zk-agreements: A privacy-preserving way to establish deterministic trust in confidential agreements

To-Wen Liu[1] and Matthew Green[1]

Johns Hopkins University
tliu96@jh.edu, mgreen@cs.jhu.edu

Abstract. Digital transactions currently exceed trillions of dollars annually, yet traditional paper-based agreements remain bottlenecks for automation, enforceability, and dispute resolution. Natural language contracts introduce ambiguity, demand manual processing, and lack computational verifiability, all of which hinder efficient digital commerce. Computable legal contracts, expressed in machine-readable formats, offer a potential solution by enabling automated execution and verification. Blockchain-based smart contracts strengthen enforceability and accelerate dispute resolution, but current implementations risk exposing sensitive agreement terms on public ledgers, creating significant privacy and competitive intelligence concerns that restrict enterprise adoption.
We introduce *zk-agreements*, a protocol designed to transition from paper-based trust to cryptographic trust while preserving confidentiality. Our design combines zero-knowledge proofs to protect private agreement terms, secure two-party computation to enable private compliance evaluation, and smart contracts to guarantee automated enforcement. Together, these components deliver both privacy preservation and computational enforceability, resolving the fundamental tension between transparency and confidentiality in blockchain-based agreements.

Keywords: zero-knowledge proofs · computable legal contract · smart contracts

1 Introduction

Contracts record the terms and conditions of an agreement between two or more parties. Traditionally, contracts are physical documents that require the contracting parties to draft, sign, and store them. These paper-based contracts have some disadvantages: they are prone to loss and errors, require effort to prevent tampering, and resolving conflicts related to them is time consuming. To resolve the issues of paper-based contracts, digital contracts have emerged as a better alternative, as they are more efficient in negotiating agreements, exchanging documents, and signing. In addition, digital contracts reduce the costs associated with storage and physical resources and can be stored electronically in encrypted local storage or a highly secure repository. Cryptography methods

can also be applied to provide confidentiality, authenticity, and integrity of a secret document.

To create a computable and legally binding digital contract, researchers at Stanford Law School have proposed a form of contract known as computable legal contracts [30]. Computable legal contracts enable natural language agreements with automated execution of contractual terms via machine-executable code. Although computable legal contracts are in their development phase, humans can easily use natural language processing models to transform textual contracts into computable legal contracts [14]. So far, we have converted physical contracts into machine-computable contracts, but these techniques still have some problems.

Lack of a deterministic way to resolve disputes. Physical contracts and computable legal contracts are supported by legal systems, which vary from country to country. However, when disputes arise, resolving them can often be a cumbersome and lengthy process, typically requiring court intervention to render a final judgment. To mitigate this problem, blockchain-based smart contracts[33] can be used to resolve disputes in a deterministic way [24, 25, 31, 32]. Computable legal contract code can be translated into a smart contract [31] that runs on a public blockchain, like Ethereum [13, 33]. In this way, contractual legal logic can be completed quickly when the terms of the contract are fully fulfilled. In addition, when one party breaches the contract, the penalty clause can also be enforced automatically. However, when we run our computable legal contract as a confidential agreement on a blockchain, another problem arises.

A privacy problem. Blockchains[28] are good at establishing trust between peers who do not know each other. However, on most of the blockchains that we use today, every piece of transactional data is public, sacrificing privacy for the parties involved. For example, in Ethereum smart contracts[13, 33], the general public can view every transaction detail from actions such as function calls or monetary transfers. These concerns have motivated privacy-enhancing payment systems such as Tornado Cash [29] to keep ether transfer transactions private and ZEXE [11] to privatize decentralized computation.

David versus Goliath. In the current commercialized society, buyers are often the weaker party compared to large merchants. When a contract dispute arises, the weaker party often lacks the resources required to fight and struggles to find an effective remedy outside of litigation, which is a lengthy and expensive process.

Generally, we propose a privacy-preserving framework for executing and resolving disputes in computable online legal contracts. This work is motivated by the limitations outlined above. Our aim is to design a cryptographic protocol on top of a distributed ledger that ensures deterministic trust while preserving confidentiality. A central challenge is the inherently public nature of blockchain transactions, which makes them unsuitable for storing sensitive agreements. Another lies in translating contracts, typically expressed in natural language, into

machine-executable code. A further challenge is the need to evaluate contractual outcomes both securely and efficiently. To overcome these obstacles, we develop the *zk-agreements* protocol, which leverages zero-knowledge proofs[19] and secure multi-party computation[17, 35]. These techniques enable trust and verifiable compliance while protecting sensitive information, allowing disputes to be resolved without disclosing the confidential terms of the agreement between the buyer and the seller.

1.1 Our Contribution

We present, *zk-agreements*, the first protocol to combine legal enforceability with cryptographic privacy for confidential business agreements. Concretely, our main contributions are:

(1) Privacy-preserving legal execution architecture: We design a novel hybrid system combining zk-SNARKs[9, 16, 20, 21], secure multiparty computation and smart contracts that enables legally binding contract execution while preserving confidential terms. Unlike existing privacy-preserving systems that sacrifice legal enforceability, our work maintains both cryptographic and legal guarantees.
(2) Verifiable off-chain contract evaluation: We develop a protocol that allows buyers and sellers to demonstrate compliance with their confidential agreements without revealing sensitive business data. Independent evaluators process real-world data inputs and forward them to a trusted execution environment, which checks whether the contractual terms are satisfied and produces succinct proofs for on-chain settlement.
(3) Legal-Cryptographic bridge: We implement automated translation from natural language contracts (via Accord Project[1, 5] tools) to cryptographically enforceable protocols, enabling seamless integration of existing legal frameworks with privacy-preserving blockchain systems.

Our approach addresses critical gaps in existing work: privacy-preserving systems lack legal enforceability, while smart legal contracts expose confidential terms on public blockchains. *zk-agreements* is the first system to get the goods of both worlds, preserving privacy while ensuring legal enforceability. A comparison with existing systems is provided in Table 1.

1.2 Related Works

Our work operates at the intersection of several domains: zero-knowledge proofs[19], secure two-party computation[18, 17, 35], computable legal contracts and privacy-preserving blockchain applications.

Zero-knowledge proofs for privacy-preserving ledgers. The evolution from Zerocoin[27] to Zerocash[8] demonstrated the validity of zero-knowledge

proofs for private transactions on public blockchains. Later on, ZEXE[11] extends zero-knowledge techniques from payment systems to general-purpose functions. Similarly, Zether[12] and Hawk[23] leverage zk-SNARKs[9, 16, 20, 21] for confidential smart contracts. Recent work by Boneh et al.[10] explores how zero-knowledge proofs can provide privacy while satisfying regulatory requirements, providing a framework for privacy-preserving systems. However, these works focus on specific use cases rather than general-purpose legal agreements.

Computable legal contracts and automated contract execution. Ethereum [33] is the first blockchain that supports programmable contracts on a public blockchain, which enables Turing-complete computation within a decentralized world computer. However, such automatic contract execution environments suffer from privacy limitations when dealing with confidential business terms. The OpenLaw project[34] is a pioneer that integrates legal contracts with smart contracts, allowing legal documents to trigger blockchain transactions. The[1] has developed comprehensive tools for converting traditional contracts into machine-executable formats (see examples in Appendix A). Ma[25] provides a comprehensive overview of blockchain-enabled smart legal contracts, highlighting the potential for automated dispute resolution and contract enforcement. However, running a smart contract that contains confidential legal content on a public blockchain is impractical. Our work seeks to preserve the privacy guarantees of the computable legal contract while ensuring that all parties fulfill their obligations under the agreement.

Table 1: Comparison of privacy-preserving contract systems.

System	**Privacy**	**Gas / On-chain**	**Scale**	**Legal**
zk-agreements	Content of agreements	Commit $\approx$ 1.2M Eval $\approx$ 0.35M	Medium	Yes
Ethereum [33]	None	Transfer $\approx$ 21K ERC-20 $\approx$ 50K	High	Limited
Zether [12]	Amount, sender/receiver	$\approx$ 7.19M	Medium	No
OpenLaw [34]	None	N/A	Medium	Yes
Tornado Cash [29]	Full unlinkability	Dep $\approx$ 1.1M Wdr $\approx$ 0.4M	Low	No

2 Preliminaries

2.1 Zero-knowledge proofs.

At a high level, a zero-knowledge proof allows a prover to convince a verifier that a statement is true without revealing any information beyond the validity

of the statement itself. For example, one might prove knowledge of a secret identity without disclosing the identity itself. Formally, such a proof asserts that a prover knows a witness w to a public instance x such that a relation $C(x, w)$ holds.

Definition 1 (Zero-knowledge proof system). A zero-knowledge proof system for an NP relation $R = \{(x, w) : C(x, w) = 1\}$ consists of a tuple of algorithms $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ with the following properties:

- **Completeness:** For any $(x, w) \in R$, if $\mathrm{crs} \leftarrow \mathsf{Setup}(1^\lambda)$ and $\pi \leftarrow \mathsf{Prove}(\mathrm{crs}, x, w)$, then $\mathsf{Verify}(\mathrm{crs}, x, \pi) = 1$ with overwhelming probability.
- **Soundness:** For any polynomial-time adversary $\mathcal{A}$ and $x \notin L$, the probability that $\mathcal{A}$ produces a proof π such that $\mathsf{Verify}(\mathrm{crs}, x, \pi) = 1$ is negligible.
- **Zero-Knowledge:** There exists a polynomial-time simulator Sim such that for any $(x, w) \in R$, the distributions $\{\mathsf{Prove}(\mathrm{crs}, x, w)\}$ and $\{\mathsf{Sim}(\mathrm{crs}, x)\}$ are computationally indistinguishable.

In our construction, we instantiate the proof system with PLONK [16] over the BN254 elliptic curve[6], achieving 128-bit security. The relation R encodes contract-compliance predicates: the public input x represents observable contract outcomes, while the witness w captures confidential agreement terms and execution traces.

2.2 Computable legal contracts

Computable legal contracts[5, 30, 34] bridge traditional legal agreements with machine-executable code, enabling automated contract execution while preserving legal validity.

Definition 2 (Computable Legal Contract). A computable legal contract is a tuple $\mathsf{CLC} = (T, L, D, E)$ where:

- T is the natural language legal text
- L is the executable logic extracted from T
- D is the data model defining contract parameters
- E is the execution environment mapping real-world events to contract states

In our protocol, we employ the Accord Project toolkit[1, 5] to convert traditional contract text into machine-executable legal agreements. First, the natural language contract is parsed to extract executable clauses and conditional statements. Next, the Ergo domain-specific language [4] formalizes the extracted logic into predicates with well-typed functions that capture the behavior of the contract. Concurrently, the Accord Project's Concerto model language [3] defines structured schemas for contract parameters and data models in a technology-agnostic manner. Finally, the Ergo code is executed within a TEE, while the commitment of the computable legal contract is encoded into an arithmetic circuit. Further technical details and illustrative examples of how Accord works are deffered to Appendix A.

2.3 Trusted Execution Environment.

A Trusted Execution Environment (TEE)[15] is a hardware-enforced, isolated execution domain that ensures the confidentiality and integrity of code and data even in the presence of a compromised operating system or other untrusted software components. In particular, TEEs protect sensitive inputs, internal state, and computation by preventing external access or tampering; guarantee that only authenticated code is executed; and support secure storage, cryptographic operations, and remote attestation. Our protocol assumes that the evaluator's computations are performed inside a TEE, so that private agreement terms or other observed data remain hidden, while outputs (such as proofs or commitments) can be safely published. The threat model for our TEE excludes physical tampering, side-channel leakage beyond those implicitly addressed by the hardware implementation, and assumes that attestation can be verified by other parties.

3 Construction of zk-agreements

3.1 Computable Legal Contract Lifecycle

In our protocol, a computable legal contract progresses through a well-defined lifecycle, which we model as a finite state machine. Let

$$\mathcal{S} = \{\mathsf{INIT},\ \mathsf{EXECUTION},\ \mathsf{EVALUATION},\ \mathsf{COMPLETED}\}$$

denote the set of lifecycle states. A contract instance *clc* begins in the state INIT when created by the contracting parties. It then transitions to EXECUTION as obligations are performed, to EVALUATION when outcomes are assessed, and finally to COMPLETED once the agreement is finalized and settled.

3.2 Setups

The setup phase initializes the foundational building blocks of our protocol, including the zero-knowledge circuits, the computable legal contract, and the cryptographic keys of the actors.

Trusted setup for zero-knowledge proofs. $(1^\lambda) \rightarrow (srs)$ This procedure takes as input a security parameter 1^λ and outputs public parameters *srs* (structured reference string) for the trusted setup ceremony of the PLONK[16] zero-knowledge proof scheme.

Computable Legal Contract Compilation. $(agr_{text}) \rightarrow (clc)$ To initialize a computable legal contract *clc*, a natural-language legal agreement agr_{text} is compiled into a machine-interpretable artifact. This procedure is invoked jointly by a buyer and a seller holding a confidential natural-language agreement. Given the public parameters pp, the parties negotiate over a secure channel, and upon completion the agreement is compiled using Accord Project's[1, 5] SDKs. The resulting *clc* is composed of two components:

- clc_{data}: a structured data model capturing the contractual terms and obligations.
- clc_{logic}: an executable specification that can be evaluated by the protocol.

After compilation, the state of the computable legal contract is set to INIT.

Key generation for participants. $(1^\lambda) \rightarrow (pk, sk)$ There are three principal actors in the protocol: the buyer, the seller, and the evaluator. Each actor invokes the KeyGen procedure to generate a pair of asymmetric keys, consisting of a public key pk and the corresponding secret key sk. Formally, the parties obtain

$$\begin{aligned}(pk_{buy}, sk_{buy}) &\leftarrow KeyGen(1^\lambda),\\ (pk_{sel}, sk_{sel}) &\leftarrow KeyGen(1^\lambda),\\ (pk_{eva}, sk_{eva}) &\leftarrow KeyGen(1^\lambda).\end{aligned}$$

The public keys are used for authentication and message binding within the protocol, while the secret keys remain private to their respective owners.

3.3 Algorithms

SignCLC $(sk_{buy}, sk_{sel}, clc) \rightarrow (clc_{signed})$. Given a computable legal contract clc, the buyer and seller each generate a digital signature on its canonical representation. Let $c = \mathsf{Hash}(clc)$ be the contract digest. The buyer computes $\sigma_{buy} = \mathsf{Sign}(sk_{buy}, c)$, and the seller computes $\sigma_{sel} = \mathsf{Sign}(sk_{sel}, c)$. The resulting signed contract is

$$clc_{signed} = (clc, \sigma_{buy}, \sigma_{sel}).$$

This step ensures that both parties are jointly bound to the same contract before it proceeds to later verification and execution.

CommitCLC $(clc_{signed}, k) \rightarrow (h,\ clc_{comm})$. Given a signed computable legal contract clc_{signed} and a nullifier k, this procedure is implemented as the Commitment circuit in Circom[7] and generates a commitment for on-chain registration as follows:

1. Compute a nullifier hash $h \leftarrow \mathsf{Hash}(k)$.
2. Derive a program digest $pd \leftarrow \mathsf{Digest}(clc_{signed})$.
3. Compute a contract commitment $clc_{comm} \leftarrow \mathsf{Hash}(k \parallel pd)$.
4. Output (h, clc_{comm}) as the registered commitment to this agreement.

SubmitCommitment $(clc_{comm}, v) \rightarrow tx$. This function records a commitment of a new computable legal contract on-chain. Given a contract commitment clc_{comm} and the total contract value v, it constructs and broadcasts a transaction $tx := (clc_{comm},\ v)$ to the blockchain. Once the transaction is finalized, the commitment clc_{comm} is inserted into the smart contract's Merkle tree[26] of registered agreements, and the state of the associated contract is updated to EXECUTION. This registration step provides the immutable reference that will later be used during proof verification.

ExecuteEvaluation $(clc_{signed}, [d_j]_{j=1}^{n}) \rightarrow rat$. Given the signed contract clc_{signed} and the measured data $[d_j]j = 1^n$ used as inputs for its evaluation, this procedure executes the contract evaluation within a TEE to preserve the confidentiality of the agreement. The contracting parties first install the evaluation code, which is generated from the contract logic and embedded within *clcsigned*. The evaluator then collects the measured data $[d_j]j = 1^n$ and securely transmits it to the TEE together with *clcsigned*. The TEE runs the embedded evaluation code on these inputs and outputs the final compliance ratio.

$$rat \leftarrow \mathsf{Eval}(clc_{signed}, [d_j]_{j=1}^{n}).$$

Upon completion, the state of the computable legal contract is updated to EVALUATION.

GenerateProof $(srs,\ k,\ clc_{signed},\ rt,\ hp,\ hd,\ pk_{buy},\ pk_{sel},\ pk_{eva},\ rat) \rightarrow \pi$. Given a secret nullifier k, a contract commitment clc_{comm}, the current Merkle root rt, a Merkle proof path hp with corresponding direction bits hd, the public keys of the buyer, seller, and evaluator $(pk_{buy},\ pk_{sel},\ pk_{eva})$, and the computed compliance ratio rat, this procedure is implemented as the Evaluation circuit in Circom[7] and generates a zero-knowledge proof of knowledge of a valid pair consisting of the nullifier k and the confidential signed contract clc_{signed} as follows:

1. Recompute a nullifier hash $h \leftarrow \mathsf{Hash}(k)$.
2. Derive a program digest $pd \leftarrow \mathsf{Digest}(clc_{signed})$.
3. Compute a contract commitment $clc_{comm} \leftarrow \mathsf{Hash}(k \parallel pd)$.
4. Verify the membership of clc_{comm} in the Merkle tree by computing $rt_{new} \leftarrow \mathsf{MerkleProof}(rt, hp, hd)$.
5. Check that $rt_{new} = rt$ to confirm the integrity of the commitment inclusion.
6. Generate a zero-knowledge proof $\pi \leftarrow \mathsf{Proof}(rt,\ h,\ clc_{comm},\ pk_{buy},\ pk_{sel},\ pk_{eva},\ rat)$ attesting to the correctness of the evaluation.
7. Output π as the succinct proof linked to this contract evaluation.

VerifyProof $(\pi,\ srs,\ h,\ rt,\ pk_{buy},\ pk_{sel},\ pk_{eva},\ rat,\ v) \rightarrow \{0, 1\}$. This function is implemented as a blockchain transaction, with a smart contract verifier already deployed on-chain. The contract maintains a nullifier table and a Merkle tree[26] that collectively store numerous commitments to computable legal contracts. Given a zero-knowledge proof π, the structured reference string srs, the public keys $(pk_{buy},\ pk_{sel},\ pk_{eva})$, and the computed compliance ratio rat, the procedure verifies the correctness of the evaluation and finalizes the contract settlement accordingly.

1. Confirm that the nullifier hash h is not in the nullifier table.
2. Check that the Merkle root rt is a valid root of the Merkle tree.
3. Invoke the on-chain verifier to check the proof: $\mathsf{valid} \leftarrow \mathsf{Verify}(\pi, srs)$.
4. If $\mathsf{valid} = 1$, release the locked funds as follows: the seller receives $v \times rat$ and the buyer receives $v \times (1 - rat)$.
5. If $\mathsf{valid} = 0$, revert the transaction.
6. Upon successful verification, update the state of the computable legal contract to COMPLETED.

4 An example of implementation: rental security deposit agreement

The *zk-agreements* protocol can be applied in domains where confidential contracts and privacy guarantees throughout the contract lifecycle are essential. We employed a template from the Accord Project[1, 5] to simulate real-world scenarios and provide a high-level description of how such agreements can be realized within our protocol. Specifically, we examine a rental security deposit agreement, demonstrating how tenants and landlords can ensure that funds are securely held in escrow and released only upon verifiable fulfillment of agreed conditions. Rental agreements often involve security deposits, which are intended to cover potential damages or unpaid rent at the end of a lease. However, disputes frequently arise between tenants and landlords regarding the fair return of the deposit. On the one hand, tenants may worry that the landlord will withhold the deposit unfairly. On the other hand, landlords face the risk that tenants may damage the property or refuse to pay outstanding obligations. Traditional arrangements rely heavily on trust or lengthy legal processes, which are costly and inefficient. Previous research has explored blockchain-based escrow solutions, but these often expose sensitive details of the agreement on chain. Our protocol offers a solution to this dilemma, ensuring the privacy of the parties while guaranteeing that the deposit is securely held and fairly released.

Commit to an agreement. The tenant constructs a computable legal contract that specifies the deposit amount and the conditions for its return. Using our protocol, the sensitive details of the agreement remain private, ensuring that neither the specific rental terms nor the tenant's identity are revealed on chain. Once the tenant commits to this computable legal contract, the commitment and corresponding deposit are securely submitted to a Merkle tree[26] on the blockchain, with the full contract details accessible only to the contracting parties.

Computable legal contract evaluation in TEE. At the end of the rental period, the landlord submits evidence of the property's condition (e.g., a digital inspection report). Within the TEE, a computable legal contract evaluation code is implemented to securely assess the submitted evidence against the terms of the rental contract. This code executes in an isolated and secure environment, ensuring that the evaluation is performed correctly without leaking sensitive information to external parties. The TEE guarantees that both the evaluation process and the evidence remain confidential, protecting the privacy of both tenant and landlord.

Automated deposit release. Once the evaluation is complete, the TEE will trigger a smart contract transaction to release the security deposit according to the result. If the property is in acceptable condition, the deposit is automatically

returned to the tenant. If damages or disputes are verified, the funds may be partially or fully released to the landlord. This "evaluate first, pay later" design ensures that the deposit is allocated fairly, while protecting the privacy of the parties and avoiding unnecessary legal conflict.

An example. To illustrate how our protocol operates in practice, we provide a detailed implementation of the rental security deposit agreement described in Appendix B. The full step-by-step specification includes the computable legal contract, TEE evaluation logic, and on-chain proof verification. This example concretely demonstrates how tenants and landlords can privately commit to an agreement, verify contract outcomes within a TEE, and enforce automated deposit release through zero-knowledge proofs and smart contracts.

5 Security analysis and benchmarks

5.1 Security analysis and properties

We analyze the security of *zk-agreements* in the standard cryptographic game-based framework. The protocol achieves privacy, integrity, and non-repudiation guarantees, assuming the hardness of the underlying cryptographic primitives, together with the zero-knowledge and soundness properties of PLONK [16] and the confidentiality guarantees provided by the TEE.

Privacy Preservation We formalize confidentiality of agreement contents via an indistinguishability game. Let $\mathcal{A}$ be a probabilistic polynomial-time adversary. $\mathcal{A}$ chooses two candidate agreements $\mathsf{clc}_0, \mathsf{clc}_1$. The challenger samples $b \xleftarrow{\$} \{0,1\}$ and randomness r, then computes

$$\mathsf{clc}_{\mathsf{comm}} \leftarrow \mathsf{Commit}(\mathsf{clc}_b; r).$$

The challenger returns $\mathsf{clc}_{\mathsf{comm}}$ together with access to public verification oracles, after which $\mathcal{A}$ outputs a guess b'. The adversary's advantage in this game is defined as

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Priv}}(\lambda) = \left| \Pr[b' = b] - \tfrac{1}{2} \right|.$$

We require that $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Priv}}(\lambda) \leq \mathrm{negl}(\lambda)$ for all PPT adversaries $\mathcal{A}$, where λ is the security parameter. This guarantee follows from three assumptions: (i) the hiding property of the commitment scheme prevents recovery of clc_b, (ii) the zero-knowledge property of PLONK ensures the existence of a simulator that produces proofs indistinguishable from real ones without knowledge of the witness, and (iii) TEE isolation prevents leakage of intermediate computation states. Hence, any adversary distinguishing with non-negligible probability must break either commitment hiding, zero-knowledgeness of PLONK, or TEE confidentiality.

Soundness If at least one contracting party is honest and the TEE operates correctly, then *zk-agreements* outputs the correct contract evaluation with probability at least $1 - 2^{-\lambda}$.
Proof Sketch. Soundness follows from three guarantees: (i) remote attestation ensures that only authenticated evaluation code executes inside the TEE, preventing malicious code injection; (ii) the soundness property of PLONK guarantees that no adversary can produce a valid proof for a false predicate except with negligible probability; and (iii) blockchain immutability ensures that once a result is verified, it cannot be modified retroactively. Therefore, any adversary violating this theorem can be transformed into one that breaks either TEE attestation, PLONK soundness, or blockchain immutability, each assumed infeasible.

Non-Repudiation We require that once a contract commitment has been signed by all parties and published on-chain, no party can later deny its participation except with negligible probability. Formally, let $\mathsf{clc}_{\mathsf{comm}} = \mathsf{Hash}(k \parallel pd)$ denote the contract commitment, where k is a nullifier and pd are the agreement parameters. Each party signs $\mathsf{clc}_{\mathsf{comm}}$ using a secure digital signature scheme, and the tuple $(\mathsf{clc}_{\mathsf{comm}}, \{\sigma_i\}_{i\in\mathcal{P}})$ is recorded on the blockchain. If the signature scheme is existentially unforgeable under chosen-message attacks (EUF-CMA) and the blockchain ledger is immutable, then any PPT adversary $\mathcal{A}$ succeeds in repudiating an honest party's commitment with probability at most $\mathrm{negl}(\lambda)$.
Proof Sketch. Digital signatures bind each participant to the value $\mathsf{clc}_{\mathsf{comm}}$, ensuring cryptographic accountability. Given signature unforgeability, no party can deny participation without invalidating its own signature. Blockchain immutability guarantees that recorded commitments cannot be deleted or altered after publication, while the nullifier mechanism enforces uniqueness of executions and prevents double-spending. Thus, any adversary that successfully repudiates must either forge a digital signature or alter the blockchain state, both assumed infeasible.

TEE-Correctness and Confidentiality We treat the TEE as an ideal functionality $\mathcal{F}_{\mathsf{TEE}}$ that guarantees two properties: (i) confidentiality, in that only the intended public output y and an attestation certificate are revealed, and (ii) correctness, in that evaluation is faithfully executed according to the prescribed contract logic. Remote attestation assures parties that the expected code is running inside a genuine enclave, while the confidentiality property ensures that no additional information about the private agreement state is leaked.

In conclusion, *zk-agreements* achieves (i) agreement confidentiality, (ii) integrity and soundness of contract evaluation, and (iii) non-repudiation of commitments. These guarantees are formally reducible to the hiding and binding properties of commitments, the zero-knowledgeness and soundness of PLONK, the EUF-CMA security of signatures, and the attestation and confidentiality properties of the TEE. Together, these results establish the protocol's suitability for deployment in privacy-critical applications.

5.2 Benchmarks: zk-circuits

Here, we evaluate circuit efficiency by benchmarking the runtime of both zk circuits integrated into our system. All benchmarks were conducted on an Apple M2 Pro laptop with 10-core CPUs at 3.5 GHz and 16 GB of RAM.

Our *zk-agreements* implementation consists of two main circuits: the commit circuit and the evaluation circuit. Both circuits are written in `circom` [7], a domain-specific language for zk-SNARK circuit design. For the proof workflow, we leverage `snarkjs` [22] to handle compilation, setup, proving, and verification. We instantiate our circuits on the BN254 elliptic curve [6], a pairing-friendly curve widely used in cryptographic operations. As our zero-knowledge proof framework, we adopt PLONK [16], which is known for its efficiency and flexibility in constructing zk-SNARKs[9, 16, 20, 21]. Table 2 presents the benchmark results, and Table 3 lists the configuration parameters of both the commit and evaluation circuits.

Table 2: Benchmark results for commitment and evaluation circuits.

Operation	**Commitment (ms)**	**Evaluation (ms)**
Compilation	520	110
Witness Generation	70	30
Setup	260	670
Proving	260	2,180
Verification	270	260

Table 3: Circuit parameters for commitment and evaluation circuits.

Metric	**Commitment**	**Evaluation**
Wires	1,870	1,781
Constraints	1,357	1,499
Private Inputs	512	277
Public Inputs	0	5
Labels	24,130	2,413
Outputs	2	0

6 Conclusion

We proposed the *zk-agreements* protocol which introduces a paradigm shift in establishing trust between parties engaging in confidential agreements. Using zero-knowledge proofs and TEE, this proposed solution ensures the privacy and

integrity of sensitive information while providing a robust framework for dispute resolution and compliance verification. As our digital landscape continues to evolve, the need for trust and privacy in agreements becomes increasingly critical. Our protocol offers a promising solution to address these challenges, providing a secure and efficient means of establishing trust while maintaining the confidentiality of sensitive information. By embracing cryptographic trust, this protocol has the potential to revolutionize the way agreements are conducted, paving the way for more transparent, fair, and dispute-free interactions between parties.

References

1. Accord Project: Smart legal contracts are recognised as being legally enforceable in england and wales. https://accordproject.org/news/smart-legal-contracts-are-recognised-as-being-legally-enforceable-in-england-and-wales/ (2021), accessed: 2024-08-28
2. Accord Project: Cicero template library. https://github.com/accordproject/cicero-template-library (2023), accessed: 2024-09-01
3. Accord Project: Concerto: A data modeling language for smart legal contracts. https://github.com/accordproject/concerto (2023), accessed: 2024-09-01
4. Accord Project: Ergo: A domain-specific language for smart legal contracts. https://github.com/accordproject/ergo (2023), accessed: 2024-09-01
5. Accord Project: Accord project. https://github.com/accordproject (2024), accessed: 2024-09-01
6. Barreto, P.S.L.M., Naehrig, M.: Pairing-friendly elliptic curves of prime order. In: Selected Areas in Cryptography. pp. 319–331. Springer (2005)
7. Bellés-Muñoz, M., Isabel, M., Muñoz-Tapia, J.L., Rubio, A., Baylina, J.: Circom: A circuit description language for building zero-knowledge applications. IEEE Transactions on Dependable and Secure Computing **20**(6), 4733–4751 (2023). https://doi.org/10.1109/TDSC.2022.3232813
8. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: Decentralized anonymous payments from bitcoin. In: IEEE Symposium on Security and Privacy. pp. 459–474. IEEE (2014)
9. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: Snarks for c: Verifying program executions succinctly and in zero knowledge. In: CRYPTO 2013. pp. 90–108. Springer (2013). https://doi.org/10.1007/978-3-642-40084-1_6
10. Boneh, D., et al.: Privacy-protecting regulatory solutions using zero-knowledge proofs. Tech. rep., Stanford Applied Cryptography Group, a16z Crypto (2022), https://a16zcrypto.com/posts/article/privacy-protecting-regulatory-solutions-using-zero-knowledge-proofs-full-paper/, full paper available via a16z Crypto
11. Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., Wu, H.: Zexe: Enabling decentralized private computation. In: IEEE Symposium on Security and Privacy. pp. 947–964. IEEE (2020)
12. Bünz, B., Agrawal, S., Zamani, M., Boneh, D.: Zether: Towards privacy in a smart contract world. In: International Conference on Financial Cryptography and Data Security. pp. 423–443. Springer (2020)
13. Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform (2014), https://ethereum.org/en/whitepaper/
14. Chen, E., Roche, N., Tseng, Y.H., Hernandez, W., Shangguan, J., Moore, A.: Conversion of legal agreements into smart legal contracts using nlp. In: Companion Proceedings of the ACM Web Conference 2023. pp. 1112–1118. ACM, New York, NY, USA (2023). https://doi.org/10.1145/3543873.3587554, https://doi.org/10.1145/3543873.3587554
15. Costan, V., Devadas, S.: Intel sgx explained. In: Cryptology ePrint Archive. No. 2016/086 (2016), https://eprint.iacr.org/2016/086.pdf
16. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019)
17. Goldreich, O.: Secure multi-party computation. Manuscript **78**(110), 1–108 (1998)

18. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: STOC. pp. 218–229. ACM (1987)
19. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on Computing **18**(1), 186–208 (1989). https://doi.org/10.1137/0218012
20. Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: ASIACRYPT 2010. pp. 321–340. Springer (2010)
21. Groth, J.: On the size of pairing-based non-interactive arguments. In: EUROCRYPT 2016. pp. 305–326. Springer (2016)
22. Iden3: snarkjs: Javascript implementation of zero-knowledge proof systems. https://github.com/iden3/snarkjs (2018), accessed: 2024-09-01
23. Kosba, A.E., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: IEEE Symposium on Security and Privacy. pp. 839–858. IEEE Computer Society (2016). https://doi.org/10.1109/SP.2016.55
24. Koulu, R.: Blockchains and online dispute resolution: smart contracts as an alternative to enforcement. SCRIPTed **13**, 40 (2016)
25. Ma, A.: Blockchain-enabled smart legal contracts. In: Blockchain Applications: Transforming Industries, Enhancing Security, and Addressing Ethical Considerations. IntechOpen (2023). https://doi.org/10.5772/intechopen.109041, https://doi.org/10.5772/intechopen.109041
26. Merkle, R.C.: Protocols for public key cryptosystems. IEEE Symposium on Security and Privacy pp. 122–134 (1980)
27. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous distributed e-cash from bitcoin. In: IEEE Symposium on Security and Privacy. pp. 397–411. IEEE (2013)
28. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008), accessed: 2024-09-01
29. Pertsev, A., Semenov, R., Storm, R.: Tornado cash privacy solution version 1.4. Tornado Cash Documentation **1**, 7 (2019)
30. Roberts, M., Selman, D., Lease, D., Tavarez, T., Brooke, T., Halford, M., Roche, N.: An introduction to computable contracts. Accord Project, Stanford, CA, 1st edn. (Apr 2024), https://www.lulu.com/shop/matt-roberts-and-dan-selman/an-introduction-to-computable-contracts/paperback/product-wp96d4.html, foreword by Dr. Megan Ma, Stanford CodeX
31. Szabo, N.: Formalizing and securing relationships on public networks. First Monday (1997)
32. Tapscott, D., Tapscott, A.: Blockchain revolution: How the technology behind bitcoin is changing money, business, and the world. Penguin (2016)
33. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper (2014), available at https://ethereum.github.io/yellowpaper/paper.pdf
34. Wright, A., Roon, D.: Openlaw: Real-world smart legal contracts on ethereum. OpenLaw Project by ConsenSys (2017)
35. Yao, A.C.: Protocols for secure computations. In: FOCS. pp. 160–164. IEEE (1982)

A The Accord Project: Computable Legal Contracts

The Accord Project[1] is an open-source initiative that provides a comprehensive toolkit for creating and executing computable legal contracts. Founded in 2017 as a collaboration between legal and technology professionals, the project aims to bridge the gap between traditional legal agreements and smart contract technology by providing standardized tools for contract digitization and automation.

A.1 Architecture Overview

The Accord Project ecosystem consists of several interconnected components:

- Ergo [4]: A domain-specific language for encoding legal contract logic
- Concerto [3]: A modeling language for defining contract data structures
- Cicero [2]: A template engine for creating reusable contract templates

For the *zk-agreements* protocol, we primarily utilize Ergo and Concerto to transform traditional legal agreements into machine-executable formats while preserving legal semantics. Figure 1 illustrates a simplified example of a contract representation, showing how parties, terms, and obligations are structured in a clear, modular form. This abstraction serves as the input to the Accord Project toolchain, where Ergo specifies executable clauses and Concerto defines the underlying data model.

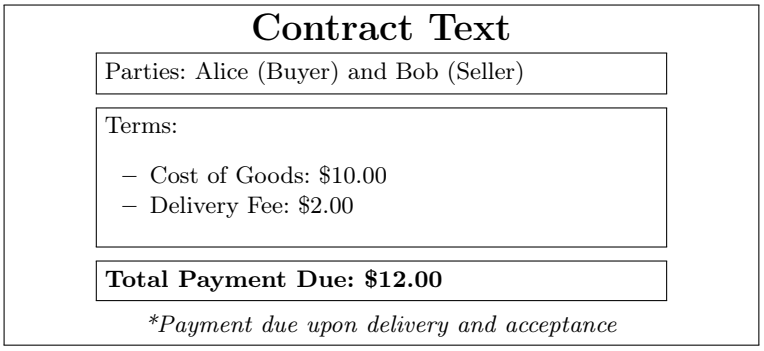

Fig. 1: `PaymentUponDelivery` contract

A.2 Ergo: Domain-Specific Language for Legal Logic

Ergo is a domain-specific language developed to express the operational logic of legal contracts in a precise and executable form. Its design bridges the gap between natural legal language and formal computation, allowing contracts to

[1] https://accordproject.org/

```
contract PaymentUponDelivery over PaymentUponDeliveryContract {
  clause delivered(request : DeliveryAcceptedRequest) : DeliveryAcceptedResponse
    emits PaymentObligation {
    enforce (contract.costOfGoods.currencyCode = contract.deliveryFee.currencyCode);

    let totalAmount = MonetaryAmount{
      doubleValue: contract.costOfGoods.doubleValue +
                   contract.deliveryFee.doubleValue,
      currencyCode: contract.costOfGoods.currencyCode
    };

    emit PaymentObligation{
      contract: contract,
      promisor: some(contract.buyer),
      promisee: some(contract.seller),
      deadline: none,
      amount: totalAmount,
      description: toString(contract.buyer) ++
        " should pay cost of goods and delivery fee to " ++
        toString(contract.seller)
    };

    return DeliveryAcceptedResponse{}
  }
}
```

Fig. 2: `PaymentUponDelivery` contract emitting a payment obligation.

be written in a style accessible to legal professionals while remaining rigorous enough for automated execution. In practice, Ergo enables contracts to specify obligations, conditions, and payments that can be enforced automatically within a digital environment.

Consider the sample contract from Fig 1 between Alice (Buyer) and Bob (Seller), where the agreement specifies a cost of goods of $10.00 and a delivery fee of $2.00, with total payment of $12.00 due upon delivery and acceptance. This legal clause can be encoded in Ergo to generate a corresponding payment obligation once the goods have been delivered and accepted by the buyer. Fig 2 illustrates this encoding in Ergo.

A.3 Concerto: Data Modeling Language

Concerto is a lightweight modeling language designed to define the structure and semantics of data used in legal contracts. It provides a clear way to represent contract parameters, parties, and assets through typed data models that

```
{
  "extends": "Contract",
  "properties": {
    "buyer": { "type": "Party" },
    "seller": { "type": "Party" },
    "costOfGoods": { "type": "MonetaryAmount" },
    "deliveryFee": { "type": "MonetaryAmount" }
  }
}
```

Fig. 3: Concerto data model for the `PaymentUponDelivery` contract.

can be validated and shared across systems. By separating data representation from contract logic, Concerto ensures that agreements have a consistent and interoperable foundation for automated execution.

Consider again the sample contract from Fig 1 between Alice (Buyer) and Bob (Seller), where the agreement specifies a cost of goods of $10.00 and a delivery fee of $2.00, with total payment of $12.00 due upon delivery and acceptance. This agreement can be represented in Concerto by defining a structured data model that captures the roles of buyer and seller, as well as the monetary amounts involved. Fig 3 illustrates this data model in Concerto.

B Detailed Explanation of the Security Deposit Agreement Use Case

Given this sample contract of the Rental Security Deposit Agreement, we can now convert this textual contract into *clc*. In Algorithm 1, we use Concerto [3] to create a machine-readable data model that extracts the computable data from the original contract. These essential data include the parties involved, the deposit value, the timing of the rental period, and the artifacts related to the property inspection and potential dispute. This step clearly defines the variables and assets within the contract. Next, we use Ergo [4] to encode the contract logic. This includes translating the textual clauses into executable functions. Ergo also manages state transitions, allowing the contract to move between different states on the basis of logical conditions. Effective state management, tracked through the variable *ContractState*, is crucial for executing contracts, such as automating the refund once the tenant approves, or enforcing an arbitrated split in case of a dispute. By automating contract execution, deterministic releases replace the need for manual oversight, reducing ambiguities, and ensuring consistent state transitions. In summary, converting a traditional legal agreement into a computable legal contract using tools like Concerto and Ergo creates self-executing, self-enforcing contracts. This modern approach enhances transparency, security, and efficiency, reducing ambiguities, minimizing disputes, and streamlining contract execution, significantly benefiting digital transactions.

Rental Security Deposit Agreement

This agreement is for the holding and release of a security deposit belonging to **tenant**. The agreement shall start at **startDate** and will remain in effect until the rental period ends.

1. Payment

A deposit of 2 ETH will be stored in a smart contract. The deposit will be released according to the conditions specified below.

2. Service

- **tenant** transfers 2 ETH into the smart contract as a refundable security deposit.
- At the end of the rental period, **landlord** must submit proof of property condition (e.g., digital inspection report hash).
- **tenant** may either approve the proof or dispute it.
- If approved, the smart contract will release the 2 ETH back to **tenant**.
- If a dispute occurs, the deposit will remain locked until arbitration provides a signed decision.

3. Obligations

- The smart contract guarantees secure custody of the 2 ETH deposit.
- **landlord** agrees to provide honest documentation of property condition.
- **tenant** agrees to fairly approve or dispute the landlord's claim.

4 Signatories

tenant	**landlord**	**startDate**
Signature	Signature	Date

Algorithm 1 ComputableLegalContractCompilation

Input: textual agreement a
Output: computable legal contract *clc*

1. Generate computable legal contract data using Concerto: $\text{clc}_{\text{data}} \leftarrow \texttt{Concerto}(a)$

Data Model:

ContractData

tenant	: String
landlord	: String
propertyId	: String
startDate	: DateTime
rentalEndDate	: DateTime
depositValue	: Integer

ContractState

lifecycle	: { INIT, EXECUTION, EVALUATION, COMPLETED }
phase	: {NONE,AWAITING_INSPECTION,AWAITING_TENANT_DECISION, APPROVED,DISPUTED}

ExternalData

inspectionReportHash	: String
tenantDecision	: {APPROVE,DISPUTE}
arbDecisionSig	: Bytes
arbTenantShare	: Integer
arbLandlordShare	: Integer

2. Generate computable legal contract logic using Ergo: $\text{clc}_{\text{logic}} \leftarrow \texttt{Ergo}(a)$

Logical Functions:
function InitializeContractState
if lifecycle = INIT **then** lifecycle ← EXECUTION; phase ← AWAITING_INSPECTION
end if
end function
function SubmitInspection(*reportHash*)
if lifecycle = EXECUTION **and** phase = AWAITING_INSPECTION **and** ValidHash(*reportHash*) **then** inspectionReportHash ← *reportHash*; lifecycle ← EVALUATION; phase ← AWAITING_TENANT_DECISION
end if
end function
function TenantDecision(*decision*)
if lifecycle = EVALUATION **and** phase = AWAITING_TENANT_DECISION **then**
if *decision* = APPROVE **then** phase ← APPROVED; ReleaseDeposit(tenant, depositValue); lifecycle ← COMPLETED
else if *decision* = DISPUTE **then** phase ← DISPUTED
end if
end if
end function
function ResolveDispute(arbTenantShare, arbLandlordShare, arbDecisionSig)
if lifecycle = EVALUATION **and** phase = DISPUTED **and** ValidArbSig(arbDecisionSig) **then** SplitDeposit(arbTenantShare, arbLandlordShare); lifecycle ← COMPLETED
end if
end function
return *clc* := (clc_{data}, $\text{clc}_{\text{logic}}$)

After compilation, we require the contracting parties to sign the computable legal contract clc through digital signatures. The contract is first hashed into a canonical digest, and then each party (buyer and seller) produces a signature over this digest using their respective secret keys. The resulting tuple, which contains the contract and both signatures, forms the signed contract clc_{signed}. This ensures that the buyer and seller are jointly committed to the same agreement before it proceeds to subsequent commitment and execution steps.

Algorithm 2 SignCLC

Input: buyer secret key sk_{buy}, seller secret key sk_{sel}, contract clc
Output: signed contract clc_{signed}
$c \leftarrow \mathsf{Digest}(clc)$
$\sigma_{buy} \leftarrow \mathsf{Sign}(sk_{buy}, c)$
$\sigma_{sel} \leftarrow \mathsf{Sign}(sk_{sel}, c)$
$clc_{signed} \leftarrow (clc,\ \sigma_{buy},\ \sigma_{sel})$
return clc_{signed}

Once the contract has been jointly signed, the next step is to generate the commitment for the signed contract clc_{signed}. The `CommitCLC` procedure is realized as a Circom circuit [7]. Given a signed computable legal contract clc_{signed} and a nullifier k, it derives a digest pd, computes a registered commitment clc_{comm}, and outputs the pair (h, clc_{comm}).

Algorithm 3 CommitCLC

Input: signed contract clc_{signed}, nullifier k
Output: nullifier hash h, contract commitment clc_{comm}
$h \leftarrow \mathsf{Hash}(k)$
$pd \leftarrow \mathsf{Digest}(clc_{signed})$
$clc_{comm} \leftarrow \mathsf{Hash}(k \parallel pd)$
return (h, clc_{comm})

Later, a transaction containing the contract commitment clc_{comm} and its value $v = 2$ ETH is submitted to the blockchain. Once the transaction is confirmed, the commitment is appended to the smart contract's Merkle tree of agreements, and the contract state is updated to EXECUTION. This step establishes an immutable on-chain reference to the agreement, while the emitted event provides Merkle proof data that can be stored in the TEE for use during later evaluation.

Algorithm 4 SubmitCommitment

Input: contract commitment clc_{comm}, contract value v
Output: transaction tx
$tx \leftarrow (clc_{comm}, v)$
broadcast tx to the blockchain
insert clc_{comm} into the Merkle tree of registered agreements
update contract state $\leftarrow$ EXECUTION
return tx

Following the on-chain commitment, the next phase is contract evaluation, the signed contract clc_{signed} is evaluated inside the TEE together with the landlord's inspection hash, the tenant's decision, and, in case of dispute, an arbitrator's signed allocation. If the tenant approves, the evaluation outputs a full release of the deposit. If the tenant disputes, the TEE verifies the arbitrator's signature and checks that the allocation splits sum to v. A valid dispute produces a payment ratio $rat = \mathsf{tenantShare}/v$, while any invalid input causes the evaluation to reject. This step ensures that the release of funds follows the contract logic while keeping the inspection details confidential inside the TEE.

Algorithm 5 ExecuteEvaluation

Input: signed contract clc_{signed},
inspection hash h_{insp}, tenant decision dec, tenant address $addr_T$,
optional arbitrator signature σ_{arb}, shares $(\mathsf{tenantShare}, \mathsf{landlordShare})$
Output: compliance ratio rat, result $\in \{\mathsf{success}, \mathsf{reject}\}$
install evaluation code embedded in clc_{signed} inside the TEE
transmit $(h_{\text{insp}}, dec, addr_T, \sigma_{\text{arb}}, \mathsf{tenantShare}, \mathsf{landlordShare})$ securely to the TEE
if $dec = \mathsf{APPROVE}$ **then**
    $rat \leftarrow 1$;
    return $(\mathsf{success}, rat)$
else if $dec = \mathsf{DISPUTE}$ **then**
    if $\textsc{Verify}(\sigma_{\text{arb}}, (h_{\text{insp}}, \mathsf{tenantShare}, \mathsf{landlordShare})) = \text{false}$ **then**
        return $(\mathsf{reject}, 0)$
    end if
    enforce $\mathsf{tenantShare} + \mathsf{landlordShare} = v$
    $rat \leftarrow \mathsf{tenantShare}/v$;
    return $(\mathsf{success}, rat)$
end if

After obtaining the evaluation result, we proceed to generate a PLONK[16] zero-knowledge proof of knowledge for the secret nullifier k and the associated contract commitment. The TEE produces this proof to attest that the evaluation code was executed correctly and that the computations were recorded faithfully. By leveraging the Merkle proof stored in the TEE, the circuit verifies commitments and records while preserving privacy. The resulting PLONK[16]

proof guarantees the integrity of the computation and validates the involvement of the correct parties and parameters without revealing sensitive data, thereby enabling secure and private verification of the transaction.

Algorithm 6 GenerateProof

Input: structured reference string srs, nullifier k, signed contract clc_{signed}, Merkle root rt, Merkle proof path hp, direction bits hd, public keys $(pk_{buy}, pk_{sel}, pk_{eva})$, compliance ratio rat
Output: zero-knowledge proof π
$h \leftarrow \mathsf{Hash}(k)$
$pd \leftarrow \mathsf{Digest}(clc_{signed})$
$clc_{comm} \leftarrow \mathsf{Hash}(k \parallel pd)$
$rt_{new} \leftarrow \mathsf{MerkleProof}(clc_{comm}, hp, hd)$
assert $rt_{new} = rt$
$\pi \leftarrow \mathsf{Proof}(srs,\ h,\ clc_{comm},\ pk_{buy},\ pk_{sel},\ pk_{eva},\ rat)$
return π

The last step explains how a zero-knowledge proof, created inside the TEE, can be verified with the use of a PLONK [16] verifier in a smart contract. Once a proof is submitted to the blockchain, the smart contract extracts the public signals from the inputs, verifies that this nullifier hash has not been spent before, and checks that the Merkle proofs are also valid. The PLONK [16] verifier is then called to verify the proof, ensuring the integrity of the transaction based on public input and commitment data. Upon successful verification, the nullifier hash is marked as spent to prevent double spending, and the commitment value is transferred according to the approved ratio between the parties (full refund if approved, or arbitrated split otherwise). Finally, after the transaction is successfully executed and confirmed by the blockchain, the state of the computable legal contract is updated to COMPLETE.

Algorithm 7 VerifyProof (Security Deposit)

Input: zero-knowledge proof π, structured reference string srs, nullifier hash h, Merkle root rt, public keys $(pk_{buy}, pk_{sel}, pk_{eva})$, compliance ratio rat, contract value v
Output: transaction tx
ensure h is not in the nullifier table
ensure rt is a valid root in the Merkle tree
$\mathsf{valid} \leftarrow \mathsf{Verify}(\pi, srs,\ h,\ rt,\ pk_{buy}, pk_{sel}, pk_{eva}, rat, v)$
if $\mathsf{valid} = 1$ **then**
    mark h as spent
    transfer $v \cdot rat$ to seller, $v \cdot (1 - rat)$ to buyer
    update contract state $\leftarrow$ COMPLETED
    return tx
else
    revert transaction
end if

By a detailed explanation of this example, we show how parties involved in a confidential agreement can run seven algorithms to fulfill an agreement in a trustless manner. That is, these two parties needless to trust each other not to cheat or rely on other dispute resolution mechanisms if there is a disagreement; instead, correctness follows from commitments, authenticated artifacts, zero-knowledge proofs, and on-chain verification.