Paralysis Proofs: Secure Access-Structure Updates for Cryptocurrencies and More

Anonymous Author(s)

Abstract

Conventional (M, N)-threshold signature schemes leave users with a painful choice. Setting $M = N$ offers maximum resistance to key compromise. With this choice, though, loss of a single key renders the signing capability unavailable, creating *paralysis* in systems that use signatures for access control. Lower M improves availability, but at the expense of security. For example, $(3, 3)$-multisig wallet experiences access-control paralysis upon loss of a single key, but a $(2, 3)$-multisig allows any two players to collude and steal funds from the third.

In this paper, we introduce techniques that address this impasse by making general cryptographic access structures *dynamic*. Our schemes permit, e.g., a $(3, 3)$-multisig, to be downgraded to a $(2, 3)$-multisig if a player goes missing. This downgrading is secure in the sense that it occurs *only* if a player is *provably* unavailable.

Our main tool is what we call a *Paralysis Proof*, evidence that players, i.e., key holders, are unavailable or incapacitated. Using Paralysis Proofs, we show how to construct a *Dynamic Access Structure System*, which can securely and flexibly update target access structures *without a trusted third party* such as a system administrator. 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 define and show how to achieve critical security and usability properties (safety, liveness, and paralysis-freeness) in a DASS.

Paralysis Proofs can help address pervasive key-management challenges in many different settings. We present DASS schemes for three important example use cases: recovery of cryptocurrency funds should players become unavailable, returning funds to users when cryptocurrency custodians fail, and remediating critical smart-contract failures such as frozen funds. We report on practical implementations for Bitcoin and Ethereum.

1 Introduction

A common paradox in key management and access control systems is the "always/never" dilemma: systems must be always available when authorized, and never used when not [70]. In general, this trade-off between availability and potential for misuse exists in all key management or authentication systems, extending even to nuclear arms [28]. One simple example is the task of securing a single private key: replicate the key broadly across geography, machine architectures, and custodians, and the attack surface for key compromise increases. Store the key in a single secured location and the probability of loss increases, as do barriers to timely access. Generally, this challenging trade-off is navigated through organizational structures for distributing control to a small number of trusted and independent parties, thereby choosing the desired optimum point on the defensibility vs. availability spectrum. Sometimes, however, this is impossible. In a cryptocurrency wallet setting, for example, large direct incentives for theft combined with suspicion among users make distributing trust extremely difficult. Worse still, a number of access-control failures have occurred through both custodial failures [11, 61] and user errors [22, 63, 79].

Consider for example three players, Alice, Bob, and Carol, who jointly own a pool of cryptocurrency. For maximum protection against key compromise, they might use a $(3, 3)$-multisignature wallet ("multisig"). Spending the money would then require the signatures of all keys sk_A, sk_B and sk_C. This provides good security: an adversary would need to compromise all three keys to steal the money. If, however, just one key became unavailable, e.g., Alice lost sk_A, the money would be permanently lost. To provide availability should a key be lost, the players might instead use a $(2, 3)$-multisig. Unfortunately, in this case, any two players could cheat the third.

Neither choice seems optimal: it appears that these three players can realize *either* good security *or* good availability.

In this paper, we show how to achieve *both* properties and obtain a better security vs. availability trade-off than either multisig option alone. The key idea is to enable an multisig to be *conditionally downgraded*, e.g., to change a $(3, 3)$-multisig to a $(2, 3)$-multisig only if a player becomes unavailable. More generally, we show how to securely downgrade (or otherwise change) any *access structure* [37], i.e., policy determining which players can control a resource.

The main technical challenge in our work is ensuring that downgrading only happens when a player is truly unavailable. Otherwise, players can cheat by simulating the disappearance of a live player. Secure downgrading relies crucially on an ability to construct strong proofs of player incapacitation, what we call *Paralysis Proofs*.

Constructing and consuming Paralysis Proofs securely without a trusted third party (TTP) such as a system administrator, as we aim to do in this paper, is challenging, as explained below. We show how two new technologies, trusted execution environments (TEEs) and blockchains, together make it feasible for the first time.

Problem setting Paralysis Proofs help address a fundamental key-management challenge that is pervasive in cryptographic systems. Cryptocurrencies vividly illustrate the problem.

Private signing keys for cryptocurrencies embody direct and total control over funds. Key theft thus leads to immediately and irrevocably stolen money, and is therefore attractive for hackers. Over 980,000 Bitcoin (worth about $9.8 billion at the time of writing) [67] have been stolen from exchanges alone.

At the same time, lost keys—resulting from, e.g., accidental deletion or corruption—can also be catastrophic. It is estimated that nearly 4 million Bitcoin (worth around $4 billion at the time of writing), have been lost forever due to lost keys [68]. Unlike traditional online banking systems where lost credentials can be recovered via out-of-band mechanisms, the decentralized nature of cryptocurrencies makes direct recovery of credentials from these systems impossible.

The main approach to dealing with such problems in cryptocurrencies is multisig addresses or wallets [59]; more generally, secret sharing is a common technique [72]. These techniques enable general access structures in which a set of players holds key shares, and predetermined subsets of players can use their shares to access the target resource (e.g., spend cryptocurrency). The most common choice is a simple (M, N)-access structure (for $M \leq N$), which gives control to any subset of M players among a total set of N. For a given N, M dictates a trade-off between security and availability. Larger M means higher security, as more players must agree to access the resource, while smaller M means fewer players must agree, lowering per-response effort and implying higher availability.

Today's systems, however, only support *fixed* or *static* access structures (i.e., for all times τ_1, τ_2, any set of players that satisfies the access structure at time τ_1 also does so at time τ_2). When no choice of M offers a good security vs. availability balance, as in our example, users face an unresolvable dilemma.

Paralysis Proofs help resolve this dilemma. They enable secure *dynamic* access structures in what we call a Dynamic Access Structure System (DASS), a system that changes M securely as players become unavailable. We call this process access-structure *migration*, as access may be changed beyond simple downgrades.

Overview of our Approach Proving to a third party that a player is available is easy: Just have her sign a fresh message. But how can Bob prove that Alice *cannot* (or refuses to) sign a message, as opposed to Bob faking a failure and not communicating with Alice?

Our Paralysis Proof constructions leverage the censorship-resistance [27] and data persistence of a public blockchain to detect and record the fact of an unavailable player. If, e.g., Alice disappears, a challenge can be issued to her *on chain*. Public blockchains are censorship-resistant, in the sense that if Alice tries to post a response within Δ epochs (blocks) for some suitable Δ she can do so with high probability even in the face of powerful network adversaries. Thanks to this property, lack of response from Alice within Δ blocks of a challenge constitutes a Paralysis Proof showing Alice's unavailability.

Given a TTP, it is then easy to migrate an access structure securely. For example, our three players might use a $(3, 3)$-multisig, with sk_A, sk_B and sk_C held by the TTP. Given a Paralysis Proof showing Alice has disappeared, the TTP could release sk_A to Bob or Carol, effectively downgrading to a $(2, 3)$-multisig.

We aim in our work, however, *to avoid TTPs*. Our DASS schemes therefore rely on two technologies that serve as trust anchors by emulating TTPs: *smart contracts* where available, and *trusted execution environments (TEEs)* such as Intel SGX, where they are not.

Example applications We explore three illustrative use cases for Paralysis Proofs, exploring forms of paralysis:

- *Cryptocurrency key loss:* Generalizing our running example, we present protocols that permit any (M, N)-multisig cryptocurrency scheme to be downgraded (i.e., have lower M) if and only if keys or players go missing. We report on a purely smart-contract based Ethereum implementation and one for Bitcoin that involves use of a TEE. The Bitcoin scripting language, despite its time-based opcodes, cannot support a Paralysis Proof System; we present some script-based alternative schemes, however, that are less secure than our main scheme and/or require proposed enhancements to Bitcoin.
- *Cryptocurrency custody failures:* The challenges of key maintenance and endpoint security make third-party custody solutions appealing to users. Custodians can themselves become paralyzed, though, and can also freeze specific users' funds. We propose a DASS which, if a custodian is failing to honor a valid withdrawal request for any reason, allows a user to migrate control of her funds to a backup key. Because this key is not usable for spending absent such a proof, the user is not trusting her own endpoint's key management security by default.
- *Smart contract failures:* Smart contract bugs can result not just in exploits, but in funds being permanently paralyzed. The famous (second) Parity Multisig Wallet bug is just one example that permanently froze approximately $150 million in Ether in 2017 [64]. 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. If (and only if) fund paralysis occurs, a paralysis proof submission triggers an "escape hatch" [51], failover logic that refunds or moves a smart contract's assets. Our techniques are general, and can help address other non-paralysis smart contract failures.

We emphasize that Paralysis Proofs are general and can be applied beyond cryptocurrency, e.g. for (M, N)-sharing of decryption keys.

We also emphasize that a TEE-based application can, of course, store a master key or all players' keys and directly mediate all access requests by players. Our schemes, however, avoid placing a TEE on the critical path for ordinary transactions, preventing vulnerability to denial-of-service and service or hardware failures.

Contributions

In summary, our main contributions are as follows:

- *Paralysis Proofs:* We introduce the concept of Paralysis Proofs, and show how to achieve them using blockchains. We also introduce Dynamic Access Structure Systems (DASSes), which combine Paralysis Proofs with a trust anchor (smart contract or TEE) to enable secure access-structure migration. They thereby offer security vs. availability trade-offs unachievable in conventional, static access-structure systems.
- *Formal definitions and framework:* We formally define key properties for a DASS (*liveness*, *safety*) and its underlying migration

policy, called a DASP (*privilege-preserving*, *paralysis-free*) (Section 3). We also present a general Universal-Composability-type ideal functionality that formally specifies security properties required for a broad range of applications (Section 4).

- *Applications:* We present three example applications: Cryptocurrency key loss (Section 4), cryptocurrency custody failures (Section 5), and smart contract failures (Section 6).
- *Implementation:* We present implementations in Ethereum and Bitcoin for the first application, using smart contracts and TEEs (Intel SGX, in particular) respectively. We offer a UC proof (sketch) of security for the latter. To illustrate the limitations of pure blockchain approaches for Bitcoin, we also explore script-based schemes in the paper appendix.
- *TEE compromise:* We explore alternative DASS designs that provide resilience to TEE compromise, such as through side-channel attacks demonstrated against SGX (Section 4.5).

2 Background

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

Trusted Execution Environments and SGX A Trusted Execution Environment (TEE) is a black-box-like execution environment that provides confidentiality and integrity for applications running on potentially malicious hosts.

Intel Software Guard EXtensions (SGX) [6, 34, 36, 54] is a realization of TEEs as a new instruction set architecture extension enabled on most new Intel processors. 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. Enclaves cannot make system calls, but can communicate with untrusted programs running in the host OS. As a result, an enclave depends upon a potentially malicious operating system for network and file system operations. Enclaves can therefore secure applications' state and execution, but cannot ensure successful network communications or file accesses. Despite these limitations, enclaves are powerful tools for building a variety of functionalities [21, 40, 89].

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 *UTXOs* or Unspent Transaction Outputs. One can check if a transaction 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 invariants of a monetary system and prevents forgery.

Beyond monetary invariants, Bitcoin must also handle access control. `script_pubkey` and `script_sig` are the authorization mechanisms for transactions that ensure funds cannot be stolen or misused. Typically the `script_pubkey` in an output 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. These requirements are represented in a stack based language known as *Bitcoin script*. 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 checks.

Smart Contracts Smart contracts are small, deterministic programs that are stored in a blockchain system's state and interpreted by a virtual machine. Beyond the *value* field associated with a simple currency transaction, transactions on smart contract blockchains contain two additional key parameters: *input data* and *code location* [85]. To process a transaction, a smart contract system looks up the code stored in the provided location, executing the code with the provided input data and processing any side effects output by the execution of the code. A key differentiator between such smart contract-enabled systems and simpler script like Bitcoin script is the capability of contract platforms to provide *rich statefulness* [83]. In a system providing rich statefulness, all executing transactions have native access to persistent state (stored across transactions, blocks, and time), and can interact with and update both local state for data storage and global blockchain state for system-wide information (like the current height of the blockchain, or the hash of the block the transaction eventually gets mined in). Rich statefulness is particularly relevant to our system, as access to state across time is required to track whether a Paralysis Proof has been initiated, is pending, or has been responded to. It is Bitcoin's lack of such native statefulness that makes trusted hardware the only practical solution for handling paralysis with large numbers of users.

3 Dynamic Access Structure Systems

In this section, we develop formal definitions and framework that we use to reason about the security of Dynamic Access Structure Systems.

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.

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. We now provide formalism for DASP specification (Section 3.1), followed by security definitions for a DASS (Section 3.2).

3.1 Specification of a Dynamic Access Structure Policy

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 when 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. 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 $s(L) \to \{\texttt{true}, \texttt{false}\}$ 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., $s(L) = \texttt{true}$ and $L \subseteq L'$ together imply that $s(L') = \texttt{true}$. A migration rule $m_{s_i,s_j} \in \mathcal{M}$ is a function $m_{s_i,s_j}(L) \to \{\texttt{true}, \texttt{false}\}$ that determines whether migrating from s_i to s_j is permitted if the set of live parties is L. We use $s_i \xrightarrow{L} s_j$ to denote $m_{s_i,s_j}(L) = \texttt{true}$.

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 $L \subseteq L_0$, denoted $s_1 \overset{L}{\rightsquigarrow} s_n$, if there exists a path $(s_1, s_2, \ldots, s_n)$ in G such that for all $i \in [1, n-1]$, $m_{s_i,s_{i+1}}(L) = \texttt{true}$.

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 3.1. The set of *least permissive* access structures for $L \subset \{P_i\}$, denoted by $S_{\text{LP}}(L)$, is as follows:

$$S_{\text{LP}}(L) = \{s \in \mathcal{S} : s(L) = \texttt{true} \wedge (\forall L' \subsetneq L, s(L') = \texttt{false})\}.$$

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 the above definition, we have the priviledge-preseving is defined as follows.

Definition 3.2. (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 $L \nsubseteq L'$ at any time t. Formall, $\forall s \in \mathcal{S}$ such that $s(L_t) = \texttt{true}$:

$$\forall s' \; s.t. \; s \xrightarrow{L_t} s', s' \in \bigcup_{L_t \subseteq L} S_{\text{LP}}(L).$$

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

Definition 3.3. (*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 $s(L_t) = \texttt{false}$:

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

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 that can get the system out of paralysis, then the DASP should permit a transition to that access structure. However, if the set of live parties is too sparse to satisfy any of the prescribed access structures, then the availability cannot be achieved.

Example 3.4. 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

$$s_0 = \mathbb{I}_{\mathcal{P}}$$
$$s_i = \mathbb{I}_{\mathcal{P}_{-i}}, 1 \leq i \leq N$$

and the condition $m_{s_0,s_i} \in \mathcal{M}$ is fulfilled for $L_t = \mathcal{P}_{-i}$.

According to Definition 3.2 and Definition 3.3, the DAS in Example 3.4 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 formally define the security of a DASS with a Universal Composability (UC) [19] 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. Byzantine corruption). We assume that the adversary has complete control of the network, with the exception that a blockchain is available to all parties, i.e. is censorship-resistant, and the maximum network latency to the blockchain is bounded by a known Δ.

3.2.2 Ideal Functionality. We specify the security goals of a Dynamic Access Structure System in the ideal functionality $\mathcal{F}_{\text{DASS}}$ defined in Figure 1.

To reduce clutter, we omit the handling of session IDs [19] 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 [19] to model the power of the network adversary. Specifically,

The Ideal Functionality of a Dynamic Access Structure System
$\mathcal{F}_{\text{DASS}}[s_0, \mathcal{R}, \mathcal{S}, \mathcal{M}]$ with parties $\{P_i\}_{i=1}^N$

1 : **On receiving**$*$ (init) from any P_i:
2 : $L_t := \{P_i\}_{i=1}^n$, $s := s_0$
3 : **On receiving** (paralysis, P_i) from $\mathcal{A}$:
4 : $L_t = L_t \setminus \{P_i\}$
5 : **On receiving** (access, inp) from $P_i \in L_t$:
6 : let current time be t
7 : // if there is an unexpired access reqeuest for inp
8 : **if** find a stored (inp, $\mathcal{P}$, T_0) and $t < T_0 + \Delta_a$ **then**:
9 : add P_i to $\mathcal{P}$
10 : // if no access reqeuest for inp or it has expired, create a new one
11 : **else** : store (inp, $\{P_i\}$, t), overwriting (inp, _, _) if exists
12 : **if** $s(\mathcal{P})$ = `true` **then**:
13 : send a public delayed output $\mathcal{R}(\mathcal{P}$, inp) to all parties in $\mathcal{P}$
14 : **On receiving** (migrate, s') from $P_i \in L_t$:
15 : assert $(s' \in \mathcal{S} \wedge m_{s,s'} \in \mathcal{M})$
16 : $L_{\text{fake-death}} = \emptyset$
17 : **for** all corrupted parites $P_c \in L_t$:
18 : ask $\mathcal{A}$ if P_c choose to pretend to be paralyzed; if so add P_c to $L_{\text{fake-death}}$
19 : **if** $m_{s,s'}(L_t \setminus L_{\text{fake-death}})$ = `true` **then**:
20 : send a public delayed output (s, s', P_i, ok) to all parties; $s = s'$

Figure 1: The ideal functionality of a Dynamic Access Structure System. The entry point marked with $*$ is only executed once.

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 the currently enforced access structure respectively. To capture the paralysis explicitly, we extend the standard corruption model [19] with a special "paralysis" corruption. Upon receipt a paralysis message from $\mathcal{A}$, a party immediately announces its paralysis and halt until the end of the protocol. In the ideal protocol, $\mathcal{A}$ sends (paralysis, P_i) to $\mathcal{F}_{\text{DASS}}$, who then 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. $s(P)$ = `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'.

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, then $\mathcal{F}_{\text{DASS}}$ 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 $s(L)$ = `true`.
- A transition to $s' \neq s$ occurs only if $m_{s,s'}(L_t)$ = `true`.

Liveness:

- If $s(L)$ = `true` for some $L \subseteq L_t$, then L can access $\mathcal{R}$ within Δ time after interacting with the DASS honestly.
- If $m_{s,s'}(L_t)$ = `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 3.4, 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 for $i > 0$ 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.

3.2.4 DASSes and Paralysis Proofs.

The main challenge in realizing $\mathcal{F}_{\text{DASS}}$ is to determine the set of live parties L_t in a trustworthy way. Our solution to that is Paralysis Proofs. Specifically, a DASS realizing $\mathcal{F}_{\text{DASS}}$ leverages the censorship-resistance of blockchain to enable parties to construct Paralysis Proofs to prove that $P_i \notin L_t$ for a given party P_i or similar facts about L_t. In particular, as we shall see, to prove that $P_i \notin L_t$, parties issue to P_i a challenge on the blockchain. If P_i does not respond within some time Δ, the challenge together with evidence of this failure to respond constitute a Paralysis Proof one that prove $P_i \notin L_t$.

4 Paralysis Proofs for Cryptocurrency

In this section, we explore the use of Paralysis Proofs to recover from cryptocurrency key loss (and related failures, e.g., player disappearance), as in our motivating example in the paper introduction. We focus on Bitcoin, which presents particular technical challenges. For comparison, we also briefly present a conceptually straightforward scheme for Ethereum.

It is challenging to implement secure Paralysis Proofs compatible with the current Bitcoin protocol because of the limited expressiveness of Bitcoin scripts. (We show in Appendix D that it is possible were a proposed feature called "covenants" available [57].) We therefore explore Paralysis Proofs for Bitcoin using TEEs—specifically, Intel SGX, a powerful TEE available in existing CPUs. Readers can refer to Section 2 for background on SGX.

We give technical preliminaries and discuss our trust model in Section 4.1. We present our main protocol (denoted Π_{SGX}) in Section 4.2, and discuss its security in the Universal Composability framework in Section 4.4, giving a proof (sketch) in the paper appendix. We discuss ways to reduce trust assumptions for SGX nodes in Section 4.5. For comparison, we present our basic Dynamic Access Structure System for Ethereum in Section 4.6.

4.1 Preliminaries

Bitcoin Transaction and CSV. In Bitcoin, spendable money is known as an Unspent Transaction Output (UTXO). We use (V, ϕ) to denote an UTXO of V coins and script ϕ. The script ϕ stipulates the condition to be satisfied in order to spend the UTXO. 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.

An essential ingredient of Π_{SGX} is Bitcoin's relative timeout script instruction, also known as `CheckSequenceVerify` or `CSV` [16]. 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. This assertion can be part of a conditional script, such that other branches of the script do not need to satisfy the CSV condition.

SGX and Attested Execution. Throughout the paper, we use SGX as a concrete building block. However, our protocol can be realized by any TEE that protects the confidentiality and integrity of computation, and can issue proofs, known as *attestations*, of computation correctness.

In our formal specification, we adopt the (local) ideal functionality $\mathcal{G}_{\mathrm{SGX}}$ by Pass et al [65] to model SGX. 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 $\mathcal{G}_{\mathrm{SGX}}.\mathsf{getpk}()$. We refer readers to [65] for details.

Ideal Blockchain. Our protocol relies on an append-only ledger. We define the ideal functionality $\mathcal{F}_{\mathrm{blockchain}}[\mathrm{succ}]$ in Figure 2 (inspired by [20]) to model a general-purpose append-only ledger implemented by common blockchain protocols. The parameter $\mathrm{succ}(\mathrm{history}, \mathrm{item}) \to \{0,1\}$ is a function that specifies the criteria for a new item to be appended to history, modeling the notion of transaction validity. We retain the append-only property of blockchains but abstract away the inclusion of items in blocks.

We assume a trustworthy time source available to $\mathcal{F}_{\mathrm{blockchain}}$ and items are timestamped when added. In practice, block numbers can serve as such timestamps. We also assume the blockchain is *censorship-resistant*, namely messages sent to $\mathcal{F}_{\mathrm{blockchain}}$ will be delivered within Δ time. In practice, this requires each party to have reliable access to the peer-to-peer network.

Trust assumptions. To summarize, our protocol relies on TEE with attestation that protects the confidentiality and integrity of computation, and an append-only, and censorship-resistant ledger. 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 the SGX properties be broken. We discuss ways to minimize the trust in SGX in Section 4.5.

$\mathcal{F}_{\mathrm{blockchain}}[\mathrm{succ}]$

Parameter: successor relationship succ : $\{0,1\}^* \times \{0,1\}^* \to \{0,1\}$
On receiving$*$ (init, genesis): storage := genesis
On receiving (read): output storage
On receiving (write, inp) from P:
  send (write, inp, P) to $\mathcal{A}$
  if succ(storage, inp) = 1 **then**
    t = clock(); storage := storage $\|$ (t, P, inp);
    output (receipt, inp)
  else output (reject, inp)

Figure 2: 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.

Protocol Π_{SGX} with $P_1, \ldots, P_N$

Hardcoded: δ (e.g. 10^{-4}), network latency Δ
For any party P_i:
On receiving (init) from environment $\mathcal{Z}$:
  $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow\$ \mathsf{KGen}(1^n)$; publish pk_i
  wait to receive $\{\mathsf{pk}_j\}_{j \neq i}$ from other parties
  send (install, $\mathsf{prog}_{\mathsf{encl}}$) to $\mathcal{G}_{\mathrm{SGX}}$ and wait to receive eid
  send (eid, resume, init, $\{\mathsf{pk}_i\}_{i=1}^{N}$) to $\mathcal{G}_{\mathrm{SGX}}$ and wait for $\mathsf{pk}_{\mathrm{SGX}}$; publish $\mathsf{pk}_{\mathrm{SGX}}$
  send (init, (δ, $\mathsf{pk}_{\mathrm{SGX}}$), ($V$, (all pk $\in \{\mathsf{pk}_i\}_{i=1}^{N}$) $\vee$ $\mathsf{pk}_{\mathrm{SGX}}$)) to $\mathcal{F}_{\mathrm{blockchain}}$
  if $\mathcal{F}_{\mathrm{blockchain}}$ 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, $\mathsf{addr}_{\mathrm{new}}$) from environment $\mathcal{Z}$:
  obtain $\mathsf{UTXO}_{\mathrm{fund}}$ from $\mathcal{F}_{\mathrm{blockchain}}$
  compute $\sigma = \mathsf{Sig}(\mathsf{sk}_i, (\mathsf{UTXO}_{\mathrm{fund}}, \mathsf{addr}_{\mathrm{new}}))$
  send (resume, (spend, σ, $\mathsf{UTXO}_{\mathrm{fund}}$, $\mathsf{addr}_{\mathrm{new}}$)) to $\mathcal{G}_{\mathrm{SGX}}$
On receiving (migrate, P') from environment $\mathcal{Z}$:
  assert $P' \subseteq \{\mathsf{pk}_i\}_{i=1}^{N}$
  obtain $\mathsf{UTXO}_{\mathrm{fund}}$ from $\mathcal{F}_{\mathrm{blockchain}}$
  send (resume, (migrate, $\mathsf{UTXO}_{\mathrm{fund}}$, P')) to $\mathcal{G}_{\mathrm{SGX}}$ and wait for t_1, t_2
  send t_1, t_2 to $\mathcal{F}_{\mathrm{blockchain}}$ // t_2 will be accepted Δ time after $\mathcal{F}_{\mathrm{blockchain}}$ accepts t_1

Figure 3: An SGX based protocol for Paralysis Proofs.

4.2 Protocol Details

We denote our DASS for Bitcoin by Π_{SGX}. We give text descriptions below of the steps involved in Π_{SGX}, so the general reader need not reference formal protocol or ideal-functionality specifications to understand the workings of our scheme.

Π_{SGX} is formally specified in Figure 3. As we prove, Π_{SGX} UC-realizes the ideal functionality $\mathcal{F}_{\mathrm{DASS}}$ in the $(\mathcal{G}_{\mathrm{SGX}}, \mathcal{F}_{\mathrm{blockchain}})$ hybrid model. Figure 4 gives the logic of the SGX enclave, the application running on $\mathcal{G}_{\mathrm{SGX}}$. The successor function in $\mathcal{F}_{\mathrm{blockchain}}$ implicitly models the logic of Bitcoin scripts.

Notation. Let N be the number of players at the start of Π_{SGX}. We denote each player as P_i for $i \in \{1, 2, \ldots, N\}$. 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 entire set of players.

Program for the SGX Enclave ($\text{prog}_{\text{encl}}$)

Hardcoded: δ, ε, network latency Δ, access grace period T_a

On input$*$ (init, P_0):

Parties := P_0

$(\text{sk}_{\text{SGX}}, \text{pk}_{\text{SGX}}) \leftarrow\$ \text{KGen}(1^n)$ and output pk_{SGX}

On input (spend, σ, $\text{UTXO}_{\text{fund}}$, addr_{new})

parse $\text{UTXO}_{\text{fund}}$ as $(V, (\text{all pk} \in P) \vee \text{pk}_{\text{SGX}})$ or abort

if received $|P|$ requests for $(\text{UTXO}_{\text{fund}}, \text{addr}_{\text{new}})$ within T_a:

  assert $\text{Vf}(\sigma_i, \text{pk}_i)$ for all $1 \le i \le n$

  sign transaction $t := \langle \text{UTXO}_{\text{fund}} \rightarrow \text{addr}_{\text{new}} \rangle$ with sk_{SGX}

  send t to $\mathcal{F}_{\text{blockchain}}$

else store σ and wait for more requests

On input (migrate, $\text{UTXO}_{\text{fund}}$, P'):

parse $\text{UTXO}_{\text{fund}}$ as $(V, (\text{all pk} \in P) \vee \text{pk}_{\text{SGX}})$ or abort

$(\text{pk}_r, \text{sk}_r) \leftarrow\$ \text{KGen}(1^n)$

// a life signal for players affected by the migration

$\phi_{\text{lifesignal}} := ((\text{any pk} \in P \setminus P') \vee (\text{pk}_r \wedge (\text{CSV} \ge \Delta)))$

sign transitions t_1, t_2 with sk_{SGX} and pk_r:

$t_1 := \langle (\delta, \text{pk}_{\text{SGX}}) \rightarrow (\varepsilon, \phi_{\text{lifesignal}}), (\delta - \varepsilon, \text{pk}_{\text{SGX}}) \rangle$

$t_2 := \langle (\varepsilon, \phi_{\text{lifesignal}}), (V, (\text{all pk} \in P) \vee \text{pk}_{\text{SGX}}) \rightarrow (V, (\text{all pk} \in P') \vee \text{pk}_{\text{SGX}}) \rangle$

output t_1 and t_2

Figure 4: The Paralysis Proof Enclave. The entry point marked with $*$ is only executed once.

Initialization. To start the protocol, some honest party needs to load an SGX instance with $\text{prog}_{\text{encl}}$ and invoke the init procedure. For now we assume a single SGX available for all honest parties; thus any honest party can initiate the enclave (once initialized, sequential initializations will be ignored). In Section 4.5 we present a distributed setup procedure that avoids the availability assumption and provides stronger guarantee.

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 that can be spent by either $\{\text{pk}_i\}_{i=1}^N$ 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 Figure 3). Hence, even in the case that all of the 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 (see also Appendix E).

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, and the set of shareholders that controls the fund will be reduced only if no appeal occurred.

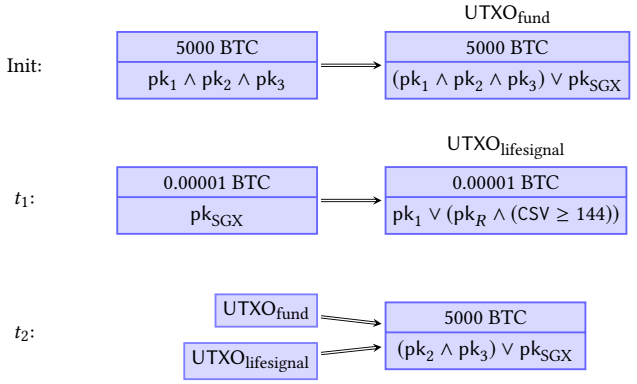

Figure 5: Example of Π_{SGX} with three players and P_1 accused of being incapacitated.

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 Figure 4 and examplified in Figure 5), as follows:

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

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. Figure 5 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'll discuss shortly, this is an important security feature because it makes Π_{SGX} more resilient to SGX's availability failure and avoids complexity of SGX with a blockchain.

4.3 Implementation

We implemented Π_{SGX} using Intel SGX SDK and Bitcoin Core. The source code is published at [2]. Our trusted functions contribute 874 lines of C++ code. The entire Trusted Computing Base (TCB) includes the Bitcoin Core implementation, two widely used cryptographic libraries (i.e., libsecp256k1 and OpenSSL), and the Panoply implementation [74].

4.4 Security of Π_{SGX}

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

Use of relative timeout. 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 [16] 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.

Note that a fresh, ephemeral key pair 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 store these ephemeral keys, as they are consumed right after generation.

No need to sync SGX with a blockchain. It is important to point out that the security of Π_{SGX} does not require the SGX enclave to have an up-to-date view of the blockchain (in fact it does not require any view of the blockchain), nor does it require a trusted clock. By contrast, protocols that require so have a larger attack surface, and in particular such protocols need additional security measures in order to be protected against rollback attacks (see, e.g., [10, 52]) and the problems of SGX clock (see, e.g. [20]).

In Appendix G we give a similar Paralysis Proof system that works with the current Bitcoin protocol and does not require SGX, but the construction has a weaker security guarantee and more than exponential overhead.

Security Proof. The security of Π_{SGX} is formally analyzed using the framework developed in Section 3. Specifically, we formulate the security goal of Π_{SGX} as a DASP, and then prove in the University Composability (UC) framework that Π_{SGX} securely realizes the ideal functionality that implements the same DASP.

Specification of the DASP The resource being managed by the Π_{SGX} is access to an oracle $\mathcal{R}(\cdot)$ that produces valid signatures authorizing Bitcoin expenditures. Initially, a Bitcoin fund is controlled by a set of N parties, denoted by P_0. Π_{SGX} aims to implement the following DASP:

$$\mathcal{S} := \{s_P(\cdot) = \mathbb{I}_P(\cdot) : \forall p \in \mathcal{P}(P_0)\} \text{ and}$$
$$\mathcal{M} := \left\{m_{s_A,s_B}(\cdot) = \mathbb{I}_B(\cdot) : A, B \subseteq P_0, B \subsetneq A\right\}.$$

We prove that Π_{SGX} realizes (formally, UC-realizes) the ideal functionality $\mathcal{F}_{\text{DASS}}[s_{P_0}, \mathcal{R}, \mathcal{S}, \mathcal{M}]$ that enforces this DASP. In particular, we prove the following theorem:

Theorem 4.1 (The Security of Π_{SGX}). *Assume $\mathcal{G}_{\text{SGX}}$'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 $(\mathcal{G}_{\text{SGX}}, \mathcal{F}_{\text{blockchain}})$-hybrid model, for static adversaries.*

Proof. See Appendix B for a proof sketch. □

4.5 Minimizing Trust in TEE

We now briefly consider some ways to minimize the trust placed in the TEE (SGX node) 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 in a power cycle. 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 the other players.

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 $\left\{\text{pk}_{\text{SGX}\,i}\right\}_{i=1}^{N}$. Finally, the enclaves then use $\left\{\text{pk}_{\text{SGX}\,i}\right\}_{i=1}^{N}$ to establish secure channels (TLS) with each other, and create a fresh shared secret key sk_{SGX} that is associated with $\left\{\text{pk}_{\text{SGX}\,i}\right\}_{i=1}^{N}$ (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 load and run the backup when needed.

Side-channel Resistance. Although SGX aims to provide confidentiality, recent work has uncovered data leakage via side-channel attacks [14, 31, 32, 39, 46, 47, 56, 62, 71, 84, 87]. 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 signatures generation (besides key generation)—which makes it more amenable to software-level side-channel mitigations, such as constant-time ECDSA implementation (e.g. [24]).

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[1] in SGX, 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 [20]). 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 $UTXO_{fund}$. 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.

4.6 Paralysis Proofs via Ethereum

An Ethereum implementation of the ideal functionality $\mathcal{F}_{DASS}$ is straightforward. Our reference implementation of a paralysis-free multisig wallet consists of 156 lines of commented Solidity code, and its main logic is shown in Figure 7.

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 before any contract action.

[1] which SGX doesn't offer, as confirmed by SGX SDK developers at `https://github.com/intel/linux-sgx/issues/161`.

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).

The full contract code, including the logic for pruning incapacitated signers and updating the signature threshold is published at [3].

5 Custodial Paralysis

Until now, this work has considered paralysis in a generalized setting– any setting in which multiple parties must authorize a transaction. Consequently, we have adopted a very narrow definition of paralysis as unavailability of a party expected to be online. Without a specific definition of what constitutes a user or a valid transaction, there is no distinction between a party who honestly refuses to sign an illegitimate message and one who maliciously paralyzes a legitimate transaction. Unfortunately this makes it impossible to construct a mechanism that distinguishes between the two. In some settings, however, the relationship between parties and their separation of responsibilities with regards to transaction signing are more well-defined.

In this section, we consider one such specific scenario and analyze the consequences of paralysis: we consider a centralized custodian, effectively a bank, responsible for providing access control to a user's funds. Here the goal is not to downgrade access if the bank becomes unavailable or paralyzed, but rather to migrate to a completely distinct recovery policy when the custodian fails to perform its assigned duties or either. Unlike in the previous setting, keys are not equal in privilege and come with defined roles.

Why centralized custodians? Centralized service providers often offer better security than a user is capable of providing on their own, and can relieve users from the burdens of key management and storage. Centralized providers can also offer layered security services including sophisticated access control, two-factor authentication, account compromise detection, account recovery options, and account insurance. Adding to these benefits, such services are convenient in their availability and do not require their users to purchase custom or dedicated hardware. Ease of key management is a major consideration for many users when choosing so called "web wallets" [42] and such services have seen remarkable success with one US-based service Coinbase at times claiming to control up to 10% of Bitcoins in active circulation [8].

Centralized custodians, however, are a single point of failure for both security and availability: they can steal funds directly or simply disappear. A number of such cases of fraudulent or otherwise insolvent services leading to losses for their users have been observed in practice [1, 73]. Such trust issues can be resolved by instead layering private key security, requiring the custodian to sign off on all transactions in a 2-of-2 multisignature scheme. This protects the user from their funds being stolen by the exchange, but does not ensure the user's funds are available to spend. It does not ensure users funds cannot be paralyzed, either maliciously or accidentally, by the failure of the centralized custodian to sign transactions in a timely fashion.

Paralysis-proof Custodians In our setting, we assume a user entrusts a centralized custodian with either a key that directly controls their funds or is part of a mullti-signature address. The custodian is responsible for authenticating the user before it authorizes any transaction with its key. Separately, the user stores a recovery key (e.g. on paper in a safety deposit box). Optimistically, the recovery key will never be required, as the custodian will remain available. If the recovery key had full control over the funds, then the custodian would offer little additional security and the funds would only be as safe as the recovery key. Instead, the recovery key is inert, controlling no funds, and can only obtain funds through a paralysis proof of the centralized custodian. If the custodian is paralyzed, control is migrated from the exchange to the recovery key.

While basic paralysis proofs would guard against unavailability of the custodian, they would not guard against a malfeasant custodian that intentionally blocks legitimately authorized transactions. To resolve this, we extend the functionality of basic paralysis proofs to included a predicate that must be met for a given party to issue a life signal. In our case, the predicate will ensure a life signal is issued only if the custodian faithfully attempted to authenticate the user and transaction and the authentication failed . The exact details of this predicate can vary depending on the authentication mechanism. We explore the set of custodian functionalities that are paralysis proof compatible.

Mechanisms for Authentication and Secondary Authentication To authenticate to the custodian, the client will need to participate in a possibly interactive authentication protocol with messages passed from the client to the custodian and potential back. As a simple example, a standard password authentication protocol would require the client to send the custodian the password. For a paralysis proof against a malicious custodian, the client needs to demonstrate that they did actually submit such a message and conversely, an honest custodian defending against a false paralysis proof needs to show no such message was sent or the message was invalid. Because the custodian is realized in an Trusted Execution Environment, we need not contend with the correctness of the messages themselves: we can rely on the TEE to produce correct messages and accept valid messages, we simply need a mechanism for assured communication between the enclave and user.

Following the techniques of [40], any interactive authentication protocol can be realized, "on-chain," 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. For example in the password authentication protocol, a paralysis proof can be realized by posting the password, encrypted under a key owned by the enclave, in a challenge transaction. In responding to the challenge, the enclave sees the included password and if and only if the password incorrect, will contest the paralysis proof. This can be applied n round protocol by posting each message, encrypted, to the blockchain and extended to, for example, integrating federated authentication protocols or challenge response based second factor authentication mechanisms.

Third Party Authentication Another option is for the enclave to contact third parties directly to provide primary or a secondary authentication factor. Town Crier [89] 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 for authentication via services such as Authy or Twillo, or ensure that a user still has an account with some service. Many of these mechanisms depend on trusting the third party service, but in the case of established and widely used services this may be more palatable than trusting the custodial service itself.

Account Recovery and Dual Paralysis Using the same mechanisms for interactive protocols or authenticating via third parties, the custodian can provide a paralysis proof compatible mechanism for account recovery in the case of lost credentials. 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 the recovery key.

6 Smart Contract Proof-Of-Paralysis

Throughout this work we have explored the relationship between smart contracts and Paralysis Proofs, using, e.g., smart contracts to implement Paralysis Proofs in Section 4.6. In the broader smart contract community, it is well known that paralysis can occur within smart contracts themselves. A classic example are the two well-publicized Parity multisignature vulnerabilities [15] [75] which together 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 [18] enumerated a number of vulnerable contracts with stuck funds and denial-of-service vulnerabilities. Some of these vulnerabilities are subtle, involving low-level platform details like the "gas" model for how computation is priced per-transaction [85].

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

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
  // (where π_i represents the code test i should run in state σ_i for expected output ω_i)
  where $\sigma_i = \{(s_1, v_1), \ldots, (s_j, v_j)\}$ // storage keys/values for test setup
  γ, β, // β is mapping from block hash to height, γ holds latest blockchain height
  Δ, // maximum time to construct a proof, in blocks
  $C = [C_P, C_R]$ // addresses of main and recovery contract
On input (access, d):
  Return $C[0](d$, msg.sender)
On input (migrate, $(i, p = \{(s_1, v_1, p_1), \ldots, (s_k, v_k, p_k)\})$):
  Assert p_j.merkle_root == p_q.merkle_root $\forall j, q . 1 \leq j, q \leq k$
  Assert $\gamma - \beta[p_1$.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$((s_j, v_j, p_j))$
    $\Sigma[s_j] = v_j$ // Initialize storage from environment
  For each $(s_j, v_j) \in \sigma_i$:
    $\Sigma[s_j] = v_j$ // Initialize static per-test storage
  If $\pi_i(\Sigma) \neq \omega_i$: // Contract paralyzed if test output differs from expected
    $C = C \setminus C[0]$
    $T = \emptyset$

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

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 code/contracts 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 [17] [33].

We find a natural solution to this class of stuck funds in software engineering tradition. When integrating various system components which may potentially be faulty, untrustworth, or unpredictable, developers often create and execute thorough integration tests [38], often continuously as software is developed, in a process known as "continuous integration" [25] [77]. Continuous integration can naturally be applied to the above paralysis scenarios: if a full continuous integration suite is run with the full blockchain and external interactions' state for every block, and failure causes a "downgrading of access control" in the transfer of funds to a simpler recovery contract, forms of paralysis described above are avoided.

Two variant implementations of this idea are possible. The first is smart-contract based, with a protocol specification for the smart contract given in Figure 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 but are not necessarily comprehensive; tests 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.

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 [85]. 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. A temporary state is instantiated, first with all the provided state entries from the environment, then from all 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 reset to prevent further migrations.

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 [48] 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 tests suggested 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 12USD per storage location proof at the time of writing: expensive but not prohibitive.

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 the expressiveness of developers' tests in an effort to make worst-case verification cost tractable in a volatile and unpredictable 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.

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 [90] is used), they can be queried or emulated by SGX, or can use a trusted off-chain oracle. Also, confidential integration tests may be useful for testing some contracts. While unsuitable in a public network due to their ability to hide backdoors, one could imagine a contract between parties where neutrally-agreed-on third parties or arbiters were responsible for developing and maintaining independent anti-paralysis test suites.

7 Related Work

Bitcoin had built-in support for threshold signatures at launch, and access-structure scripts for Bitcoin have been discussed since at least 2012 (see, e.g., [66]). The witness for the built-in Bitcoin opcode is a list that consists of the individual signatures that meet the threshold, which increases the on-chain verification complexity (this is undesirable, cf. [49]).

Gennaro, Goldfeder and Narayanan[30] presented a novel ECDSA threshold signature construction that reduces the on-chain complexity of multi-signature control over a Bitcoin address. However, this contruction requires a rather complex setup using ZK proofs, and does not support arbitrary access structures. Threshold Schnorr signature is significantly more efficient [76], with planned support on the roadmap of Bitcoin developers [86].

Mesh signatures [13] can be used to implement an arbitrary access structure. Attribute-based signatures [50] is an alternative approach that utilizes a trusted third party to implement arbitrary access structures. These constructions rely on complex cryptographic primitives such as bilinear pairings (which have no native support in Bitcoin script language). By themselves, mesh signatures can only support a *static* access structure (cf. Section 1).

Ethereum wallets such as Mist [55] and Gnosis [82] 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 (paid according to gas) when the complexity of the access structure is greater.

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 the players in order to update the access structure (see [23, 88]). By contrast, a paralysis proof system enables any player to remove the incapacitated players. Privacy-preserving cloud services can allow remote administrators to modify the access-control policies dynamically, via cryptographic constructions (see, e.g., [35, 41]). Dynamic access-control policies for a non-confidential cloud service may also benefit from dynamic access-control policies [58]. In all of these constructions, a policy modification affects the ability of end-users to interact with the server, but the set of administrators that are authorized to perform the modifications is static.

Credential-recovery schemes Password-recovery systems allow users to recover from the loss of a secret, but they require a trusted third party. See [12] for a survey.

Blockchains as censorship-resistant channels Bitcoin and other cryptocurrencies have been proposed as a means to facilitate a censorship-resistant channel. For instance, ZombieCoin [5] analyzes the prospects of a botnet command&control center using Bitcoin. Another recent work [4] allows users behind a government firewall to discover Tor entry nodes, via a challenge-response protocol in which the messages are transmitted on a cryptocurrency ledger.

8 Beyond Paralysis Proof Systems and Cryptocurrencies

The techniques we have introduced for Paralysis Proof System 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 Appendix F.
- *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., [89], 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 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 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 supporting formalism includes a formal DASP framework, and security and functionality property definitions for DASSes and DASPs, as well as UC-type ideal functionality for a DASS.

Paralysis Proofs and DASSes can be applied in a number of settings, as we show by exploring three in the paper: cryptocurrency key loss, cryptocurrency custody failures, and smart contract failures, proposing practical schemes for all three. We report on a straightforward DASS for cryptocurrency key loss in Ethereum,

and show through detailed exploration that a DASS for Bitcoin is only practical using our TEE-based techniques.

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.

References

[1] How Hackers Stole $500 Million in Digital Currency. http://fortune.com/2018/01/31/coincheck-hack-how/.

[2] Paralysis Proofs Implementation (Bitcoin and SGX), 2018. https://s3.amazonaws.com/anonymous-code/pp.tar.gz.

[3] Paralysis Proofs Implementation (Ethereum), 2018. https://s3.amazonaws.com/anonymous-code/Paralysis.sol.

[4] R3c3: Cryptographically-secure censorship resistant rendezvous using cryptocurrencies, 2018. Preprint.

[5] Syed Taha Ali, Patrick McCorry, Peter Hyun-Jeen Lee, and Feng Hao. Zombiecoin: Powering next-generation botnets with bitcoin. In Michael Brenner, Nicolas Christin, Benjamin Johnson, and Kurt Rohloff, editors, *Financial Cryptography and Data Security - FC 2015 International Workshops, BITCOIN, WAHC, and Wearable, San Juan, Puerto Rico, January 30, 2015, Revised Selected Papers*, volume 8976 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2015.

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *HASP'13*, pages 1–7, 2013.

[7] Gavin Andresen. P2SH. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki.

[8] Brian Armstrong. Coinbase is not a wallet, February 2016.

[9] Bellare and Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *13th CCS*. ACM SIGSAC, 2006.

[10] Iddo Bentov, Yan Ji, Fan Zhang, Yunqi Li, Xeuyuan Zhao, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. https://eprint.iacr.org/2017/1153, 2017.

[11] Nirupama Devi Bhaskar and David LEE Kuo Chuen. Bitcoin exchanges. In *Handbook of Digital Currency*, pages 559–573. Elsevier, 2015.

[12] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy*, pages 553–567. IEEE Computer Society, 2012.

[13] Xavier Boyen. Mesh signatures. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 210–227. Springer, 2007.

[14] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*, page 33, 2017.

[15] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the Parity multisig bug, Jul. 2017. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[16] BtcDrak, Mark Friedenbach, and Eric Lombrozo. Checksequenceverify. BIP 112, https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki, 2015.

[17] Vitalik Buterin. Reclaiming of ether in common classes of stuck accounts (eip #156), 2016. https://github.com/ethereum/EIPs/issues/156.

[18] Vitalik Buterin. Thinking about smart contract security, Jun. 2016. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.

[19] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.

[20] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *ArXiv e-prints*, April 2018.

[21] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 719–728, 2017.

[22] Frank Chung. 'don't tell my wife': Melbourne man cries over lost bitcoins as price surges past $us10,000, 2017. 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.

[23] Yvo Desmedt and Sushil Jajodia. Redistributing secret shares to new access structures and its applications. Technical report, November 22 2009.

[24] Bitcoin devlopers. Optimized C library for EC operations on curve secp256k1, 2018. https://github.com/bitcoin-core/secp256k1/.

[25] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[26] Pieter Wuille Eric Lombrozo, Johnson Lau. Segregated witness. BIP 141, https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki, December 2015.

[27] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59, 2016.

[28] Peter D. Feaver. Command and control in emerging nuclear nations. *International Security*, 17(3):160–187, 1992.

[29] Ben A. Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: Functional encryption using intel sgx, 2017. https://eprint.iacr.org/2016/1071.

[30] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *Applied Cryptography and Network Security - 14th ACNS*, volume 9696 of *Lecture Notes in Computer Science*, pages 156–174. Springer, 2016. https://freedom-to-tinker.com/2015/03/08/threshold-signatures-for-bitcoin-wallets-are-finally-here/.

[31] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017.

[32] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA*, pages 299–312, 2017.

[33] Yoichi Hirai. Who can recover stuck funds on ethereum?, 2018. https://medium.com/@pirapira/who-can-recover-stuck-funds-on-ethereum-345ba7566c9c.

[34] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13*, pages 1–1, 2013.

[35] Garrison III, William C., Adam Shull, Steven Myers, and Adam J. Lee. On the practicality of cryptographically enforcing dynamic access control policies in the cloud (extended version), April 26 2016. Comment: 26 pages; extended version of the IEEE S&P paper.

[36] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, 2016.

[37] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

[38] Syed Roohullah Jan, Syed Tauhid Ullah Shah, Zia Ullah Johar, Yasin Shah, and Fazlullah Khan. An innovative approach to investigate various software testing techniques and strategies. *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN*, pages 2395–1990, 2016.

[39] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, page 5. ACM, 2017.

[40] Gabriel Kaptchuk, Ian Miers, and Matthew Green. Giving state to the stateless: Augmenting trustworthy computation with ledgers. Cryptology ePrint Archive, Report 2017/201, 2017. https://eprint.iacr.org/2017/201.

[41] Jongkil Kim and Surya Nepal. A cryptographically enforced access control with a flexible user revocation on untrusted cloud storage. *Data Science and Engineering*, 1(3):149–160, 2016.

[42] Katharina Krombholz, Aljosha Judmayer, Matthias Gusenbauer, and Edgar Weippl. The other side of the coin: User experiences with bitcoin security and privacy. In Jens Grossklags and Bart Preneel, editors, *Financial Cryptography and Data Security*, pages 555–580, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[43] Johnson Lau. Bip 114. https://github.com/bitcoin/bips/blob/master/bip-0114.mediawiki, 2017.

[44] Johnson Lau. P2wpk. https://github.com/jl2012/bips/blob/vault/bip-0VVV.mediawiki, 2017.

[45] Johnson Lau. Pushtxdata. https://github.com/jl2012/bips/blob/vault/bip-0ZZZ.mediawiki, 2017.

[46] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent B Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security*, pages 523–539, 2017.

[47] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, pages 16–18, 2017.

[48] Loi Luu and Nate Rush. Peacerelay merkle-patricia trie proof verification. https://github.com/loiluu/peacerelay/blob/48cab51e6638a6614c86299f1b880698631f8738/contracts/PeaceRelay.

sol#L84, 2017.
[49] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 706–719. ACM, 2015.
[50] Hemanta K. Maji, Manoj Prabhakaran, and Mike Rosulek. Attribute-based signatures. In Aggelos Kiayias, editor, *CT-RSA*, volume 6558 of *Lecture Notes in Computer Science*, pages 376–392. Springer, 2011.
[51] Bill Marino and Ari Juels. Setting standards for altering and undoing smart contracts. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 151–166. Springer, 2016.
[52] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. 2017. http://eprint.iacr.org/2017/048.pdf.
[53] Gregory Maxwell. Coincovenants using scip signatures, an amusingly bad idea. https://bitcointalk.org/index.php?topic=278122.0, 2013.
[54] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy - HASP '13*, pages 1–1, 2013.
[55] Mist. https://github.com/ethereum/mist.
[56] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.
[57] Malte Möser, Ittay Eyal, and Emin Gün Sirer. Bitcoin covenants. In *Financial Cryptography Bitcoin Workshop*, 2016.
[58] Prasad Naldurg and Roy H. Campbell. 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)*, pages 231–237, New York, June 2–3 2003. ACM Press.
[59] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.
[60] Russell O'Connor and Marta Piekarska. Enhancing bitcoin transactions with covenants. In *Financial Cryptography Bitcoin Workshop*, 2017.
[61] Ouriel Ohayon. The sad state of crypto custody, 2018. https://techcrunch.com/2018/02/01/the-sad-state-of-crypto-custody/.
[62] Dan et al. O'Keeffe. SGXSpectre, 2018. https://github.com/lsds/spectre-attack-sgx.
[63] Nick Ortega. 'i forgot my pin': An epic tale of losing $30,000 in bitcoin, 2017. https://www.wired.com/story/i-forgot-my-pin-an-epic-tale-of-losing-dollar30000-in-bitcoin/.
[64] Charlie Osborne. Parity shakes up wallet audits, but funds remain frozen. *ZDNet*, 16 November 2017.
[65] Rafael Pass, Elaine Shi, and Florian Tramer. Formal Abstractions for Attested Execution Secure Processors. Cryptology ePrint Archive, Report 2016/1027, 2016. https://eprint.iacr.org/2016/1027.
[66] Alan Reiner. New wallet file ideas. https://bitcointalk.to/index.php?topic=128119.0, 2012.
[67] Reuters. Cryptocurrency Exchanges Are Increasingly Roiled With These Problems.
[68] JEFF JOHN ROBERTS and NICOLAS RAPP. Exclusive: Nearly 4 Million Bitcoins Lost Forever, New Study Says, 11 2017.
[69] Jeremy Rubin, Manali Naik, and Nitya Subramanian. Merkelized abstract syntax tree. http://www.mit.edu/~jlrubin/public/pdfs/858report.pdf, 2014.
[70] Eric Schlosser. Always / Never. *The New Yorker*, jan 2014.
[71] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 3–24. Springer, 2017.
[72] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
[73] Daniel Shane. $530 million cryptocurrency heist may be biggest ever, January 2018.
[74] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with SGX enclaves. In *NDSS*. The Internet Society, 2017.
[75] Jutta Steiner. Security is a process: A postmortem on the parity multi-sig library self-destruct, 2017. https://blog.ethcore.io/security-is-a-process-a-postmortem-on-the-parity-multi-sig-library-self-destruct/.
[76] Stinson and Strobl. Provably secure distributed schnorr signatures and a (t,n) threshold scheme for implicit certificates. In *ACISP: Information Security and Privacy: Australasian Conference*, 2001.
[77] Sean Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE'09.*, pages 369–374. IEEE, 2009.
[78] Peter Todd. Checklocktimeverify, 2014. https://github.com/petertodd/bips/blob/checklocktimeverify/bip-checklocktimeverify.mediawiki.
[79] Ruth Umoh. 3 of the craziest things people are doing to recover their lost bitcoin, 2017. https://www.cnbc.com/2017/12/21/3-crazy-things-people-are-doing-to-recover-lost-bitcoin.html.
[80] https://github.com/dedis/doc/issues/1.
[81] https://medium.com/@ConsenSys/thoughts-on-utxo-by-vitalik-buterin-2bb782c67e53.
[82] Gnosis Multisig Wallet. https://github.com/gnosis/MultiSigWallet.
[83] Kyle Wang. Ethereum: Turing-completeness and rich statefulness explained, 2017. https://hackernoon.com/ethereum-turing-completeness-and-rich-statefulness-explained-e650db7fc1fb.
[84] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434. ACM, 2017.
[85] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.
[86] Pieter Wuille et al. Schnorr signatures and signature aggregation. https://bitcoincore.org/en/2017/03/23/schnorr-signature-aggregation/.
[87] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015.
[88] Jia Yu, Fanyu Kong, Xiangguo Cheng, and Rong Hao. Two protocols for member revocation in secret sharing schemes. In Michael Chau, G. Alan Wang, Xiaolong Zheng, Hsinchun Chen, Daniel Zeng, and Wenji Mao, editors, *Intelligence and Security Informatics - Pacific Asia Workshop, PAISI 2011, Beijing, China, July 9, 2011. Proceedings*, volume 6749 of *Lecture Notes in Computer Science*, pages 64–70. Springer, 2011.
[89] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 270–282. ACM, 2016.
[90] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town Crier: An authenticated data feed for smart contracts. *23rd ACM Conference on Computer and Communications Security, CCS 2016*, pages 270–282, 2016.

A Addition Formalism

B Proof of Theorem 4.1

We recall Theorem 4.1:

Theorem 4.1 (The Security of Π_{SGX}). *Assume $\mathcal{G}_{\text{SGX}}$'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 $(\mathcal{G}_{\text{SGX}}, \mathcal{F}_{blockchain})$-hybrid model, for static adversaries.*

Proof. For simplicity, we write $\mathcal{F}_{\text{DASS}}[s_{P_0}, \mathcal{R}, \mathcal{S}, \mathcal{M}]$ as $\mathcal{F}_{\text{DASS}}$ from now on. To prove Theorem 4.1, it suffices to show that for the "dummy adversary" $\mathcal{A}$, there exists a PPT adversary Sim such that for any PPT environment $\mathcal{Z}$

$$\text{EXEC}_{\Pi_{\text{SGX}}, \mathcal{A}, \mathcal{Z}} \approx \text{EXEC}_{\mathcal{F}_{\text{DASS}}, \text{Sim}, \mathcal{Z}} \ . \tag{1}$$

Basically, the dummy adversary simply relays messages between the environment $\mathcal{Z}$ and parties. In particular, $\mathcal{A}$ corrupts parties when instructed by $\mathcal{Z}$ and passes all gathered information to $\mathcal{Z}$. We refer readers to Section 4.4.1 of [19] for details on emulation with respect to the dummy adversary.

We first present the construction of Sim, then we show that Sim satisfies Equation (1).

B.1 Construction of Sim.

Sim generally proceeds as follows: if a message is sent by an honest party to $\mathcal{F}_{\text{DASS}}$, Sim emulates the appropriate real world "network traffic" for $\mathcal{Z}$ using the information obtained from $\mathcal{F}_{\text{DASS}}$. If a message is sent to $\mathcal{F}_{\text{blockchain}}$ or $\mathcal{G}_{\text{SGX}}$ by a corrupted party, Sim intercepts the input and interact with $\mathcal{A}$ with the help of $\mathcal{F}_{\text{DASS}}$. We provide further details on the processing of specific messages.

Initialization. For an honest party P_h, Sim faithfully emulates Π_{SGX} as if P_h is called with a init message. In particular, Sim generates the key pair and simulates the initialization of $\mathcal{G}_{\text{SGX}}$ and $\mathcal{F}_{\text{blockchain}}$, if not already initialized. If a malicious party P_m sends init with corrupted parameters (i.e. different from those of $\mathcal{F}_{\text{DASS}}$), Sim aborts after simulating a abort message to all parties.

Access. For an honest party P_h calling $\mathcal{F}_{\text{DASS}}$ with (access, inp), Sim computes a signature σ over $\text{UTXO}_{\text{fund}}$ and inp (using the secret key generated in the initialization phase) and simulates $\mathcal{G}_{\text{SGX}}$ faithfully.

If $\mathcal{G}_{\text{SGX}}$ (simulated by Sim) is activated a malicious party P_c with input (spend, σ, $\text{UTXO}_{\text{fund}}$, addr_{new}), Sim checks that 1) the signature is valid against the public key (distributed in the initialization phase) and 2) $\text{UTXO}_{\text{fund}}$ is indeed unspent on $\mathcal{F}_{\text{blockchain}}$.

- If both checks pass, Sim proceeds as if P_c is honest.
- If the first check fails, Sim aborts.
- If the second check fails, Π_{SGX} will proceed normally since $\mathcal{G}_{\text{SGX}}$ doesn't have an up-to-date view of the blockchain. Therefore Sim simulates $\mathcal{G}_{\text{SGX}}$ faithfully, but without sending any message to $\mathcal{F}_{\text{DASS}}$.

Migration from S_P to $S_{P'}$. Without loss of generality, we only consider the cases where $P = P' \cup \{\text{pk}_k\}$, namely the migrations that remove P_k from the access structure. Other cases can be analyzed similarly.

If $\mathcal{F}_{\text{DASS}}$ is activated by an honest party P_h with input (migrate, P') that removes P_k from the current access structure, Sim proceeds as follows:

- If P_k is honest or paralyzed, Sim emulates P_i's part of Π_{SGX} by computing t_1 and t_2. If $\mathcal{F}_{\text{DASS}}$ authorizes the migration (indicated by a public ok output), Sim delay the output for Δ time, put t_2 on the blockchain, and then permits $\mathcal{F}_{\text{DASS}}$ to deliver the output to all parties. If $\mathcal{F}_{\text{DASS}}$ rejects the migration, Sim spends the life signal on P_k's behalf.
- If P_i is malicious, Sim waits until $\mathcal{F}_{\text{DASS}}$ asks whether P_k chooses to pretend to be paralyzed. Sim generates t_1, t_2 and sends t_1 to $\mathcal{F}_{\text{blockchain}}$. If $\mathcal{A}$ spends the output on behalf of P_k within Δ, Sim responds "yes" to $\mathcal{F}_{\text{DASS}}$. If not, Sim sends "no" to $\mathcal{F}_{\text{DASS}}$ and put t_2 to $\mathcal{F}_{\text{blockchain}}$.

If $\mathcal{G}_{\text{SGX}}$ (simulated by Sim) is activated by a corrupted party P_c with input (migrate, $\text{UTXO}_{\text{fund}}$, P') that removes P_k from the current access structure, Sim computes t_1, t_2 and send both to $\mathcal{A}$ as if from $\mathcal{F}_{\text{blockchain}}$, and then proceeds as follows:

- If P_k is *honest and alive*: If $\mathcal{A}$ sends t_1 to $\mathcal{F}_{\text{blockchain}}$, Sim spends t_1 on P_k's behalf, and send (migrate, $s_{P'}$) to $\mathcal{F}_{\text{DASS}}$ on P_c's behalf at the same time.
- If P_k is *paralyzed*: If $\mathcal{A}$ sends both t_1 and t_2 to $\mathcal{F}_{\text{blockchain}}$, Sim sends (migrate, $s_{P'}$) to $\mathcal{F}_{\text{DASS}}$ on P_c's behalf.
- If P_k is *malicious*: If $\mathcal{A}$ sends both t_1 and t_2 to $\mathcal{F}_{\text{blockchain}}$, and $\mathcal{A}$ doesn't spend t_1, then Sim sends (migrate, $s_{P'}$) to $\mathcal{F}_{\text{DASS}}$ on P_c's behalf, and sends "no" to $\mathcal{F}_{\text{DASS}}$ when asked whether P_k chooses to be paralyzed.

B.2 Validity of Sim.

We show that no environment can distinguish an interaction with $\mathcal{A}$ and Π_{SGX} from one with Sim and $\mathcal{F}_{\text{DASS}}$ by hybrid arguments. Consider a sequence of hybrids, starting with the real-world execution of Π_{SGX}. H_1 lets Sim to emulate $\mathcal{G}_{\text{SGX}}$ and $\mathcal{F}_{\text{blockchain}}$. H_2 filters out the forgery attacks against Σ_{SGX} and H_3 filters out the forgery attacks against the signature scheme.

Hybrid H_1 proceeds as in the real world protocol, except that Sim emulates $\mathcal{G}_{\text{SGX}}$ and $\mathcal{F}_{\text{blockchain}}$. Specially, Sim generates a key pair $(\text{pk}_{\text{att}}, \text{sk}_{\text{att}})$ for Σ_{SGX} and publishes pk_{att}. Whenever $\mathcal{A}$ wants to communicate with $\mathcal{G}_{\text{SGX}}$, Sim records $\mathcal{A}$'s messages and faithfully emulates $\mathcal{G}_{\text{SGX}}$'s behavior. Similarly, Sim emulates $\mathcal{F}_{\text{blockchain}}$ by storing items internally.

As $\mathcal{A}$'s view in H_1 is perfectly simulated as in the real world, no $\mathcal{Z}$ can distinguish between H_1 and the real execution.

Hybrid H_2 proceeds as in H_1, except for the following modifications. If $\mathcal{A}$ invoked $\mathcal{G}_{\text{SGX}}$ with a correct message (install, $\text{prog}_{\text{encl}}$), then for all sequential resume calls, Sim records a tuple $(\text{outp}, \sigma_{\text{SGX}})$ where outp is the output of $\text{prog}_{\text{encl}}$ and σ_{SGX} is an attestation under sk_{att}. Let Ω denote the set of all such tuples. Whenever $\mathcal{A}$ sends an attested output $(\text{outp}, \sigma_{\text{SGX}}) \notin \Omega$ to Sim or an honest party, Sim aborts.

The indistinguishability between H_1 and H_2 can be shown by the following reduction to the the EU-CMA property of Σ_{SGX}: In H_1, if $\mathcal{A}$ sends forged attestations to Sim, signature verification will fail with all but negligible probability. If $\mathcal{Z}$ can distinguish H_2 from H_1, $\mathcal{Z}$ and $\mathcal{A}$ can be used to win the game of signature forgery.

Hybrid H_3 proceeds as in H_2, except for the following modifications. Suppose the set of public keys belonging to corrupted parties is $\{\text{pk}_i\}_{i=1}^N$. If $\mathcal{A}$ sends (spend, σ, _, _) and σ verifies under a public key $\text{pk} \notin \{\text{pk}_i\}_{i=1}^N$, Sim aborts.

Similarly, the indistinguishability between H_2 and H_3 can be shown by a reduction to the EU-CMA property of signature scheme.

It remains to observe that H_3 is identical to the ideal protocol with Sim.

□

```
// Approve a proposal and execute if signature threshold is reached
function spend(uint256 proposal_id) public updateRequired {
  require(is_active_keyholder(msg.sender));
  require(proposal_id < proposals.length);

  // add sender's signature to approval
  proposal_sigs[proposal_id][msg.sender] = true;

  // if enough proposers approved, send money
  uint256 num_signatures = 0;
  for (uint256 i = 0; i < keyholders.length; i++) {
    if (!paralyzed[keyholders[i]]) {
      if (proposal_sigs[proposal_id][keyholders[i]]) {
        num_signatures++;
      }
    }
  }

  if (((num_signatures) * mu2) >=
   (num_responsive_keys() * mu1)) {

    if (!proposals[proposal_id].filled) {
      proposals[proposal_id].filled = true;
      proposals[proposal_id].to.transfer(
              proposals[proposal_id].amount);
    }

  }
}

// Accuse a user of being paralyzed
function accuse(address accused) public
 updateRequired returns(uint256) {
  address accuser = msg.sender;

  // users cannot accuse themselves
  // (ensures always at least one active keyholder;
  // prevent stuck funds)
  require(accuser != accused);

  // both requester and accused must be active keyholders
  require(is_active_keyholder(accuser));
  require(is_active_keyholder(accused));

  // shouldn't be any outstanding claims against accused
  require(!(paralysis_claims[accused].expiry >= now));

  // create and insert an Paralysis Claim
  uint256 expiry = now+delta;
  paralysis_claims[accused] = ParalysisClaim(expiry, false);
  NewAccusation(accused, expiry); // notify the accused
  return expiry;
}

function respond() public {
  require(paralysis_claims[msg.sender].expiry > now);
  paralysis_claims[msg.sender].responded = true;
}
```

Figure 7: Partial Solidity implementation on Ethereum.

C Additional Code Sample

C.1 Paralysis Proofs via Ethereum

Figure 7 shows the code snippet of the Ethereum implementation of Paralysis Proofs.

D Paralysis Proofs via Covenants

In the context of Bitcoin scripts, the notion of a *covenant* allows to put restrictions on the way that an output can be spent. Covenants were introduced by Moser, Eyal and Sirer [57], following an early idea by Maxwell [53]. Another generic method for covenants was given by O'Connor and Piekarska [60], and an efficient implementation of covenants (`OP_PUSHTXDATA`) was created by Lau [45]. So far, covenants support has not been enabled on the Bitcoin mainnet.

The mechanism of [57] supports a recursive covenant by letting the interpreter replace the `Pattern` keyword with the covenant itself. The recursion is required in our paralysis use-case, because the funds must be restored back to the original covenant whenever an accusation attempt fails. However, the single `Pattern` capability of [57] is inadequate for the paralysis covenant, because we wish to move the funds between different covenants that depend on the subset of remaining shareholders. Fortunately, the implementation of [45] supports multiple recursive patterns, by hashing fixed and variable data and then comparing the result to the output P2SH address [7] or the SegWit P2WPKH/P2WSH [44] (as well as Merkelized syntax trees [43, 69]).

An exemplary paralysis covenant is illustrated in Figure 8, using syntax that is similar to that of [57]. In this example, three shareholders can spend the entire amount `V` with no restrictions, by using the 3-out-of-3 multisig condition of the `Pattern123` covenant. Any two shareholders can accuse the third shareholder of being paralyzed, by moving the entire fund of `V` coins into an `PatternIJ` covenant that lets them spend the coins after a relative timeout of 150 blocks. While the 150 blocks are still being created, the third shareholder can move the funds back into the initial covenant `Pattern123`. Similarly, any single shareholder can accuse the two other shareholders of being paralyzed, by moving the `V` coins into the `PatternI` covenant.

Note that the covenants `PatternIJ` and `PatternI` must be distinct for different values of `I`, `J`, in order to avoid collusion attacks. For example, if `PatternIJ` allowed any 2-out-of-3 to spend the `V` funds after the timeout, then two malicious shareholders P_2, P_3 could pretend that P_3 is paralysed, so that P_1, P_2 would accuse P_3, and after the 150 blocks timeout P_2, P_3 will spend the funds arbitrarily (without the consent of the honest P_1).

There is certain similarity between the SGX protocol of Figure 3 and the covenants implementation of Figure 8. The main difference is that the $\mathsf{pk}_\mathtt{I}, \mathsf{pk}_\mathtt{J}$ multisig replaces $\mathsf{pk}_{\mathrm{SGX}}$ in the condition ($\mathsf{pk}_{\mathrm{SGX}} \wedge \mathtt{CSV} \geq \Delta$). Hence, by taking the paralysis use-case as an example, it can be inferred that the *complexity of the covenants approach is significantly higher than that of an SGX implementation* (in terms of conceptual as well as on-chain complexity, see also Appendix E). As there have been recent proposals to support stateless covenants in Ethereum (for better scalability, cf. [81]), the comparative advantages of our SGX-based design may prove useful in other contexts too.

E The Complexity of Access Structure Realizations

The ideal functionality of Section 4 is threshold predicates that require consent from μN of the N live shareholders in order to spend the funds. ($\mu = 1$ would mean unanimous consent.)

However, it is also possible to consider an extended functionality that requires signatures according to a more complex access structure. E.g., any subset of 35 out of $\{P_1, P_2, \ldots, P_{40}\}$ can spend the

Basic paralysis covenants for three shareholders

```
Pattern123
IF
    3 <pk_1> <pk_2> <pk_3> 3 CheckMultiSig
ELSE IF
    <V> <Pattern12> CheckOutputVerify 2 <pk_1> <pk_2> 2 CheckMultiSig
ELSE IF
    <V> <Pattern13> CheckOutputVerify 2 <pk_1> <pk_3> 2 CheckMultiSig
ELSE IF
    <V> <Pattern23> CheckOutputVerify 2 <pk_2> <pk_3> 2 CheckMultiSig
ELSE IF
    <V> <Pattern1> CheckOutputVerify <pk_1> CheckSig
ELSE IF
    <V> <Pattern2> CheckOutputVerify <pk_2> CheckSig
ELSE IF
    <V> <Pattern3> CheckOutputVerify <pk_3> CheckSig
ENDIF ENDIF ENDIF ENDIF ENDIF ENDIF ENDIF

PatternIJ for (I, J) ∈ {(1, 2), (1, 3), (2, 3)}
IF
    <150> CheckSequenceVerify 2 <pk_I> <pk_J> 2 CheckMultiSig
ELSE IF
    <V> <Pattern123> CheckOutputVerify 1 <pk_1> <pk_2> <pk_3> 3 CheckMultiSig
ENDIF ENDIF

PatternI for I ∈ {1, 2, 3}
IF
    <150> CheckSequenceVerify <pk_I> CheckSig
ELSE IF
    <V> <Pattern123> CheckOutputVerify 1 <pk_1> <pk_2> <pk_3> 3 CheckMultiSig
ENDIF ENDIF
```

Figure 8: Basic paralysis covenants for three shareholders.

funds, otherwise the funds can be spent with the consent of all shareholders in the privileged set $\{P_1, P_2, P_3, P_4\}$, or otherwise the privileged $P_1, P_{35}, P_{36}, P_{37}, P_{38}, P_{39}, P_{40}$ can spend the funds.

Such an access structure can be accomplished using cryptographic constructions, in particular mesh signatures [13] and attribute-based signatures [50]. However, these schemes involve bilinear pairings and are rather complex, which entails that the on-chain verification of such schemes will be impractical or costly.

Recently, the use of SGX has been suggested for functional encryption that is far more efficient in comparison to a standard cryptographic variant [29]. In a similar fashion, the use of SGX for an access structure based signature scheme implies substantial efficiency gains too. Thus, the improved efficiency applies both to the off-chain protocol that produces signatures, and to the on-chain cost of verifying the signature (i.e., the on-chain complexity is reduced to just one ordinary signature verification against $\mathsf{pk}_{\mathrm{SGX}}$).

The ideal functionality of Section 4 can therefore be replaced with an extended functionality that supports an access structure, and the Bitcoin protocol of Figure 3 will essentially remain the same. This is because the off-chain complexity of creating the signature will be handled by the SGX enclave code, and the on-chain complexity will be absorbed into a verification against $\mathsf{pk}_{\mathrm{SGX}}$.

It is worth considering whether it is inherently that case that the high efficiency requires SGX, or whether it is possible to design a cryptocurrency with built-in support to access structure based signatures. In fact, certain support is offered via the use of Merklized Abstract Syntax Trees [43, 69] and Schnorr aggregate signatures [86]. As in the "Large multi-signature constructs" of [43], we can for example have a Merkle tree with $2+\binom{40}{35}-\binom{36}{31}-\binom{33}{28}+\binom{30}{25} < 2^{18}$ leaves, such that all but two of the leaf nodes require a multisig by a specific subset of $\{P_1, P_2, \ldots, P_{40}\}$ of size 35 (excluding subsets that already include the privileged sets $\{P_1, P_2, P_3, P_4\}$ and $\{P_1, P_{35}, P_{36}, P_{37}, P_{38}, P_{39}, P_{40}\}$, without double counting), put only the root hash on the blockchain, and expect a valid Merkle authentication path to spend the coins. Further, the script of the leaf can use a single aggregated public key that is created from the public keys of the 35 signers (using delinearization [9, 80]), so that the on-chain complexity is on par with that of verifying one ordinary signature. Regarding the total on-chain complexity, we have that transaction that spends the funds consists of one aggregated signature for the leaf node and a Merkle authentication path of 18 sibling hashes.

However, per the discussion of `OP_EVAL` in [43], the use of a Merklized Abstract Syntax Trees becomes significantly more challenging for a predicate that involves a more complex relation than a logical `OR` among the leaves. For instance, if the access structure specified that P_1, P_2, P_3 must consent, and either P_4, P_5 or P_6, P_7 must also consent, then this cannot be handled by the implementation of [43]. By contrast, SGX can handle this instance just as easily as the previous example.

As the above discussion illustrates, harnessing SGX to spend funds according to an access structure can be highly useful even for a cryptocurrency with a Turing-complete scripting language (such as Ethereum). Let us point out that as long as [86] is not yet operational, it can be quite beneficial to employ SGX even for threshold signatures, since an ECDSA threshold scheme (without a trusted dealer) is rather complex, cf. [30].

The use of access structures in a cryptocurrency can also incorporate a notion of time, which in turn can help to avoid system paralysis that is caused by disagreement together with the disappearance of some players. For instance, the functionality can require 75% of the active players to agree on how to spend the funds, but require only 50% of the active players after one year, and only 20% after three years. In the UTXO model of Bitcoin, this can be accomplished via trusted hardware: whenever the players agree to spend the funds they will specify *absolute* timeouts for the 50% and 20% cutoffs (using CLTV [78]), and whenever the SGX enclave is asked to remove an incapacitated player it will create a transaction whose output hardcodes the same absolute timeouts as the input that is being spent. If the access structures for the different points in time are complex, the trusted hardware based implementation will be particularly beneficial (otherwise covenants could be used).

F Daily Withdrawal Limit using SGX

Let us consider a functionality $\mathcal{F}_{\mathrm{daily}}$ that allows N shareholders to spend the funds if at least μN of them reach an agreement (for $\mu \leq 1$), and allows each individual shareholder to spend a small

portion of the funds (e.g., 0.1%) each day. Moreover, the functionality allows ρN shareholders to disable the daily spending of funds by individual shareholders (for $\rho \leq \mu$, which is useful in the case that some shareholders appear to spend too much). By using $\rho < \mu$, it is easier to block the daily withdrawals than to reach consensus on a large expenditure.

It may be quite useful to combine $\mathcal{F}_{\text{daily}}$ with a functionality for paralysis proofs, but for simplicity we focus in this section on a bare implementation of just $\mathcal{F}_{\text{daily}}$ itself. Given an expressive enough covenants support for Bitcoin (such as [45]), it is possible — though quite complex — to implement $\mathcal{F}_{\text{daily}}$ using similar methods to the ones that we describe in Appendix D. However, let us present here the more efficient implementation that relies on trusted hardware, and can be deployed on the current Bitcoin mainnet.

The SGX-based protocol Π_{daily} that implements $\mathcal{F}_{\text{daily}}$ is given in Figure 9.

Since Bitcoin outputs must be fully consumed, Π_{daily} does not realize $\mathcal{F}_{\text{daily}}$ exactly, but instead lets each one of the shareholders perform a daily withdrawal, in sequential order. Thus, the first shareholder has the privilege to withdraw a small amount on the first day, the second shareholder can withdraw a small amount on the second day, the third shareholder on the third day, and so on. If for example the third shareholder did not withdraw, then on the forth day any single shareholder can withdraw a small amount (on a first come first served basis), but on the fifth day the sequential order resumes and the forth shareholder will have the privilege to withdraw.

It should be noted that in a cryptocurrency that uses the accounts model rather than the UTXO model (e.g., Ethereum), a more expressive realization of $\mathcal{F}_{\text{daily}}$ is possible. E.g., multiple shareholders can withdraw small amounts as long as the daily limit has not yet been reached.

The gist of Π_{daily} is an embedding of a public key $\mathsf{pk}_{\mathrm{SGX}_j}$ into the spending transaction, corresponding to the shareholder P_j who currently has the daily withdrawal privilege. Since the secret key $\mathsf{sk}_{\mathrm{SGX}_j}$ is known only to the SGX enclave, P_j cannot spend the funds arbitrarily, but instead has to submit to enclave a request to spend a small amount V' of the V coins to an arbitrary destination T'. The enclave will thus also produce a new output for the rest of the $V - V'$ funds, with $\mathsf{pk}_{\mathrm{SGX}_{j+1}}$ embedded into it.

Since P_j may not necessarily wish to withdraw, the output that the enclaves produces also allows spending of a small amount with a special master public key $\mathsf{pk}_{\mathrm{SGX}_0}$, but only after a relative timeout of Δ blocks (since Bitcoin blocks are created once every 10 minutes on average, $\Delta = 144$ blocks implies ≈ 1 day). Hence, any shareholder who submitted a request to withdraw from the current funds will be able to spend the signed transaction that the enclave produced for her, but only after Δ blocks so that P_j has the opportunity to spend first.

In case μN shareholders wish to spend an arbitrary amount, or in case ρN shareholders wish to disable the daily withdrawal feature, they can submit their μN (or ρN) signatures to the enclave and receive a signed transaction that takes precedence over any daily withdrawal transaction. This is accomplished by using a small relative timeout Δ' in the condition that allows the current privileged shareholder to perform a daily withdrawal, so that the transaction that was agreed upon by μN (or ρN) shareholders can be incorporated into the blockchain earlier (e.g., $\Delta' = 3$ is reasonable).

Other parts of the Π_{daily} protocol (in particular the setup procedure) are identical to Π_{SGX}, see Section 4 for details.

Protocol Π_{daily}

Hardcoded: $\left\{\mathsf{pk}_i\right\}_{i=1}^{N}, \mu, \rho, V_{\max}, \Delta, \Delta'$

Init:

(1) Setup: securely generate and share $(\mathsf{sk}_{\mathrm{SGX}}, \mathsf{pk}_{\mathrm{SGX}})$, $(\mathsf{sk}_{\mathrm{SGX}_0}, \mathsf{pk}_{\mathrm{SGX}_0})$ and $\left\{(\mathsf{sk}_{\mathrm{SGX}_k}, \mathsf{pk}_{\mathrm{SGX}_k})\right\}_{k=1}^{N}$ among the enclaves.
(2) Let $\phi_\mu \triangleq [\mu N\text{-out-of-}N \text{ multisig among } \mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_N]$.
(3) For $j \in [N]$, let $\psi_j \triangleq [\phi_{\text{all}} \vee \mathsf{pk}_{\mathrm{SGX}} \vee (\mathsf{pk}_{\mathrm{SGX}_0} \wedge (\mathsf{CSV} \geq \Delta)) \vee (\mathsf{pk}_{\mathrm{SGX}_j} \wedge (\mathsf{CSV} \geq \Delta'))]$.
(4) $\{P_i\}$ escrow $\mathsf{UTXO}_{\text{fund}}$ by publishing $\langle \mathsf{UTXO}_{\text{fund}} \rightarrow (V, \psi_1)\rangle$.

Spend:

(1) $\{P_i\}$ send μN signatures, the escrowed (V, ψ_{curr}), and T_{new} to the enclave.
(2) The enclave returns a signed transaction $t = \langle (V, \psi_{\text{curr}}) \rightarrow T_{\text{new}}\rangle$.

Daily withdrawal:

(1) P_k sends a signed request with the escrowed (V, ψ_{curr}) and (V', T') to the enclave.
(2) The enclave verifies that $V' \leq V_{\max}$.
(3) The enclave fetches curr by parsing ψ_{curr}.
(4) If $k = \text{curr}$ then $\mathsf{sk} := \mathsf{sk}_{\mathrm{SGX}_k}$ else $\mathsf{sk} := \mathsf{sk}_{\mathrm{SGX}_0}$.
(5) If $k < N$ then $\ell := k + 1$ else $\ell := 1$.
(6) The enclave uses sk to create the signed transaction $t = \langle (V, \phi_{\text{curr}}) \rightarrow (V - V', \psi_\ell), (V', T')\rangle$, and returns t.

Disallow daily withdrawals:

(1) $\{P_i\}$ send ρN signatures and the escrowed (V, ψ_{curr}) to the enclave.
(2) The enclave returns a signed transaction $t = \langle (V, \psi_{\text{curr}}) \rightarrow (V, [\phi_{\text{all}} \vee \mathsf{pk}_{\mathrm{SGX}}])\rangle$.

Figure 9: An SGX-based realization of $\mathcal{F}_{\text{daily}}$.

G Purely Bitcoin-Based Paralysis Proofs

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 $\left\{t_{i,j,k}\right\}_{j \in [N]\setminus\{i\}, k \in [K]}$ 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. Figure 10 illustrates the transactions in the aforementioned scheme.

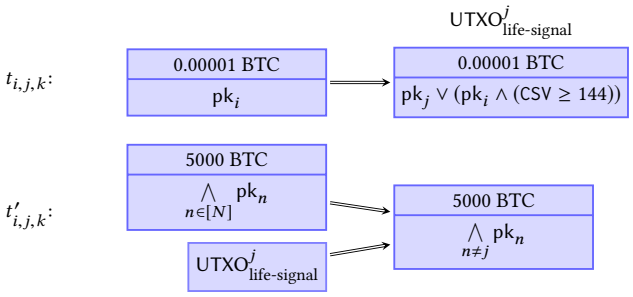

Figure 10: Bitcoin-based Paralysis Proofs with N players (with public keys $\{\mathsf{pk}_n\}_{n\in[N]}$). Each player P_i will prepare unsigned transactions $\{t_{i,j,k}\}_{j\in[N]\setminus\{i\},k\in[K]}$. All players will sign transactions $t'_{i,j,k}$.

This scheme can be implemented post-SegWit [26], 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

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

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