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

Fan Zhang[1], Philip Daian[2], Iddo Bentov[3], and Ari Juels[4]

[1] Cornell University, fanz@cs.cornell.edu
[2] Cornell Tech, Jacobs Institute, phil@cs.cornell.edu
[3] Cornell University, iddobentov@cornell.edu
[4] Cornell Tech, Jacobs Institute, juels@cornell.edu

Abstract. Suppose that N players share cryptocurrency using an M-out-of-N multisig scheme. If $N - M + 1$ players disappear, the remaining ones have a problem: They've permanently lost their funds.

We introduce *Paralysis Proofs*. A Paralysis Proof is a proof that players cannot act in concert, e.g., some players have become unavailable. Paralysis Proofs can support the construction of a *Paralysis Proof System*, which helps maintain resource availability by updating (e.g., downgrading) the resource's access structure when critical players, i.e., key-share holders, become unavailable.

We present a very general Paralysis Proof System implementation that combines trusted hardware, specifically Intel SGX, with a censorship-resistant channel in the form of a blockchain. Active players may issue a challenge to inactive or missing ones. A failure to respond in a timely way, as recorded on the blockchain, generates a Paralysis Proof that authorizes the trusted hardware to change the access structure, for instance, to allow cryptocurrency to be spent without the missing players.

Paralysis Proofs help address a pervasive key-management problem in cryptocurrencies and many other settings. We present specific instantiations for Ethereum (without trusted hardware) and for Bitcoin (with and without trusted hardware). We show that for any cryptocurrency system, versions with trusted hardware can be far more efficient than those without.

We also show how extensions of our techniques can encompass a rich array of access-structure policies addressing problems well beyond paralysis.

1 Introduction

Key management is an ever-present security challenge that highlights the tension between security and convenience. It arises prominently in the context of cryp-

tocurrencies, where secret keys are used to authorize transactions. It is further aggravated by some of the unique characteristics of cryptocurrencies.

First, as secret keys embody direct and total control over the money, key theft immediately enables funds to be stolen. As secret keys are accessed for each transaction, and potentially from multiple platforms, the keys of frequent users may be highly exposed to attack. In addition to theft, accidental deletion or corruption of keys is also fatal. 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 the system impossible. Therefore, lost keys equate with lost money.

Motivated by these practical concerns, there have been various proposals for protection and risk control mechanisms (see, e.g., [11]). However, the current key management support that cryptocurrencies offer is scant and inadequate. Particularly in Bitcoin, due to the limited expressiveness of its scripts, specifying complex access-control logic is difficult. The proposed Merklized Abstract Syntax Tree [27] can help to avoid the high transaction fees and latency that complex scripting logic entails, but only in some settings (e.g., M-out-of-N multisig where $M \ll N$, see Appendix B for more details).

Cryptocurrency funds associated with real world business activities, however, require flexible and sometimes complex *access structures* [14], meaning subsets of players authorized to control resources. A fundamental problem then arises that has received little attention, a problem we call access-control *paralysis*.

1.1 The paralysis problem

Consider, for instance, a cryptocurrency fund that is controlled by N players via an N-out-of-N Shamir secret sharing [28] of a single secret key or using N-out-of-N multisigs. If one of the key holders / players and her key share disappear due to an unforeseen event (e.g., a car accident, a broken hardware wallet, etc.), the remaining $N-1$ key holders will be *unable to access the fund*. Such paralysis, i.e., the inability to achieve a quorum of players, can alternatively take the form of a key-withholding attack, where a malicious key-share holder blackmails other players by withholding her key share. Similar problems can arise in other access structures. For instance, given a M-out-of-N threshold setting, inability to access $N - M + 1$ shares results in paralysis.

Existing secret sharing and multisig schemes cannot resolve the paralysis problem securely because of a basic paradox. Consider our N-out-of-N example from above and suppose that we wish to avoid paralysis should one share go missing. The only clear way to do so is to allow the $N - 1$ remaining players to update to an $(N - 1, N)$-threshold access structure should paralysis occur. If $N - 1$ players have this ability, though, then they can simply pretend that one share has gone missing, downgrade the access structure, and access the fund on their own. The system therefore effectively has an $(N - 1, N)$-threshold access structure.

To put it another way, the only way to avoid paralysis given a particular access structure is not to have that access structure to begin with, but instead to have a weaker one with lower system security.

One might argue for simply starting with an $(N-1, N)$-threshold in this example. But this is strictly and unnecessarily weaker than a conditionally downgraded (N, N)-threshold. Moreover, as we show, as we show, one may want to support the option of *multiple* access-structure downgrades. Starting with the weakest possible access structure in place would clearly then be a bad idea.

1.2 Our solution

We propose a novel technique called a *Paralysis Proof System* that is, to the best of our knowledge, the first to resolve this paradox. A Paralysis Proof System supports securely conditional migration from one access structure to another; the access structures can be arbitrarily rich and the policy governing migration in cases of paralysis can be quite flexible. A Paralysis Proof System can tolerate system paralysis both in settings where players disappears and more generally when players fail to act in concert. The principal tools that we employ to build a general version of a Paralysis Proof System are trusted hardware—Intel SGX, in particular—and a censorship-resistant channel in the form of a blockchain.

The critical idea behind Paralysis Proof Systems is that when some players in an access structure fail to act, the remaining players construct a *proof*, called a *Paralysis Proof*, that a quorum of players cannot act in concert. In our N-out-of-N access-structure example, $N-1$ players might construct a proof that an Nth player has disappeared. The proof itself takes the form of a challenge issued by the $N-1$ players to the Nth on the blockchain. Given the censorship-resistant nature of the blockchain, if the Nth player is available, she can respond reliably to the challenge. If not, the challenge of the $N-1$ players, together with evidence on the blockchain of a lack of response by the Nth player, constitute a Paralysis Proof within the access structure. This proof can be fed to a piece of trusted hardware to generate a signature using the underlying shared credentials of the full set of N players.

A Paralysis Proof System can be realized relatively easily for Ethereum using a smart contract, and we can even avoid trusted hardware in that setting. Scripting constraints in Bitcoin, however, necessitate the use of trusted hardware and also introduce some technical challenges that we address in this paper. Prime among these is the fact that without significant bloat in its trusted computing base, an SGX application cannot easily synch securely with the Bitcoin blockchain. We show how to avoid a need for an SGX application realizing a Paralysis Proof System to have a trustworthy view of the blockchain. We also present in the paper appendix a (somewhat less efficient) approach without trusted hardware that makes use of *covenants*, a proposed Bitcoin feature, and a scheme that works in Bitcoin today, but has security and efficiency limitations.

In the limit, it is possible, of course, to have all access requests mediated by an enclaved application, enabling arbitrarily defined access to a key. This approach, however, renders the availability of the enclave critical in a manner that we seek to avoid in our designs.

Let us emphasize that the capability to express timeouts does not, in itself, enable a Paralysis Proof System. For example, a simple Bitcoin script can dictate that the N players can spend the money before time τ_0, any subset of the players of size $N-1$ can spend the money in the time interval $[\tau_0, \tau_1]$, and any subset of size $N-2$ can spend the money after time τ_1. However, a malicious coalition of $N-2$ players could simply wait until time τ_2 and then spend the money without the consent of the other two players, even if these two players are active. By contrast, a Paralysis Proof can be used to transfer ownership of the funds to the coalition of $N-2$ players *only* if the other two players are incapacitated. In Appendix B we describe a combination of a Paralysis Proof System and time-based expirations that lower the threshold should paralysis occur.

In general, Paralysis Proof Systems are conceptually simple and offer a powerful new capability that is widely realizable today for the first time thanks to the emergence of Intel SGX and blockchains. This capability applies to settings well beyond cryptocurrency, e.g., file decryption, and can also be extended, as we explain to a rich set of access-structure modification options beyond paralysis.

Paper Organization.

In Section 2, we formally define policies and security for Paralysis Proof Systems. We explore implementation of a Paralysis Proof System for Bitcoin using SGX in Section 3. For comparison, we also present implementation with an Ethereum smart contract in Section 4. In Section 5, we very briefly discuss extensions of the techniques and concepts in this paper, showing how they can enable access-control policies in settings beyond cryptocurrency and paralysis. We conclude in Section 6 with a brief discussion of future work.

In the paper appendices, we describe an approach to Paralysis Proof System for Bitcoin using covenants, show how SGX can greatly improve the efficiency of complex access-structure support for Bitcoin, and given an example of an enriched access-control policy that enforces daily cryptocurrency withdrawal limits.

2 Paralysis Proof Systems

Paralysis Proofs demonstrate conditions, e.g., player incapacitation, that justify migration from one access structure to another. They serve within a Paralysis Proof System, a system that utilizes Paralysis Proofs to enforce enriched access control policies that can tolerate access-structure paralysis. We explain in

this section how these policies are formally specified and how the security of a Paralysis Proof System can be formally defined.

2.1 Policy Specification

We define a Paralysis Proof System *policy* as 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 $L_0 = \{P_i\}_{i=1}^{N}$ denote the set of N players at beginning of the protocol, and L_t the set of *live* (i.e. not incapacitated) players at time t. As we shall see shortly, correctly determining L_t, i.e. which players are actually live, is the main technical challenge in deploying Paralysis Proofs. We use L_t to denote the ground truth. We assume that if a player 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 players $L \subseteq L_0$ is allowed to access the managed resource. Access structures are monotonic, i.e., $s(L) = \texttt{true}$ and $L \subseteq L'$ together imply that $s(L') = \texttt{true}$.

For a given policy, the set of access structures $\mathcal{S}$ and the associated migration rules $\mathcal{M}$ may be represented as a directed graph $G = (\mathcal{S}, \mathcal{M})$. A node $s_i \in \mathcal{S}$ is an access structure and an enhanced edge $c_{s_i,s_j} = (s_i, s_j) \in \mathcal{M}$ represents a *migration condition* under which access structure s_i may change to access structure s_j. Condition is a function that takes as input the set of live players L_t and outputs `true` (migration permitted) or `false` (migration not permitted). For simplicity, we let c_{s_i,s_j} denote a given edge or its associated function.

The goal of a policy is to specify proper access structures and migration rules to retain access even a certain set of players become incapacitated. Depending on the application scenarios, one may design policies to defend against different levels of access-structure paralysis, e.g. the disappearance of up to M players. In what we call a *paralysis-free* policy, if the current access structure cannot be satisfied, switching to another satisfiable access structure should be permitted, as long as the switch doesn't put any of the live players at disadvantage. More precisely, we define the set of *least permissive* access structures for L_t as

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

That is, all players in L_t must be live to satisfy access structures in S_{LP}. Formally, a policy $(\mathcal{R}, \mathcal{S}, \mathcal{M})$ is *paralysis-free* if $\forall s \in \mathcal{S}$

$$s(L_t) = \texttt{false} \wedge S_{\text{LP}}(L_t) \neq \emptyset \implies \exists s' \in S_{\text{LP}}(L_t), c_{s,s'} \in \mathcal{M} \; s.t \; c_{s,s'}(L_t) = \texttt{true}.$$

Note that a paralysis-free policy doesn't imply the availability of the resource. What a paralysis-free policy can guarantee is the *best possible availability*: if

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

2.2 Security of a Paralysis Proof System

A Paralysis Proof System can be implemented in various ways, e.g. using SGX or Bitcoin scripts. We therefore model it in a very general way as an interactive agent that grants access upon request according to the current access structure in force. The Paralysis Proof System will migrate to another access structure given a suitable input, specifically a Paralysis Proof.

Adversarial model. We assume an adversary that may control an arbitrary number of players. An honest player always follows the protocol, while a malicious player controlled by the adversary may deviate arbitrarily. We assume that the adversary has complete control of the network, with the exception that a blockchain is available to all players, i.e. is censorship-resistant, and the maximum network latency to the blockchain is bounded by Δ.

Let s denote the state, i.e. the current effective access structure, of a Paralysis Proof System enforcing policy $(\mathcal{R}, \mathcal{S}, \mathcal{M})$. We say that a Paralysis Proof System is *secure* if the following security properties are preserved in all states $s \in \mathcal{S}$ at any time t:

Safety:
- A set of players $L \subseteq L_t$ can access $\mathcal{R}$ only if $s(L) = \texttt{true}$.
- A transition to $s' \neq s$ occurs at time t only if $c_{s,s'}(L_t) = \texttt{true}$.

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

Example 1. Let's take the example of N shareholders who wish to retain access to a resource $\mathcal{R}$ should one player disappear. Let $\mathcal{P} = L_0 = \{P_i\}_{i=1}^N$ denote the set of N players, and $\mathcal{P}_{-i} = \mathcal{P} \backslash \{P_i\}$ denote the set of $N-1$ players that excludes P_i. Let $\mathbb{I}(\cdot)$ denote an indicator function. A policy $(\mathcal{R}, \mathcal{S}, \mathcal{M})$ 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 $c_{s_0,s_i} \in \mathcal{M}$ is fulfilled for $L_t = \mathcal{P}_{-i}$.

In Example 1, the **Safety** property ensures that access is enforced by the current access structure at any time, and that the access structure can be downgraded to allow access by $N-1$ shareholders only if $|L_t| < N$, i.e., a collusion of $N-1$ shareholders cannot maliciously accuse the Nth 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 players. Moreover, if allowed by the policy, the Liveness property ensures that the access structure will be downgraded within a bounded time should players 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 players may not immediately activate an access-structure migration; in fact, if all players are incapacitated, such migration cannot happen.

Paralysis Proofs. Because a Paralysis Proof System cannot directly infer L_t, our system instead leverages the censorship-resistance of blockchain to enable players to construct Paralysis Proofs. Note that a Paralysis Proof does not necessarily prove L_t. It may merely prove that $P_i \notin L_t$ for a given player P_i or similar facts about L_t.

A Paralysis Proof System can be implemented by a program in an Intel SGX *enclave* (c.f. Section 3) or Bitcoin scripts enhanced with covenants (c.f. Appendix A). To determine L_t in a trustworthy way, the program relies on Paralysis Proofs presented by players in the system. In particular, as we shall see, to prove that $P_i \notin L_t$, players 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 $P_i \notin L_t$.

2.3 Basic Ideal Functionality

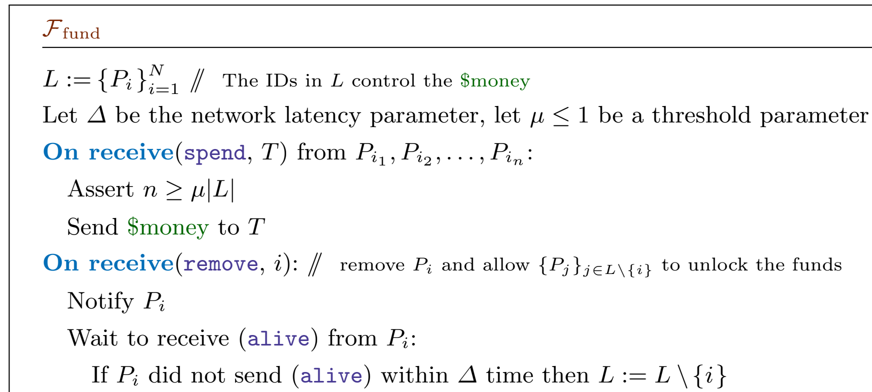
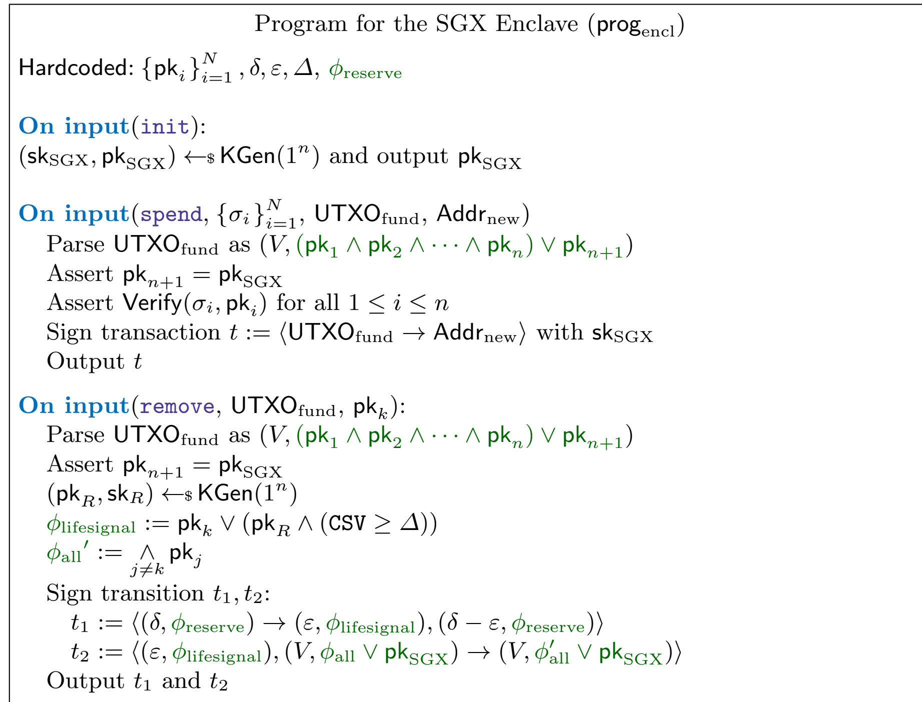


Fig. 1. The ideal functionality $\mathcal{F}_{\text{fund}}$

We model the goal of a basic Paralysis Proof System by means of the ideal functionality $\mathcal{F}_{\text{fund}}$ in Figure 1. The functionality $\mathcal{F}_{\text{fund}}$ requires some threshold of M-out-of-N to spend an atomic unit of money \$money (where M/N is defined by μ) and allows players to accuse a player P_i of paralysis and remove her from the access structure. It is straightforward to implement $\mathcal{F}_{\text{fund}}$ with an Ethereum contract; see Section 4. In Section 3 and Appendix A we explore Bitcoin implementations of $\mathcal{F}_{\text{fund}}$. For simplicity, our Bitcoin protocols are restricted to $\mu = 1$, but it is possible to extend them (Figures 3 and 6) to $\mu < 1$ too. In Appendix B we discuss the complexity of extended functionalities, where some of the challenges apply to Ethereum as well.

3 Paralysis Proofs via SGX


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

Hardcoded: $\{\mathsf{pk}_i\}_{i=1}^N, \delta, \varepsilon, \Delta, \phi_{\text{reserve}}$

On input(init):
$(\mathsf{sk}_{\text{SGX}}, \mathsf{pk}_{\text{SGX}}) \leftarrow_{\$} \mathsf{KGen}(1^n)$ and output pk_{SGX}

On input(spend, $\{\sigma_i\}_{i=1}^N$, $\mathsf{UTXO}_{\text{fund}}$, $\mathsf{Addr}_{\text{new}}$)
Parse $\mathsf{UTXO}_{\text{fund}}$ as $(V, (\mathsf{pk}_1 \wedge \mathsf{pk}_2 \wedge \cdots \wedge \mathsf{pk}_n) \vee \mathsf{pk}_{n+1})$
Assert $\mathsf{pk}_{n+1} = \mathsf{pk}_{\text{SGX}}$
Assert $\mathsf{Verify}(\sigma_i, \mathsf{pk}_i)$ for all $1 \leq i \leq n$
Sign transaction $t := \langle \mathsf{UTXO}_{\text{fund}} \rightarrow \mathsf{Addr}_{\text{new}} \rangle$ with sk_{SGX}
Output t

On input(remove, $\mathsf{UTXO}_{\text{fund}}$, pk_k):
Parse $\mathsf{UTXO}_{\text{fund}}$ as $(V, (\mathsf{pk}_1 \wedge \mathsf{pk}_2 \wedge \cdots \wedge \mathsf{pk}_n) \vee \mathsf{pk}_{n+1})$
Assert $\mathsf{pk}_{n+1} = \mathsf{pk}_{\text{SGX}}$
$(\mathsf{pk}_R, \mathsf{sk}_R) \leftarrow_{\$} \mathsf{KGen}(1^n)$
$\phi_{\text{lifesignal}} := \mathsf{pk}_k \vee (\mathsf{pk}_R \wedge (\mathtt{CSV} \geq \Delta))$
$\phi_{\text{all}}' := \bigwedge_{j \neq k} \mathsf{pk}_j$
Sign transition t_1, t_2:
$t_1 := \langle (\delta, \phi_{\text{reserve}}) \rightarrow (\varepsilon, \phi_{\text{lifesignal}}), (\delta - \varepsilon, \phi_{\text{reserve}}) \rangle$
$t_2 := \langle (\varepsilon, \phi_{\text{lifesignal}}), (V, \phi_{\text{all}} \vee \mathsf{pk}_{\text{SGX}}) \rightarrow (V, \phi_{\text{all}}' \vee \mathsf{pk}_{\text{SGX}}) \rangle$
Output t_1 and t_2

Fig. 2. The Paralysis Proof Enclave

Due to the limited expressiveness of Bitcoin scripts, it is problematic to implement secure Paralysis Proofs that are compatible with the current Bitcoin protocol (see Appendix A for a Bitcoin implementation with covenants [24]). In this section, we describe a secure protocol using trusted hardware, namely SGX.

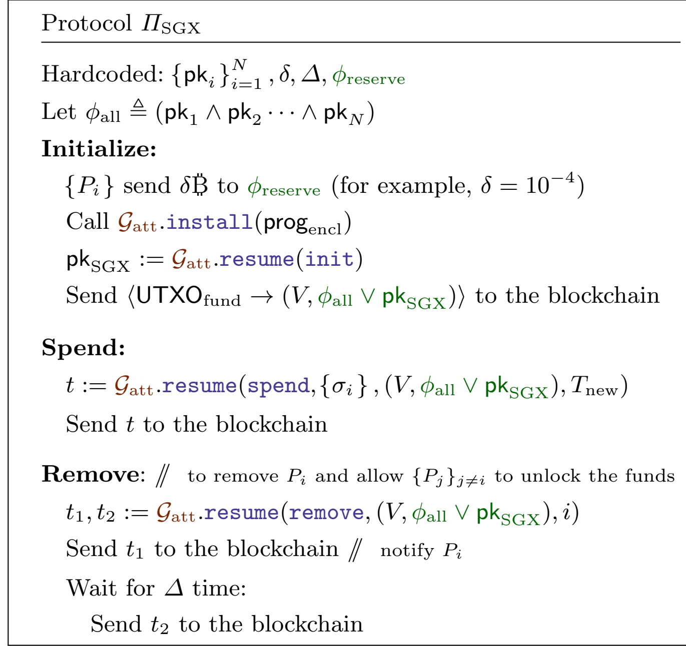


Fig. 3. An SGX based protocol for Paralysis Proofs.

Intel *Software Guard Extensions* (SGX) is an instruction set architecture extension that permits code execution in an isolated, tamper-free environment, and can prove to remote users that an output represents the result of such execution. We refer readers to [1,13,12,23] for further details on SGX.

Compared to the covenants-based scheme, the SGX-based protocol is simpler and has better on-chain efficiency. Let us stress that the trust assumption in SGX is *local*, i.e., only the players in the Paralysis Proofs protocol will be affected should the SGX properties be broken.

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. We use $\langle I_1, I_2, \ldots, I_n \rightarrow O_1, O_2, \ldots, O_m \rangle$ to denote a transaction with input $\{I_i\}_{i=1}^n$ and output $\{O_i\}_{i=1}^m$. We use (V, ϕ) to denote an UTXO of V coins and script ϕ.

Modeling SGX. We adopt a simplified version of the ideal functionality $\mathcal{G}_{\text{att}}$ defined in [26]. Compared with the original definition, we drop the parameters *sid* and *eid* for notational simplicity. Figure 2 gives the logic of the SGX enclave, which is used to program $\mathcal{G}_{\text{att}}$, and Figure 3 gives the protocol Π_{SGX} that realizes the basic ideal functionality of Section 2.3. The reader is advised to refer to Figure 4 for an illustration of Π_{SGX}.

Setup of Enclaves. To avoid the reliance on a centralized SGX server, each player in Π_{SGX} runs her own SGX enclave with an identical program. This way, any individual player (or set of players) can always use all the capabilities of the protocol without being dependent on the other players. This is particularly relevant with a signature scheme (cf. Section 2.3), but even with an N-out-of-N multisig it is important that the enclave is not controlled by only one player, in case the player becomes incapacitated. Each enclave first generates a fresh key pair $(\mathsf{pk}_{\text{SGX}_i}, \mathsf{sk}_{\text{SGX}_i})$ and outputs $\mathsf{pk}_{\text{SGX}_i}$ while keeping $\mathsf{sk}_{\text{SGX}_i}$ secret. Each player uses her identity P_i to endorse $\mathsf{pk}_{\text{SGX}_i}$, and all the players reach agreement on the list of SGX identities $\{\mathsf{pk}_{\text{SGX}\,i}\}_{i=1}^{N}$. The enclaves then use $\{\mathsf{pk}_{\text{SGX}\,i}\}_{i=1}^{N}$ to establish secure channels (TLS) with each other, and create a fresh shared secret key sk_{SGX} that is associated with $\{\mathsf{pk}_{\text{SGX}\,i}\}_{i=1}^{N}$ (i.e., another invocation of the setup procedure will generate a different shared key). Given 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 by encrypting it (which mainly consists of sk_{SGX}) 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.

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

Spend. 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 B).

Remove. In this procedure, the enclave received the unspent escrow fund as input, and generates two signed transactions: t_1 is the life signal for P_k, and t_2 spends both the life signal (i.e., the unspent t_1) and the escrowed fund to a script that $(\{\mathsf{pk}_i\}_{i=1}^{N} \setminus \{\mathsf{pk}_k\}) \vee \mathsf{pk}_{\text{SGX}}$ can spend. 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 4 demonstrates an example with three players.

The **Remove** procedure resolves *system paralysis* by letting $N-1$ shareholders spend the money if one shareholder 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.

The core idea of implementing an "appeal" in Bitcoin is to use what we call *life signals*. A life signal request for party P_k is a UTXO of a negligible amount of εɃ, that can be spent either by P_k — thereby signaling her liveness — or by $\mathsf{pk}_{\mathrm{SGX}}$ but only after a delay.

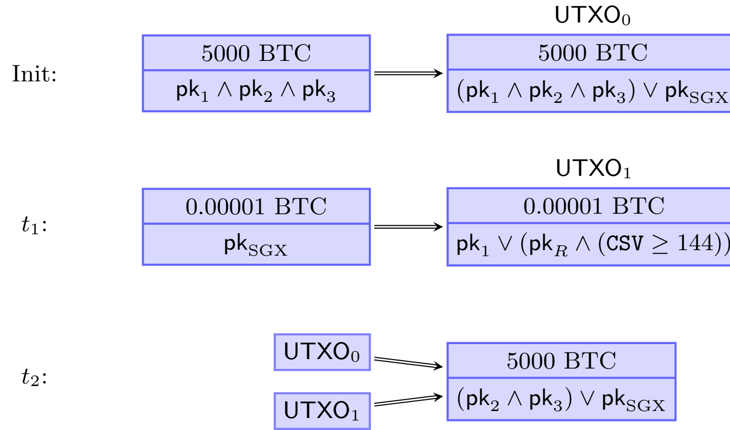


Fig. 4. Example of Π_{SGX} with three players and P_1 accused of being incapacitated.

The security of Π_{SGX} stems from the use of relative timeout (`CheckSequenceVerify` [7]) 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 (known as `ScriptSig` in Bitcoin) of each of its inputs is correct. The witness that the SGX enclave produced for spending the escrow fund is immediately valid, but the witness for spending t_1 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). Since the witnesses are bound to the entire t_2, it is impossible to maul t_2 into a different transaction that uses only the witness for the escrow fund.

Thus, setting the parameter Δ to a large value serves two purposes: (1) giving P_k enough time to respond, and (2) making sure 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. Also notice that the enclave code needs to parse the unspent escrow input and fetch the list of current shareholders, so that the list in the output for the new escrow fund will not include shareholders that already proved to be incapacitated earlier. Given the aforementioned setup procedure, one may regard our enclave request format (i.e., supplying the unspent escrow fund as input) as a method to synchronize the N local enclaves by using the blockchain itself.

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 a view of the blockchain 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., [21,5]).

In Appendix D 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.

4 Paralysis Proofs in Ethereum

An Ethereum implementation of the ideal functionality $\mathcal{F}_{\mathrm{fund}}$ of Figure 1 is straightforward. Our reference implementation consists of 117 lines of Solidity code, and its main logic is shown in Figure 5.

This implementation differs from the ideal functionality only in minor engineering changes and optimizations. Firstly, all spends must occur in multiple contract calls, first initiating a send transaction and then accepting a signature from each potential shareholder until the threshold is met, rather than performing the operation atomically. This is necessary, as it removes the requirement to verify Ethereum signatures on-chain, and allows the signatures in the contract transactions of each shareholder to serve as their signatures in $\mathcal{F}_{\mathrm{fund}}$. It would be possible to more closely mirror the atomic nature of our ideal functionality by not allowing calls to `remove` while a withdrawal is in progress, and adding a corresponding timeout to each withdrawal, forcing this series of signature transactions to occur together.

Furthermore, 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

```
function spend(uint256 proposal_id) public {
  // Get rid of any paralyzed keyholders
  prune_paralyzed_keyholders();

  require(is_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
  uint 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) >= required_sigs) {
    if (!proposals[proposal_id].filled) {
      proposals[proposal_id].filled = true;
      proposals[proposal_id].to.transfer(proposals[proposal_id].amount);
    }
  }
}

function remove(address accused) public {
  // Get rid of any paralyzed keyholders (prevent paralyzed requester)
  prune_paralyzed_keyholders();
  // both requester and accused must be keyholders
  require(is_keyholder(msg.sender));
  require(is_keyholder(accused));

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

  // Create and insert an Paralysis Claim
  paralysis_claims[accused] = ParalysisClaim(now+delta, false);
  NewAccusation(accused, now + delta); // Notify the accused
}

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

Fig. 5. A Solidity-based solution on Ethereum.

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.

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 Etheruem-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).

Our full contract code, which includes the logic for pruning incapacitated signers and updating the signature threshold according to $\mu|L|$ as in $\mathcal{F}_{\text{fund}}$, is published at https://github.com/pdaian/paralysis_proofs.

5 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 C.
- *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., [34], 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.

6 Conclusion

We have shown how Paralysis Proof Systems enrich existing access-control policies in a way that was previously unachievable. They allow an access structure to be modified—typically downgraded—given the incapacitation of a player or the inability of a set of players to act in concert. While this basic capability is realizable in Ethereum, it is only practical in Bitcoin using our techniques. Additionally, we have also shown how Paralysis Proof Systems can be used to control decryption keys and how the techniques introduced in this paper can given rise to a range of interesting and sophisticated event-driven access-control policies.

Side-channel attacks against SGX [8,19,33] are a source of concern about our proposed schemes. In future work, we will explore techniques to mitigate such attacks. For example, in a system that permits an N-out-of-N secret sharing scheme to be downgraded to $(N-1)$-out-of-N, it is possible for an enclave application to store and conditionally release a single share, rather than controlling a master secret. This would limit the damage of a side-channel attack should the node containing the application be compromised. It is also possible to distribute secrets across enclaves, which is useful in mitigating the impact of node compromises, but also helpful for resilience to node failures [21], another topic of future work.

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 new access-control regimes and we hope that our work stimulates exploration of other novel capabilities.

References

1. Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *HASP'13*, pages 1–7, 2013.
2. Gavin Andresen. P2SH. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki.

3. Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. Fair two-party computations via bitcoin deposits. In *International Conference on Financial Cryptography and Data Security*, pages 105–121, 2014.
4. Bellare and Neven. Multi-signatures in the plain public-key model and a general forking lemma. In *13th CCS*. ACM SIGSAC, 2006.
5. 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.
6. Xavier Boyen. Mesh signatures. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 210–227. Springer, 2007.
7. BtcDrak, Mark Friedenbach, and Eric Lombrozo. Checksequenceverify. BIP 112, https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki, 2015.
8. Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.
9. Pieter Wuille Eric Lombrozo, Johnson Lau. Segregated witness. BIP 141, https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki, December 2015.
10. Ben A. Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. Iron: Functional encryption using intel sgx, 2017. https://eprint.iacr.org/2016/1071.
11. 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.
12. 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.
13. Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide, 2016.
14. 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.
15. Ranjit Kumaresan and Iddo Bentov. How to Use Bitcoin to Incentivize Correct Computations. In *CCS*, 2014.
16. Johnson Lau. Bip 114. https://github.com/bitcoin/bips/blob/master/bip-0114.mediawiki, 2017.
17. Johnson Lau. P2wpk. https://github.com/jl2012/bips/blob/vault/bip-0VVV.mediawiki, 2017.
18. Johnson Lau. Pushtxdata. https://github.com/jl2012/bips/blob/vault/bip-0ZZZ.mediawiki, 2017.
19. 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. *arXiv preprint arXiv:1611.06952*, 2016.
20. 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.
21. 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.
22. Gregory Maxwell. Coincovenants using scip signatures, an amusingly bad idea. https://bitcointalk.org/index.php?topic=278122.0, 2013.

23. 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.
24. Malte Möser, Ittay Eyal, and Emin Gün Sirer. Bitcoin covenants. In *Financial Cryptography Bitcoin Workshop*, 2016.
25. Russell O'Connor and Marta Piekarska. Enhancing bitcoin transactions with covenants. In *Financial Cryptography Bitcoin Workshop*, 2017.
26. 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.
27. Jeremy Rubin, Manali Naik, and Nitya Subramanian. Merkelized abstract syntax tree. http://www.mit.edu/~jlrubin/public/pdfs/858report.pdf, 2014.
28. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
29. Peter Todd. Checklocktimeverify, 2014. https://github.com/petertodd/bips/blob/checklocktimeverify/bip-checklocktimeverify.mediawiki.
30. https://github.com/dedis/doc/issues/1.
31. https://medium.com/@ConsenSys/thoughts-on-utxo-by-vitalik-buterin-2bb782c67e53.
32. Pieter Wuille et al. Schnorr signatures and signature aggregation. https://bitcoincore.org/en/2017/03/23/schnorr-signature-aggregation/.
33. Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.
34. 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.

A 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 [24], following an early idea by Maxwell [22]. Another generic method for covenants was given by O'Connor and Piekarska [25], and an efficient implementation of covenants (`OP_PUSHTXDATA`) was created by Lau [18]. So far, covenants support has not been enabled on the Bitcoin mainnet.

The mechanism of [24] 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 [24] 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 [18] supports multiple recursive patterns, by hashing fixed and variable data and then comparing the

```
                              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 $(\mathtt{I},\mathtt{J}) \in \{(1,2),(1,3),(2,3)\}$

```
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 $\mathtt{I} \in \{1,2,3\}$

```
IF
   <150> CheckSequenceVerify <pk_I> CheckSig
ELSE IF
   <V> <Pattern123> CheckOutputVerify 1 <pk_1> <pk_2> <pk_3> 3 CheckMultiSig
ENDIF ENDIF
```

Fig. 6. Basic paralysis covenants for three shareholders.

result to the output P2SH address [2] or the SegWit P2WPKH/P2WSH [17] (as well as Merkelized syntax trees [16,27]).

An exemplary paralysis covenant is illustrated in Figure 6, using syntax that is similar to that of [24]. 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 6. The main difference is that the $\mathsf{pk_I}, \mathsf{pk_J}$ 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 B). As there have been recent proposals to support stateless covenants in Ethereum (for better scalability, cf. [31]), the comparative advantages of our SGX-based design may prove useful in other contexts too.

B The Complexity of Access Structure Realizations

The ideal functionality of Section 2.3 and focus of the paper are 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 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 [6] and attribute-based signatures [20]. 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 [10]. 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 Figure 1 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 [16,27] and Schnorr aggregate signatures [32]. As in the "Large multi-signature constructs" of [16], 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 [4,30]), 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 [16], 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 [16]. 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 [32] 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. [11].

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 [29]), 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).

C Daily Withdrawal Limit using SGX

Let us consider a functionality $\mathcal{F}_{\text{daily}}$ 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 [18]), it is possible — though quite complex — to implement $\mathcal{F}_{\text{daily}}$ using similar methods to the ones that we describe in Appendix A. 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 7.

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 $\mathsf{pk}_{\text{SGX}_j}$ into the spending transaction, corresponding to the shareholder P_j who currently has the daily withdrawal privilege. Since the secret key $\mathsf{sk}_{\text{SGX}_j}$ 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 $\mathsf{pk}_{\text{SGX}_{j+1}}$ 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 $\mathsf{pk}_{\text{SGX}_0}$, 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

Protocol Π_{daily}

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

Init:

1. Setup: securely generate and share $(\mathsf{sk}_{\text{SGX}}, \mathsf{pk}_{\text{SGX}}), (\mathsf{sk}_{\text{SGX}_0}, \mathsf{pk}_{\text{SGX}_0})$ and $\left\{(\mathsf{sk}_{\text{SGX}_k}, \mathsf{pk}_{\text{SGX}_k})\right\}_{k=1}^{N}$ among the enclaves.
2. Let $\phi_\mu \triangleq [\mu N\text{-out-of-}N$ multisig among $\mathsf{pk}_1, \mathsf{pk}_2, \ldots, \mathsf{pk}_N]$.
3. For $j \in [N]$, let $\psi_j \triangleq [\phi_{\text{all}} \vee \mathsf{pk}_{\text{SGX}} \vee (\mathsf{pk}_{\text{SGX}_0} \wedge (\texttt{CSV} \geq \Delta)) \vee (\mathsf{pk}_{\text{SGX}_j} \wedge (\texttt{CSV} \geq \Delta'))]$.
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 $t = \langle (V, \psi_{\text{curr}}) \rightarrow T_{\mathsf{new}} \rangle$.

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 $k = \mathsf{curr}$ then $\mathsf{sk} := \mathsf{sk}_{\text{SGX}_k}$ else $\mathsf{sk} := \mathsf{sk}_{\text{SGX}_0}$.
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 $t = \langle (V, \psi_{\text{curr}}) \rightarrow (V, [\phi_{\text{all}} \vee \mathsf{pk}_{\text{SGX}}]) \rangle$.

Fig. 7. An SGX-based realization of $\mathcal{F}_{\text{daily}}$.

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 3 for details.

D 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 3. In the initial setup phase, each player P_i will prepare unsigned transactions $\{t_{i,j,k}\}_{j\in[N]\setminus\{i\},k\in[K]}$ that accuse P_j (these transactions are similar to t_1), and all players will sign transactions $t'_{i,j,k}$ that take UTXO_0 and the output of $t_{i,j,k}$ as inputs (these transactions are similar to t_2).

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.

This scheme can be implemented post-SegWit [9], as the `txid` hash excludes the `ScriptSig` witness. Each Bitcoin transaction references the `txid` hash of prior transactions, attaches witnesses (`ScriptSig`) that satisfy the predicates of those prior transactions, and defines new outputs with predicates (`ScriptPubKey`). By excluding the `ScriptSig` witnesses when hashing this transaction data into a new `txid` digest, it is possible to prepare smart contracts by signing different branches of transactions where a future transaction becomes effective only if a certain chain of prior transactions was incorporated into the blockchain (cf. [3, Section 3] and [15, Section 5.1]). In particular, it allows one to prepare $t'_{i,j,k}$ and condition its validity on that of $t_{i,j,k}$, a transaction that is not yet included the blockchain.

The parameter K specifies the number of accusation attempts that can be made; hence a malicious player that pretends to be incapacitated more than K times

will break this scheme. The SGX scheme does not exhibit this deficiency, because any player can send a fresh small amount of bitcoins to the enclave and thereby create an accusation transaction.

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

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

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