Private Data Objects: an Overview

MIC BOWMAN, ANDREA MIELE, MICHAEL STEINER, BRUNO VAVALA, Intel Labs, U.S.

We present Private Data Objects (PDOs), a technology that enables mutually untrusted parties to run smart contracts over private data. PDOs result from the integration of a distributed ledger and Intel Secure Guard Extensions (SGX). In particular, contracts run off-ledger in secure enclaves using Intel SGX, which preserves data confidentiality, execution integrity and enforces data access policies (as opposed to raw data access). A distributed ledger verifies and records transactions produced by PDOs, in order to provide a single authoritative instance of such objects. This allows contracting parties to retrieve and check data related to contract and enclave instances, as well as to serialize and commit contract state updates. The design and the development of PDOs is an ongoing research effort, and open source code is available and hosted by Hyperledger Labs [5, 7].

1 INTRODUCTION

As the community makes progress on the development of distributed ledger (DL) and smart contract (SC) execution architectures, the need for verifiability raises privacy and confidentiality concerns. On one hand, the participants in a DL need all the data involved in a transaction in order to perform the appropriate checks and to ensure that it is valid. In the case of SCs, this includes having access to the input and output data for executing and validating a SC. On the other hand, this requirement clearly clashes with the sensitive nature of information such as, for example, personal data, health records, or confidential corporate documents. As a consequence, this problem ends up restricting the application domain of SCs.

The solutions that have been proposed to address this issue range from private groups to cryptographically-secure computation. Private groups can achieve good performance by simply implementing access control mechanisms for group members, under the assumption that such members are trusted. The latter instead uses complex zero-knowledge constructions or multi-party computation protocols, which are based on cryptographic assumptions and are relatively expensive in practice.

In this paper we introduce Private Data Objects (PDOs), a solution based on a trusted execution environment (TEE) and a DL (Fig. 1). The TEE allows us to run contract code in isolation (from OS and other applications), ensuring execution integrity and data confidentiality. The DL allows us to track the registered contracts and enclaves, to serialize the state transitions (or updates) of contracts and to coordinate their interactions.

The combination of these technologies allows PDOs to achieve the desired balance between security, performance, efficiency and programmability. More precisely, PDOs leverage Intel SGX to set up a TEE for running contract code. Since SGX provides execution verifiability, contracts can run off-ledger (on SGX-capable machines). This improves

Author's address: Mic Bowman, Andrea Miele, Michael Steiner, Bruno Vavala, firstname.lastname@intel.com, SPR, Intel Labs, U.S..

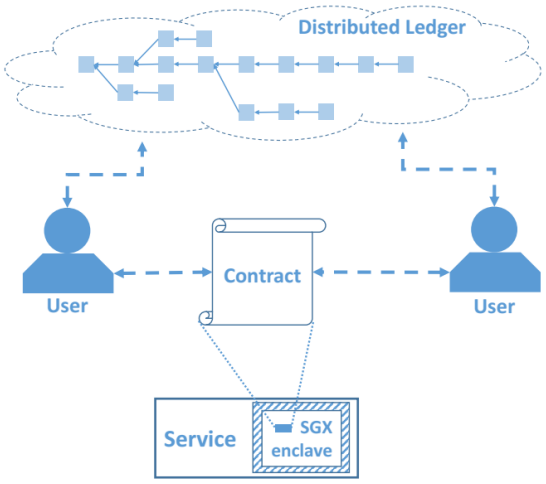

Fig. 1. PDO concept for private off-chain contract execution.

efficiency by avoiding the execution of a contract code by possibly numerous mutually-untrusted ledger nodes. In addition, PDO contracts can implement arbitrary programs, without therefore limiting the application domain.

Applications of PDOs are meant to enable mutually-distrustful individuals and organizations to share and process data according to pre-agreed policies, which are defined in the contract itself. Such policies to access and update data are carried with the object and enforced by trusted hardware. This holds regardless of where the object resides or how often it changes hands. However, although in principle any SGX platform can deliver such property, users only accept transactions that are recorded and thus validated on the ledger. The responsibility of submitting a transaction (on the ledger) is on the party who initiates the operation resulting in that transaction. PDOs let parties use pseudonyms for this task in order to guarantee privacy and unlinkability on the ledger.

The rest of this paper is organized as follows. We briefly outline some related projects for privacy preservation in DLs and SCs in Section 2. We specify a list of security threats that guides our design efforts in Section 3. Then we introduce the main components of PDOs and explain how they interact with each other in Section 4. We elaborate on the security aspects of PDOs and on the consequences of hardware compromise in Section 5. Finally, we summarize our findings and conclusions in Section 6.

2 RELATED WORK

Bitcoin and Ethereum are arguably the most popular distributed systems for implementing and executing SCs today. Also, they provide the underlying distributed ledger. However, they were not designed to ensure data confidentiality, since the information is public on the ledger.

Privacy and confidentiality in distributed ledgers. Several solutions have been proposed to solve the problem of information leakage in distributed ledgers. Zerocash [14] and Hawk [11] are of particular interest since they provide strong transactional and user

privacy as well as programmability. They both rely on cryptographic primitives (Zero-Knowledge arguments). Hawk additionally relies on a facilitating party which can be instantiated with Trusted Computing technology (e.g., Intel SGX).

Enigma [20] proposes the use of Secure Multi-Party Computation as a building block for the decentralized execution of SCs on private data, while relying on an external blockchain-based DL for identity management, access control and a tamper-proof event log. As this may require the action of several nodes, Enigma reduces redundancy by replicating computation and storage to a subset of the participating nodes.

Solidus [3] achieves transaction-graph confidentiality by enhancing Oblivious RAM with public verifiability. In particular, the new construction enables clients to update a public (on the ledger, though encrypted) physical memory and to generate a zero-knowledge proof of correct update.

The Global Synchronization Log (GSL) [6] bears similarities with PDOs in that it proposes to separate contract execution and data from the ledger. This allows GSL to limit the exposure of sensitive data to the entitled participants. Differently from PDOs, and similarly to Fabric (described next), GSL does not constrain the trusted computing base, nor it enforces the as-intended execution of contracts (i.e., entitled parties can misbehave).

Ekiden [4] provides a platform for SC execution that is close to PDOs. The main similarities lie in the off-ledger contract execution, the SGX-based data privacy approach, and persisting the contract state on the ledger for availability and consistency. Ekiden and PDOs however diverge on some key aspects, such as: enclaves are stateless and not contract-specific in PDOs, thereby providing a contract execution service on-demand; PDOs ecosystem includes key provisioning services which enable enclave selection and migration of contract executions, while Ekiden has a set of key managers that replicate keys on all of the compute nodes. In general, PDOs focuses more on real-world aspects such as scalability – e.g., expressing dependencies via CCL allows more asynchronous in execution of interacting contracts than enforcing them post-facto via proof of publications – while at the same time providing secondary layers of defense to reduce trust required in trusted hardware.

Lastly, Fabric (see below) has been extended [1] to execute chaincode on private data using SGX. Similarly to PDOs, only the contract code runs in the chaincode enclave, except that PDOs additionally make use of a contract interpreter. Differently from PDOs, it uses a ledger enclave to maintain (hashes of) the ledger state. Although it allows to verify the latest ledger state (assuming final consensus), it has to implement part of the Fabric peer and manage a possibly large state. PDOs instead are agnostic to the distributed ledger. Additional guarantees (e.g., last state verification, commitments, etc.) can be provided as required by leveraging features of the used distributed ledger (e.g., final consensus).

Distributed ledger technologies. Hyperledger Sawtooth [9] and Hyperledger Fabric [2, 8] are recent distributed ledger platforms. The former is made to scale to a large number of participants. Also, it enables customized applications and SCs through the concept of *transaction families*, which abstracts the ledger. The latter is designed for smaller sets of known participants and can run arbitrary SCs called Chaincodes. Also, it provides privacy

and confidentiality for transactions through its *channel*[1] concept, which defines a private network for a subset of the members to make transactions. PDOs can be easily built on Sawtooth by implementing custom transaction families. Also, they are able to provide stronger privacy and confidentiality guarantees with respect to Fabric.

Coco [13] is a framework for high-throughput and confidential blockchains for the enterprise. Differently from PDOs (and similarly to Fabric), it is designed for a known and approved set of participants. Also, it implements leader-based consensus protocols, though its support for pluggable consensus hints that it may support decentralized protocols as Sawtooth does. Similarly to PDOs, it is based on SGX for security and performance on trusted hardware. However, SGX is used to secure most of Coco's functionality, including core and SCs. PDOs instead limit SGX usage to protect the execution and data of SCs and do not require SGX-capable ledger validators.

Programming model. From a programming perspective, PDOs can be seen as the secure extension of the abstract data objects introduced in [12]. The extension is two-fold: (i) policies strongly bound to the data, (ii) data and policy integrity and confidentiality. In addition, the policies defined to access and process the data can be designed to free the user from any data-related representation, or to prevent the user from using implementation details, thereby abstracting it.

Additional related work. ELI [10] proposes a technique based on an abstract distributed ledger to implement stateful functionalities for otherwise stateless enclaves. PDOs use a ledger to address a similar problem for state management.

Town Crier [19] is a system designed to augment SCs with authenticated data feeds. It intermediates between a user SC on a blockchain and a trusted data source (i.e., HTTPS-enabled web services) to deliver source-authenticated data. PDOs can be extended to use Town Crier in future applications.

3 MODEL

We outline the functionality and the security properties required by PDOs.

3.1 Functionality

We assume the existence of a DL, whose validators receive and verify transactions and answer queries about accepted transactions and stored state. DL protocols such as consensus, gossip, node authentication, etc. are out of scope.

We assume the existence of SGX-enabled cloud service providers, which can provide on-demand hardware-protected execution of contracts. In addition, we assume such enclave services can contact the manufacturer (in our case, the Intel Attestation Service) for the verification of hardware-based attestations.

3.2 Security

We assume that the DL is available and reliable. So valid transactions can be submitted and are accepted by validators, which make the data available to other parties. Clearly, the

[1]Readers should be warned that the definition of *channel* is counterintuitive and varies across projects.

security properties of PDOs inherit the assumptions of the underlying distributed ledger (e.g., whether a majority or two-thirds of the validators must be honest).

We assume a computationally-bounded adversary as we rely on cryptographic primitives such as digital signatures and cryptographic hashes. The adversary can have access to the ledger's state and tries to link transactions, contracts and data to specific users. Also, the adversary can have access to any out-of-enclave data.

The Intel Attestation Service is assumed to be trusted. In particular, IAS correctly and completely reports whether an SGX-based attestation was issued with cryptographic material that belongs to SGX-enabled hardware.

Although SGX enclaves increase the cost of attacks, we assume that an adversary with enough resources can subvert an enclave. However, we assume that it is difficult for such an adversary to mount a scalable attack on the SGX platforms used in the PDO's distributed setting.

We distinguish between attacks that compromise data confidentiality and others that target integrity. In the former case, the data of all contracts executed by the compromised enclave is potentially exposed, though it maintains its correctnetss. In the latter case, the correctness of the data itself can be additionally compromised.

To mitigate these attacks, it is assumed that users are able to take action based on informed risk to calibrate the level of decentralization (e.g., how many enclaves are involved, what organization they belong to, where they are geographically located, etc.) for the provisioning of cryptographic material and the execution of contracts.

We assume that the contract interpreter and the contracts are correct. Specifically, the contract interpreter executes the contracts as they are intended to. Also, neither the interpreter nor the contract leak secret data. Although the correctness property is critical for the security of PDOs, we leave the burden of guaranteeing it to orthogonal research work.

4 OVERVIEW

PDOs (Fig. 1) allow mutually-untrusted individuals and organizations to access and update private data through pre-agreed policies. In particular, a PDO implements these policies in the SC code (C) and binds them to the data using Intel SGX. Such data constitutes the SC state ($\mathcal{S}_C$). Any state update is (and can only be) the result of a SC method invocation.

Method invocations (Fig. 2) are triggered by the user and performed by the SC interpreter ($\mathcal{I}$). In particular, the user grabs the most recent data from the ledger and requests an invocation to the Enclave Service (ES) who hosts the SGX enclave. $\mathcal{I}$ executes SC code over the input SC state, possibly resulting in a state transition and thus in a new state. This procedure is performed inside an SGX enclave ($\mathcal{E}$), which protects integrity and confidentiality of SC code and data. SC state and (optionally) code are stored in encrypted form while outside the enclave's trust boundary. The final results are returned to the user. The user eventually submits a transaction to the ledger about the new state which is verified and logged for validity and later audits.

The ES registers (Fig. 3) the contract enclave $\mathcal{E}_{\mathcal{I}}$ on the DL. The enclave is provided in the form of a SC-execution-as-a-service business model. The registration includes an enclave attestation verification step with the Intel Attestation Service (IAS), which verifies

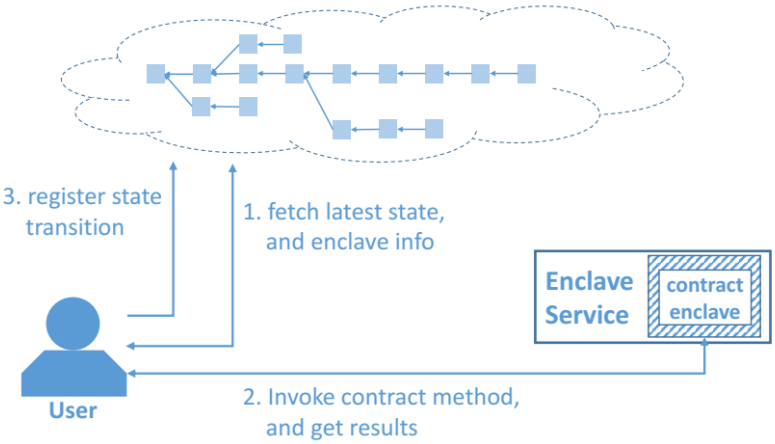

Fig. 2. PDO's contract method invocation.

the enclave's hardware root-of-trust. The ES then submits the registration transaction to the ledger together with the enclave's verification report.

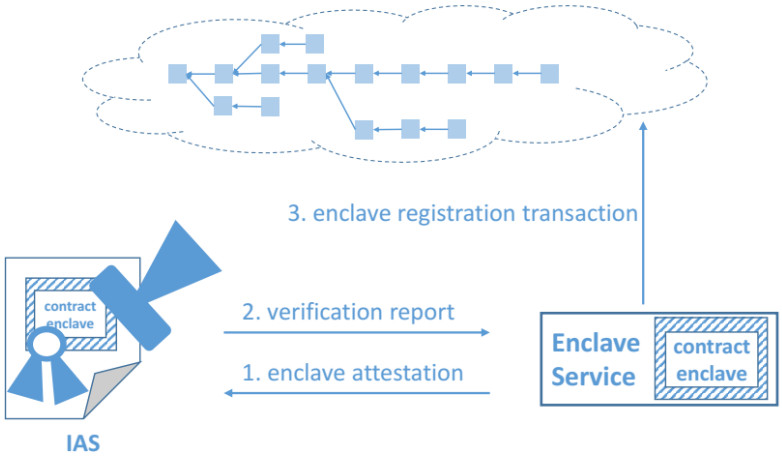

Fig. 3. PDO's contract enclave registration sequence.

Although the SC execution occurs off-chain, the SC is registered on the DL by the owner, who also authorizes its execution at known ESs (Fig. 4). The registration happens on the DL, while the execution authorization is realized through key provisioning. The latter is achieved by means of a set of Provisioning Services (PS) which provide cryptographic material for state encryption to a registered enclave. The chosen PSs and the actually provisioned enclaves are finally exposed on the DL in the contract registration transaction.

The DL ensures that there is a single authoritative instance and log of a PDO. Proofs (i.e., cryptographic signatures) of the operations and identities of the involved parties are exposed to enable verification and trust establishment by end-users. Additionally, the DL helps guarantee atomicity of updates across interacting PDOs and validates dependencies

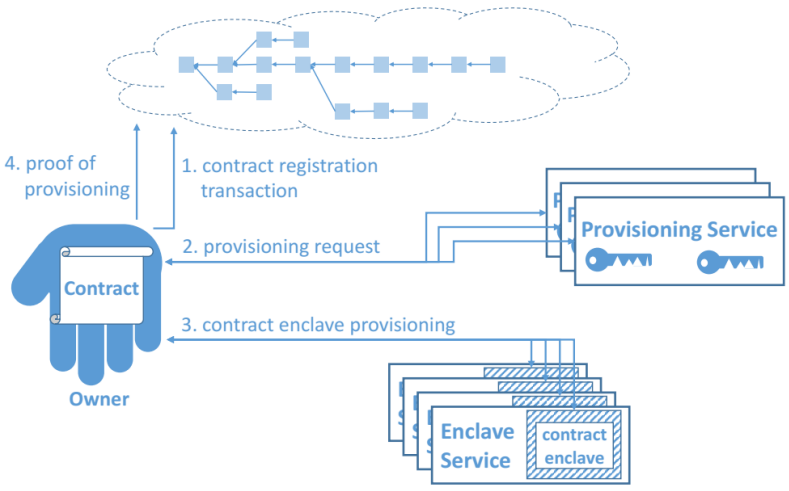

Fig. 4. PDOs phases for contract registration and contract enclave provisioning.

between transactions (if any). PDO state transitions are not trustworthy until the associated transaction is submitted to, and accepted by, the DL.

4.1 Distributed Ledger

PDOs are designed to work jointly with a DL. The DL is necessary to keep track of committed transactions, dependencies, registered/updated/revoked enclaves and contracts. Also, the DL validators enforce policies. For example, on enclave registration, they check that the enclave's attestation has been verified by the Intel Attestation Service (IAS).

Although PDOs are currently built on Hyperledger Sawtooth [9], in principle they do not require a dedicated DL. The reason is that several DLs allow customization of the logic used by the validators to validate and accept transactions. Hence, as long as such logic is expressive enough to address the requirements of PDOs, the ledger and PDOs can be seen as independent components.

Our Sawtooth-based implementation supports PDOs by means of three custom transaction families. A *transaction family* is the Sawtooth abstraction which allows ledger developers to implement custom rules to modify the ledger's state. Submitted transactions have to abide by such rules in order to pass the validation test and be appended to the ledger.

4.1.1 Contract Registry. This transaction family implements operations and rules for managing contracts. In particular, it enables a contract owner to register and revoke a contract, and to add/remove enclaves that can run it. Also, the contract owner uses this registry to specify the list of provisioning services (PSs) for a contract.

4.1.2 Enclave Registry. This transaction family implements operations and rules for managing enclaves. The available operations include registering, updating and revoking enclaves. Enclave services (ESs) have to comply with rules such as providing an enclave owner (who can update or revoke the enclave), the enclave's public keys (see Section 4.4)

and the IAS-verified enclave attestation. This information is validated in order to accept the submitted transactions.

4.1.3 Coordination and Commit Log (CCL). The CCL is one the unique charactestics of PDOs as it separates transaction ordering and semantics, similarly to the Global Synchronization Log (GSL) [6]. The CCL transaction family allows the expression of dependencies between state update transactions. Similarly to the GSL, the state update logic defined inside contracts is off-chain, while the contract identifiers are on-chain and used to build a dependency graph. The validators then enforce these dependencies by rejecting transactions whose dependencies are not committed.

These dependencies enable the coordination of the updates to one or multiple objects. In particular, they allow the enforcement of the serial progression of the contract state. Also, they allow the specification and enforcement of the progression of a contract state conditional to the progression of other contract states, while enabling maximal concurrency among interacting contracts and ledger interactions. The validity of a progression is based on the digital signature of a contract enclave (see Section 4.4).

4.2 Provisioning Services

PSs answer the contract owner (O_C)'s requests to provision secrets to $\mathcal{E}_I$. The secrets are eventually used to encrypt the state $\mathcal{S}_C$. Most importantly, only a provisioned $\mathcal{E}_I$ has access to all secrets and so is able to decrypt the $\mathcal{S}_C$.

The provisioning protocol is intermediated by the contract owner (Fig. 4). The PSs participate in the protocol as long as they can successfully verify the registration of contract and enclaves on the DL. The protocol can be implemented on a generic interface, which allows to customize it according to a specific application/deployment scenario. The currently implemented protocol let the PSs generate secrets, and it guarantees security as long as one of the participating PSs is honest.

The purpose of using PSs is to make $\mathcal{S}_C$ encryption (i) enclave-independent and (ii) ES-independent. The first property is important to ensure that contract enclaves are stateless with respect to contracts (i.e., they only need a contract and the contract state to perform a method invocation). This prevents the problem of how to move the state between two enclaves running on separate platforms. The second property is necessary because an ES is considered untrusted by other parties. Once the enclave terminates and transactions are committed on the ledger, the ES cannot be trusted to persist critical enclave's data (e.g., sealed storage and encrypted state). This would open it to trivial albeit severe denial of service attacks.

Notice that the DoS issue is strictly related to $\mathcal{S}_C$, and not to any enclave's data in sealed storage. The enclave in fact has some private cryptographic material, which is sealed and should be persisted for later invocations. However, the unavailability of this data only makes the associated enclave's execution environment unavailable, since the enclave cannot recover its private cryptographic material. In particular, this problem does not prevent the use of other enclaves, possibly at other ESs.

4.3 Enclave Hosting Service

An Enclave Service (ES) is an entity that owns SGX-enabled platforms, and provides generic SGX-based Contract Enclaves as a service (e.g., a cloud provider). However, ownership of the SGX-enabled hardware is not sufficient for PDOs. ESs additionally have to comply with the enclave registration/update/revocation requirements on the DL, and to prove that such enclaves run the correct software. The latter is achieved by verifying SGX enclaves with IAS, while the former is achieved by submitting to the DL nodes valid transactions, which have to include IAS signed verification reports.

4.4 Contract Enclave

Contract Enclaves ($\mathcal{E}_{\mathcal{I}}$) are SGX enclaves which include a contract interpreter ($\mathcal{I}$) and thus can run contracts. For the sake of making them general, they do not include any contract or data. Hence, one $\mathcal{E}_{\mathcal{I}}$ can handle many PDO instances, i.e., $C, \mathcal{S}_C$ pairs. In this sense, an ES provides a $\mathcal{E}_{\mathcal{I}}$-as-a-service.

During its lifetime, a $\mathcal{E}_{\mathcal{I}}$ is registered into/revoked from the enclave registry (Section 4.1.2), added to/removed from the list of enclaves authorized to execute a specific contract C in the contract registry (Section 4.1.1). Also, it generates execution results, which are described in transactions submitted to the CCL (Section 4.1.3) for validation.

An ES leverages IAS to let others (i.e., DL validators) establish trust in a $\mathcal{E}_{\mathcal{I}}$. Then, $\mathcal{E}_{\mathcal{I}}$-generated and attested key pairs are used to preserve trust and extend it to the enclave's signed statements. In particular, the $\mathcal{E}_{\mathcal{I}}$ SGX attestation is issued and submitted to the validators as a proof of correct execution. The attestation covers (and thus extends trust to) two $\mathcal{E}_{\mathcal{I}}$ public keys, namely: a public verification key, which is used to check the enclave's signatures and to limit the interactions with IAS; and a public encryption key, which is used to provision secrets to the enclave. The two public/private key pairs are generated by the enclave, so they are bound to a specific enclave instance. The public keys are attested, registered and retrievable from the DL, while their private counterparts are saved by the enclave in sealed storage.

Although the $\mathcal{E}_{\mathcal{I}}$ is responsible for the confidentiality of $\mathcal{S}_C$, the encryption key is not bound to the enclave itself, but rather securely provisioned to it on a per-contract and per-contract-owner basis. $\mathcal{E}_{\mathcal{I}}$ verifies the signatures of the PSs over the provisioned secrets. Eventually, it signs a proof of provisioning, which includes the PSs and the (encrypted) secrets. Ultimately, it is the contract owner that adds such provisioned enclave to the list of enclaves (on the DL) which are authorized to run the contract.

4.4.1 Contract Interpreter. Similarly to the Ethereum EVM [18], the contract interpreter $\mathcal{I}$ executes a contract code C registered on the DL by a contract owner O_C. Most importantly, $\mathcal{I}$ is not bound to a specific C, so $\mathcal{E}_{\mathcal{I}}$ can run any contract written in the interpreter's language.

Since $\mathcal{I}$ runs in an SGX enclave, there are some features that are desirable in an interpreter's implementation. (i) *Contract code sandboxing*. This prevents (possibly) malicious contracts implemented by (possibly) untrusted third-parties to tamper with the rest of the enclave and the ES platform. (ii) *Small code footprint/TCB*. This arguably reduces the complexity of the code, thereby simplifying code audit. (iii) *Unique contract representation.*

This makes a contract easily verifiable by the participants. Also, all participants can agree on the same copy, which can be identified on the DL with one cryptographic hash, thereby avoiding multiple documents due to non-deterministic compilations. Finally, it easily enables the authentication and execution of the agreed copy within the enclave. (iv) *Efficient contract state management.* This allows a contract to process the state without requiring to load it upfront inside the enclave.

PDOs leverage a small abstraction inside $\mathcal{E}_I$ for generic contract interpreters. The current default implementation relies on *TinyScheme* [15], which is based on the *Scheme* [16, 17] functional language. We plan to support additional interpreters.

4.4.2 Contract. A contract (C) defines and implements methods for accessing and updating data in the contract state. As parties need to agree on a C before using it in a PDO, it is expected that they communicate with each other offline in order to share the contract code offchain. So the contract must not necessarily be public. Its execution is designed to occur natively off-chain inside SGX enclaves at the chosen ESs (Section 4.3).

4.4.3 Contract State. The contract state $\mathcal{S}_C$ indicates the (private) data which a PDO associates to a specific contract. At C runtime, the contract has access to plaintext state, which is managed by $\mathcal{I}$. In the other circumstances, $\mathcal{S}_C$ is encrypted while stored outside of the contract enclave.

Let us provide some details of the workflow. Upon the first contract method invocation, $\mathcal{I}$ initializes an empty state $\mathcal{S}_C$ for C. In subsequent invocations, $\mathcal{I}$ reads or modifies $\mathcal{S}_C$ according to the logic implemented by C. $\mathcal{I}$ can access $\mathcal{S}_C$ in plaintext, since SGX maintains the confidentiality of the data. $\mathcal{E}_I$ decrypts (resp. encrypts) $\mathcal{S}_C$ before (resp. after) C requires data. These operations are performed using a contract-specific key, which is derived from the provisioned secrets (see Section 4.2).

4.5 Contract Owner

The contract owner O_C is the person/organization that creates a PDO instance by registering a contract C and mediating the provisioning of one or more $\mathcal{E}_I$s. The registration is done by submitting a transaction on the DL. The provisioning happens by contacting the chosen PSs and ensuring that $\mathcal{E}_I$ was provisioned. The proof of correct provisioning is exposed on the DL, thereby showing that an $\mathcal{E}_I$ is able to run the registered C. The contract creation happens through method invocation in a $\mathcal{E}_I$, which (being the first invocation) initializes an empty $\mathcal{S}_C$.

4.6 Users

Users transact together in three phases. First, they send method invocation messages to contracts. User authentication can be performed by $\mathcal{E}_I$ or by a contract according to the implemented policies. In this way only authorized participants are able to invoke methods. Second, they check the results returned by a $\mathcal{E}_I$ and delivered to them through the ES. Third, they submit the resulting transaction to the DL.

The design rationale for such protocol stems from the mutual-distrust between the parties (including the ESs). At the ESs, as the helper process for $\mathcal{E}_I$ is not in the TCB, it cannot be trusted to submit a transaction to the ledger or handle communication between

objects. In a specular manner, if a user decides not to commit the results, the helper process cannot be trusted not to do so.

PDOs leverage the user's motivation to invoke a method and get results to solve this problem. In particular, a user acts as a *channel* for communication between the enclave and the ledger (and additionally between objects).

PDOs implement the channel concept by enforcing that only the one who invokes a method can submit the resulting transaction to the DL. The channel is set up as follows: the user generates a fresh key pair at each invocation, includes the public key in the method invocation message and signs the message; after the method invocation, $\mathcal{E}_I$ includes the public key in the results; finally, the DL validators only accept the resulting transaction if it is also signed by the holder of the corresponding (channel's) private key. As a different key pair is used for each transaction commit, this prevents DL validators from correlating transactions, thereby protecting the user's privacy.

5 SECURITY ASPECTS OF PDO'S DECENTRALIZED MODEL

A critical objective of PDOs is to avoid the presence of single points of failures that can jeopardize the entire system. Once this can be assured, the next step is how to detect and remove failed end-points, such as compromised enclaves. In this section we elaborate on these key security aspects.

5.1 Trust but verify what is on the ledger

As participants are mutually-untrusted, all of their interactions are based on mechanisms that allow to verify each other's work. We briefly summarize the verification capabilities of the participants.

First, the DL is designed to distribute trust among the validators. As we mentioned however, its design is orthogonal to PDOs.

Second, the contract owner entrusts PSs to provision an enclave, and an ES to execute it. However, the owner later verifies the correct enclave execution, secret provisioning and contract/enclave registration transaction on the DL. An ES submits but verifies enclave registration/revocation transactions. PS's verify with the DL that they are going to provision actual enclaves before moving forward. Users entrust an ES to execute a contract enclave, but verify the enclave's work and directly submit transactions.

Additionally, the users are able to verify the provisioning procedure. The contract owner in fact publishes on the DL the identities of the selected PSs, together with their provisioned encrypted secrets and a proof of correct enclave provisioning. So users can check on the DL whether the PSs belong to one or more possibly reputable organizations and whether enough PSs have been selected. Ultimately, this allows users to trust that the encryption key is secure and only available to the provisioned enclave.

The encryption key plays a critical role in state confidentiality but not in its integrity. If an adversary compromises the key, then exposing the state is a trivial task. However, the adversary cannot tamper with the integrity of the state in order to corrupt a future committed state. The reason is that DL validators only accept a CCL transaction (Section 4.1.3) if an enclave signs a valid state update (i.e., a transition from the last committed

state to another). So if the adversary tampers with the state before/after the execution, the validators would be unable to accept the transaction.

5.2 Compromised Enclaves

PDOs aim at taking advantage of SGX without ruling out the possibility that enclaves could be compromised. As enclaves deal with confidential data, this would trivially result in the exposure of the handled data. Since PDOs natively use per-PDO state encryption keys, the problem would impact only the contracts running on the compromised enclave.

The design however enables mechanisms for limiting the confidentiality breach and for maintaining integrity. The first mechanism is to limit the explicitly provisioned enclaves and possibly to run them at reputable ESs. The mechanism enables a contract owner to make these decisions and also allows the users to take informed risk. In fact, the details of the PSs are exposed on the DL.

Another mechanism is the verification of an enclave's work. So while confidentiality is compromised, this still allows to preserve the integrity of the computation. PDOs can implement verification by letting additional enclaves re-execute a method using the same input parameters. If their outcomes match, the output of an enclave can be used to commit the transaction. Otherwise, the majority of matching responses (ideally $n - 1$ out of n) can be considered as a proof of misbehavior of the remaining enclaves. This could result for example in enclave revocation and/or state commit revocation transactions.

6 CONCLUSIONS

PDOs introduce a new way to securely run smart contracts over private data by leveraging Intel SGX. They are designed following a decentralized trust model in order to work in a network of mutually-untrusted entities and participants. In addition, they take advantage of a distributed ledger which can maintain a single authoritative instance of the objects. Their design is ledger-agnostic, so the implementation can be adapted for different ledger architectures.

From a security perspective, PDOs protect user privacy on the ledger and data confidentiality in secure enclaves. The security architecture exposes enclave hosting services and enclave provisioning services on the ledger (upon registration). This enables participants to establish trust in the enclaves based on informed risk (i.e., informing about who runs them and who provisions them). Most importantly, PDOs are designed to limit confidentiality breaches, to provide mechanisms to guarantee integrity in spite of an enclave compromise, and to revoke misbehaving enclaves.

REFERENCES

[1] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. 2018. *Blockchain and Trusted Computing: Problems, Pitfalls, and a Solution for Hyperledger Fabric.* Technical Report arXiv:1805.08541v1 [cs.DC]. arXiv.org.

[2] Christian Cachin. 2016. Architecture of the Hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*.

[3] Ethan Cecchetti, Fan Zhang, Yan Ji, Ahmed Kosba, Ari Juels, and Elaine Shi. 2017. Solidus: Confidential Distributed Ledger Transactions via PVORM. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 701–717.

[4] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2018. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *arXiv preprint arXiv:1804.05141* (2018).

[5] Intel Corp. 2018. Private Data Objects. https://github.com/hyperledger-labs/private-data-objects (2018).

[6] DigitalAsset. 2018. The Global Synchronization Log. http://hub.digitalasset.com/hubfs/Documents/TheGlobalSynchronizationLog.pdf (2018).

[7] Linux Foundation. 2018. Hyperledger Labs. https://hyperledger-labs.github.io/ (2018).

[8] IBM. 2018. Hyperledger Fabric. https://www.hyperledger.org/projects/fabric (2018).

[9] Intel. 2018. Hyperledger Sawtooth. https://www.hyperledger.org/projects/sawtooth (2018).

[10] Gabriel Kaptchuk, Ian Miers, and Matthew Green. 2017. *Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers.* Technical Report. Cryptology ePrint Archive, Report 2017/201, 2017. https://eprint. iacr. org/2017/201.

[11] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on.* IEEE, 839–858.

[12] Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages.* 50–59.

[13] Microsoft. 2018. The Coco Framework. https://github.com/Azure/coco-framework/blob/master/docs/Coco%20Framework%20whitepaper.pdf (2018).

[14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on.* IEEE, 459–474.

[15] Dimitrios Souflis and J Shapiro. 2005. TinyScheme. http://tinyscheme.sourceforge.net (2005).

[16] Gerald Jay Sussman and Guy L. Steele, Jr. 1998. Scheme: A Interpreter for Extended Lambda Calculus. *Higher Order Symbol. Comput.* 11, 4 (Dec. 1998), 405–439.

[17] Gerald J Sussman and Guy L Steele Jr. 1975. *SCHEME: An Interpreter for Extended Lambda Calculus.* Technical Report AI Memo n.349. MIT Artificial Intelligence Laboratory.

[18] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.

[19] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS).* 270–282.

[20] Guy Zyskind, Oz Nathan, and Alex Pentland. 2015. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471* (2015).