test output differs from expected
    $C = C \setminus C[0]$; $T = \emptyset$
```

Figure 6: An example implementation of Paralysis Proofs for smart contracts.

If a smart contract is paralyzed, it must be failing some unit test i at block b. A user who notices this submits a proof by sending migrate, including Merkle proofs-of-inclusion for all state items consumed by the test *from the environment* at block b in the global Ethereum state trie [64]. The implementation prog_{ci} checks that submitted Merkle proofs are all from the same block, that the block is recent (to prevent DoS via stale proofs), and that all proofs are valid.

For this check, prog_{ci} instantiates a temporary state. First all the validated state entries from the environment are added, then updated with the hardcoded static initialization state in σ_i. If observed test output differs from expected, paralysis is detected and prog_{ci} migrates any access to a recovery contract that returns user funds, preventing paralysis of contract funds (assuming the test set is sufficiently rich and recovery operational). The test set is nulled to prevent further migrations.

Both users of the target contract and companies interested in maintaining availability of the contract (like the developers, for example Parity Technologies in [49]) are strongly incentivized to submit such proofs when they detect paralysis. Contracts desiring economic robustness can also incentivize a market of test executors by adding a substantial cryptocurrency bounty, paid to any user who successfully executes migrate.

The smart-contract based scheme has important advantages. Primarily, any user can prove paralysis using only on-chain data at any time, minimizing the trust surface required to just the code governing proof verification and recovery (and excluding complex attestation schemes and trust in enclave confidentiality). The scheme is also practical and optimistically efficient. We tested an example implementation provided at [37] of a Merkle-Patricia state item proof checker in a smart contract, usable for the `is_valid` function referenced in prog_{ci}. In the optimistic case where funds are not paralyzed, our scheme adds no on-chain overhead or additional cost to contract operation. In the exceptional case of paralysis, our scheme's cost is justified by the potential recovery of funds. Our initial exploration suggests a cost of about 1.36 million gas per invocation of the functionality required by `is_valid`; this is approximately 1/6 of a full Ethereum block, or \$12 per storage location proof at the time of writing: expensive but not prohibitive.

SGX-based implementation. Several issues are however present with this SGX-free smart contract scheme. The on-chain Merkle proof verification is still somewhat costly, and a transaction/test can potentially access many storage locations. This may be acceptable in smart contract form, as tests can be potentially broken up into small/short pieces. This scheme financially incentivizes short tests, however, which may limit expressiveness of developers' tests. Developers may choose to optimize against worst-case verification cost, which is difficult in a volatile and unpredictable transaction fee market. By making the cost of executing a test less dependent on the size of the test using SGX and off-chain computation, longer and more expressive tests become tractable. Such a scheme can optionally be operated in the Sealed-Glass Proof model in [60], enabling resilience against unbounded side channel leakage and removing confidentiality requirements, relying only on attestations for security.

An SGX-based solution can also leverage attestations and confidential execution. For example, if any off-chain or legacy systems are required in the integration test (e.g., when an oracle such as Town Crier [68] is used), they can be queried or emulated by SGX, or can use a trusted off-chain oracle. Also, confidential tests may be useful for some contracts. While unsuitable in a public network due to their ability to hide backdoors, one could imagine a contract between parties with neutrally-agreed-on third parties or arbiters responsible for maintaining independent anti-paralysis test suites.

This approach can also be applied within a modified version of N-of-N version programming (NNVP) as described in the Hydra framework [16]. To reduce on-chain gas costs, one or more heads can be executed off-chain within an enclave, which can issue an attestation when a bug (divergence in component outputs) is identified. Unfortunately, off-chain execution does not harmonize well with the Hydra bug-bounty system, as it leaves on-chain logic exposed to exploitation. But it does provide an objective means for identifying vulnerabilities, creating a Paralysis Proof and triggering fault logic within a paralyzed smart contract.

7 RELATED WORK

Static access-structures for cryptocurrencies. Bitcoin had built-in support for threshold signatures at launch, and access-structure scripts for Bitcoin have been discussed since at least 2012 (e.g. [51]). However the built-in implementation (`CHECKMULTISIG`) simply takes a list of individual signatures and check if the threshold are met, which increases the on-chain verification complexity (this is undesirable, cf. [38]).

[29] presented a novel ECDSA threshold signature construction that reduces on-chain complexity. However, this construction requires a rather complex setup using ZK proofs, and does not support arbitrary access structures. Threshold Schnorr signatures are far more efficient [57], with support planned for Bitcoin [65].

Ethereum wallets such as Mist [43] and Gnosis [63] support multi-signature access structures, along with other features such as daily limits. However, these wallets are implemented via on-chain code, which implies that users will incur higher costs when the complexity of the access structure is greater. New versions of the Gnosis wallet [42] allow for arbitrary, unrestricted challenge-based policy migration, but do not formalize security or suggest secure policies.

Access-control policies with dynamic access-structures. Secret sharing schemes with revocation support do not provide the same guarantees as a Paralysis Proof system, since such schemes require actions by at least a threshold of players to update the access structure (see [24, 67]). By contrast, a Paralysis Proof system enables any player to remove incapacitated players. Privacy-preserving cloud services can allow remote administrators to modify access-control policies dynamically, via cryptographic constructions (see, e.g., [32, 36]). Dynamic access-control policies for a non-confidential cloud service may also benefit from dynamic access-control policies [45]. All of these constructions requires a static set of administrator set authorized to perform modifications.

Credential-recovery. Password systems allow recovery from secret loss but require a TTP. See [14] for a survey.

8 PARALYSIS PROOF SYSTEMS BEYOND CRYPTOCURRENCIES

The techniques we have introduced for Paralysis Proof Systems in combining SGX with blockchains can be applied to settings other than Paralysis Proofs and even to settings other than cryptocurrencies. We give some examples here:

Daily spending limits. It is possible to enforce limits on the amount of BTC that set of players can spend in a given interval of time. For example, players might be able to spend no more than 0.5 BTC per day. We explore this objective, and technical limitations in efficient solutions in the full version [69].

Decryption. The credentials controlled by a Paralysis Proof System need not be signing keys, but instead can be *decryption keys*. It is possible then, for example, to create a deadman's switch. For example, a document can be decrypted by any of a set of journalists should its author be incapacitated.

Event-driven policies. Using an oracle, e.g., [68], it is possible to condition access-control policies on real-world events. For example, daily spending limits might be denominated in USD by accessing oracle feeds on exchange rates. Similarly, decryption credentials for a document might be released for situations other than incapacitation, e.g., if a document's author is prosecuted by a government. (This latter example would in all likelihood require natural language processing, but this is not beyond the capabilities of an enclaved application.)

The last example involving prosecution does not require use of a blockchain, of course. Many interesting SGX-enforceable access-control policies do not. But use of a blockchain as a censorship-resistant stateful channel can help ensure that policies are enforced. For example, release of a decryption key might be *entangled* with the spending of cryptocurrency. A certain amount of cryptocurrency, say, 10 BTC, might be spendable on condition that an oracle is recently queried and the result consumed by an enclave application. This approach provides an economic assurance of a censorship-resistant stateful channel from the blockchain to the enclave.

9 CONCLUSION

We have shown how Paralysis Proofs can enrich existing access-control policies in a way that was previously unachievable without a trusted third party. By leveraging Paralysis Proofs, DASSes allow an access structure to be securely migrated—typically downgraded—given the incapacitation of a player, the inability of a set of players to act in concert, or the functional paralysis of a smart contract.

Our formalisms include a formal DASP framework, and security and functionality definitions for DASSes and DASPs, as well as UC-type ideal functionality for a DASS. Our ideal functionality suggests a natural proof sketch for security.

Paralysis Proofs and DASSes can be applied in many settings, and we showcase three in the paper: cryptocurrency key loss, cryptocurrency custody failures, and smart contract failures, proposing practical schemes for all three. We report on a simple DASS for cryptocurrency key loss in Ethereum, and on a detailed exploration concluding that DASS for Bitcoin is only practical using a TEE.

In summary, we believe that the combination of the advent of two pivotal technologies, blockchains and trusted hardware (specifically SGX), is a powerful one. It enables a powerful new range of access-control regimes without the need for trusted third parties and, we believe, will stimulate exploration of a broad spectrum of other novel capabilities with applications beyond cryptocurrencies.

ACKNOWLEDGEMENTS

This work is funded in part by NSF grants CNS-1330599, CNS-1514163, CNS-1564102, and CNS-1704615, as well as ARO grant W911NF-16-1-0145.

REFERENCES

[1] 2017. `sgx_get_trusted_time`. https://github.com/intel/linux-sgx/issues/161.
[2] 2018. Paralysis Proofs Implementation (Bitcoin and SGX). https://s3.amazonaws.com/anonymous-code/pp.tar.gz.
[3] 2018. Paralysis Proofs Implementation (Ethereum). https://s3.amazonaws.com/anonymous-code/Paralysis.sol.
[4] 2018. R3C3: Cryptographically-secure Censorship Resistant Rendezvous using Cryptocurrencies. Preprint.
[5] 2018. Scenario A: the appeal transaction that spends t_1. https://tbtc.bitaps.com/9f512c103dad80c012ea63212a173e939193e67716e6456a44ee45946cc5c6ad.
[6] 2018. Scenario A: the life signal transaction t_1. https://tbtc.bitaps.com/3adb9a9048695b1a6684b5803e43c8aaa79b603589c627a19dbba89b8c64f4ba.
[7] 2018. Scenario A: the migration transaction t_2. https://tbtc.bitaps.com/a75e045b9c74a3f803787087e3958d6345236d77c8507b0ed0796d47bb6c91b6.
[8] 2018. Scenario B: the life signal transaction t_1. https://tbtc.bitaps.com/eb9588aad0b080dc3ba836ab19c358306f9d508b81ea78335eafac5a76b71656.
[9] Syed Taha Ali, Patrick McCorry, Peter Hyun-Jeen Lee, and Feng Hao. 2015. ZombieCoin: Powering Next-Generation Botnets with Bitcoin. In *BITCOIN Workshop at Financial Cryptography and Data Security (FC 2015)*. Springer.
[10] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *HASP*.
[11] Brian Armstrong. Feb. 25, 2018. Coinbase is not a wallet. https://blog.coinbase.com/coinbase-is-not-a-wallet-b5b9293ca0e7.
[12] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xeuyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. 2017. Tesseract: Real-Time Cryptocurrency Exchange using Trusted Hardware. https://eprint.iacr.org/2017/1153.
[13] Nirupama Devi Bhaskar and David LEE Kuo Chuen. 2015. Bitcoin exchanges. In *Handbook of Digital Currency*. Elsevier, 559–573.
[14] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. 2012. The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *IEEE Symposium on Security and Privacy (S&P)*.
[15] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. 2017. An In-Depth Look at the Parity Multisig Bug. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.
[16] Lorenz Breindenbach, Phil Daian, Florian Tramèr, and Ari Juels. 2018. Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts. In *USENIX Security*.
[17] BtcDrak, Mark Friedenbach, and Eric Lombrozo. 2015. *BIP 112: CHECKSEQUENCEVERIFY*.
[18] Vitalik Buterin. 2016. Reclaiming of ether in common classes of stuck accounts (EIP #156). https://github.com/ethereum/EIPs/issues/156.
[19] Vitalik Buterin. 2016. Thinking About Smart Contract Security. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.
[20] Ran Canetti. 2000. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067. https://eprint.iacr.org/2000/067.
[21] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. In *EuroS&P*.
[22] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. 2017. Fairness in an Unfair World: Fair Multiparty Computation from Public Bulletin Boards. In *CCS*.
[23] Frank Chung. 2017. "Don't tell my wife": Melbourne man cries over lost bitcoins as price surges past $10,000. http://www.news.com.au/finance/money/investing/dont-tell-my-wife-melbourne-man-cries-over-lost-bitcoins-as-price-surges-past-us10000/news-story/bd18b6f6aa123dca017f9cc75544fd01.
[24] Yvo Desmedt and Sushil Jajodia. 2009. *Redistributing Secret Shares to New Access Structures and Its Applications*. Technical Report.
[25] Bitcoin devlopers. 2018. Optimized C library for EC operations on curve secp256k1. https://github.com/bitcoin-core/secp256k1/.
[26] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
[27] Pieter Wuille Eric Lombrozo, Johnson Lau. 2015. Segregated Witness. BIP 141, https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki.
[28] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-NG: A Scalable Blockchain Protocol. In *NSDI*.
[29] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. 2016. Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security. In *Applied Cryptography and Network Security (ACNS)*.
[30] Yoichi Hirai. 2018. Who can recover stuck funds on Ethereum? https://medium.com/@pirapira/who-can-recover-stuck-funds-on-ethereum-345ba7566c9c.
[31] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. In *HASP*.
[32] William C. Garrison III, Adam Shull, Steven Myers, and Adam J. Lee. 2016. On the Practicality of Cryptographically Enforcing Dynamic Access Control Policies

in the Cloud. In *IEEE Symposium on Security and Privacy (S&P)*.
[33] Mitsuru Ito, Akira Saito, and Takao Nishizeki. 1989. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* 72, 9 (1989), 56–64.
[34] Syed Roohullah Jan, Syed Tauhid Ullah Shah, Zia Ullah Johar, Yasin Shah, and Fazlullah Khan. 2016. An Innovative Approach to Investigate Various Software Testing Techniques and Strategies. *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN* (2016), 2395–1990.
[35] Gabriel Kaptchuk, Ian Miers, and Matthew Green. 2017. Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers. Cryptology ePrint Archive, Report 2017/201. https://eprint.iacr.org/2017/201.
[36] Jongkil Kim and Surya Nepal. 2016. A Cryptographically Enforced Access Control with a Flexible User Revocation on Untrusted Cloud Storage. *Data Science and Engineering* 1, 3 (2016), 149–160.
[37] Loi Luu and Nate Rush. 2017. PeaceRelay Merkle-Patricia Trie Proof Verification. https://github.com/loiluu/peacerelay/blob/48cab51e6638a6614c86299f1b880698631f8738/contracts/PeaceRelay.sol.
[38] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. 2015. Demystifying Incentives in the Consensus Computer. In *ACM Conference on Computer and Communications Security*. ACM, 706–719.
[39] Bill Marino and Ari Juels. 2016. Setting standards for altering and undoing smart contracts. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*. Springer, 151–166.
[40] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. http://eprint.iacr.org/2017/048.pdf.
[41] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP*.
[42] Richard Meissner. 2018. https://github.com/gnosis/safe-contracts/blob/48185fcac93e73a9a6b556648f60ea59f517435d/contracts/extensions/SingleAccountRecoveryExtension.sol.
[43] Mist. [n.d.]. https://github.com/ethereum/mist.
[44] Malte Möser, Ittay Eyal, and Emin Gün Sirer. 2016. Bitcoin Covenants. In *Financial Cryptography Bitcoin Workshop*.
[45] Prasad Naldurg and Roy H. Campbell. 2003. Dynamic access control: preserving safety and trust for network defense operations. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT-03)*. ACM Press, New York, 231–237.
[46] John P. Njui. 2018. Coinbase Custody Service Secures Major Institutional Investor Worth $20 Billion. https://ethereumworldnews.com/coinbase-custody-service-secures-major-institutional-investor-worth-20-billion/. *Ethereum World News* (2018).
[47] Ouriel Ohayon. 2018. The sad state of crypto custody. https://techcrunch.com/2018/02/01/the-sad-state-of-crypto-custody/.
[48] Nick Ortega. 2017. 'I Forgot My PIN': An Epic Tale of Losing $30,000 in Bitcoin. https://www.wired.com/story/i-forgot-my-pin-an-epic-tale-of-losing-dollar30000-in-bitcoin/.
[49] Charlie Osborne. 16 November 2017. Parity shakes up wallet audits, but funds remain frozen. *ZDNet* (16 November 2017).
[50] Rafael Pass, Elaine Shi, and Florian Tramèr. 2017. Formal Abstractions for Attested Execution Secure Processors. In *EUROCRYPT*.
[51] Alan Reiner. 2012. New wallet file ideas. https://bitcointalk.to/index.php?topic=128119.0.
[52] Reuters. 2017. Cryptocurrency Exchanges Are Increasingly Roiled With These Problems. http://fortune.com/2017/09/29/cryptocurrency-exchanges-hackings-chaos/
[53] Jeff John Roberts and Nicolas Rapp. 2017. Exclusive: Nearly 4 Million Bitcoins Lost Forever, New Study Says. http://fortune.com/2017/11/25/lost-bitcoins/
[54] Eric Schlosser. 2014. Always / Never. *The New Yorker* (Jan. 2014). https://www.newyorker.com/news/news-desk/always-never
[55] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.
[56] Jutta Steiner. 2017. Security is a process: A Postmortem on the Parity Multi-Sig Library Self-Destruct. https://blog.ethcore.io/security-is-a-process-a-postmortem-on-the-parity-multi-sig-library-self-destruct/.
[57] Douglas R. Stinson and Reto Strobl. 2001. Provably Secure Distributed Schnorr Signatures and a (t, n) Threshold Scheme for Implicit Certificates. In *ACISP*.
[58] Sean Stolberg. 2009. Enabling agile testing through continuous integration. In *Agile Conference*.
[59] Paul Syverson, R Dingledine, and N Mathewson. 2004. Tor: the Second Generation Onion Router. In *USENIX Security*.
[60] Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. 2017. Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge. In *EuroS&P*.
[61] Ruth Umoh. 2017. 3 of the craziest things people are doing to recover their lost bitcoin. https://www.cnbc.com/2017/12/21/3-crazy-things-people-are-doing-to-recover-lost-bitcoin.html.
[62] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*.
[63] Gnosis Multisig Wallet. [n.d.]. https://github.com/gnosis/MultiSigWallet.
[64] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. http://yellowpaper.io/
[65] Pieter Wuille et al. 2017. Schnorr signatures and signature aggregation. https://bitcoincore.org/en/2017/03/23/schnorr-signature-aggregation/.
[66] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P)*.
[67] Jia Yu, Fanyu Kong, Xiangguo Cheng, and Rong Hao. 2011. Two Protocols for Member Revocation in Secret Sharing Schemes. In *Intelligence and Security Informatics - Pacific Asia Workshop, PAISI 2011, Beijing, China, July 9, 2011. Proceedings*.
[68] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *ACM CCS*.
[69] Fan Zhang, Philip Daian, Iddo Bentov, and Ari Juels. 2018. Paralysis Proofs: Safe Access-Structure Updates for Cryptocurrencies and More. *IACR Cryptology ePrint Archive* 2018 (2018), 96.

$\mathcal{F}_{\mathbf{chain}}[\mathsf{succ}]$

```
1 :  Parameter: validity $\mathsf{succ} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, and $\Delta$.
2 :  On receiving* (init, genesis): storage := genesis
3 :  On receiving (read): output storage
4 :  On receiving (write, inp) from $P$:
5 :      send (write, inp, $P$) to $\mathcal{A}$ and start a timer of $\Delta$
6 :      block until $\mathcal{A}$ acknowledges or the timer fires
7 :      if succ(storage, inp) = 1 then
8 :          $t$ = clock(); storage := storage $\|$ ($t$, $P$, inp);
9 :          output (receipt, inp)
10 :     else output (reject, inp)
```

Figure 7: Ideal blockchain. The entry point marked with $*$ is only executed once. The parameter succ defines the validity of new items. A new item can only be appended to the storage if the evaluation of succ outputs 1.

A ADDITIONAL FORMALISM

Fig. 7 defines the ideal blockchain used in $\mathcal{F}_{\text{DASS}}$. Note that $\mathcal{F}_{\text{chain}}$ is more than a standard bulletin board because it captures the notion of transaction validity (by the succ function). Figs. 8 and 9 formally specify Π_{SGX} in Section 4.

B PURELY SCRIPT-BASED PARALYSIS PROOFS FOR BITCOIN

A Paralysis Proof mechanism can also be implemented without SGX (on the current Bitcoin mainnet), albeit with subpar security and more than exponential overhead.

Our construction utilizes the "life signal" method of Section 4. In the initial setup phase, each player P_i will prepare unsigned transactions $\{t_{i,j,k}\}_{j\in[N]\setminus\{i\},k\in[K]}$ that accuse P_j (these transactions are similar to t_1), and all players will sign transactions $t'_{i,j,k}$ that take UTXO_0 and the output of $t_{i,j,k}$ as inputs (these transactions are similar to t_2). K is a security parameter specifying the number of accusation attempts that can be made. Fig. 10 illustrates the transactions in the aforementioned scheme.

```
Protocol $\Pi_{\text{SGX}}$ with $P_1, \ldots, P_N$
Hardcoded: $\delta$ (e.g. $10^{-4}$), network latency $\Delta$
For any party $P_i$:
On receiving (init) from environment $\mathcal{Z}$:
(pk$_i$, sk$_i$) $\leftarrow\$$ KGen($1^n$); publish pk$_i$
wait to receive $\{\text{pk}_j\}_{j \neq i}$ from other parties
send (install, prog$_{\text{encl}}$) to $\mathcal{F}_{\text{SGX}}$ and wait to receive eid
send (eid, resume, init, $\{\text{pk}_i\}_{i=1}^N$) to $\mathcal{F}_{\text{SGX}}$ and publish pk$_{\text{SGX}}$
send (init, (($\delta$, pk$_{\text{SGX}}$), ($V$, (all pk $\in \{\text{pk}_i\}_{i=1}^N$) $\vee$ pk$_{\text{SGX}}$)) to $\mathcal{F}_{\text{chain}}$
if $\mathcal{F}_{\text{chain}}$ is not properly initialized: broadcast abort
else broadcast ok
wait to receive ok from others and abort if a abort is received
On receiving (access, addr$_{\text{new}}$) from environment $\mathcal{Z}$:
obtain UTXO$_{\text{fund}}$ from $\mathcal{F}_{\text{chain}}$
compute $\sigma$ = Sig(sk$_i$, (UTXO$_{\text{fund}}$, addr$_{\text{new}}$))
send (resume, (spend, $\sigma$, UTXO$_{\text{fund}}$, addr$_{\text{new}}$)) to $\mathcal{F}_{\text{SGX}}$
On receiving (migrate, $P'$) from environment $\mathcal{Z}$ :
assert $P' \subseteq \{\text{pk}_i\}_{i=1}^N$
obtain UTXO$_{\text{fund}}$ from $\mathcal{F}_{\text{chain}}$
send (resume, (migrate, UTXO$_{\text{fund}}$, $P'$)) to $\mathcal{F}_{\text{SGX}}$ and wait for $t_1$, $t_2$
send $t_1$, $t_2$ to $\mathcal{F}_{\text{chain}}$ // $t_2$ will be accepted $\Delta$ time after $t_1$
```

Figure 8: An SGX based protocol for Paralysis Proofs.

```
Program for the SGX Enclave (prog$_{\text{encl}}$)
Hardcoded: $\delta$, $\varepsilon$, network latency $\Delta$, access grace period $T_a$
On input* (init, $P_0$):
Parties := $P_0$
(sk$_{\text{SGX}}$, pk$_{\text{SGX}}$) $\leftarrow\$$ KGen($1^n$) and output pk$_{\text{SGX}}$
On input (spend, $\sigma$, UTXO$_{\text{fund}}$, addr$_{\text{new}}$)
parse UTXO$_{\text{fund}}$ as ($V$, (all pk $\in P$) $\vee$ pk$_{\text{SGX}}$) or abort
if received $|P|$ requests for (UTXO$_{\text{fund}}$, addr$_{\text{new}}$) within $T_a$:
  assert Vf($\sigma_i$, pk$_i$) for all $1 \leq i \leq n$
  sign transaction $t$ := $\langle$UTXO$_{\text{fund}} \rightarrow$ addr$_{\text{new}}\rangle$ with sk$_{\text{SGX}}$
  send $t$ to $\mathcal{F}_{\text{chain}}$
else store $\sigma$ and wait for more requests
On input (migrate, UTXO$_{\text{fund}}$, $P'$):
parse UTXO$_{\text{fund}}$ as ($V$, (all pk $\in P$) $\vee$ pk$_{\text{SGX}}$) or abort
(pk$_r$, sk$_r$) $\leftarrow\$$ KGen($1^n$)
$\phi_{\text{lifesignal}}$ := ((any pk $\in P \setminus P'$) $\vee$ (pk$_r$ $\wedge$ (CSV $\geq \Delta$)))
sign transitions $t_1$, $t_2$ with sk$_{\text{SGX}}$ and pk$_r$:
$t_1$ := $\langle(\delta$, pk$_{\text{SGX}}) \rightarrow (\varepsilon, \phi_{\text{lifesignal}})$, $(\delta - \varepsilon$, pk$_{\text{SGX}})\rangle$
$t_2$ := $\langle(\varepsilon, \phi_{\text{lifesignal}})$, ($V$, (all pk $\in P$) $\vee$ pk$_{\text{SGX}}$) $\rightarrow$
($V$, (all pk $\in P'$) $\vee$ pk$_{\text{SGX}}$)$\rangle$
output $t_1$ and $t_2$
```

Figure 9: The Paralysis Proof Enclave. The entry point marked with $*$ is only executed once.

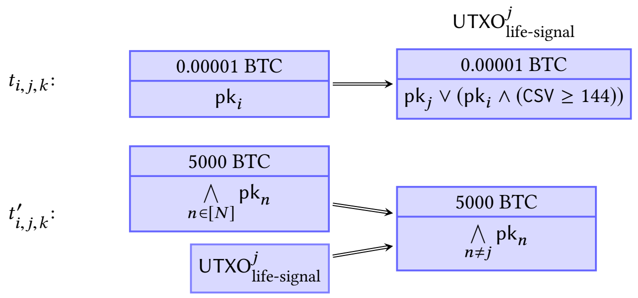

$\text{UTXO}^j_{\text{life-signal}}$
$t_{i,j,k}$:
0.00001 BTC
pk$_i$
0.00001 BTC
pk$_j \vee$ (pk$_i \wedge$ (CSV $\geq$ 144))
$t'_{i,j,k}$:
5000 BTC
$\bigwedge_{n \in [N]}$ pk$_n$
$\text{UTXO}^j_{\text{life-signal}}$
5000 BTC
$\bigwedge_{n \neq j}$ pk$_n$

Figure 10: Bitcoin-based Paralysis Proofs with N players (with public keys $\{\text{pk}_n\}_{n\in[N]}$). Each player P_i will prepare unsigned transactions $\{t_{i,j,k}\}_{j\in[N]\setminus\{i\},k\in[K]}$. All players will sign transactions $t'_{i,j,k}$.

This scheme can be implemented post-SegWit [27], where transaction hash (`txid`) excludes the `ScriptSig` witness. In particular, SegWit allows one to prepare $t'_{i,j,k}$ and condition its validity on that of unsigned $t_{i,j,k}$.

After every player receives all the signed transactions, the players will move the high-value fund into UTXO$_0$. This guarantees atomicity: either every player will have the ability to eliminate all the incapacitated player, or none of the player will have this ability. The output of $t_{i,j,k}$ requires a signature from P_j before the CSV timeout and a signature from P_i after the CSV timeout, and P_i may embed this signature into $t'_{i,j,k}$ after P_j failed to spend the output of $t_{i,j,k}$ on the blockchain. Since UTXO$_0$ requires the signatures of all parties, the only way to eliminate an incapacitated player is by using the signed transactions $t'_{i,j,k}$ that were prepared in advance.

The parameter K specifies the number of accusation attempts that can be made; hence a malicious player that pretends to be incapacitated more than K times will break this scheme. The SGX scheme does not exhibit this deficiency, because any player can send a fresh small amount of bitcoins to the enclave and thereby create an accusation transaction.

Furthermore, in order to support sequences of $\ell > 1$ incapacitated players, the N players will need to prepare in advance additional transactions that spend the outputs of $t'_{i,j,k}$ in order to eliminate another player, and so on. The scheme offers the most safety when $\ell = N - 1$, as this implies that any lone active player (i.e., all other players became incapacitated) will be able to gain control over the fund. The number of signed transactions that need to be prepared in advance is

$$f(\ell, N, K) \triangleq KN(N-1) \cdot K(N-1)(N-2) \cdots$$
$$K(N-\ell+1)(N-\ell) \geq \Omega(K^\ell N^\ell).$$

Thus, $\ell = N-1$ implies that $f(\ell, N, K)$ grows faster than $g(N) = 2^N$.