POSE: **P**ractical **O**ff-chain **S**mart Contract **E**xecution (Full Version)

Tommaso Frassetto*, Patrick Jauernig*, David Koisser*, David Kretzler†,
Benjamin Schlosser†, Sebastian Faust† and Ahmad-Reza Sadeghi*
Technical University of Darmstadt, Germany
*first.last@trust.tu-darmstadt.de
†first.last@tu-darmstadt.de

***Abstract*—Smart contracts enable users to execute payments depending on complex program logic. Ethereum is the most notable example of a blockchain that supports smart contracts leveraged for countless applications including games, auctions and financial products. Unfortunately, the traditional method of running contract code *on-chain* is very expensive, for instance, on the Ethereum platform, fees have dramatically increased, rendering the system unsuitable for complex applications. A prominent solution to address this problem is to execute code *off-chain* and only use the blockchain as a trust anchor. While there has been significant progress in developing off-chain systems over the last years, current off-chain solutions suffer from various drawbacks including costly blockchain interactions, lack of data privacy, huge capital costs from locked collateral, or supporting only a restricted set of applications.**

In this paper, we present *POSE*—a practical off-chain protocol for smart contracts that addresses the aforementioned shortcomings of existing solutions. *POSE* leverages a pool of Trusted Execution Environments (TEEs) to execute the computation efficiently and to swiftly recover from accidental or malicious failures. We show that *POSE* provides strong security guarantees even if a large subset of parties is corrupted. We evaluate our proof-of-concept implementation with respect to its efficiency and effectiveness.

I. Introduction

More than a decade ago, Bitcoin [46] introduced the idea of a decentralized cryptocurrency, marking the advent of the blockchain era. Since then, blockchain technologies have rapidly evolved and a plethora of innovations emerged with the aim to replace centralized platform providers by distributed systems. One particularly important application of blockchains concerns so-called *smart contracts*, complex transactions executing payments that depend on programs deployed to the blockchain. The first and most popular blockchain platform that supported complex smart contracts is Ethereum [57]. However, Ethereum still falls short of the decentralized "world computer" that was envisioned by the community [50]. For example, contracts are replicated among a large group of miners, thereby severely limiting scalability and leading to high costs. As a result, most contracts used in practice in the Ethereum ecosystem are very simple: 80% of popular contracts consist of less than 211 instructions, and almost half of the most active contracts are simple token managers [48]. More recently proposed computing platforms in permissionless decentralized settings (e.g., [1], [33]) suffer from similar scalability limitations.

In recent years, numerous solutions have been proposed to address these shortcomings of blockchains, one of the most promising being so-called *off-chain execution* systems. These protocols move the majority of transactions off-chain, thereby minimizing the costly interactions with the blockchain. A large body of work has explored various types of off-chain solutions including most prominently state-channels [45], [26], [22], Plasma [51], [36] and Rollups [47], [5], which are actively investigated by the Ethereum research community. Other schemes use execution agents that need to agree with each other [59], [58], rely on incentive mechanisms [35], [56], or leverage Trusted Execution Environments (TEEs) [20], [25]. A core challenge that arises while designing off-chain execution protocols is to handle the possibility of parties who stop responding, either maliciously or accidentally. Without countermeasures, this may cause the contract execution to stop unexpectedly, which violates the *liveness* property. Despite major progress towards achieving liveness in a off-chain setting, current solutions come with at least one of these limitations: $\boxed{\text{i}}$ participating parties need to lock large amounts of collateral; $\boxed{\text{ii}}$ costly blockchain interactions are required at every step of the process or at regular intervals; and finally $\boxed{\text{iii}}$ the set of participants and the lifetime need to be known beforehand, which limits the set of applications supported by the system. Additionally, existing solutions often $\boxed{\text{iv}}$ do not support keeping the contract state confidential, which is required, e.g., for eBay-style proxy auctions [9] and games such as poker. We refer the reader to Table II for an overview on related work and to Section X for a detailed discussion.

Addressing all of these limitations in one solution while guaranteeing liveness is highly challenging. Currently, there are two ways to address the risks of unresponsive parties. The first approach is to require collateral, i.e., parties have to block large amounts of money, which is used to disincentivize malicious behavior and to compensate parties in case of premature termination (cf. $\boxed{\text{i}}$). Since the amount of collateral depends on the number of participants and the amount of money in the contract, both must be fixed for the whole lifetime of the contract. To ensure payout of the collateral, the lifetime of the contract must be fixed as well (cf. $\boxed{\text{iii}}$). The second approach is to store contract state on the blockchain to enable other parties to resume execution. However, this is both expensive and leads to long waiting times due to frequent synchronization with the blockchain (cf. $\boxed{\text{ii}}$). Further, if the contract state needs to be confidential, and hence, is not publicly verifiable, verifying the correctness of the contract execution is harder (cf. $\boxed{\text{iv}}$). Realizing a system tackling all these challenges in a holistic way could pave the way towards the envisioned "world computer". We will further elaborate on the specific challenges in Section III.

Our goals and contributions: We present *POSE*, a novel off-chain execution framework for smart contracts in permissionless blockchains that overcomes these challenges, while achieving correctness and strong liveness guarantees. In *POSE*, each smart contract runs on its own subset of TEEs randomly selected from all TEEs registered to the network. One of the selected TEEs is responsible for the execution of a smart contract.

However, as the system hosting the executing TEE may be malicious (e.g., the TEE could simply be powered off during contract execution), our protocol faces the challenge of dealing with malicious operator tampering, withholding and replaying messages to/from the TEE. Hence, the TEE sends state updates to the other selected TEEs, such that they can replace the executing TEE if required. This makes *POSE* the first off-chain execution protocol with strong liveness guarantees. In particular, liveness is guaranteed as long as at least one TEE in the execution pool is responsive. Due to this liveness guarantee, there is no inherent need for a large collateral in *POSE* (cf. $\boxed{\text{i}}$). The state remains confidential, which allows *POSE* to have private state (cf. $\boxed{\text{iv}}$). Furthermore, *POSE* allows participants to change their stake in the contract at any time. Thus, *POSE* supports contracts without an a-priori fixed lifetime and enables the set of participants to be dynamic (cf. $\boxed{\text{iii}}$). Above all, *POSE* executes smart contracts quickly and efficiently without any blockchain interactions in the optimistic case (cf. $\boxed{\text{ii}}$).

This enables the execution of highly complex smart contracts and supports emerging applications to be run on the blockchain, such as federated machine learning. Thus, *POSE* improves the state of the art significantly in terms of security guarantees and smart contract features. To summarize, we list our main contributions below:

- We introduce *POSE*, a fast and efficient off-chain smart contract execution protocol. It provides strong guarantees without relying on blockchain interactions during optimistic execution, and does not require large collaterals. Moreover, it supports contracts with an arbitrary contract lifetime and a dynamic set of users. An additional unique feature of *POSE* is that it allows for confidential state execution.
- We provide a security analysis in a strong adversarial model. We consider an adversary which may deviate arbitrarily from the protocol description. We show that *POSE* achieves correctness and state privacy as well as strong liveness guarantees under static corruption, even in a network with a large share of corrupted parties.
- To illustrate the feasibility of our scheme, we implement a prototype of *POSE* using ARM TrustZone as the TEE and evaluated it on practical smart contracts, including one that can merge models for federated machine learning in 238ms per aggregation.

II. Adversary Model

The goal of *POSE* is to allow a set of users to run a complex smart contract on a number of TEE-enabled systems. Note, that *POSE* is TEE-agnostic and can be instantiated on any TEE architecture adhering to our assumptions, similar to, e.g., FastKitten [25]. In order to model the behavior and the capabilities of every participant of the system, we make the following assumptions:

A1: We assume the TEE to protect the enclave program, in line with other TEE-assisted blockchain proposals [62], [25], [20], [17], [63], [42]. Specifically:

A1.1: We assume the TEE to provide integrity and confidentiality guarantees. This means that the TEE ensures that the enclave program runs correctly, is not leaking any data, and is not tampering with other enclaves. While our proof of concept is based on TrustZone, our design does not depend on any specific TEE. In practice, the security of a TEE is not always flawless, especially regarding information leaks. However, plenty of mitigations exist for the respective commercial TEEs; hence, we consider the problem of information leakage from any specific TEE, as well as TEE-specific vulnerabilities in security services, orthogonal to the scope of this paper. We discuss some mitigations to side-channel attacks to TrustZone, as well as the possible grave consequences of a compromised or leaking TEE for the executed smart contract, in Section VII-B.

A1.2: We further assume the adversaries to be unable to exploit memory corruption vulnerabilities in the enclave program. This could be ensured using a number of different approaches, e.g., by using memory-safe languages, by deploying a run-time defense like CFI [11], or by proving the correctness of the enclave program using formal methods. The existence of these defenses can be proven through remote attestation (cf. A3).

A2: We assume the TEE to provide a good source of randomness to all its enclaves and to have access to a relative clock according to the GlobalPlatform TEE specification [31].

A3: We assume the TEE to support *secure remote attestation*, i.e., to be able to provide unforgeable cryptographic proof that a specific program is running inside of a genuine, authentic enclave. Further, we assume the attestation primitive to allow differentiation of two enclaves running the same code under the same data. Note that today's industrial TEEs support remote attestation [3], [6], [8], [34], [55].

A4: We assume the TEE operators, i.e., the persons or organizations owning the TEE-enabled machines, to have full control over those machines, including root access and control over the network. The operators can, for instance, provide wrong data to an enclave, delay the transmission of messages to it, or drop messages completely. The operators can also completely disconnect an enclave from the network or (equivalently) power off the machine containing it. However, as stated in A1.1, the operators cannot leak data from any enclave or influence its computation in any way besides by sending (potentially malicious) messages to it through the official software interfaces.

A5: We assume static corruption by the adversary. More precisely, a fixed fraction of all operators is corrupted while an arbitrary number of users can be malicious (including the case where they all are). We model each of the malicious parties as *byzantine adversaries*, i.e., they can behave in arbitrary ways.

A6: We assume the blockchain used by the parties to satisfy the following standard security properties: common prefix (ignoring the last γ blocks, honest miners have an identical chain prefix), chain quality (blockchain of honest miner contains significant fraction of blocks created by honest miners), and chain growth (new blocks are added continuously). These properties imply that valid transactions are included in one of the next α blocks and that no valid blockchain fork of length at least γ can grow with the same block creation rate as the main chain. We deem protection against network attacks (e.g., network partition attacks), which violate these standard properties, orthogonal to our work.

III. Design

POSE is a novel off-chain protocol for highly efficient smart contract execution, while providing strong correctness, privacy, and liveness guarantees. To achieve this, *POSE* leverages the integrity and confidentiality guarantees of TEEs to speed up contract execution and make significantly more complex contracts practical[1]. This is in contrast to executing contracts on-chain, where computation and verification is distributed over many parties during the mining process. *POSE* supports contracts with arbitrary lifetime and number of users, which includes complex applications like the well-known CryptoKitties [2]. We elaborate more on interaction between contracts in Appendix A. Our protocol involves users, operators and a single on-chain smart contract. *Users* aim to interact with smart contracts by providing inputs and obtaining outputs in return. *Operators* own and manage the TEE-enabled systems and contribute computing power to the *POSE* network by creating protected execution units, called *enclaves*, using their TEEs. These enclaves perform the actual state transitions triggered by users. A simple on-chain smart contract, which we call *manager*, is used to manage the off-chain enclave execution units. In the optimistic case, when all parties behave honestly, *POSE* requires only on-chain transactions for the creation of a *POSE* contract as well as the locking and unlocking of user funds. The smart contract execution itself is done without any on-chain transactions.

A. Architecture Overview

Figure 1 illustrates the high-level working of *POSE*. Before contract creation, there is already a set of enclaves that are registered with the on-chain manager contract. The registration process is explained in detail in Section V-E1. To create a *POSE* contract, a user will initialize a contract creation with the manager (Step 1), which includes a chosen enclave—out of the registered set—to execute the off-chain contract creation. In Step 2, the chosen *creator* enclave will setup the *execution pool* for the given smart contract. In Figure 1, the pool size is set to three; thus, the *creator* enclave will randomly select three enclaves from the set of all enclaves registered in the system (Step 3). In Step 4, the *creator* enclave will submit the finalized contract information to the manager. This includes the composition of the execution pool, i.e., a selected *executor* enclave, which is responsible for executing the *POSE* contract, as well as the *watchdogs*, ensuring availability. We elaborate on this in-depth in Section V-E2. In Step 5, another user can now call the new contract by directly contacting the executor. Finally, for Step 6, the executor will execute the user's contract call and distribute the resulting state to the watchdog enclaves, which confirm the state update. See Section V-E3 for a detailed specification of the execution protocol. If one of the enclaves stops participating (e.g., due to a crash), the dependent parties can challenge the enclave on the blockchain (see Section V-E4). The dependent party can either be the user awaiting response from the executor or the executor waiting for the watchdogs' confirmation. For example, if the executor stops executing the contract, the executor is challenged by the user. A timely response constitutes a successful state transition as requested by the user. Otherwise, if the current executor does not respond, one of the watchdogs will fill in as the new executor. This makes *POSE* highly available, as long as at least one watchdog enclave is dependable; thus, avoiding the need for collateral to incentivize correct behavior. Further, *POSE* supports private state, as the state is only securely shared with other enclaves.

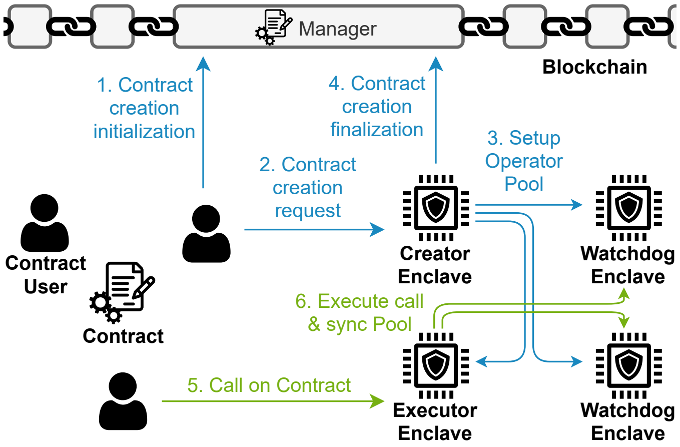

Fig. 1. Exemplary overview how *POSE* contracts are created (in blue) and executed (in green).

B. Design Challenges

We encountered a number of challenges while designing *POSE*. We briefly discuss them below.

Protection Against Malicious Operators. *POSE*'s creator, executor, and watchdogs are protected in isolated enclaves running within the system, which is itself still under control of a potentially malicious operator. Hence, operators can provide arbitrary inputs, modify honest users' messages, execute replay attacks, and withhold incoming messages. Moreover, the system and its TEE (i.e., enclaves) can be turned off completely by its operator. In order to protect honest users from malicious operators, we incorporate several security mechanisms. While malicious inputs and modification of honest users' messages can easily be prevented using standard measures like a secure signature scheme, preventing withholding of messages is more challenging. One particular reason is that for unreceived messages, an enclave cannot differentiate between unsent and stalled messages by the operator. Hence, we incorporate an on-chain challenge-response procedure, which provides evidence about the execution request and the existence of a response to the enclave.

[1] We design *POSE* without depending on any specific TEE implementation. In Section VII-B, we discuss the implications of using ARM TrustZone to realize our scheme.

Achieving Strong Liveness Guarantees. We enable dependent parties to challenge unresponsive operators via the blockchain. The challenged operators either provide valid responses over the blockchain that dependent parties can use to finalize the state transition, or they are dropped from the execution pool. In case an executor operator has been dropped, we use the execution pool to resume the execution; this requires state updates to be distributed to all watchdogs. With at least one honest operator in the execution pool, the pool will produce a valid state transition. Our protocol tolerates a fixed fraction of malicious operators as stated in our adversary model (cf. Section II). By selecting the pool members randomly, we guarantee with high probability that at least one enclave—controlled by an honest operator—is part of the execution pool. We show in Section VII-A that our protocol achieves strong liveness guarantees.

Synchronization with the Blockchain. Some of the actions taken by an enclave depend on blockchain data, e.g., deposits made by clients. Hence, it is crucial to ensure that the blockchain data available to an enclave is consistent and synchronized with the main chain. As an enclave does not necessarily have direct access to the (blockchain) network, it has to rely on the blockchain data provided by the operator. However, the operator can tamper with the blockchain data and, e.g., withhold blocks for a certain time. Thus, a major challenge is designing a synchronization mechanism that (i) imposes an upper bound on the time an enclave may lag behind the main chain, (ii) prevents an operator from isolating an enclave onto a fake side-chain, and (iii) ensures correctness and completeness of the blockchain data provided to the enclave, without (iv) requiring the enclave to validate or store the entire blockchain. We present our synchronization mechanism addressing these challenges in Section V-D.

Reducing Blockchain Interactions. Our system aims to minimize the necessary blockchain interactions to avoid expensive on-chain computations. In the optimistic scenario, the only on-chain transactions necessary are the contract creation and the transfer of coins. The transfer transactions can also be bundled to further reduce blockchain interactions. Note that the virtualization paradigm known from state channels [26] can be applied to our system. This enables parties to install virtual smart contracts within existing smart contracts, and hence, without any on-chain interactions at all. In the pessimistic scenario, i.e., if operators fail to provide valid responses, they have to be challenged, which requires additional blockchain interactions.

Support of Private State. To support private state of randomized contracts, careful design is required to avoid leakage. While the confidentiality guarantees of TEEs prevent any data leakage during contract execution, our protocol needs to ensure that an adversary cannot learn any information except the output of a successful execution. In particular, in a system where the contract state is distributed between several parties, we need to prevent the adversary from performing an execution on one enclave, learning the result, and exploiting this knowledge when rolling back to an old state with another enclave. This is due to the fact that a re-execution may use different randomness or different inputs resulting in a different output. We prevent these attacks by outputting state updates to the users only if all pool members are aware of the new state. Moreover, by solving the challenge of synchronization between enclaves and the blockchain, we prevent an adversary from providing a fake chain to the enclave, in which honest operators are kicked from the execution pool. Such a fake chain would allow an attacker to perform a parallel execution. While results of the parallel (fake) execution cannot affect the real execution, they can prematurely leak private data, e.g. the winner in a private auction.

IV. Definitions & Notations

In the following, we introduce the cryptography primitives, definition, and notations used in the *POSE* protocol.

Cryptographic Primitives. Our protocol utilizes a public key encryption scheme $(GenPK, Enc, Dec)$, a signature scheme $(GenSig, Sign, Verify)$, and a secure hash function $H(\cdot)$. All messages sent within our protocol are signed by the sending party. We denote a message m signed by party P as $(m; P)$. The verification algorithm $Verify(m')$ takes as input a signed message $m' := (m; P)$ and outputs $\texttt{ok}$ if the signature of P on m is valid and $\texttt{bad}$ otherwise. We identify parties by their public keys and abuse notation by using P and P's public key pk_P interchangeably. This can be seen as a direct mapping from the identity of a party to the corresponding public key.

TEE. We comprise the hardware and software components required to create confidential and integrity-protected execution environments under the term TEE. An operator can instruct her TEE to create new *enclaves*, i.e., new execution environments running a specified program. We follow the approach of Pass et al. [49] to model the TEE functionality. We briefly describe the operations provided by the ideal functionality formally specified in [49, Fig. 1]. A TEE provides a $TEE.install(prog)$ operation which creates a new enclave running the program $prog$. The operation returns an enclave id eid. An enclave with id eid can be executed multiple times using the $TEE.resume(eid, inp)$ operation. It executes $prog$ of eid on input inp and updates the internal state. This means in particular that the state is stored across invocations. The $resume$ operation returns the output out of the program. We slightly deviate from Pass et al. [49] and include an attestation mechanism provided by a TEE that generates an attestation quote ρ over $(eid, prog)$. ρ can be verified by using method $VerifyQuote(\rho)$. We consider only one instance $\mathcal{E}$ running the *POSE* program per TEE. Therefore, we simplify the notation and write $\mathcal{E}(inp)$ for $TEE.resume(eid, inp)$.

Blockchain. We denote the blockchain by BC and the average block time by τ. A block is considered final if it has at least γ confirmation blocks. Throughout the protocol description in Section V-E, enclaves consider only transactions included in final blocks. Finally, we define that any smart contact deployed to the blockchain is able to access the current timestamp using the method $\mathsf{BC}.now$ and the hash of the most recent 265 blocks [7] using the method $\mathsf{BC}.bh(i)$ where i is the number of the accessed block. These features are available on Ethereum.

V. The *POSE* Protocol

The *POSE* protocol considers four different roles: a manager smart contract deployed to the blockchain, operators that run TEEs, enclaves that are installed within TEEs, and users that create and interact with *POSE* contracts. In the following, we will shortly elaborate on the on-chain smart contract and the program executed by the enclaves, explain the *POSE* protocol, and finally explain further security mechanisms that are omitted in the protocol description.

A. Manager

We utilize an on-chain smart contract in order to manage the *POSE* system's on-chain interactions. We call this smart contract *manager* and denote it by M. On the one hand, M keeps track of all registered *POSE* enclaves. This enables the setup of an execution pool whenever an off-chain smart contract instance is created. On the other hand, it serves as a registry of all *POSE* contract instances. M stores parameters about each contract to determine the instance's status. We denote the tuple describing a contract with identifier id as M^{id}. In particular, the manager stores the creator enclave ($creator$), a hash of the program code ($codeHash$), the set of enclaves forming the execution pool ($pool$), a total amount of locked coins ($balance$), and a counter of withdrawals ($payouts$). We set the field $creator$ to $\perp$ after the creation process has been completed to identify that a contract is ready to be executed. Moreover, for both executor and watchdog challenges, the contract allocates storage for a tuple containing the challenge message ($c1Msg$ resp. $c2Msg$), responses ($c1Res$ resp. $c2Res$), and the timestamp of the challenge submission ($c1Time$ resp. $c2Time$). A non-empty field $c1Time$ resp. $c2Time$ signals that there is a running challenge.

Every *POSE* enclave maintains a local version of the manager state extracted from the blockchain data it receives from the operator when being executed. This enables all enclaves to be aware of on-chain events, e.g., ongoing challenges.

B. POSE Program

All enclaves registered within the system run the *POSE* program that enforces correct execution and creation of *POSE* contracts. In practice, the *POSE* program's source code will be publicly available, e.g., in a public repository, so that the community can audit it. Our protocol ensures that all registered enclaves run this code using remote attestation (cf. Section V-E1: Enclave registration). We present methods required for the execution protocol in Program 1 and defer methods for the contract creation into Program 3 in Appendix B.

Whenever an enclave is invoked, it synchronizes itself with the blockchain network and receives the relevant blockchain data in a reliable way (cf. Section V-D). This way, the POSE program has access to the current state of the manager. In order to support arbitrary contracts, we define a common interface in Section V-C that is used by the POSE program to invoke contracts.

Enclaves running the *POSE* program only accept signed messages as input. The public keys of pool members for signature verification are derived from the synchronized blockchain data. According to our adversary model (cf. Section II), the adversary cannot read or tamper messages originating from honest users or the enclave itself. Further, the contracts themselves keep track of already received execution requests and do not perform state transitions for duplicated requests. (cf. Section V-C). This prevents replay attacks against both, executive and watchdog enclaves.

Program 1: *POSE* Program (execution) executed by enclave T

```
Upon invocation with input blockchain data BC, store BC.
Upon receiving $m := (\texttt{execute}, id, r, move; U)$, do:
  1) If $M^{id}.pool[0] \neq T$ or $\mathcal{T}^{id}_{wait} \neq \emptyset$, return $(\texttt{bad})$.
  2) Execute $C_{id}.nextState(U, \mathsf{BC}, move, H(m))$.
  3) Store $\mathcal{T}^{id}_{wait} = M^{id}.pool$ and $h^{id} = H(m)$, set
     $c = Enc(C_{id}.getState(all); key^{id})$ and return
     $(\texttt{update}, id, c, h^{id}; T)$.
Upon receiving $m := (\texttt{update}, id, c, h; T')$, do:
  1) If $T' \neq M^{id}.pool[0]$ or $T \notin M^{id}.pool$, return $(\texttt{bad})$.
  2) Define $state = Dec(c; key^{id})$ and call
     $C_{id}.update(state, h)$.
  3) Return $(\texttt{confirm}, id, h; T)$.
Upon receiving $\{m_i := (\texttt{confirm}, id, h_i; T_i)\}_i$, do:
  1) If $M^{id}.pool[0] \neq T$ or $\mathcal{T}^{id}_{wait} = \emptyset$, return $(\texttt{bad})$.
  2) Set $\mathcal{T}^{id}_{wait} = \mathcal{T}^{id}_{wait} \cap M^{id}.pool$.
  3) For each $m_i$ do:
     • If $h_i \neq h^{id}$ or $T_i \notin \mathcal{T}^{id}_{wait}$, skip $m_i$.
     • Otherwise remove $T_i$ from $\mathcal{T}^{id}_{wait}$.
  4) If $\mathcal{T}^{id}_{wait} \neq \{T\}$, return $(\texttt{bad})$. Otherwise, set $\mathcal{T}^{id}_{wait} = \emptyset$,
     $state := C_{id}.getState(pub)$ and return
     $(\texttt{ok}, id, state, h^{id}; T)$.
```

C. POSE Contracts

Although our system supports the execution of arbitrary smart contracts, the contracts need to implement a specific interface (cf. Program 2). This allows any *POSE* enclave to trigger the execution without knowing details about the smart contract functionality. Upon an execution request from some user, the *POSE* enclave provides the user's identity U, blockchain data BC, the description of the user's request, $move$, and the request hash, h, to the smart contract's method $nextState$. The smart contract first processes the relevant blockchain data and marks the current length of the blockchain as processed. This feature is mainly used to enable smart contracts to deal with money, i.e., to detect on-chain deposits and withdrawals. We elaborate on the processing of blockchain data in Section V-D, and on the money mechanism of the *POSE* system in Appendix D.

Note that double spending within a contract is prevented due to sequential processing of any execution request, and double spending of on-chain payouts is prevented by the mechanism explained in Appendix D. After the blockchain data is processed, $nextState$ executes the move requested by the user and updates the state accordingly. Method $update$ takes state new and hash h (for preventing replay attacks) as input and sets new as the contract state. This includes the length of the blockchain that is marked as processed. Further, the smart contract provides method $getState$. If called with $flag = all$, it returns the whole smart contract state. Otherwise, if called with $flag = pub$, it returns only the public state. In order to prevent replay attacks, each smart contract maintains a list with the hashes of already received execution requests, Rec. In case of duplicated requests, i.e., $h \in Rec$, both the $nextState$ method and the $update$ method, do not perform any state transition. Instead, they interpret the request as a dummy

Program 2: Interface of a contract C executed within a *POSE* enclave

```
Function: $nextState(U, \mathsf{BC}, move, h)$
Function: $update(new, h)$
Function: $getState(flag)$
```

move that has no effect on the state. If executed successfully, the $nextState$ method adds the executed request to Rec, i.e., $Rec = Rec \cup \{h\}$. As Rec is part of the state, it is updated by the $update$ method as well. While it might seem counter intuitive to overwrite the list of received requests, this feature is required to ensure that all enclaves are aware of the same transition history; even if an executor distributes a state update to just a subset of watchdogs before getting kicked [2].

We consider the initial state of a smart contract to be hard-coded into the smart contract description. If an enclave creates a new smart contract instance, the initial state is automatically initialized. A contract state additionally contains a variable to store the highest block number of the already processed blockchain data. This variable is used to detect which transactions of received blockchain data have already been handled.

D. Synchronization

As some of the actions taken by an enclave depend on blockchain data, e.g., deposits to the contract, it is crucial to ensure that the blockchain state available to a registered enclave $\mathcal{E}$ is consistent and synchronized with the main chain. In particular, blocks that are considered final by some party, will eventually be considered final by all parties. We design a synchronization mechanism that allows $\mathcal{E}$ to synchronize itself without having to validate whole blocks. Note that $\mathcal{E}$ has access to a relative time source according to our adversary model (see Section II).

Upon initialization, $\mathcal{E}$ receives a chain of block headers BCH of length $\gamma + 1$. Note that the first block p of BCH can be considered final since it has γ confirmation blocks. First, $\mathcal{E}$ checks that BCH is consistent in itself and sets its own clock to be the one of the latest block's timestamp. Second, $\mathcal{E}$ signs block p as blockchain evidence that needs to be provided to the manager. The registration mechanism (cf. Section V-E1) uses this evidence to ensure that $\mathcal{E}$ has been initialized with a valid sub-chain of the main-chain up to block p. Further, the registration mechanism checks that p is at most τ^{on}_{slack} blocks behind the current one; τ^{on}_{slack} needs to account for the confirmation blocks and the fact that transactions are not always mined immediately. Via this parameter, we can set an upper bound to the time $\tau^{\mathit{off}}_{slack}$ an enclave may lag behind; $\tau^{\mathit{off}}_{slack}$ additionally considers potential block variance and the fact that miners have some margin to set timestamps. In the following, we call $\tau^{\mathit{off}}_{slack}$ *slack* [3]. Clients that want a contract execution to capture on-chain effects, e.g., deposits, wait until the enclave considers the corresponding block as final, even when being at slack.

Once successfully initialized, $\mathcal{E}$ synchronizes itself with the blockchain. Whenever a registered enclave is executed throughout the protocol, it receives the sub-chain of block headers BCH' that have been mined since the last execution. $\mathcal{E}$ checks that BCH' is a valid successor of BCH where blocks in BCH that have not been final may change. Further, $\mathcal{E}$ checks that the latest block in BCH' is at most $\tau_{variance}$ behind the own clock; $\tau_{variance}$ captures the variance in the block creation time and the fact that miners have some margin to set timestamps. When receiving a block that is before the own clock, the clock is adjusted.

Finally, we need to prevent an operator from isolating its enclave by setting up a valid sidechain with manipulated timestamps. To this end, we require the operators to periodically provide new blocks to $\mathcal{E}$ even if $\mathcal{E}$ does not need to take any action. In particular, we require that the operator provides at least L blocks within time τ_p where τ_p accounts for potential block time variances. The system is secure as long as the attacker cannot mine L blocks within time τ_p while the honest miners can. Hence, the selection of τ_p and L has some implications on the fraction of adversarial computing power that can be tolerated by the system. Since 2018, an interval of 50 (100, 200, 300) blocks took at most 33 (28, 26, 25) seconds per block [10], which might all be reasonable choices for L and $\frac{\tau_p}{L}$. As the average block time is around 13 seconds [4], the adversary gets $2-3$ times more time to mine the blocks of its sidechain. This means that the system can tolerate adversarial fractions from a third (when instantiated with $L = 300$ and $\tau_p = 25 \cdot L$) to a forth (when instantiated with 50 and $33 \cdot L$).

While the above techniques allow an enclave to synchronize itself, the enclave does not have access to the block data, yet. Instead of requiring enclaves to validate whole blocks, we require operators to filter the relevant transactions and provide them to the enclave while enabling the enclaves to check correctness and completeness of the received data itself. For the latter, we introduce $incrTxHash$, a hash maintained by the manager and all initialized enclaves that is based on all relevant transactions. Whenever the manager receives a relevant transaction tx, it updates $incrTxHash$, such that $incrTxHash_{i+1}$ is defined as

$$H(incrTxHash_i \mid\mid tx.data \mid\mid tx.sender \mid\mid tx.value)$$

where $tx.data$ is the raw data of tx, $tx.sender$ denotes the creator of tx, and $tx.value$ contains the amount of any deposits or withdrawals. Whenever enclaves are invoked with new blocks, operators additionally provide all relevant transactions. This way, enclaves can re-compute the new incremental hash and compare the result to the on-chain value of $incrTxHash$. In order to verify that the on-chain $incrTxHash$ is indeed part of the main chain, operators additionally provide a Merkle proof showing that $incrTxHash$ is part of the state tree. The proof can be validated using the state root, which is part of the block headers provided to the enclaves. This way, enclaves can ensure that operators have not omitted or manipulated any relevant transactions.

[2] In practice, the state update removes at most the last element from the request history; a fact that can be exploited to reduce the size of state updates.

[3] We can reduce the slack assuming an absolute source of time realized via trusted NTP servers, cf. [20], by enabling the enclave to check if she was invoked with the most recent block headers up to some variance of the timestamps.

E. Protocol Description

In this section, we dive into a detailed description of our protocol. We present 1) enclave registration, 2) contract creation, 3) contract execution, and 4) the challenge-response parts of our protocol. The *POSE* program running inside the operators' enclaves is stated in Section V-B. For the sake of exposition, we extracted the validation steps performed by the manager on incoming messages into Program 4 in Appendix B. Further, we elaborate in Appendix D on the coin flow within the protocol.

1) Enclave Registration: Operator O controlling some TEE unit can contribute to the *POSE* system by instructing his TEE to create a new *POSE* enclave $\mathcal{E}_O$. The protected execution environment $\mathcal{E}_O$ needs to be initialized with the *POSE* program presented in Section V-B. During the creation of $\mathcal{E}_O$, an asymmetric key pair (pk_O, sk_O) is generated. The secret key sk_O is stored inside the enclave and hence is only accessible by the *POSE* program running in $\mathcal{E}_O$. The public key pk_O is returned as output to the operator. Furthermore, operator O uses the TEE to produce an attestation ρ_O stating that the freshly generated enclave $\mathcal{E}_O$ runs the *POSE* program and controls the secret key corresponding to pk_O.[4]

Finally, O sends the latest $\gamma+1$ block headers BCH together with the relevant blockchain data to the enclave which validates the consistency of the block headers and completeness of the blockchain data (cf. Section V-D) and returns a blockchain evidence ρ_O^{BC}, i.e., a signed tuple containing the blockhash and the number of the latest final block known to the enclave. After operator O created a new *POSE* enclave $\mathcal{E}_O$, O can register $\mathcal{E}_O$ by sending $m := (\mathtt{register}, \mathcal{E}_O, \rho_O, \rho_O^{\mathsf{BC}}; O)$ to manager M. M verifies that ρ_O is a valid attestation and that ρ_O^{BC} refers to a block on the blockchain known to M that is not older than τ_{slack}^{on} blocks. If the check holds and the signature of the operator is valid, i.e., $\mathit{Verify}(m) = \mathtt{ok}$, M adds $\mathcal{E}_O$ (identified by its public key pk_O) to the set of registered enclaves, i.e., $M.\mathit{registered} := M.\mathit{registered} \cup \{\mathcal{E}_O\}$. This procedure ensures that all registered enclaves run the *POSE* program and that the secret key sk_O remains private. Hence, re-attesting enclaves during later protocol steps is not needed.

2) Contract Creation: The creation protocol is initiated by a user U who wants to install a new smart contract, with program code code, into the *POSE* system. We outline the protocol in the following and provide a full explanation and specification in Appendix B.

U picks an arbitrary registered enclave $\mathcal{E}_C$ and sends a creation initialization to M containing $H(\mathit{code})$ and $\mathcal{E}_C$. The manager M allocates a new contract tuple with a fresh identifier id. Next, U sends a creation request, containing code, to $\mathcal{E}_C$ which randomly selects n enclaves for the contract execution pool and samples a symmetric pool key. The generated information is distributed in a confidential way to all pool enclaves, which install a new smart contract with code code and confirm the installation to $\mathcal{E}_C$. Finally, $\mathcal{E}_C$ signs a creation confirmation, which is submitted to M that marks the contract as created.

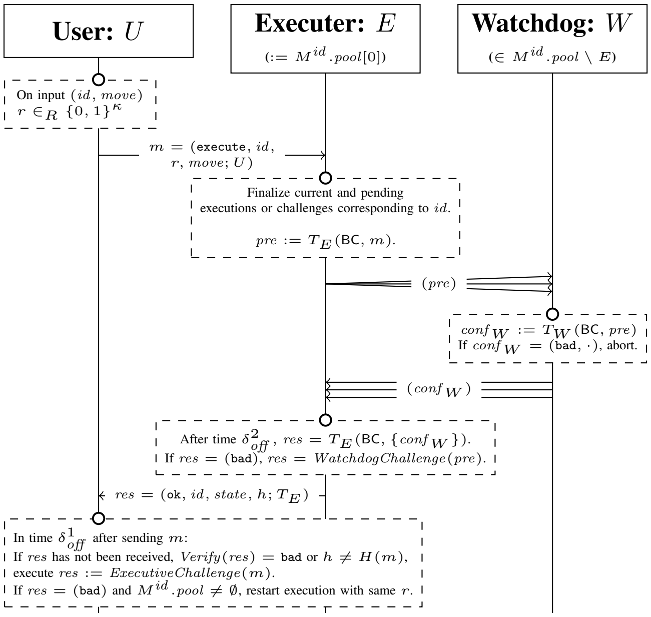
User: U
Executer: E
$(:= M^{id}.\mathit{pool}[0])$
Watchdog: W
$(\in M^{id}.\mathit{pool} \setminus E)$
On input (id, move)
$r \in_R \{0,1\}^\kappa$
$m = (\mathtt{execute}, id, r, \mathit{move}; U)$
Finalize current and pending executions or challenges corresponding to id.
$\mathit{pre} := T_E(\mathsf{BC}, m)$.
(pre)
$\mathit{conf}_W := T_W(\mathsf{BC}, \mathit{pre})$
If $\mathit{conf}_W = (\mathtt{bad}, \cdot)$, abort.
(conf_W)
After time δ_{off}^2, $\mathit{res} = T_E(\mathsf{BC}, \{\mathit{conf}_W\})$.
If $\mathit{res} = (\mathtt{bad})$, $\mathit{res} = \mathit{WatchdogChallenge}(\mathit{pre})$.
$\mathit{res} = (\mathtt{ok}, id, \mathit{state}, h; T_E)$
In time δ_{off}^1 after sending m:
If res has not been received, $\mathit{Verify}(\mathit{res}) = \mathtt{bad}$ or $h \neq H(m)$,
execute $\mathit{res} := \mathit{ExecutiveChallenge}(m)$.
If $\mathit{res} = (\mathtt{bad})$ and $M^{id}.\mathit{pool} \neq \emptyset$, restart execution with same r.

Fig. 2. Detailed execution protocol.

If the contract is not created within a certain time, U starts a creation challenge. If any pool member does not respond to $\mathcal{E}_C$ timely, $\mathcal{E}_C$ starts a pool challenge (cf. Section V-E4).

3) Contract Execution: The execution protocol is initiated by a user U who wants to execute an existing smart contract, identified by id, with input move. The protocol is specified in Figure 2. Program 1 specifies the parts of the *POSE* program that are relevant for the contract execution.

To trigger the execution, U sends an execution request to operator E controlling the executor enclave $\mathcal{E}_E$, the first enclave in the contract pool stored at M. $\mathcal{E}_E$ executes the request and securely propagates the new state to all other pool members, called watchdogs. If any watchdog does not confirm in time, it is challenged by E (cf. *Challenge-Response*). Eventually, $\mathcal{E}_E$ receives confirmations from all watchdogs or the unresponsive watchdogs are kicked out of the pool. Either way, $\mathcal{E}_E$ outputs the new public state to U. We want to stress that this way no party gets to know the result of an update before all pool members agree on the update. If E does not respond in time, it is challenged by U (cf. *Challenge-Response*). If E does not respond to the challenge, it is kicked from the pool by U. The next enclave in the pool, $\mathcal{E}'_E$, takes over as the new executor. At this point, the new executor might be on a different state than the other pool members, since $\mathcal{E}'_E$ might have received the previous state update but some other pool members not, or vice versa.

Our system automatically ensures that all enclaves share the same contract state after the next successful execution, in which $\mathcal{E}'_E$ distributes its state to the other enclaves. Let us call the previous incompletely distributed update update and the new updated initiated by $\mathcal{E}'_E$ update'. In case $\mathcal{E}'_E$ has received update, update' is a successor of update, and hence, covers both updates. This way, a watchdog that updates to update'

[4] An attestation mechanism can be designed based on a chain of trust, where the TEEs manufacturer's public key represents the root. This way a smart contract knowing a list of public keys can verify an attestation quote without further interaction. We omit further details about the practical implementation and refer the reader to [49].

essentially contains both executions, $update$ and $update'$. In case $\mathcal{E}'_E$ has not received $update$ but the other watchdogs have, $\mathcal{E}'_E$ either propagates the update already known to the watchdogs, i.e., $update = update'$, or a concurrent one, i.e., $update \neq update'$. For the former, the watchdogs interpret the update as a dummy update without any effect as the corresponding execution request is already within their list of received request hashes (cf. Section V-C). For the latter, the update of the watchdogs is overwritten by the one of the executive enclave. As $update$ has been incomplete, and hence, produced no public output, it is safe to overwrite this update. To produce a public output for $update$, all pool enclaves including $\mathcal{E}'_E$ would have to confirm $update$.

Finally, U can just submit the previous execution request with the same random nonce r to $\mathcal{E}'_E$. In case the enclave has already seen this request, it is interpreted as empty dummy move which prevents a duplicated execution.

4) Challenge-Response: If any party does not receive a *timely* response to its messages during the off-chain execution, it challenges the receiver on-chain. Therefore, all operators need to monitor the blockchain for any on-chain challenges. We will elaborate on the timeouts ($\delta^{\dagger}_{\star}$), where $\dagger \in \{0, 1\}$ and $\star \in \{\mathit{off}, \mathit{on}\}$, which define the notion of *timely* in Appendix C. In particular, we describe the relation between δ^1_* and δ^2_*. The challenge-response procedure is executed in all of the following cases.

(a) The creator enclave has not responded to the user within time δ^1_{off} during the contract creation protocol.
(b) At least one pool enclave has not responded to the creator enclave within time δ^2_{off} during the contract creation protocol.
(c) The executor enclave has not responded to the user within time δ^1_{off} during the contract execution protocol.
(d) At least one watchdog enclave has not responded to the executor enclave within time δ^2_{off} during the contract execution protocol.

Since (a) is conceptually identically to (c) and (b) to (d), we present the executor challenge (c) and the watchdog challenge (d) in Figure 3 and Figure 4. The specifications of (a) and (b) are provided in the Appendix B in Figure 6 and Figure 7.

For the executor challenge as shown in Figure 3, suppose user U has not received a result from the executor enclave $\mathcal{E}_E$ within time δ^1_{off}, then, U starts the challenge-response protocol. To this end, U sends the execution request to the manager M who verifies the validity of the message (cf. Program 4). If all checks hold, M stores the challenge message and then starts timeout δ^1_{on} by storing the current timestamp. As soon as the challenge message is recorded on-chain, the operator of the executor enclave $\mathcal{E}_E$ extracts the execution request from the challenge and starts the execution. Performing the execution request is identical to the standard execution as described in Section V-E3. However, the operator prioritizes challenges over off-chain execution requests to avoid getting kicked. Additionally, if $\mathcal{E}_E$ already performed the state update and state propagation, the operator may use the already obtained result as response. Either way, if the operator sends a response message in time, the manager M checks the validity of the message and whether or not it matches the stored challenge. If all checks succeed, M stores the result and removes the challenge message. This finalizes the challenge procedure. If the operator does not send a valid response in time δ^1_{on}, user U sends message `finalize` to M. This triggers the manager to kick $\mathcal{E}_E$ from the execution pool of this contract and assign the next enclave in the list as the new executor enclave, if possible. Then, if the pool is not empty, U restarts the execution. As M only accepts a response if the operator executed the challenged request correctly, the described procedure ensures that there is either a consistent state transition or $\mathcal{E}_E$ is kicked from the execution pool, hence, ensuring liveness as long as there remains one active operator.

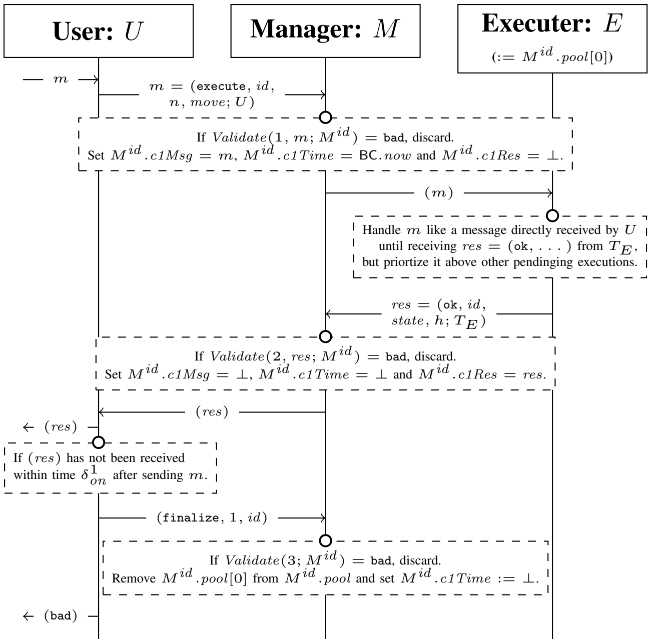
User: U
Manager: M
Executer: E
$(:= M^{id}.pool[0])$
— m →
$m = (\texttt{execute}, id, n, move; U)$
If $Validate(1, m; M^{id}) = \texttt{bad}$, discard.
Set $M^{id}.c1Msg = m$, $M^{id}.c1Time = \texttt{BC}.now$ and $M^{id}.c1Res = \bot$.
(m)
Handle m like a message directly received by U
until receiving $res = (\texttt{ok}, \ldots)$ from T_E,
but priortize it above other pendinging executions.
$res = (\texttt{ok}, id, state, h; T_E)$
If $Validate(2, res; M^{id}) = \texttt{bad}$, discard.
Set $M^{id}.c1Msg = \bot$, $M^{id}.c1Time = \bot$ and $M^{id}.c1Res = res$.
(res)
← (res)
If (res) has not been received
within time δ^1_{on} after sending m.
$(\texttt{finalize}, 1, id)$
If $Validate(3; M^{id}) = \texttt{bad}$, discard.
Remove $M^{id}.pool[0]$ from $M^{id}.pool$ and set $M^{id}.c1Time := \bot$.
← $(\texttt{bad})$

Fig. 3. Detailed executor challenge protocol.

Since the executor enclave $\mathcal{E}_E$ is dependent on the confirmation message from all watchdog enclaves, it is necessary to allow $\mathcal{E}_E$ to challenge the watchdog enclaves as well (Figure 4). In this case, the executor enclave acts as the challenger and all watchdog enclaves need to provide a confirmation message as response. At the end of this challenge-response protocol, all unresponsive watchdog enclaves are removed from the execution pool. The executor enclave then continues performing the execution with all confirmations obtained during this procedure. Again, M only accepts responses if the watchdog executed the state update correctly, hence, ensuring that a watchdog either performs the correct state update or is kicked from the pool.

F. Security Remarks

To keep the protocol description compact, we omitted some security features from the specification, which we explain in this section.

Allowing unrestricted execution requests comes with the problem that malicious users can send requests whose execution takes a disproportional amount of time, e.g., due to infinite loops. If the execution time exceeded the boundaries defined by the on-chain timeouts, malicious users could exploit

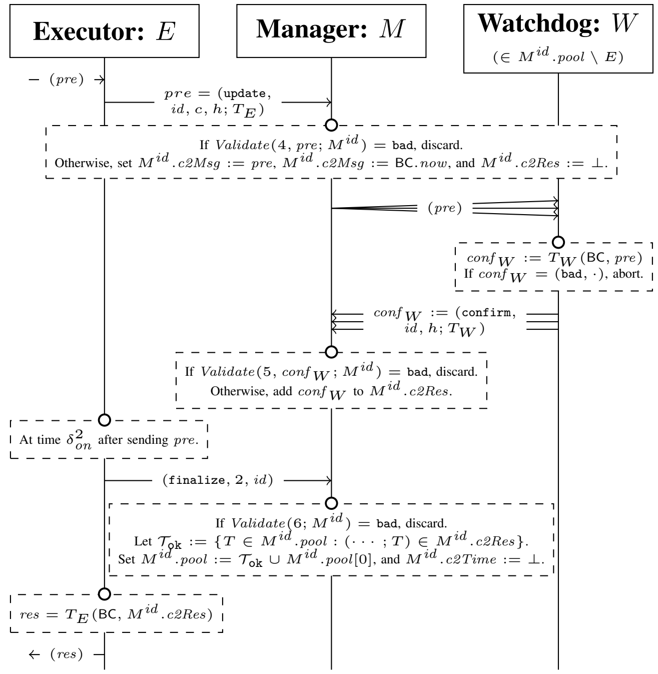
Executor: E
Manager: M
Watchdog: W
$(\in M^{id}.pool \setminus E)$
(pre)
$pre = (\texttt{update}, id, c, h; T_E)$
If $Validate(4, pre; M^{id}) = \texttt{bad}$, discard.
Otherwise, set $M^{id}.c2Msg := pre$, $M^{id}.c2Msg :=$ BC$.now$, and $M^{id}.c2Res := \perp$.
(pre)
$conf_W := T_W(\text{BC}, pre)$
If $conf_W = (\texttt{bad}, \cdot)$, abort.
$conf_W := (\texttt{confirm}, id, h; T_W)$
If $Validate(5, conf_W; M^{id}) = \texttt{bad}$, discard.
Otherwise, add $conf_W$ to $M^{id}.c2Res$.
At time δ^2_{on} after sending pre.
$(\texttt{finalize}, 2, id)$
If $Validate(6; M^{id}) = \texttt{bad}$, discard.
Let $\mathcal{T}_{\texttt{ok}} := \{T \in M^{id}.pool : (\cdots; T) \in M^{id}.c2Res\}$.
Set $M^{id}.pool := \mathcal{T}_{\texttt{ok}} \cup M^{id}.pool[0]$, and $M^{id}.c2Time := \perp$.
$res = T_E(\text{BC}, M^{id}.c2Res)$
(res)

Fig. 4. Detailed watchdog challenge protocol.

this behavior to kick honest operators from an execution pool. This operator *denial of service* attack harms the liveness property of the system. In order to mitigate the vulnerability, we introduce an upper bound to the computation complexity of a single contract execution. Once the bound is reached, the executor enclave stops executing and reverts the state but still provides a valid output. The timeouts in the system are set such that an honest operator cannot be kicked from an execution pool even if an execution takes the maximum amount of computation. The same applies to update and creation requests, where failed creations return a *fail confirmation* that can be submitted to the manager instead of the creation confirmation. A fail confirmation triggers the manager to mark the contract as crashed. Note that the *POSE* system still supports the execution of arbitrary complex smart contracts as the timeouts and hence the upper bounds can be set arbitrarily high (cf. Appendix C). Additionally, all contracts of an operator are executed and challenged independently, and thus, contracts do not block each other.

While we have assumed that all operators run only one *POSE* enclave, multiple enclaves can be created in practice. This enables the opportunity of a *sybil attack*, where a malicious operator generates multiple *POSE* enclaves to increase its share in the system and hence harm the liveness property. This attack can be mitigated by forcing an operator to deposit funds at each enclave registration and which will be paid back to the operator only if she behaves honestly. We note that this deposit is independent of any contract and its parties. Now, such an attack is directly linked to financial loss. See Section VI for more discussions about incentives and fees.

In order to enhance *privacy*, neither users nor operators send inputs or respectively execution results in clear. Instead, users encrypt inputs using hybrid encryption based on the public key of the executor enclave. Additionally, users specify a symmetric key in their execution request, which is used to encrypt the result of the execution when sent back to the user. This way, inputs and results are private and cannot be eavesdropped by a malicious operator.

The term *griefing* denotes attacks where an adversary forces an honest party to interact with the blockchain in order to generate financial damage to this party. Especially when blockchain transactions require high fees, such attacks pose serious vulnerabilities. In regards to challenges within the *POSE* protocol, we mitigate the attack surface for griefing attacks by incorporating a mechanism in the manager that fairly splits the fees for challenge and response between the challenger and the challenged party. The same mechanism can be used for the contract creation process.

An adversary executing a *clogging* attack sends many transactions to the system to prevent honest users from issuing transactions. In the context of *POSE*, an off-chain clogging attack results in honest clients making an on-chain challenge to ensure that their requests will be processed. Hence, a successful clogging attack has to be performed on-chain. For the on-chain challenge, our system inherits the vulnerabilities of the underlying blockchain.

VI. Extensions

We simplified some protocol steps in order to make the protocol description more compact and easier to understand. We discuss the most important extensions and their benefits in this section.

Contract & Operator Lifecycle. A mechanism that releases enclaves from their execution duty can be integrated. This allows operators to voluntarily withdraw their enclaves from an execution pool. On the one hand, terminated contracts can be closed, which releases all pool enclaves from their execution duty. On the other hand, it enables to withdraw a single enclave and exchanging it by a randomly chosen replacement enclave. Additionally, a replacement strategy is also applicable to the scenarios in which enclaves are kicked. The latter extension reduces the chance of a contract crash, the event in which no more operator remains. We stress that these extensions can easily be achieved by adding the functionality to our *POSE* program and the manager. In case a contract is idle for a long time, an extension may be implemented that allows operators to *hibernate* their respective enclave. The enclave state can be stored on disk by encrypting it with a key that is kept alive in the hibernating enclave; thus, only requiring minimal overhead in memory. The *POSE* program ensures freshness by synchronizing with the blockchain; thus, preventing rollback attacks.

Incentives. Although *POSE* provides security not only against rational but also byzantine adversaries, it is beneficial to introduce incentives for operators to join the system and act honestly. Moreover, operators can be compensated for on-chain transactions. Such incentives can be achieved by introducing execution fees paid by the users to the operators. We expect these fees to be significantly lower than Ethereum transaction fees since replication of computation is only required among a small pool. Additionally, registration fees for operators can be used to mitigate the risk for sybil attacks.

By mitigating these attacks and due to the random assignment of enclaves to contract pools, operators can only actively enforce centralization at high cost.

Efficiency Improvements. Instead of propagating each contract invocation, a more fine-grained distinction based on the action can be added. In particular, a simple state retrieval must not be propagated. In order to improve the efficiency of the manager, messages and responses are not stored persistently. Instead, only their hashes are stored and the actual data is propagated via events. Moreover, the total on-chain transactions can be reduced by letting the executor enclave challenge only the unresponsive watchdog enclaves.

VII. Security Analysis

In this section, we present security considerations of *POSE* based on the adversary model stated in Section II.

A. Protocol Security

We analyze the security of our protocol under the assumption of an IND-CPA secure encryption scheme, an EU-CMA secure signature scheme and a collision resistant hash function in the following. We present definitions of correctness, ϵ-liveness and state privacy.

1) Correctness: We define a state update as the evaluation of a transition function f, which receives as inputs a user U, a user input $move$ and a copy of the blockchain BC. The *correctness* property states that each state update evaluates the transition function as defined by the contract code with valid inputs, i.e., U is the (potentially malicious) client triggering the transition, $move$ the input of U and BC a valid copy of the blockchain that is at most τ_{slack}^{off} behind the main chain.

Claim 1 (Correctness)**:** *The POSE protocol satisfies correctness.*

We first note that according to our adversary model, a corrupted operator may delete any message intended for her enclave or generated from her enclave. However, the correct execution of the *POSE* program inside the enclave cannot be influenced. When an operator creates a *POSE* enclave, the registration process ensures that the new enclave indeed runs the *POSE* program. To this end, our protocol utilizes the TEE attestation mechanism, which generates a verifiable statement that the enclave is running a specific program. Upon registration with the manager M, M checks the validity of the attestation statement as well as the blockchain evidence, the signed hash and number of the latest block known to the enclave. M only registers the enclave in the system if the new enclave is running the *POSE* program and is not further behind than maximally τ_{slack}^{off}. Finally, the TEE integrity and confidentiality guarantees ensure that a malicious operator cannot modify the enclave's code, tamper with its state or access its private data, in particular, its signature keys.

During the creation of a contract, the pool enclaves attest the code of the installed contract to the creation enclave. The creator checks that the code is consistent with the hash stored in the manager before signing a creation confirmation. Hence, it is not possible, without breaking the EU-CMA security of the signature scheme or the collision resistance of the hash function, to create a valid creation confirmation for a contract with different code than specified by the creation request.

Next, contract state updates can only be triggered by invoking the executor enclave with an execution request or invoking a watchdog enclave with an update request. The correctness of the latter is reduced to the correctness of the former. To see this, we observe that any update request to a watchdog enclave requires to be signed by the executor enclave. Clearly, the executor enclave only signs updates corresponding to its own executions. Therefore, an adversary cannot forge incorrect update request without breaking the unforgeability of the signature scheme. Also, the executor enclave can only issue a new state update if all watchdogs confirmed the previous one. Hence, it is not possible to tamper with the order in which the update requests are provided to a watchdog enclave.

As stated before, the TEE integrity guarantees ensure the correct execution of the program code and hence the correct execution of the smart contract. It follows that a state update can only be achieved by providing inputs to the executor enclave. The executor enclave receives a signed message containing the action $move$ from user U and the relevant blockchain data from its operator. In Section V-D, we describe how our protocol achieves secure synchronization between the executor enclave and the blockchain. In particular, the synchronization mechanism ensures that the blockchain data accepted by an enclave is correct and complete in regard to a correct blockchain copy that is at most τ_{slack}^{off} behind the main chain. This guarantees that BC, represented by the received blockchain data, is a synchronized copy of the current blockchain. In order to protect inputs by honest users U, $move$ needs to be signed by U. This means an adversary cannot tamper with the input without breaking the signature scheme.

Finally, we note that each *POSE* enclave maintains a list of received messages. Since an honest user randomly selects a fresh nonce for each execution request, replay attacks can be detected and prevented by any executor enclave.

2) Liveness: The liveness property states that every contract execution initiated by an honest user U will eventually be processed with high probability. In case of a successful execution, a valid execution response is given by the executor. Unsuccessful execution can only happen in case of a contract *crash*. In this event, the contract execution halts and neither honest nor malicious users can perform successful contract executions anymore. We emphasize that the pool size can be set such that crashes happen only with negligible probability. In particular, for ϵ-liveness, the probability of a crash is bounded by $1-\epsilon$.

Claim 2 (ϵ-Liveness)**:** *Let n be the total number of enclaves in the system, m be the number of malicious operators' enclaves and s be the contract pool size. The POSE protocol satisfies ϵ-liveness for $\epsilon = 1 - \Pi_{i=0}^{s-1}(\frac{m-i}{n-i}) > 1 - (\frac{m}{n})^s$.*

Whenever user U sends an execution request to the executor enclave $\mathcal{E}_E$, U either directly receives a response or U challenges $\mathcal{E}_E$ via the manager M. If $\mathcal{E}_E$ does not respond within some predefined timeout, it will be kicked out of the execution pool and one of the watchdog enclaves takes over the executor role. User U can now trigger the execution again by interacting with the new executor enclave. During the execution, the executor enclave $\mathcal{E}_E$ requires confirmations from all watchdog enclaves in order to produce a valid result. However, watchdog enclaves cannot stall the execution forever, since $\mathcal{E}_E$ is able to challenge them via the manager as well. All unresponsive watchdog enclaves will be kicked out of the execution pool and the confirmations from the remaining watchdogs are enough to create a result. We stress that all timeouts are defined in Appendix C with great care to ensure that honest operators have enough time to respond. For example, the timeout for the executor challenge is sufficient to allow the executor enclave to challenge the watchdog enclaves twice; once for a currently running off-chain execution and once for the challenged on-chain execution.

Although the protocol guarantees that honest operators' enclaves will never be kicked, there is a small probability that an execution pool consists only of malicious operators' enclaves. If all enclaves are kicked out of the execution pool, the contract execution crashes. Let n be the number of total registered enclaves and m denote the number of enclaves controlled by malicious operators. The execution pool size is given by s. The probability of a crash is equal to the probability that only malicious operators' enclaves are within an execution pool. This is bounded by $\epsilon = 1 - \Pi_{i=0}^{s-1}(\frac{m-i}{n-i}) > 1 - (\frac{m}{n})^s$. Hence, *POSE* achieves ϵ-liveness.

Assuming a total of $n = 100$ registered enclaves and $m = 70$ of them are controlled by malicious operators. Even in this setting with a large share of malicious operators, *POSE* achieves liveness with $\epsilon > 92\%$ for a pool size of just 7. If only half of the operators are malicious, i.e., $m = 50$, *POSE* achieves liveness with $\epsilon > 99\%$ for the same pool size of 7. For $m = 10$ malicious operators, a pool size of only 3 yields a liveness with $\epsilon > 99\%$. For the same scenario of 10% malicious operators and assuming 40 millions contracts running in *POSE*, the pool size of 11 results in a probability of more than 99% that there is no crash at all in the whole system. See Fig. 8 for an illustration of the probability of no crashes depending on the number of contracts for different pool sizes.

3) State Privacy: The *state privacy* property says that the adversary cannot obtain additional information about a contract state besides what she learns from the results of contract executions alone.

Claim 3 (State Privacy)**:** *The POSE protocol satisfies state privacy.*

The smart contract's state is maintained by the enclaves within the execution pool. According to our adversary model (see Section II), the TEE provides confidentiality guarantees. In particular, the execution of an enclave does not leak any data. Hence, the smart contract's state is hidden from the adversary, even if the enclave's operator is corrupted. The only point in time during the *POSE* protocol when information about the contract's state is revealed is at the end of the execution protocol. However, the data provided as a result contains only public state and hence does not reveal anything about the private state. During the execution protocol, the executor enclave propagates the new state to all watchdog enclaves. However, the transferred data is encrypted using an IND-CPA secure encryption scheme. The security of the scheme guarantees that an adversary seeing the message cannot extract information from it.

While an enclave only publishes outputs after successful executions, we need to show that each produced output is final. In particular, a succeeding executor must not be able to revert to a state in which a published output should not have been produced. To this end, the state of the executor enclave producing a particular output needs to be replicated among all other enclaves before revealing the actual output. This property is achieved by the state propagation mechanism in our protocol. An enclave only returns an output if all enclaves in the pool confirm the corresponding state update. The EU-CMA secure signature scheme guarantees unforgeability of the confirmations. Hence, each confirmation guarantees that the corresponding enclave has updated its state correctly. Further, the correctness property of our protocol (cf. Section VII-A1) ensures that an enclave is always executed with a correct blockchain copy, and hence, is always aware of the correct pool composition. This means that an output can only be returned if the whole pool has received the corresponding state update.

B. Architectural Security

We further examine the architectural security of enclaves. The case of a user or TEE operator going offline by turning off their machine is covered in the protocol security (cf. Section VII-A); here we focus on parties that follow the protocol, trying to gain an unfair advantage in various ways.

The adversary might try to perform a memory corruption attack on the client used by users to interact with the executor (e.g., to send inputs). To mitigate this risk, the software should be implemented in a memory-safe language, like Python or Rust, and be open source so that it can be easily inspected.

A malicious TEE operator can also try mounting a memory-corruption or a side-channel attack on its TEE. As mentioned in A1.1, we assume that the TEE protects the confidentiality of the enclave and prevents leakage. However, in practice, cache-based side-channel attacks have been successfully demonstrated also on ARM processors [43]. While we want to stress that our ARM TrustZone-based implementation is a research prototype and the design is TEE-agnostic, the risk of these attacks can be mitigated by making the TEE opt-out of shared caches and flush private caches upon context switch, as proposed in [19]. Alternatively, a more advanced TEE design can be used [24], [19], [16]. Moreover, if the enclave code has an exploitable memory-corruption vulnerability, it is possible to mount a memory-corruption attack against it. One way to mitigate this risk, and hence, realize our assumption A1.2, is to use a memory-safe language for our smart contracts (in our case, Lua), or to deploy a run-time mitigation (like CFI [11]). Yet, in practice, an adversary might still be able to compromise an enclave. In this case, only the contracts of this enclave are affected. The consequences depend on the role of the enclave: for an executor enclave, the adversary gets full control over the contract; for a watchdog enclave, the adversary can only break state privacy.

Finally, an adversary might build a malicious smart contract with the goal of compromising secrets owned by other contracts or blocking an enclave by entering into an infinite loop. We mitigate against the first scenario by ensuring that only one smart contract is executing at any given time in an enclave, so that no foreign plain text secrets are present in memory at any point during contract execution. In case of multiple enclaves running on the same system, the TEE is isolating enclaves from each other such that no contract can tamper with another (cf. assumption A1.1). To handle infinite loops, we leverage a Lua sandbox [14], which interrupts the execution of the Lua code after a predetermined number of instructions has been issued and disables access to unsafe functions and modules.

VIII. Implementation

In order to evaluate *POSE*, we implemented a prototype for the manager and the enclaves, which uses TrustZone for the enclaves themselves and Lua as the smart contract programming language. We open source our prototype implementation to foster future research in this area[5]. We describe each of them in the following.

Manager. For the manager we use an Ethereum smart contract written in Solidity, which we will refer to as *manager* in the following. Even if this implementation is based on Ethereum, we note that our design can be realized on any blockchain supporting rich smart contracts. The manager keeps a list of all registered enclaves in the network as well as a list of all deployed contracts, including their public information, e.g., the address of the current executor. As mentioned in the protocol described in Section V-E, the manager provides functions to register an enclave, create a new *POSE* contract, deposit or withdraw money, and functions to challenge the current executor or any of the watchdogs. To synchronize all participants, every time a challenge related function was called it will throw an appropriate Solidity event.

Enclaves. The contract creator, executor, and watchdogs are enclaves running in a TEE. As our protocol is TEE-agnostic and all commercial TEEs exceed smart contracts' on-chain requirements on memory/computational-power capabilities significantly, we chose to use ARM TrustZone [15] for our prototype. TrustZone features a traditional programming model (OS, and user-space applications with standard library), and the Open Portable Trusted Execution Environment (OP-TEE) OS [41] already supports a large fraction of standard functionality, and hence, does not force us to reimplement this for the contract execution environment. TrustZone supports two execution modes: secure world and normal world. The system's memory can be freely distributed among these worlds. The secure world is an trusted OS which is completely independent from the normal OS, which in our case is Linux. Code running in the secure world is called a *Trusted App* (TA). A TA may only communicate with the normal world via shared memory regions, which are explicitly allocated as such. We implement the *POSE* enclaves as TAs. Computations in the secure world have native performance; yet, switching between worlds has a constant but negligible overhead (in our tests around 449µs). TrustZone does not impose memory limits for secure world. While we leverage the traditional TrustZone concept, recent versions add support for a S-EL2 hypervisor to allow multiple strongly isolated enclaves that allows *POSE* to scale better on these platforms. Most basic cryptographic functions are provided by the OP-TEE TA library, such as AES and TLS. Note that TrustZone itself does not standardize a remote attestation implementation itself, but industry [3], [6], [8] and OP-TEE implementations exist[6]. Remote attestation can also be used to prove a certain set of software defenses is active in the enclave. In our prototype, we leveraged OP-TEE's remote attestation functionality to attest the enclave after setting up the runtime. To leverage this feature, the *POSE* enclave requests a signed attestation report from the attestation PTA (Pseudo Trusted App), essentially a kernel module of the OP-TEE OS in secure world. The keys for signing the attestation report are derived using hardware device information and stored persistently after generation (using Secure Storage, or ”Trusted Storage”, as defined by GlobalPlatform's TEE Internal Core API specification).

To properly interact with the Ethereum-based manager, we also adapted and deployed an Ethereum wallet for embedded devices [13], enabling the enclaves to create ECDSA signatures, Keccak hashes, handle encoding, and create transactions to call the manager. For *POSE* contracts, we use the scripting language Lua [52]. It is a well-established, fast, powerful, yet simple language written in C. Lua as well as the enclave itself allow arbitrary computation. We ported the Lua interpreter to run inside the TA, by stripping out operations unsupported by the TA, such as file access. After each execution step, the enclave returns to the normal world while keeping the contract's Lua session alive. When the normal world receives an input from a user, it invokes the TA with these inputs to continue the Lua execution. To update the enclave runtime, different approaches are possible in practice, e.g., the manager could announce an update and all outdated enclaves would shut themselves down after a timeout. Honest operators then would incrementally trigger an enclave replacement during the timeout period.

IX. Evaluation

This section examines *POSE* regarding complexity and performance. In the following, we will report absolute performance numbers and discuss these in relation to Ethereum itself, but also compare to existing works based on TEEs, namely FastKitten and Bitcontracts. FastKitten has a highly similar set of tested smart contracts, so a comparison can put our numbers in perspective. For Bitcontracts, we reimplemented Quicksort with the same experiment setup. Note, that the smart contracts can still be implemented differently, and the performance and the TEE differ.

Complexity. Running a *POSE* contract in the benign case, i.e., if all involved enclaves respond, requires exactly two blockchain interactions for the setup. Each user of a contract also needs one blockchain interaction each time the user deposits or withdraws money regarding the contract. However, as *POSE* does not require a fixed collateral for the setup, the money transactions do not inherently prevent the contract from execution—except the specific contract demands it. Otherwise, when either the executor or any watchdog fails to respond, each challenge requires two blockchain interactions. The delay incurred by our challenge protocol is dominated by the on-chain transactions. This holds also for other off-chain solutions, e.g., state-channels [45], [26], [22], Plasma [51], [36], Rollups [47], [5] and FastKitten [25]. For instance, the time it takes for an honest executor to kick a watchdog is $325s$

[5] https://github.com/AppliedCryptoGroup/PoseCode

[6] https://github.com/OP-TEE/optee_os/pull/5025

TABLE I. Cost of executing the methods of the *POSE* manager. The USD costs were estimated based on the prices (Gas to GWei and ETH to USD) on May. 8, 2022 [27], [21]. *For comparison, these are the costs of popular operations on Ethereum.

Method	**Cost**	
	Gas	**USD**
`registerEnclave`	175 910	13.23
`initCreation`	198 436	14.91
`finalizeCreation`	79 545	5.98
`deposit`	37 255	2.80
`withdraw`	36 997	2.78
`challengeExecutor`	54 654	4.11
`executorResponse`	51 478	3.87
`executorTimeout`	53 327	4.01
`challangeWatchdogsCreation`	231 286	17.38
`challengeWatchdog`	131 362	9.87
`watchdogResponse`	36 257	2.72
`watchdogTimeout`	52 142	3.92
simple Ether transfer*	21 000	1.58
create CryptoKitty*	250 000	18.78

on average. We discuss timeout parameters and the challenge delay more thoroughly in Appendix C. In the worst-case, a malicious operator does not respond to the off-chain messages but to the challenges in every execution step, which would effectively reduce *POSE*'s execution speed beneath that of the blockchain. However, such an attack requires continuous blockchain interactions from the malicious party and hence entails costs for every execution step (cf. Section IX "Manager").

Test Setup. We deployed a test setup with our prototype implementation for performance measurements. The test setup consists of five devices. For the enclaves we deployed three Raspberry Pi 3B+ with four cores running at 1.4GHz. These are widely available and cheap devices that support ARM TrustZone. As state updates are small (just the delta to the previous state) and watchdogs receive and process the state updates in parallel, we do not expect an increase of the pool size to significantly influence the evaluation. Further, we used `ganache-cli` (6.10.2) to emulate a Ethereum blockchain in our local network, which runs the Solidity contract that implements the manager. Finally, a fifth device emulates multiple users by simply sending out network requests to both the manager and enclave operators, which are all connected via Ethernet LAN.

Manager. As the *POSE* manager is implemented as an Ethereum smart contract, interactions with it incur some costs in the form of Gas. The costs of all implemented methods of the Solidity contract are listed in Table I. The first five methods are used for benign *POSE* contract execution. The second part of the table shows methods that are required for challenges, including the response and timeout methods to resolve them. In terms of storage, each additionally registered enclave will require 64 bytes and each contract 288 bytes + (pool size × 32 bytes) of on-chain storage.

Contract Execution. To measure and demonstrate the efficiency of *POSE* contract execution, we implemented three applications as Lua code in our test setup. All time measurements are averaged over 100 runs. Regardless of the used contract, setting up an executor or watchdog enclave with a Lua contract takes 189ms. Creating an attestation report for the enclave takes another 367ms with OP-TEE's built-in remote attestation using a one-line dummy contract. For our biggest contract, Poker, the attestation takes 377ms, resulting in a total setup time of 566ms. In contrast, FastKitten needs 2s for enclave setup. Note that FastKitten needs an additional blockchain interaction. Multiple contracts run by a single operator are executed in parallel, including network communication. Thus, the number of enclaves, contracts and transactions a single operator can process depends on the operator's hardware. As modern servers CPUs feature 128 cores [23], and servers often feature multiple CPUs, we do not expect parallel execution to affect performance significantly. However, to prevent overload, the number of pools an operator participates in can be limited.

Rock paper scissors. This is an implementation of the popular game with two players. Unlike traditional smart contracts, we can leverage *POSE*'s private state to simply store each player's input, instead of having to use much more complex multi-round commitments. The resulting smart contract is 27 lines of code (LoC). Disregarding the delay caused by human players, the execution time of one round with two user inputs is 32ms. In comparison, FastKitten only needs 12ms, but is also running on a much more powerful machine. In contrast, executing this game on Ethereum would take around 5 minutes for each round (20 confirmation blocks, 15s block time each).

Poker. We have also implemented Poker as a multi-party contract running over multiple rounds. Note that in *POSE*, the poker game can be implemented as an ongoing cash game table, i.e., players may join or leave the table at any time, as contracts in *POSE* do not have to be finite. Each round consists of three phases each requiring an input from all users. The resulting smart contract is 209 lines of code (LoC). We execute the contract with five players who have their deposit ready at the start, with a total execution time of 199ms (vs. 45ms in FastKitten, but again, on a more powerful machine). Playing this game on Ethereum would take 5 minutes per player input.

Federated Machine Learning. For this application, users can submit locally trained models, which will be aggregated to a single model by the contract. Any user can then request the new model from the contract. For our measurements, each user trained a convolutional neural network consisting of 431 080 individual weights on the MNIST handwritten digits dataset [61]. For aggregation, the contract averages every existing weight with the corresponding weight sent by the user. The smart contract itself is only 5 LoCs, as we load the existing weights separately. Each aggregation took 238ms, which demonstrates the efficiency of *POSE*. Trying to execute the same function on Ethereum, for each aggregation, storage of the weights alone would exceed 1 billion gas (assuming 4 bytes float per weight) and the calculation over 3.4 million gas (8 gas per weight).

Quicksort. We have also implemented Quicksort to sort a hardcoded input array of 2048 random integers, as done in Bitcontracts [58]. The resulting smart contract is 29 lines of code (LoC). The total execution time of the contract is 20ms. Compared to the 6ms in Bitcontracts, we use a less powerful machine (Bitcontracts uses an AWS T2.micro instance with a recent Intel processor at 3.3Ghz), while our performance measurement also includes additional steps like context switches

TABLE II. OVERVIEW OF RELATED WORK, n DENOTES THE NUMBER OF TRANSACTIONS.

	No collateral	Private state	Blockchain interactions (optimistically)	Non-fixed lifetime & group
Ethereum [57]	✓	✗	$O(n)$	✓
MPC [39], [40], [38]	✗	✓	$O(1)$	✗
State Channels [45], [26], [22]	✗	✗	$O(1)$	✗
VM-based [35], [59], [58]	✗	✗	$O(n)$	✓
Ekiden [20]	✗	✓	$O(n)$	✗
FastKitten [25]	✗	✓	$O(1)$	✗
POSE	✓	✓	$O(1)$	✓

and the setup of the enclave runtime. Executing this Quicksort contract on Ethereum would cost around 6.5 million gas.

Watchdog State Updates. When an executor operator has been dropped, a watchdog takes over execution. For this to work, state changes are distributed to the watchdogs. Storing the current state and restoring it on a watchdog takes 17ms for the poker contract (averaged over 100 runs, corrected for network latency), which also has the biggest state among the ones we implemented.

Enclave Teardown. After an executor enclave is not expecting further inputs and finished the smart contract execution, the execution environment has to be cleaned up for the next smart contract, i.e., cryptographic secrets and the smart contract in the shared memory need to be zeroed. This takes 25ms.

X. RELATED WORK

Ethereum [57] is the most prominent decentralized cryptocurrency with support for smart contract execution. However, it is suffering from very high transaction costs and data used by smart contracts is inherently public.

Hawk [37] aims for improving the privacy by automatically creating a cryptographic protocol from a high-level program in order to allow computation on private data without disclosing it. However, this complex cryptographic layer further decreases performance of the system and increases costs. Similarly, approaches based on Multiparty Computation (MPC) [39], [40], [38] distribute the computation between multiple parties such that no party can access the cleartext data. These approaches have substantial overhead in performance, communication and collateral required.

One approach to alleviate the complexity limitation are state channels [45], [26], [22], which enable parties to lock some funds on the blockchain, execute complex contracts off-chain, and finally commit the results of the contract to the blockchain. This is efficient if all parties agree on the results; otherwise, the dispute can be solved on-chain, which takes longer and is more expensive.

Arbitrum [35] represents a smart contract as a virtual machine (VM), which is executed privately by a number of "managers". After execution, if all managers agree on the result of the computation, this result can be simply signed and committed to the blockchain, without the need to perform the computation on chain. In case managers disagree, a bisection algorithm is used to compare subsets of the execution on chain and find which is the first instruction on which the managers disagree, then punish the malicious manager(s). Hence, as long as at least one manager is honest, the correct result is computed. While computationally efficient, this on-chain protocol is still relatively expensive, so Arbitrum also includes financial incentives to encourage the managers to behave. The managers have full access to the data used by the VM, so confidentiality is broken if even one manager is malicious. Note that unlike Arbitrum, *POSE* does not require multiple parties to execute the smart contract: the watchdog enclaves just need to acknowledge the new states, unless the executor enclave fails.

ACE [59] and Bitcontracts [58] are similar to Arbitrum, but they allow the results of contract executions to be approved by a configurable quorum of service providers, not necessarily all of them. Unlike *POSE*, ACE does not support private state and requires on-chain communication per contract invocation. Although the transaction is computed off-chain, the invocation and the result are registered on-chain. Further, Arbitrum and ACE require changes to the blockchain infrastructure, hence, they are harder to deploy in practice.

Ekiden [20] is also an off-chain execution system that leverages TEE-enabled *compute nodes* to perform computation and regular *consensus nodes* that interact with a blockchain. The major drawback of Ekiden is that it requires every computation step to retrieve its initial status from the blockchain, and it only supports input from one client at a time. Moreover, the atomic delivery of the output of each step requires to wait for publication of the updated state before the output is made available to the client. Hence, any highly interactive protocol with multiple participants (like a card game, for instance) would incur significant delays between turns just to wait for the blockchain. The paper evaluates Ekiden on a fast blockchain, Tendermint, but it does not quantify its latency for interactive protocols on mainstream blockchains like Ethereum or Bitcoin. The Oasis Network uses an updated version of Ekiden [30]; yet, this version still requires to store state on the blockchain after each call.

FastKitten [25] also leverages TEEs to perform off-chain computation. It assumes a rational attacker model, with financial incentives to convince all participants to follow the protocols. If they all do, the communication happens directly between the TEE and them, thus dispensing with the high latency due to blockchain roundtrips. However, FastKitten only supports contracts with a predefined list of participants and a limited lifespan. It also requires the TEE operator to deposit as much as every participant combined as collateral. *POSE* lifts those restrictions: it enables long-lived smart contracts with an unknown set of participants and requires no collateral from the TEE owners. Further, *POSE* achieves strong liveness guarantees in the presence of byzantine adversaries, while FastKitten assumes a rational adversary.

ROTE [44] is a system that detects rollback attacks on TEEs by storing a counter on a number of other TEEs. This approach is similar to the watchdog enclaves used in *POSE* to ensure that execution of a smart contract continues. However, unlike *POSE*, ROTE can only detect rollback attacks, but cannot prevent malicious operators from withholding the state. SlimChain [60] primarily aims at reducing on-chain storage, while still requiring blockchain interactions to store state commitments. Further, the paper does not address storage nodes crashing, which would lead to a liveness violation. Pointproofs [32] proposes a new vector commitment scheme to reduce the storage requirements on blockchain validators. Although validators do not need to store all values of a smart contract, once a transaction provides these values, the execution is still performed on-chain. However, *POSE* works entirely off-chain in the optimistic case and ensures liveness with its protocol.

Chainspace [12] proposes an entirely new distributed ledger platform focusing on sharding combined with a directed acyclic graph structure, while POSE extends established blockchains (e.g., Ethereum). ResilientDB [53] proposes a consensus protocol that clusters validators' geo-location to minimize network overheads. In contrast, *POSE* is a off-chain execution protocol for smart contracts. Hyperledger Fabric Private Chaincode [29] requires trust in handling the encryption key by the client or an *admin*; thus, we deem it not applicable to permissionless blockchains, targeted by *POSE*. Hyperledger *Private Data Objects* [18], an alternative to Private Chaincode, requires periodic blockchain interactions to store the state on-chain. This slows execution on contract calls to the speed of the blockchain, unlike *POSE*, which executes contracts entirely off-chain in the optimistic case. Hyperledger *Avalon* [28] can outsource workloads to TEE enclaves. However, these workloads have to be self-contained, and thus, interactions by participants still require on-chain transactions, while *POSE* can run interactive contracts completely off-chain (e.g., Poker).

XI. Conclusion

Smart contracts have become an indispensable tool in the era of blockchains; yet, current approaches suffer from various shortcomings. In this paper, we introduce *POSE*, a novel off-chain execution protocol that addresses all of these shortcomings to enable much more versatile smart contracts. We showed *POSE*'s security and demonstrated its feasibility with a prototype implementation.

Acknowledgements

This work was supported by the European Space Operations Centre with the Networking/Partnering Initiative, the German Federal Ministry of Education and Research within the StartUpSecure funding program *Sanctuary* (16KIS1417) and within the *iBlockchain project* (16KIS0902), by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) *SFB 1119 – 236615297 (CROSSING Project S7)*, by the European Union's Horizon 2020 Research and Innovation program under Grant Agreement No. 952697 (ASSURED), by the BMBF and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*.

References

[1] Cardano. https://cardano.org/. (Accessed on 05/20/2021).

[2] Cryptokitties - collect and bread furrever friends! https://www.cryptokitties.co/. Accessed 14-08-2022.

[3] Enhanced attestation (v3). https://docs.samsungknox.com/dev/knox-attestation/about-attestation.htm. Accessed 20-04-2022.

[4] Etherscan - ethereum average block time chart. https://etherscan.io/chart/blocktime. Accessed 20-09-2021.

[5] Optimistic rollups - ethhub. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/optimistic_rollups/. (Accessed on 05/20/2021).

[6] Qualcomm® trusted execution environment (tee) v5.8 on qualcomm® snapdragon™ 865 security target lite. https://www.tuv-nederland.nl/assets/files/cerfiticaten/2021/08/nscib-cc-0244671-stlite.pdf. Accessed 20-04-2022.

[7] Solidity documentation. https://docs.soliditylang.org/en/v0.8.7/. Accessed 20-09-2021.

[8] Upgrading android attestation: Remote provisioning. https://android-developers.googleblog.com/2022/03/upgrading-android-attestation-remote.html. Accessed 20-04-2022.

[9] Proxy bid. https://en.wikipedia.org/w/index.php?title=Proxy_bid&oldid=968758683, July 2020.

[10] Google cloud bigquery: Block variance. https://console.cloud.google.com/bigquery, 2021. Query: SELECT b.timestamp FROM 'bigquery-public-data.ethereum_blockchain.live_blocks' AS b ORDER BY b.timestamp; Accessed 20-09-2021.

[11] Martın Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. CFI: Principles, implementations, and applications. In *Proc. ACM Conference and Computer and Communications Security (CCS)*, 2005.

[12] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *25th Annual Network and Distributed System Security Symposium, (NDSS 2018)*, 2018.

[13] AnyLedger. Embedded Ethereum wallet library GitHub. https://github.com/Anylsite/embedded-ethereum-wallet, 2020.

[14] APItools. sandbox.lua. https://github.com/APItools/sandbox.lua, 2017.

[15] ARM Limited. ARM Security Technology: Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2008.

[16] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[17] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1521–1538, 2019.

[18] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. Private data objects: an overview. *CoRR*, abs/1807.05686, 2018.

[19] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. SANCTUARY: ARMing TrustZone with user-space enclaves. In *NDSS*, 2019.

[20] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE, 2019.

[21] CoinMarketCap. Ethereum (ETH) price. https://coinmarketcap.com/currencies/ethereum/, 2020.

[22] Jeff Coleman, Liam Horne, and Li Xuanji. Counterfactual: Generalized state channels, Jun 2018. https://l4.ventures/papers/statechannels.pdf.

[23] Ampere Computing. Ampere Altra Max 64-Bit Multi-Core Processor Features. https://amperecomputing.com/processors/ampere-altra/, 2022.

[24] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[25] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. Fastkitten: practical smart contracts on bitcoin. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[26] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC*

Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, 2018.

[27] Etherscan. Ethereum Average Gas Price Chart. https://etherscan.io/chart/gasprice, 2020.

[28] Hyperledger Foundation. Hyperledger avalon. https://wiki.hyperledger.org/display/avalon/Hyperledger+Avalon. Accessed 04-08-2022.

[29] Hyperledger Foundation. Hyperledger fabric private chaincode github. https://github.com/hyperledger/fabric-private-chaincode. Accessed 04-08-2022.

[30] Oasis Foundation. An implementation of ekiden on the oasis network. https://oasisprotocol.org/papers. Accessed 04-08-2022.

[31] GlobalPlatform. TEE Internal Core API Specification. https://globalplatform.org/specs-library/tee-internal-core-api-specification-v1-2/, 2019.

[32] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 2007–2023, 2020.

[33] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[34] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: Epid provisioning and attestation services. *White Paper*, 1(1-10):119, 2016.

[35] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 2018)*. USENIX Association, 2018.

[36] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. Commit-chains: Secure, scalable off-chain payments. *Cryptology ePrint Archive, Report 2018/642*, 2018.

[37] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016.

[38] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[39] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[40] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to secure computation with penalties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[41] Linaro, Inc. OP-TEE Documentation. https://readthedocs.org/projects/optee/downloads/pdf/latest/, 2020.

[42] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Peter Pietzuch, and Emin Gün Sirer. Teechain: Reducing storage costs on the blockchain with offline payment channels. In *Proceedings of the 11th ACM International Systems and Storage Conference*, pages 125–125, 2018.

[43] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[44] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[45] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR*, abs/1702.05812, 2017.

[46] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2008.

[47] Offchain Labs, Inc. Arbitrum rollup: Off-chain contracts with on-chain security. 2020.

[48] Gustavo A Oliva, Ahmed E Hassan, and Zhen Ming Jack Jiang. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empirical Software Engineering*, 2020.

[49] Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2017.

[50] Travis Patron. What's the big idea behind Ethereum's world computer. https://www.coindesk.com/whats-big-idea-behind-ethereums-world-computer/, 2016.

[51] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. 2017.

[52] PUC-Rio. The programming language Lua. https://www.lua.org/, 2020.

[53] Sajjad Rahnama, Suyash Gupta, Thamir M Qadah, Jelle Hellings, and Mohammad Sadoghi. Scalable, resilient, and configurable permissioned blockchain fabric. *Proceedings of the VLDB Endowment*, 13(12), 2020.

[54] Andrey Sergeenkov. How to check your ethereum transaction. https://www.coindesk.com/learn/how-to-check-your-ethereum-transaction/. Accessed 24-08-2022.

[55] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.

[56] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *CoRR*, abs/1908.04756, 2019.

[57] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[58] Karl Wüst, Loris Diana, Kari Kostiainen, Ghassan Karame, Sinisa Matetic, and Srdjan Capkun. Bitcontracts: Adding expressive smart contracts to legacy cryptocurrencies. 2019.

[59] Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. ACE: asynchronous and concurrent execution of complex smart contracts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[60] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. Slimchain: scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment*, 14(11):2314–2326, 2021.

[61] Yann LeCun and Corinna Cortes and Christopher J.C. Burges. THE MNIST DATABASE. http://yann.lecun.com/exdb/mnist/, 2020.

[62] 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, 2016.

[63] Fan Zhang, Philip Daian, Iddo Bentov, and Ari Juels. Paralysis proofs: Safe access-structure updates for cryptocurrencies and more. *IACR Cryptol. ePrint Arch.*, 2018:96, 2018.

Appendix

A. Supported Contracts

The *POSE* system supports contracts with a dynamic set of users of arbitrary size and an unrestricted lifetime. The timeouts need to be set reasonable with respect to the expected execution time of the contracts to allow the execution of complex contracts and to prevent denial of service attacks at the same time.

Interaction between *POSE* contracts can be realized by letting the TEE of the calling contract instruct its operator to request an execution of the second contract via the respective executive operator and wait for the response. We deem the exact specification, e.g., enforce an upper bound on (potentially recursive) external calls to guarantee timely request termination, an engineering effort. Calls from POSE contracts to on-chain contracts can be supported similarly to our payout concept (Appendix D).

B. Further Protocol Blocks

To keep the specification of the *POSE* protocol in the main body simple and compact, we have excluded the formal specification of the creation process and the validation algorithms. In this section, we present the full specification of these protocol parts.

1) Creation protocol and creation challenges: In the following, we will describe the creation protocol and the corresponding challenges in detail. The Figures 5, 6, and 7 specify the creation protocol, the challenge of the creator and the challenge of the pool members during creation, respectively. Program 3 specifies the parts of the *POSE* program that are relevant for the creation.

The creation is initiated by a user U that wants to install a contract with program code $code$. The initial state of the contract is hard-coded in $code$. U obtains the set of registered enclaves from the manager M. U is free to choose one of these enclaves as creator $\mathcal{E}_C$. Next, a creation initialization message is sent from the user to the manager M. This message contains a hash of $code$ as well as the selected creator enclave. M picks an unused contract id id and initializes the on-chain information about the contract including the information received from U and the latest block number p. The manager returns id and p to the user.

Next, user U sends a creation request containing the contract id id and the program code $code$ to the creator enclave $\mathcal{E}_C$. Upon receiving a creation request, $\mathcal{E}_C$ randomly selects n enclaves which will form the smart contract execution pool. One of pool enclaves is assigned as the executor enclave for contract id while all others act as watchdog enclaves. In addition, $\mathcal{E}_C$ samples a symmetric pool encryption key. The generated information, i.e., the execution pool, the assigned executor enclave, and the encryption key, as well as the creation request is distributed to all pool enclaves.

Upon receiving the message from $\mathcal{E}_C$, each pool enclave executes $initContract(code)$ which creates a new instance of the contract defined by $code$. The method locally allocates memory to set up the initial state of the contract. Afterwards, a confirmation message is sent back to the creator enclave.

If $\mathcal{E}_C$ receives confirmations from all pool enclaves in a predefined time period, $\mathcal{E}_C$ finally generates a successful creation statement. Otherwise, if any pool enclave has not responded timely, the creator enclave starts a challenge-response protocol. The challenge procedures during the creation process are similar to the ones of the execution protocol explained in Section V-E4. The differences to the challenge procedures are captured in the protocol specifications in Figure 6 and 4.

Eventually, the creation statement is sent to the manager M which updates the on-chain information about the contract. In particular, M marks the creation as final and stores the pool information. If the creation process is not finalized by the creator enclave in time, user U starts the challenge-response protocol (see Section V-E4). If the creator does not respond to the creation challenge, the creation has failed and U has to try again.

2) POSE Program (Creation): All enclaves participating in our *POSE* protocol need to run a specific program. We divided

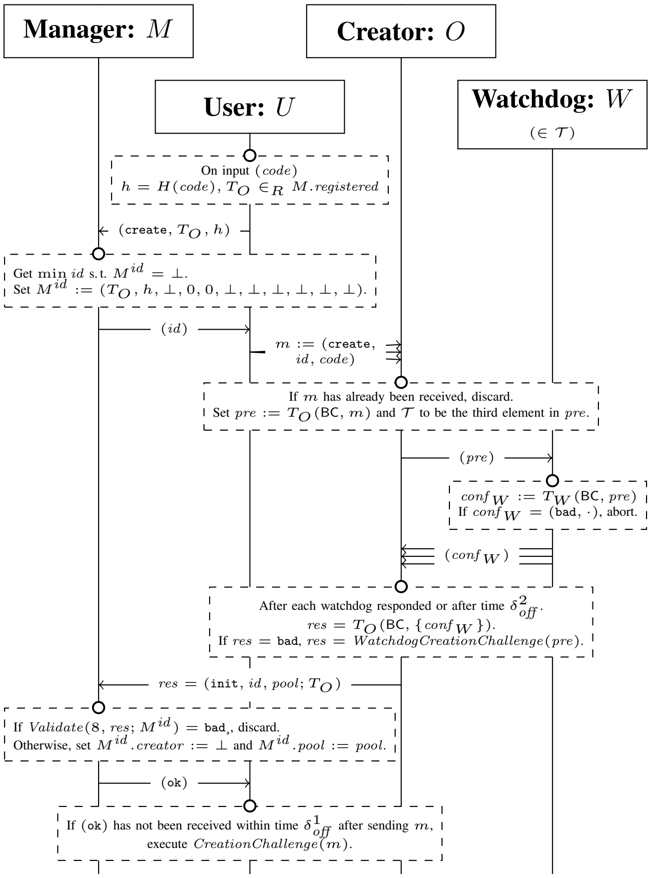
Manager: M
Creator: O
User: U
Watchdog: W
$(\in \mathcal{T})$
On input $(code)$
$h = H(code)$, $T_O \in_R M.registered$
$(\texttt{create}, T_O, h)$
Get $\min id$ s.t. $M^{id} = \bot$.
Set $M^{id} := (T_O, h, \bot, 0, 0, \bot, \bot, \bot, \bot, \bot, \bot)$.
(id)
$m := (\texttt{create}, id, code)$
If m has already been received, discard.
Set $pre := T_O(\mathsf{BC}, m)$ and $\mathcal{T}$ to be the third element in pre.
(pre)
$conf_W := T_W(\mathsf{BC}, pre)$
If $conf_W = (\texttt{bad}, \cdot)$, abort.
$(conf_W)$
After each watchdog responded or after time δ^2_{off}.
$res = T_O(\mathsf{BC}, \{conf_W\})$.
If $res = \texttt{bad}$, $res = WatchdogCreationChallenge(pre)$.
$res = (\texttt{init}, id, pool; T_O)$
If $Validate(8, res; M^{id}) = \texttt{bad}$,, discard.
Otherwise, set $M^{id}.creator := \bot$ and $M^{id}.pool := pool$.
$(\texttt{ok})$
If $(\texttt{ok})$ has not been received within time δ^1_{off} after sending m,
execute $CreationChallenge(m)$.

Fig. 5. Detailed creation protocol.

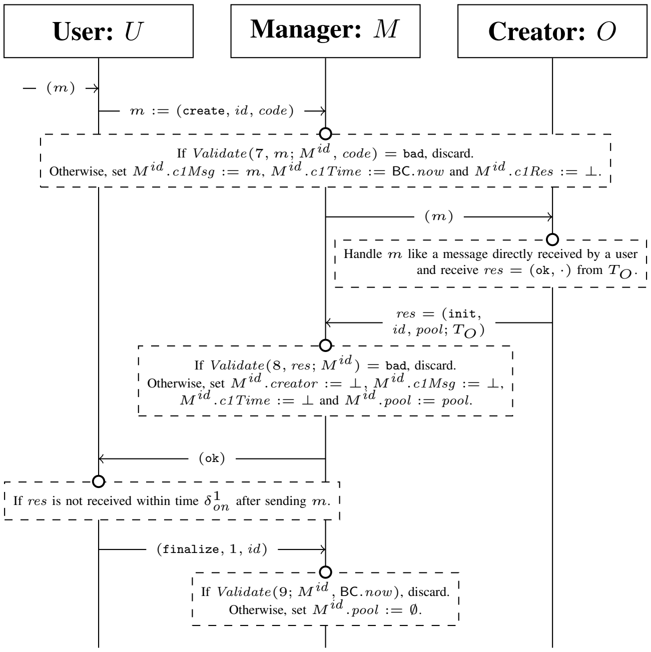
User: U
Manager: M
Creator: O
(m)
$m := (\texttt{create}, id, code)$
If $Validate(7, m; M^{id}, code) = \texttt{bad}$, discard.
Otherwise, set $M^{id}.c1Msg := m$, $M^{id}.c1Time := \mathsf{BC}.now$ and $M^{id}.c1Res := \bot$.
(m)
Handle m like a message directly received by a user
and receive $res = (\texttt{ok}, \cdot)$ from T_O.
$res = (\texttt{init}, id, pool; T_O)$
If $Validate(8, res; M^{id}) = \texttt{bad}$, discard.
Otherwise, set $M^{id}.creator := \bot$, $M^{id}.c1Msg := \bot$,
$M^{id}.c1Time := \bot$ and $M^{id}.pool := pool$.
$(\texttt{ok})$
If res is not received within time δ^1_{on} after sending m.
$(\texttt{finalize}, 1, id)$
If $Validate(9; M^{id}, \mathsf{BC}.now)$, discard.
Otherwise, set $M^{id}.pool := \emptyset$.

Fig. 6. Detailed creator challenge protocol.

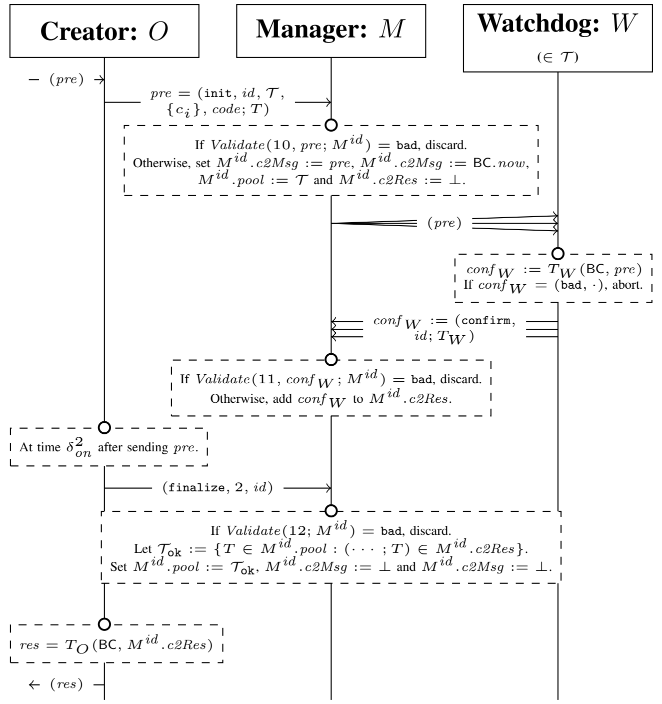
Creator: O
Manager: M
Watchdog: W
$(\in \mathcal{T})$
$-\ (pre) \rightarrow$
$pre = (\texttt{init}, id, \mathcal{T}, \{c_i\}, code; T)$
If $Validate(10, pre; M^{id}) = \texttt{bad}$, discard.
Otherwise, set $M^{id}.c2Msg := pre$, $M^{id}.c2Msg := \mathsf{BC}.now$,
$M^{id}.pool := \mathcal{T}$ and $M^{id}.c2Res := \perp$.
(pre)
$conf_W := T_W(\mathsf{BC}, pre)$
If $conf_W = (\texttt{bad}, \cdot)$, abort.
$conf_W := (\texttt{confirm}, id; T_W)$
If $Validate(11, conf_W; M^{id}) = \texttt{bad}$, discard.
Otherwise, add $conf_W$ to $M^{id}.c2Res$.
At time δ^2_{on} after sending pre.
$(\texttt{finalize}, 2, id)$
If $Validate(12; M^{id}) = \texttt{bad}$, discard.
Let $\mathcal{T}_{\texttt{ok}} := \{T \in M^{id}.pool : (\cdots; T) \in M^{id}.c2Res\}$.
Set $M^{id}.pool := \mathcal{T}_{\texttt{ok}}$, $M^{id}.c2Msg := \perp$ and $M^{id}.c2Msg := \perp$.
$res = T_O(\mathsf{BC}, M^{id}.c2Res)$
$\leftarrow (res)$

Fig. 7. Detailed pool challenge protocol.

Program 3: *POSE* program (creation) executed by enclave T

```
Upon receiving $(\texttt{create}, id, code)$, do:
  1) If $M^{id} = \perp$, $M^{id}.creator \neq T$ or
     $M^{id}.codeHash = H(code)$, return $(\texttt{bad})$.
  2) Generate an encryption key $key^{id}$ and sample a random
     subset $\mathcal{T}^{id}$ of size $n$ from $M.registered$.
  3) Let $T_i$ be the $i$-th member in $\mathcal{T}^{id}$ and $pk_i$ this member's
     public encryption key. Calculate $c_i := Enc(key^{id}; pk_i)$ for
     each $i$.
  4) Store $\mathcal{T}^{id}$ and $\mathcal{T}^{id}_{wait} := \mathcal{T}^{id}$ and return
     $(\texttt{init}, id, \mathcal{T}^{id}, \{c_i\}_i, code; T)$.
Upon receiving $(\texttt{init}, id, \mathcal{T}, \{c_j\}_j, code; T')$, do:
  1) If $T \notin \mathcal{T}$, $T' \neq M^{id}.creator$ or $C^{id} \neq \perp$, return $(\texttt{bad})$.
  2) Let $T$ be the $i$-th element in $\{c_j\}_j$. Calculate and store
     $key^{id} := Dec(c_i; sk_T)$, $C^{id} := initContract(code)$ and
     $\mathcal{T}^{id}_{wait} := \emptyset$.
  3) Return $(\texttt{confirm}, id; T)$.
Upon receiving message $\{m_i = (flag_i, id; T_i)\}_i$, do:
  1) If $\mathcal{T}^{id}_{wait} = \emptyset$, $M^{id}.creator \neq T$, return $(\texttt{bad})$.
  2) Set $\mathcal{T}^{id}_{wait} = \mathcal{T}^{id}_{wait} \cap M^{id}.pool$.
  3) For each $m_i$ do:
     • If $T_i \notin \mathcal{T}^{id}_{wait}$, skip $m_i$.
     • Otherwise remove $T_i$ from $\mathcal{T}^{id}_{wait}$.
  4) If $\mathcal{T}^{id}_{wait} \neq \emptyset$, return $(\texttt{bad})$. Otherwise, return
     $(\texttt{init}, id, \mathcal{T}^{id}; T)$.
  5) If $T \notin \mathcal{T}^{id}$, delete all stored variables with respect to $id$.
```

the program description into Program 1 and 3. The later one contains all methods executed during the creation protocol. In particular, the creator enclave invokes the *POSE* program with input `create`, id and contract code $code$, where id denotes the id assigned by the manager M. After an execution pool of size n and an encryption key are sampled, the *POSE* program returns message `init`. This message is forwarded to all execution pool members. Upon receiving message `init`, the *POSE* program initializes the new contract and returns a confirmation message. When invoked with the confirmation messages, the creator enclave checks if a confirmation of each pool member has been received. Only if this check is true, it returns a successful creation statement.

3) Validation: All of the different messages sent to the manager throughout the protocol need to be validated with several checks. In order to keep the description compact, we did not include the validation steps in the protocol figures but extracted them into a validation algorithm specified in Program 4. The algorithm is invoked with an counter specifying the checks that should be performed, an optional message that should be checked and the contract state tuple maintained by the manager. The validation returns `ok` if all requirements are satisfied and M can continue executing and `bad` if M should discard the received request.

C. Timeouts

Our protocol incorporates several timeouts δ^*_{off}, which define until when an honest user or operator expects a response to a request, and δ^*_{on}, which define until when the manager expects a response to a challenge. These timeouts have to be selected carefully such that each honest party has the chance to answer each message and challenge before the respective timeout expires. In this section, we elaborate on the requirements on the timeouts. We neglect message transmission delays and also assume that each challenge sent to the manager will directly be received by all operators (already before it is included into a final block)[7]. We recall the maximum blockchain delay which is defined as $\delta_{\mathsf{BC}} = \alpha \cdot \tau$ (cf. II and IV).

The off-chain propagation timeout δ^2_{off} describes the time an execution or creation operator maximally waits for a confirmation from the (other) pool members. It needs to be larger than the maximal update respectively installation time of a contract. Timeout $\delta^2_{on} \geq \delta^2_{off} + \delta_{\mathsf{BC}}$ describes the maximal time after which M expects a response to any watchdog challenge, either during creation or execution. The off-chain execution timeout δ^1_{off} describes the maximal time a user waits for a response to an execution request. Note that there might be a running execution and both running and new execution might require a watchdog challenge. In case watchdogs are dropped in the process of such a challenge, the executor needs to be able to notify its enclave about the new pool constellation, and hence, wait until the finalization of the challenge is within a final block. This takes additional time $\Delta = \tau \cdot \gamma$ (cf. IV). Hence, δ^1_{off} needs to be high enough to enable the challenged executor to perform two contract executions and run two watchdog challenges each taking up to time $\delta^2_{on} + \delta_{\mathsf{BC}} + \Delta$. We elaborate on maximal execution, update, and installation times of contracts in Section V-F. Finally, $\delta^1_{on} \geq \delta^1_{off} + \delta_{\mathsf{BC}}$ defines the maximal time after which M expects a response to an execution challenge. As the creation is comparable to the execution, we set the timeouts for off-chain creation and creation-challenge according to the ones of the execution.

[7] Instead, we could also add two times the maximum message delay to each off-chain timeout δ^*_{off} and the blockchain confirmation time $\Delta = \tau \cdot \gamma$ to each on-chain timeout δ^*_{on}.

Program 4: Algorithm *Validate*

The validation algorithm performs the following checks. If input $C = \bot$, the parsing of a message fails or any require is not satisfied, the algorithm outputs `bad`. Otherwise, it outputs `ok`.

- On input $(1, m; C)$, parse m to $(\texttt{execute}, id, \cdot, \cdot; U)$. Require that $C.\mathit{creator} = \bot$, $C.\mathit{c1Time} = \bot$ and $\mathit{Verify}(m) = \texttt{ok}$.
- On input $(2, \mathit{res}; C)$, parse res to $(\texttt{ok}, id, \cdot, h; T)$. Require that $C.\mathit{creator} = \bot$, $H(C.\mathit{c1Msg}) = h$, $C.\mathit{c1Time} + \delta^1_{on} > \mathsf{BC}.\mathit{now},\ \mathit{Verify}(\mathit{res}) = \texttt{ok}$ and $C.\mathit{pool}[0] = T$.
- On input $(3; C)$, require that $C \neq \bot$, $C.\mathit{creator} = \bot$, $C\mathit{c1Msg} \neq \bot$ and $C.\mathit{c1Time} + \delta^1_{on} \leq \mathsf{BC}.\mathit{now}$.
- On input $(4, \mathit{pre}; C)$, parse pre to $(\texttt{update}, id, c, h; T)$. Require that $C.\mathit{creator} = \bot$, $C.\mathit{c2Time} = \bot$, $C.\mathit{pool}[0] = T$ and $\mathit{Verify}(\mathit{pre}) = \texttt{ok}$.
- On input $(5, \mathit{conf}; C)$, parse conf to $(\texttt{confirm}, id, h; T_i)$ and $C.\mathit{c2Msg}$ to $(\cdot, \cdot, \cdot, h'; \cdot)$. Require that $C.\mathit{creator} = \bot$, $C.\mathit{c2Time} + \delta^2_{on} > \mathsf{BC}.\mathit{now},\ \mathit{Verify}(\mathit{conf}) = \texttt{ok}$, $h = h'$ and $T \in C.\mathit{pool}$.
- On input $(6; C)$, require that $C.\mathit{creator} = \bot$, $C.\mathit{c2Time} \neq \bot$ and $C.\mathit{c2Time} + \delta^2_{on} \leq \mathsf{BC}.\mathit{now}$.
- On input $(7, m; C, \mathit{code})$, parse m to $(\texttt{create}, id, \mathit{code})$. Require that $C.\mathit{creator} \neq \bot$, $C.\mathit{c1Time} = \bot$ and $C.\mathit{codeHash} = H(\mathit{code})$.
- On input $(8, \mathit{res}; C)$, parse res to $(\texttt{init}, id, \mathcal{T}; T)$. Require that $C.\mathit{creator} = T$, $C.\mathit{c1Time} + \delta^1_{on} > \mathsf{BC}.\mathit{now}$ and $\mathit{Verify}(\mathit{res}) = \texttt{ok}$.
- On input $(9; C)$, require that $C.\mathit{creator} \neq \bot$, $C\mathit{c1Time} \neq \bot$ and $C.\mathit{c1Time} + \delta^1_{on} \leq \mathsf{BC}.\mathit{now}$.
- On input $(10, \mathit{pre}; C)$, parse pre to $(\texttt{init}, id, \cdot, \cdot, \cdot; T)$. Require that $C.\mathit{creator} = T$, $C.\mathit{c2Time} = \bot$ and $\mathit{Verify}(\mathit{pre}) = \texttt{ok}$.
- On input $(11, \mathit{conf}; C)$, parse conf to $(\texttt{confirm}, id; T_i)$. Require that $C.\mathit{creator} \neq \bot$, $C.\mathit{c2Time} \neq \bot$, $C.\mathit{c2Time} + \delta^2_{on} > \mathsf{BC}.\mathit{now},\ \mathit{Verify}(\mathit{conf}) = \texttt{ok}$ and $T \in C.\mathit{pool}$.
- On input $(12; C)$, require that $C.\mathit{creator} \neq \bot$, $C.\mathit{c2Time} \neq \bot$ and $C.\mathit{c2Time} + \delta^2_{on} \leq \mathsf{BC}.\mathit{now}$.

The timeouts are an upper bound of the delay that can be enforced by malicious operators by withholding messages. To decrease the delays in a practical setting our implementation incorporates dynamic timeouts. Such a timeout is initially set to match an optimistic scenario where all operators directly answer. Only if the executor signals that a watchdog is not responding, the timeout is increased. For example, the δ^1_{on} timeout is initially set by the manager just high enough to allow the executor to perform the execution offline and to send one on-chain transaction. This on-chain transaction is either the response or a watchdog challenge. In case the executor creates a watchdog challenge this triggers the manager to increase the δ^1_{on} timeout for the executor. Similarly, the timeout δ^1_{on} is increased by the manager if any watchdog is not responding and the executor sends a transaction that kicks this watchdog. The increased timeout allows the executor to provide the kick transaction together with enough confirmation blocks to its enclave to finalize the execution. This dynamic timeout mechanism still allows the executor to respond in time even if a watchdog is not responding but at the same prevents the executor to stall the execution to the maximum although the watchdogs have already responded. While the executor still can create a watchdog challenge to increase the delay, this attack is costly since the executor needs to pay for the on-chain transaction. The value of the off-chain timeout δ^1_{off} is handled similarly. The client only needs to account for watchdog challenges in the previous execution in the timeout if there is indeed a running on-chain challenge. If there are no running challenges, a client can decrease δ^1_{off} to δ_{BC} plus two times the time it takes a TEE to execute and update a contract. Hence, if the executor is unresponsive, the client submits its executor challenge much earlier.

We give a concrete evaluation for the case of Ethereum, as this is the platform on which our implementation works. Let $\alpha = 20$ be the number of blocks until a transaction is included in the blockchain in the worst case, and $\alpha_{avg} = 10$ in the average case. Further, we consider the block creation time to be $\tau = 44s$ per block in the worst case and $\tau_{avg} = 15s$ in the average case[8]. Finally, we assume that blocks are final, when they are confirmed by $\gamma = 15$ successive blocks. Since the network delay and the computation time of enclaves are at most just a few seconds, which is insignificant compared to the time it requires to post on-chain transactions, we neglect these numbers for simplicity in the following example. In case the executor (resp. a watchdog) is not responding, it is challenged by the the client (resp. the executor). The creation of such a challenge takes $\alpha_{avg} \cdot \tau_{avg} = 150s$ on average. In what follows, due to the dynamic timeout mechanism, the on-chain timeout for both, executor challenge (δ^1_{on}) and watchdog challenge (δ^2_{on}), is initially set to $\alpha \cdot \tau = 880s$. For on-chain timeouts, we need to consider the worst-case parameters to allow honest operators to respond timely in every situation. While a dishonest operator can delay up to the defined timeout, an honest operator responds, and hence, finalizes the challenge in $150s$ on average. In case the challenged operator gets kicked, the (next) executor enclave needs to provide the kick transaction together with enough confirmation blocks to its enclave to finalize the execution. This takes $(\alpha_{avg} + \gamma) \cdot \tau_{avg} = 375s$ on average. For executor challenges, it can happen that the executor submits a watchdog challenge during the timeout period. In this case, which can happen at most twice, the timeout is increased by $880s$. If the challenged watchdog does not reply, and consequently is kicked from the pool, the timeout is increased by $(\alpha + \gamma) \cdot \tau = 1\,540s$. Note, this worst case is very costly to provoke, and in the general case, an honest executor can finalize the kick of the watchdog in $375s$.

D. Coin Flow

The *POSE* protocol supports the off-chain execution of smart contracts that deal with coins, e.g., games with monetary stakes. To this end, we provide means to send coins to and receive coins from a contract. In this section, we explain the mechanisms that enable the transfer of money and the intended coin flow of *POSE* contracts.

In order to deposit money to a *POSE* contract, identified by id, a user U sends a message $(\texttt{deposit}, id, \mathit{amount}; U)$ with amount coins to M. Upon receiving a deposit message, M checks whether a contract with identifier id exists and validates the signature, i.e., $M^{id} \neq \bot$ and $\mathit{Verify}(\texttt{deposit}, id, \mathit{amount}; U) = \texttt{ok}$. If the checks hold, M increases the contract balance $M^{id}.\mathit{balance}$ by amount.

[8] For setting α and α_{avg}, we consider a transaction to be included into the blockchain after at most 20 resp. 10 blocks according to [54]. To determine τ, we analyzed the Ethereum history via Google-BigQuery and identified that since 2018 every interval of 20 blocks took at most $44s$ per block. For τ_{avg}, we take the average parameter for Ethereum (cf. https://etherscan.io/chart/blocktime).

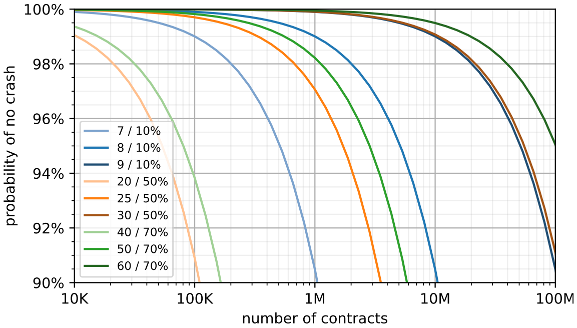

Fig. 8. Cumulative probabilities of no contracts crashing with a large number of POSE contracts in the system for different pool sizes s and adversary shares m, labeled "s / m".

As deposits are part of blockchain data that are provided by the operator to an enclave (cf. V-D) and the enclave forwards the data to the $nextState$ function of the contract C^{id}, U is ensure that C^{id} processes the deposit once the corresponding block is final. However, it is upon to the application logic to decide how deposits are processed.

A contract C can transfer coins to users by outputting *withdrawals* as part of the public state. It is upon the application logic to decide how and when coins are transferred to the users. For example, a game can issue withdrawals once the winner has been determined or leave the coins locked for another round unless a user explicitly requests a withdrawal via a contract execution. However, once a withdrawal has been issued, the coins are irreversible transferred.

Technically, contract C with identifier id maintains a list of all unspent withdrawals $\{amount_i, U_i\}$ and a counter $payouts$ for the number of spent payouts. Each public state returned by C contains a payout, a signed message $m :=$ $(\texttt{withdraw}, id, payouts, \{amount_i, U_i\}; \mathcal{E}_E)$ where $\mathcal{E}_E$ is the executor enclave of the contract. This message can be sent to M to spent all withdrawals within the payout. M checks the validity of the payout, i.e., $Verify(m) = \texttt{ok}$, $\mathcal{E}_E =$ $M^{id}.pool[0]$, and $payouts = M^{id}.payouts$. If the checks hold, M transfers coins to the users according to the withdrawal list $\{amount_i, U_i\}$. Finally, M sets $M^{id}.payouts := payouts + 1$ and $M^{id}.balance := M^{id}.balance - sum$, where sum is the sum of all withdrawals. Once C processes a final block with a payout transaction, it updates its list of unspent withdrawals $\{amount_i, U_i\}$ accordingly and increments $payouts$ by 1.

This mechanism ensures that a malicious user can neither double spent withdrawals nor prevent an honest user from withdrawing his coins as long as the contract remains live. Note that for each value of $payouts$ only one payout can be submitted successfully. Further, a contract only issues a payout for the next value of $payouts$ once it has processed a final block containing the current value of $payouts$. As the contract removes the already spent withdrawals from the list, it prevents that any withdrawal is double spent. Although a payout temporarily invalidates all other payouts for the same value of $payouts$ and hence might invalidate same withdrawals, the unspent withdrawals will be included in each payout of the incremented $payouts$ and hence are spent with the next payout submission.