LedgerHedger: Gas Reservation for Smart Contract Security

Itay Tsabary[1], Alex Manuskin[2], Roi Bar-Zur[1], and Ittay Eyal[1]

[1] Technion, Israel
itaytsabary@gmail.com
roi.bar-zur@campus.technion.ac.il
ittay@technion.ac.il
[2] alex@manuskin.org

Abstract. In *smart contract* blockchain platforms such as Ethereum, users interact with the system by issuing *transactions*. System operators called *miners* or *validators* add those transactions to the blockchain. Users attach to each transaction a fee, which is collected by the miner who placed it in the blockchain. Miners naturally prioritize better-paying transactions. This process creates a volatile fee market due to limited throughput and fluctuating demand. The fee required to place a transaction in the future is unknown; yet, ensuring timely transaction confirmation is critical for securing smart contracts that represent billions of dollars and underpin prominent blockchain scaling solutions.

We present LedgerHedger, a novel mechanism that guarantees the confirmation of a transaction within a specified time frame. Due to the absence of external enforcement in decentralized systems, LedgerHedger uses incentives. Its core is a *hedging* agreement between a transaction issuer and a second party, possibly a miner. The issuing party pays for the transaction upfront while the second party commits to paying any necessary fees when the transaction is issued in the future, even if they exceed the original payment.

LedgerHedger gives rise to a strategic game, where the issuing party deposits the transaction payment and the committing party deposits a collateral. During the target time frame, the latter is required to confirm the transaction if it exists, or they have the option to withdraw the payment and the collateral if the transaction is not presented.

We demonstrate that for a broad range of parameters, a subgame perfect equilibrium exists where both parties are incentivized to act as desired, thereby guaranteeing transaction confirmation. We implement LedgerHedger and deploy it on an Ethereum test network, showcasing its efficacy and minor overhead.

Keywords: Blockchains, Cryptocurrency, Smart Contracts, Hedging, Gas Price

1 Introduction

Decentralized smart contract platforms like Ethereum [1, 2], Solana [3], Avalanche [4, 5], and Binance Smart Chain [6] have reached market caps of hundreds of billions of dollars [7]. These platforms facilitate *transactions* of virtual cryptocurrency tokens and allow users to interact via stateful programs called *smart contracts*. They support a diverse range of interactions, from token transfers to executing complex business logic. For these transactions to be finalized, they require *confirmation* by system operators known as *miners* or *validators*. Confirmed transactions are recorded on a decentralized ledger, the *blockchain*.

The blockchain is constrained in terms of its transaction capacity [2, 8, 9]. Transaction *issuers*, therefore, assign *fees* to their transactions, paid to the confirming miner. Naturally, miners prioritize transactions that offer higher fees, leaving behind those with lower bids. Due to demand fluctuation [10–16], the required confirmation fee at a future time frame is unknown and volatile [17,18].

This unpredictability is not just an inconvenience; it's a critical security vulnerability. A growing range of smart contract applications, collectively worth billions of dollars [19, 20], critically rely on timely transaction confirmations.

Among these applications are financial instruments such as *vaults* [21–24], *atomic swaps* [25–30] and *contingent payments* [31–35], and a prominent blockchain scaling solution, *off-chain channels* [36–43]. These are based on *Hash Time Locked Contracts* (HTLCs) for their operation [44,45]. Conducted between two parties, an HTLC pays the first party for providing a suitable hash preimage (hash lock), or the other party after a timeout elapses (time lock). A delay in confirmation can lead to an elapsed timeout, resulting in irreversible unjust token transfers.

Other blockchain scaling solutions, *optimistic* and *zero-knowledge roll-ups* [46–55] also heavily depend on timely confirmations [56, 57]. For example, optimistic roll-ups work under the presumption that on-chain summaries are correct, but any delay during their limited dispute period risks token theft. Zero-knowledge roll-ups, on the other hand, demand swift on-chain validations of correctness proofs, with delays potentially halting system progress.

Given the critical nature of timely transaction confirmations and the volatility of fees, there's a clear need for a reliable mechanism to ensure timely transaction confirmation without the risk of unpredictable costs. In response, we present LedgerHedger, a novel mechanism for blockchain reservation. LedgerHedger is a smart contract that facilitates an agreement between a *Buyer* (transaction issuer) and a *Seller* (naturally, but not necessarily, a miner). This agreement ensures that *Seller* will incorporate a transaction issued by *Buyer* in a future block for a predetermined fee. This arrangement protects both parties from unexpected fluctuations in fee rates.

To reason about LedgerHedger, we use a model (§2) with an append-only log of transactions called the blockchain. Miners batch transactions in *blocks* and append the blocks to the blockchain; this confirms the added transactions. Transactions consume system resources, measured in *gas*. Each block has a *gas-price*, a tokens-per-gas-unit metric indicating the required transaction fee for confirmation based on a transaction's gas consumption. *Seller* has a future gas allocation, and *Buyer* needs to have a transaction confirmed during that period. Our model employs a common price variation framework for predicting future *gas-price*, backed by data from Ethereum's history. *Buyer* and *Seller* are *risk-averse* [58–64], that is, their utilities are concave [61, 65, 66, 66–70] functions of their token holdings, leading them to prefer stable and predictable outcomes over uncertain ones, even if they more profitable in expectation.

The mechanics of LedgerHedger (§3) revolve around the concept of *hedging* [67, 71, 72]. In conventional markets, external authorities, such as courts, ensure the enforcement of hedging contracts. However, in the decentralized world of cryptocurrencies, miners hold the exclusive power to decide on transaction confirmations. Faced with clear incentives as potential contract participants, the inherent power asymmetry invalidates solutions that rely on parties' altruistic behavior [73]. LedgerHedger is designed to incentivize both parties to act as desired. For that, *Seller* deposits a collateral as part of the contract initiation [44, 74, 75], which is later returned only if she abides by the contract, and confirms *Buyer*'s transaction. LedgerHedger also protects *Seller*, ensuring she is paid even if *Buyer* misbehaves, and does not supply *Seller* with a transaction to confirm. The contract operates in two distinct phases: an initiation phase and an execution phase.

We prove LedgerHedger is *incentive compatible* (§4), meaning that both parties are incentivized to act as desired. To that end, we model the incentives of *Buyer* and *Seller* as a game with two phases. In the first phase, both parties decide whether to commit to a LedgerHedger contract or to proceed without any hedging. Upon reaching the target time frame, if a contract is in place, the two parties can interact according to its terms. Otherwise, they operate at the prevailing market *gas-price*. At the end of the game, strategies chosen by the participants, coupled with the stochastic market *gas-price*, determine the final token count for each party, and consequently their utilities.

We analyze the game using the *subgame-perfect-equilibrium (SPE)* solution concept, suitable for its dynamic, turn-based nature. An SPE comprises a strategy of *Buyer* and a strategy of *Seller* such that both cannot increase their utility by deviating at any stage of the game. Through our analysis, we ascertain that initiating and adhering to the contract is a mutually beneficial strategy for both parties in a variety of practical conditions.

We implement LedgerHedger as an Ethereum smart contract and deploy it on a test network (Appendix A). LedgerHedger's overhead is low – three orders of magnitude lower than the hedged gas for prevalent use cases. Moreover, we demonstrate LedgerHedger's efficacy (§5) through a sensitivity analysis with respect to contract parameters, *gas-price* distribution, and utility functions. We then identify concrete parameters under which LedgerHedger is viable.

Despite recent advances in addressing the issue of timely confirmation, existing solutions are not without their limitations (§6). Some, such as those that seek to estimate the required fee to allow swift confirmation [76–82], or to redesign the fee market for better predictability [83–85], do not address the inherent uncertainty of future fee fluctuations. Another approach, *congestion detection* [86], introduces dynamic timeouts that adjust once the blockchain is congested. This approach, however, only replaces safety violations with liveness violations.

An alternative method to hedge gas prices, involves buying and later selling *gas tokens*. Some gas tokens, which rely on external *oracles* to derive their price, are susceptible to adversarial manipulation [87]. Another type of gas token has relied on the *refund mechanism* of the Ethereum protocol [88–90]. However, these gas tokens were inefficient, and were broken as Ethereum evolved [91, 92].

Compared to existing solutions, LedgerHedger brings substantial benefits. It enables long-term reservations for transaction processing at predetermined fees, eliminating the need for fee estimation. The two-party interaction basis enhances its resilience to external manipulations, congestion, and protocol changes. Furthermore, it achieves this robustness without requiring an external oracle or modifications to the underlying blockchain, offering a self-contained and reliable solution.

In summary, our contributions (§7) are: (1) a *gas-price* fluctuation model, confirmed by Ethereum measurements; (2) LedgerHedger, the first mechanism for addressing the prevalent requirement of timely blockchain confirmation; (3) an analysis showing LedgerHedger's security and applicability for a wide range of parameters; and (4) an open-source implementation for Ethereum, deployed on a test network.

2 Model

To reason about blockchain reservation, we first describe a general model for an underlying blockchain-based cryptocurrency (§2.1). We then present the setup for a future transaction inclusion deal (§2.2), and the stochastic value of fees (§2.3). Finally, we describe the participants' utility functions (§2.4).

2.1 Cryptocurrency System

The blockchain system tracks internal cryptocurrency *tokens* that its *users* can *transact*. To apply their transactions, users broadcast them across a peer-to-peer network. A subset of users, called *miners*, batch transactions in data structures called *blocks*.

Miners add blocks to a global data structure, called the *blockchain*, forming an append-only list of blocks. Blocks have indexes matching their append order, and we denote the i'th block by b_i. A transaction is *confirmed* when it is included in the blockchain.

We follow the common assumption [1, 25, 26, 41, 42, 44, 74, 83, 93–95] that all miners create blocks according to the above description, and all published transactions and all created blocks are instantaneously available to all system users and miners.

The *system state* is the association of tokens to *smart contracts*, predicates that need to be satisfied in order to transact their associated tokens. Parties infer the state by sequentially parsing the blockchain. Only transactions that satisfy the contract predicates can be confirmed, and we disregard transactions that do not.

The smart contract predicates can verify that the transaction is digitally signed, for an *existentially unforgeable under chosen message attacks (EU-CMA)* [44,74,96–98] digital signature algorithm; that the transaction is included in a block numbered higher or lower than a parameter; that the transaction transfers a number of tokens; or a combination of the above. We say a user *owns* tokens if she is the only user that is able to satisfy the contract predicate.

Transactions are measured by their *gas* requirement – an internal measure of transaction resource consumption. Each operation in a transaction requires a certain amount of gas, and the total transaction gas is the sum of all operations' gas. When considering a transaction *tx*'s gas requirement, denoted by g_{tx}, we consider it with respect to the system state when it is confirmed.

Each block has a bound on the total gas of its transactions. Transactions offer tokens as a fee for the including miner. This fee is set by the transaction issuer, determining a non-negative tokens-per-gas ratio, which we denote by π_{tx} for transaction *tx*. When confirmed, transaction *tx* pays $g_{tx} \cdot \pi_{tx}$ tokens to the miner that included it in a block.

Miners choose which transactions to include in a block based on their offered π_{tx} values. We refer to the minimal required value to be included in a block by *gas-price*. For simplicity, we assume there are always sufficiently many transactions that offer *gas-price* to exactly fill a block [95, 99], and that any transaction offering at least *gas-price* is confirmed.

2.2 Future Transaction Setup

Consider two system participants, *Buyer* and *Seller*, with the following interests: *Buyer* requires g_{alloc} gas allocated to a transaction of her choice in future blocks; *Seller* has a gas allocation of g_{alloc} in such a suitable block, which she can sell for tokens.

We denote the transaction that *Buyer* wants to be included by $tx_{payload}$, and the relevant block interval for its inclusion by $[b_{start}, b_{end}]$. Note that the content of $tx_{payload}$ is not necessarily known up to b_{start}. We also denote the block interval in which *Buyer* wishes to assure the future allocation by $[b_{init}, b_{acc}]$ such that $init \leq acc < start \leq end$.

If *Seller* is a miner, a suitable choice of block interval will guarantee with overwhelming probability that *Seller* will obtain the necessary gas allocation regardless if block generators are chosen probabilistically, as in Ethereum, or deterministically, as in planned Central Bank Digital Currencies (CBDCs) [100, 101] (Appendix B). Furthermore, *Seller* does not have to be a miner, as she can simply purchase the necessary gas allocation when the time comes.

2.3 Gas Price

To reason about hedging, one requires a prediction of the commodity future price. We assume the future *gas-price* is drawn from some price distribution. We assume both *Buyer* and *Seller* have perfect knowledge of this distribution.

Previous work [77–82] provides *gas-price* predictions, but focuses exclusively on prediction for the *next* block. We are not aware of work modeling the *gas-price* for a further future (e.g., a week ahead), hence we assume it follows the prevalent *random-walk price model* [102–105]. According to this model, the *gas-price* follows a Gaussian random walk, where in each block it changes according to a random sample from a normal distribution $N\left(\mu, \sigma^2\right)$. It follows [106, 107] that the future *gas-price* change after n blocks is also drawn from a normal distribution with parameters $N\left(n \cdot \mu, n \cdot \sigma^2\right)$.

For simplicity, we assume the random walk is without a *drift*, meaning $\mu = 0$. We also assume that σ^2 is small [108], so in the short term the *gas-price* has a low variance.

We validate this model using the Kolmogorov–Smirnov [109] test on historical Ethereum gas prices over a month (Appendix C).

We slightly enhance the price model to neglect rare events. Specifically, the *gas-price* cannot be negative, as that would imply the miner pays users to transact, instead of the obvious option of leaving blocks empty; excessively high *gas-price* is also impossible, as that removes any incentive to transact and renders the system unusable.

In summary, denote by $\mathcal{F}$ the *gas-price* distribution in the target interval. $\mathcal{F}$ is a *truncated* normal distribution [110]; its mean value is the *gas-price* for block b_{init}; its lower tail is truncated such that the *gas-price* is non-negative, and we truncate the upper tail symmetrically with respect to the mean. Denote the *probability density function (PDF)* of $\mathcal{F}$ by $\mathcal{F}_{\text{pdf}}$.

Denote the *gas-price* for block b_{init} by π_{setup}. We assume that $[b_{init}, b_{acc}]$ is relatively short, and make the simplifying assumption that the *gas-price* for this entire interval is π_{setup}. Similarly, we assume that $[b_{start}, b_{end}]$ is relatively short, and denote the *gas-price* for this interval by $\pi_{exec} \sim \mathcal{F}$.

2.4 Wealth and Utility

The interaction with the contract concludes with each player having some number of tokens – their resultant *wealth*. We model the exogenous motivation of *Buyer* from having a transaction $tx_{payload}$ that consumes at least g_{alloc} gas confirmed during φ_{exec} as her receiving tokens from doing so, denoting their number by w^{exo}. We capture the player's happiness from having wealth using a *utility* function.

We denote the initially available tokens of *Buyer* and *Seller* by w^{init}_{Buyer} and w^{init}_{Seller}, respectively. Each player's resultant wealth therefore depends on these values, their paid and received transaction fees, and the values of π_{setup} and π_{exec}. A player's utility $U : W \to \mathbb{R}$ is a function describing happiness from eventually having W tokens, including the exogenous motivation w^{exo} for *Buyer*.

We assume both *Seller* and *Buyer* are *risk-averse* [62–66, 111–114], that is, they value the certainty of their resultant wealth. This implies that they might not prefer to maximize their expected wealth. For example, a risk-averse player might prefer getting 4 tokens with probability 1 over getting 10 token with probability 0.5, despite the latter higher expected value of 5. Risk aversion justifies

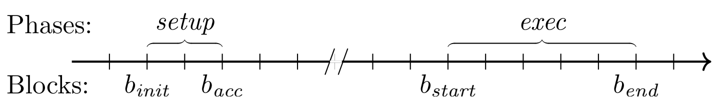


Fig. 1: LEDGERHEDGER interaction block ranges.

actions like individuals purchasing insurance [115, 116], or airlines hedging oil prices [117,118]. Risk and *ambiguity* [119,120] aversion also capture that players do not have perfect knowledge of $\mathcal{F}$.

The common practice [61, 65–70] to model risk aversion is using a utility function $U(W)$ with the following two properties: (1) $U(W)$ is strictly increasing in W, meaning a player is strictly happier with having more tokens, and (2) $U(W)$ is concave, where higher curvature implies a stronger risk aversion tendency. Hereinafter, we consider utility functions that meet this definition.

3 LedgerHedger

We present LEDGERHEDGER, our construction enabling a *Buyer* and a *Seller* to hedge future block gas for a predetermined *gas-price*. We begin by detailing LEDGERHEDGER's design (§3.1), and follow by formalizing its security guarantees (§3.2).

3.1 LedgerHedger Design

LEDGERHEDGER operates in two phases, *setup* and *exec*, representing its setup and execution in the block intervals of interest, presented in Figure 1. Throughout the following functions, the contract verifies identities using the EU-CMA digital signature algorithm.

In the *setup* phase, *Buyer* initiates a LEDGERHEDGER instance using a transaction. The initiation sets the contract parameters, including the block ranges in which interactions can be made with the contract instance, the required gas for the future transaction, and a required collateral to be deposited by *Seller*. She also deposits the token payment for the future transaction confirmation.

Following its initiation, the contract starts an *acceptance block countdown*, during which a *Seller* can accept it using a transaction. Additionally, accepting the contract requires *Seller* to deposit tokens as a collateral matching the collateral parameter. The collateral is returned conditioned on *Seller* further interacting with the contract. Either if *Seller* accepted the contract, or if the acceptance countdown is completed, the contract accepts no further interactions until the *exec* phase.

Towards or even during the *exec* phase, *Buyer* can publish $tx_{payload}$. This allows *Seller* to *apply* it, executing $tx_{payload}$, and getting the payment and collateral tokens from the contract. This is the main functionality of LEDGERHEDGER – enabling *Seller* to execute a transaction provided by *Buyer*.

Alternatively, *Seller* can *exhaust* the contract, consuming the hedged gas on null operations, and then receiving its tokens. The motivation for this functionality is to enable *Seller* to claim the tokens, regardless if *Buyer* provides a transaction or not; this protects *Seller* from a faulty or malicious *Buyer*. However, the naive solution of letting *Seller* report *Buyer* as faulty is not sufficient: It allows a *Seller* to falsely accuse a correct *Buyer*, getting the contract tokens without providing the confirmation service. By making *Seller* waste equivalent gas, we remove her incentive to do so.

If *Seller* has not accepted the contract, then *Buyer* can *recoup* the contract tokens using a transaction.

LedgerHedger comprises these functions, which we now describe in detail and present in Alg. 1.

Initiate *Buyer* initiates the contract through the invocation of the *Initiate* function (lines 1–6), setting the contract parameters. These include *acc*, the block number by which *Seller* is required to accept the contract; *start* and *end*, the range in block numbers during which block *Seller* is required to confirm the transaction; g_{alloc}, a positive number of gas units *Buyer* wishes to use; *col*, the non-negative token collateral required by *Seller*; and, ε, an additional non-negative number of tokens that will be transferred to *Seller* for confirming the provided *Buyer* transaction. For simplicity, we consider the block confirming the initiation transaction is b_{init}.

The contract verifies the provided parameters are valid according to the above specification: the block numbers are ascending, the gas parameter is positive, and the token parameters are non-negative (lines 2– 3).

After this verification, the contract derives the offered *payment*: the additional ε tokens are subtracted from the sent tokens *sentTokens*. This is the number of tokens that will be paid to *Seller* for either executing a transaction or exhausting the contract. This implies the contract's offered *gas-price* is $\pi_{contract} = \frac{payment}{g_{alloc}}$ (line 4). It also stores the public identifier of *Buyer* as PK_{Buyer} (line 5). Finally, the contract sets its status variable *status* to initiated (line 6), indicating the contract has been initiated, but no further transactions have interacted with it. We denote the gas consumption of the *Initiate* function by g_{init}.

Accept Once the contract is initiated, a *Seller* can accept it through the invocation of the *Accept* function (lines 6–12). This enables only a single *Seller* to accept the contract, and only before the timeout set by *Buyer* expires. It also requires *Seller* to deposit the requested collateral.

For that, this function first verifies that this invocation is no later than b_{acc} (line 8), that the contract has been initiated, but not further interacted with (line 9), and that the sent tokens collateral suffices (line 10).

The contract then stores *Seller* public identifier as PK_{Seller} (line 11), and updates its status variable *status* to accepted, indicating the contract has been accepted (line 12). We denote the gas consumption of the *Accept* function by g_{accept}.

The previous *Initiate* and *Accept* functions facilitate the initiation and acceptance of LEDGERHEDGER. The following three functions detail its conclusion.

Recoup The *Recoup* function (lines 12–18) enables *Buyer* to withdraw her deposited tokens from LEDGERHEDGER if no *Seller* accepts it prior to b_{acc}.

For that, it first verifies the invocation is within $[b_{start}, b_{end}]$ (line 14), that the contract is initiated, but no *Seller* had accepted it (line 15), and that the invocation is by *Buyer* (line 15). We discuss earlier recouping in Appendix D.

Then, the contract marks its status completed (line 17), and sends *Buyer* her deposited $payment + \varepsilon$ tokens (line 18). We denote the gas consumption of the *Recoup* function by g_{done}.

Apply The *Apply* function (lines 18–26) implements the main functionally of LEDGERHEDGER: *Seller* executing a transaction tx_{provided} provided by *Buyer*, and receiving the agreed-upon payment for doing so. Let g_{pub} be the gas consumption of tx_{provided}.

This function takes as an input a transaction tx_{provided}, and first verifies tx_{provided} was issued by *Buyer* (line 20). Then, it verifies the invocation is within $[b_{start}, b_{end}]$ (line 21), that *Seller* had previously accepted (line 22), and that the invocation is by *Seller* (line 23).

The contract then executes the operations of tx_{provided} as a subroutine (line 24), marks its status completed (line 32), and sends $payment + \varepsilon + col$ tokens to *Seller* (line 33).

Considering all operations except the execution of tx_{provided}, the *Apply* function performs similar operations to those of *Recoup*. We therefore consider its gas consumption, aside from execution of tx_{provided}, is also g_{done}.

Exhaust The *Exhaust* function (lines 26–33) allows *Seller* to get $payment + col$ tokens for expending g_{alloc} gas during the required block interval. Its goal is to protect *Seller* from a spiteful *Buyer*, specifically from the case where *Buyer* does not publish a tx_{provided} transaction, or publishes ones that consume more than g_{alloc} gas.

When *Exhaust* is invoked, the contract first verifies it is within $[b_{start}, b_{end}]$ (line 28), that *Seller* had previously accepted (line 29), and that the invocation is by *Seller* (line 30).

The contract then performs null operations consuming g_{alloc} gas (line 31), marks its status completed (line 32), and sends *Seller* $payment + col$ tokens (line 33). Note that executing *Exhaust* results with the remaining ε being forever locked in the contract.

Similarly, the operations of *Exhaust*, aside from the exhaustion, resemble those of *Recoup*. Therefore, its gas cost, aside from the exhaustion, is also g_{done}.

3.2 Possible LedgerHedger Interactions

Following immediately from the functions of LEDGERHEDGER (Alg. 1) and the EU-CMA digital signature algorithm, we get the following properties, which define all possible interactions of the participants with the contract:

Contract parameters are immutable The contract parameters are set only once by *Buyer* at its initiation and are immutable.

These parameters are set before π_{exec} is drawn. Moreover, *Buyer* must transfer $payment + \varepsilon$ tokens to the contract at its initiation.

Single *Seller* accepting Only a single *Seller* can accept the contract, only after it is initiated, and only before b_{acc}. That means *Seller* can accept the contract only after its parameters are set, and only after *Buyer* has already transferred $payment + \varepsilon$ tokens to it. *Seller* can accept the contract only before π_{exec} is known, and only by transferring *col* tokens.

Contract token extraction Extracting the contract tokens requires successfully invoking either *Recoup*, *Apply* or *Exhaust*, which all require to be invoked during $[b_{start}, b_{end}]$.

***Buyer* extracting tokens** Only *Buyer* can successfully invoke *Recoup*, only during $[b_{start}, b_{end}]$, and only if *Seller* had not accepted the contract.

***Seller* extracting tokens** Only *Seller* that accepted the contract can successfully invoke *Apply* or *Exhaust*, but not both. For either function, a successful invocation can be made only during $[b_{start}, b_{end}]$, and only if *Seller* had accepted the contract before b_{start} (specifically, before b_{acc} which precedes b_{start}).

Additionally, *Seller* can only successfully invoke *Apply* by providing a transaction $tx_{payload}$ published by *Buyer*.

4 Incentive Compatibility

To demonstrate incentive compatibility, we identify the conditions under which it is in the best interest of both parties to fulfill a given contract. In addition, we find contract parameters for which *Buyer* and *Seller* initiate and accept a contract in the first place. Our analysis culminates in the following theorem.

Theorem 1. *There exist utility functions, a distribution $\mathcal{F}$, and contract gas and token parameters such that: Buyer is incentivized to initiate the contract; Seller is incentivized to accept the initiated contract; Buyer is incentivized to publish $tx_{payload}$ with gas consumption g_{pub} equal to g_{alloc}; and, Seller is incentivized to fulfill the contract by confirming $tx_{payload}$.*

To prove the theorem, we first model LedgerHedger as a game between *Buyer* and *Seller*. We then consider a *subgame perfect equilibrium (SPE)*, capturing the dynamic, turn-based nature of the game. We express the equilibrium strategy as a function of the distribution, the utility functions, and the contract parameters. Afterward, we prove there are scenarios where engaging and fulfilling the contract is an SPE. The game definition and analysis of the SPE are deferred to Appendix E.

5 Efficacy

To show the efficacy of LedgerHedger, we first review relevant contract parameters, *gas-price* distributions, and utility functions (§5.1). We then show how to set the contract parameters to assure its fulfillment (§5.2), and conclude by describing concrete ranges where both parties benefit from the contract (§5.3).

5.1 Contract Parameters, Distributions, Utility Functions

Contract parameters We set $g_{alloc} = 5e6\left(5 \cdot 10^6\right)$ as a representative example of a ZK roll-up proof gas requirement [121,122], and arbitrarily choose $w^{exo} = w^{init}_{Buyer} = w^{init}_{Seller} = 1e9$. Considering our implementation gas requirements (presented in Appendix A), we fix the contract function gas requirements at $g_{init} = 0.1e6, g_{accept} = 75e3$ and $g_{done} = 20e3$. We still consider $\varepsilon = 1$, and derive the desired values of *payment* and *col* throughout this section.

Distribution $\mathcal{F}$ The resultant players' wealth depends on their strategies and on the *gas-price* value π_{exec}, which is drawn from $\mathcal{F}$. Therefore, towards our analysis, we need to instantiate $\mathcal{F}$. Inspired by Ethereum current *gas-price* [123], we set the *gas-price* at initiation to $\pi_{setup} = 100$. For the distribution $\mathcal{F}$, we consider normal distributions with a mean value of π_{setup}, and truncate them symmetrically at 0 and 200. We consider three different distributions, denoted $\forall i \in [1,3] : \mathcal{F}_i$, differing in their variance $\sigma_i^2 = 10^{i+1}$.

Utility functions Agent risk aversion is modeled through the concavity of its utility function. However, the optimal strategy is not affected by affine transformations of the utility function [70,124], so simply measuring the curvature fails to capture this preference. Instead, the risk preference of a utility function $U(W)$ is typically measured using its *Arrow-Pratt Relative Risk Aversion (RRA)* [65,66], $RRA = -\frac{W \cdot U''(W)}{U'(W)}$, where $U'(W)$ and $U''(W)$ are the first and second derivatives of $U(W)$, respectively.

For our instantiation we use a few common options for utility functions [66–70,112–114]: Linear utility $U(W) = W$ with $RRA = 0$, exhibiting risk-neutrality; Sqrt utility $U(W) = \sqrt{W}$ with $RRA = 0.5$, exhibiting mild risk-aversion; and, Log utility $U(W) = \log(W)$ with $RRA = 1$, exhibiting higher risk-aversion.

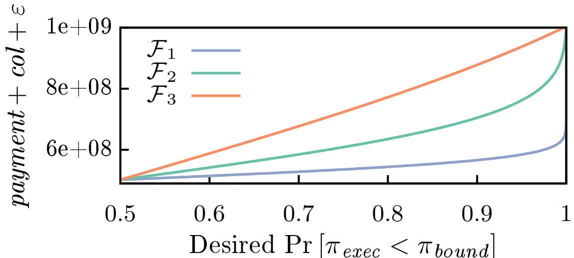


Fig. 2: Required contract funds $payment + col + \varepsilon$ to achieve desired fulfillment probability $\Pr[\pi_{exec} < \pi_{bound}]$.

5.2 Contract Fulfillment

With the contract parameters, distributions, and utility functions set, we are first interested in finding the *payment* and *col* parameters for *Seller* to confirm $tx_{payload}$. By Lemma 1, this occurs when $\pi_{exec} < \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}}$ (Appendix E). Let us denote $\pi_{bound} = \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}}$, hence we are interested in finding when $\pi_{exec} < \pi_{bound}$.

Recall $\pi_{exec} \sim \mathcal{F}$, so the condition holds only with some probability. This is not a predicament specific to LedgerHedger but to hedging in general – in extreme cases one party might be better off violating the contract, as the incurred punishment is smaller than the cost of abiding by the contract [71]. However, setting a sufficient incentive can achieve any desired probability. For bounded probabilities, we can achieve deterministic success.

The probability that $\pi_{exec} < \pi_{bound}$ is given by the distribution's *cumulative distribution function (CDF)* at π_{bound}. Figure 2 shows the required $payment + col + \varepsilon$ value to achieve $\Pr[\pi_{exec} < \pi_{bound}]$.

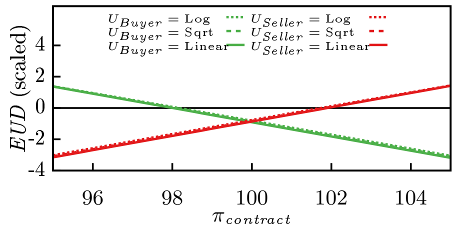
Figure 2 illustrates that increasing *payment* and *col* results with higher fulfillment probability, as they increase the incentive for *Seller* to fulfill the contract.

Additionally, Figure 2 shows the effect of the distribution variance on meeting the π_{bound} bound. As expected, the more variant distributions have heavier right tails, requiring more funds to achieve the same success probability.

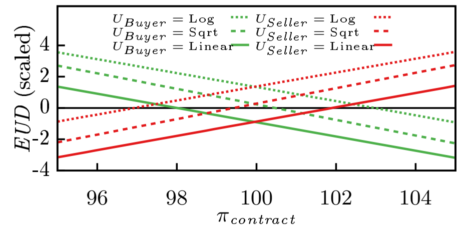
If there exists an upper bound on the distribution value, like in the truncated normal distribution, simply setting $payment + \varepsilon + col$ such that π_{bound} exceeds that upper bound assures success deterministically. In case of an unbounded distribution, the failure probability is negligible in $payment + \varepsilon + col$ according to the Chernoff bound [125]. As such, hereinafter, we consider *payment* and *col* values such that $\Pr[\pi_{exec} < \pi_{bound}] = 1$.

5.3 Initiation and Acceptance

Let us begin by considering the effect of the *payment* and the *col* parameters. *Buyer* pays *payment* tokens to *Seller* for g_{alloc} gas. Too high *payment* values disincentivize *Buyer* from initiating the contract, as she can buy g_{alloc} for the *gas-price* instead. Too low *payment* values disincentivize *Seller* from accepting



(a) Distribution $\mathcal{F}_1$



(b) Distribution $\mathcal{F}_3$

Fig. 3: Normalized expected utility difference.

the contract, as she can instead sell g_{alloc} for *gas-price*. The *col* tokens are used to incentivize *Seller* to abide by an accepted contract, as she loses them otherwise.

We now analyze the contract initiation and acceptance for concrete values of *payment* and *col*, for a specific $\mathcal{F}$, and assuming *Buyer* and *Seller* each have a utility function $\text{Utility} \in \{\text{Linear}, \text{Sqrt}, \text{Log}\}$.

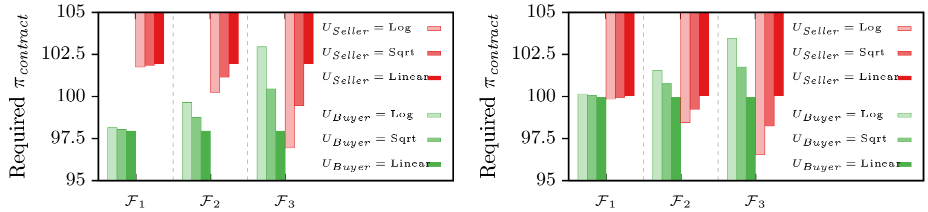
Let EUD_{Buyer} represent *Buyer*'s additional expected utility from initiating a LedgerHedger contract, as opposed to directly purchasing a gas allocation when the target timeframe arrives. Similarly, let EUD_{Seller} represent *Seller*'s additional expected utility from accepting a LedgerHedger contract, as opposed to directly selling her gas allocation in the target timeframe. Thus, when the expected utility difference is positive, the corresponding participant will interact with LedgerHedger (Corollary 1 and Corollary 2,Appendix E).

We arbitrarily set $col = 1e9$ to satisfy $\Pr[\pi_{exec} < \pi_{bound}] = 1$ (lower values suffice as well, as we need $payment + col > 1e9$), and numerically calculate EUD_{Buyer} and EUD_{Seller} for the various distributions and utility functions, as a function of $\pi_{contract} = \frac{payment}{g_{alloc}}$.

Figure 3 presents these values, scaled for comparison, for the various utility functions, and for the lowest-variance distribution $\mathcal{F}_1$ (Figure 3a) and for highest-variance distribution $\mathcal{F}_3$ (Figure 3b). As expected, the higher the agreed price $\pi_{contract}$ is, engaging in a contract becomes less profitable for *Buyer* and more for *Seller*, since the utility functions are strictly increasing. That is, *Buyer* agrees to initiate up to a maximal price, and *Seller* agrees to accept for no less than a minimal price. We denote these by $\pi^{\max}_{Buyer}$ and by $\pi^{\min}_{Seller}$, respectively, and refer to these as the *required* prices.

Determining the $\pi_{contract}$ that *Buyer* and *Seller* agree upon is a matter of negotiation, outside the scope of this work. We focus on finding conditions for such a price to exist, i.e., for $\pi^{\max}_{Buyer} > \pi^{\min}_{Seller}$.

Figure 3 shows that utility functions with higher RRA are more amenable to engage in the contract. Specifically, it shows that $\pi^{\max}_{Buyer}$ is the highest in case of a logarithmic utility function Log ($RRA = 1$), followed by the price in case of a square root utility function Sqrt ($RRA = 0.5$), and then by the price in case of a linear utility function Linear ($RRA = 0$). This is expected – higher RRA means higher preference for certainty, which is achieved through engaging in the



(a) With friction

(b) Without friction

Fig. 4: $\pi_{contract}$ for initiation and acceptance.

contract. Symmetrically, it shows that $\pi_{Seller}^{\min}$ is the lowest with a logarithmic utility function, and highest with a linear utility function.

Lastly, Figure 3 highlights how the distribution $\mathcal{F}$ affects the existence of a $\pi_{contract}$ such that $\pi_{Buyer}^{\max} > \pi_{Seller}^{\min}$. For $\mathcal{F}_1$ (Figure 3a), there is no $\pi_{contract}$ where for any combination of utility function for *Buyer* and *Seller* both utility differences are positive, i.e., $\pi_{Buyer}^{\max} < \pi_{Seller}^{\min}$. However, for $\mathcal{F}_3$ (Figure 3b), there is a range of $\pi_{contract}$ values where $\pi_{Buyer}^{\max} > \pi_{Seller}^{\min}$ for some utility function combinations. This is due to the different variance values of the distributions. Intuitively, a distribution with higher variance offers less certainty about π_{exec}, making the contract-induced certainty more appealing for risk-averse ($RRA > 0$) participants.

To further emphasize the distribution effect, Figure 4a presents the required prices for the various utility functions and distributions. It shows the distributions with lower variance values $\mathcal{F}_1$ and $\mathcal{F}_2$ both result with $\pi_{Buyer}^{\max} < \pi_{Seller}^{\min}$, i.e., no contract. However, for a high variance value, there exist combinations of U_{Buyer} and U_{Seller} that result with $\pi_{Buyer}^{\max} > \pi_{Seller}^{\min}$. For example, the above is satisfied for $\mathcal{F}_3$ when U_{Buyer} is Log and U_{Seller} is Linear, or vice versa. This implies that the parties engage in a contract even if one of them is risk neutral.

Figure 4a also shows that the required prices are fixed for the linear utility function, for both *Buyer* and *Seller*, for any considered $\mathcal{F}$. Broadly speaking, this holds due to $\Pr\left[\pi_{exec} < \pi_{bound}\right] = 1$, the linearity of the utility function, and the fact all considered distributions have the same mean value. We bring a thorough explanation in Appendix F.

Finally, as a theoretical exercise, we consider the cost of *friction* [126] – the inherent costs of g_{init}, g_{accept} and g_{done} that *Buyer* and *Seller* incur. The reason this experiment might be of interest is due to further optimizations in LedgerHedger that result with even lower overheads. We set $g_{init} = g_{accept} = g_{done} = 0$ and find the required prices for the various utility functions and distributions (Figure 4b). As expected, reducing the friction results with both *Buyer* and *Seller* being more amenable to initiate and accept the contract. Specifically, this relaxation facilitates the contract creation even for $\mathcal{F}_1$ and $\mathcal{F}_2$.

6 Related work

We are not aware of previous work that guarantees future transaction confirmation in a timely manner, despite this being a security requirement of prominent cryptocurrency applications.

Recently, Lotem et al. [86] suggested extending Ethereum's contract capabilities to allow applications to monitor the blockchain congestion level. The applications can then extend their timeouts in case of congestion. This mechanism replaces safety with liveness violations – the timeout does not expire, but the application cannot progress to its post-timeout state. In contrast, LedgerHedger assures confirmation at the desired interval, and is directly applicable to Ethereum and similar blockchains.

Infura's any.sender [76] service gets issuers' transactions confirmed at competitive fees using estimation and dynamic fee update. Unlike LedgerHedger, it does not address long-term reservation and its necessary mechanisms.

Several projects suggest mitigating *gas-price* changes using gas tokens. These are managed by designated smart contracts, whose value follows the *gas-price*. To protect against *gas-price* rising (falling), one buys (sells) gas-tokens, and later sells (buys) them. A future transaction issuer can acquire gas-tokens beforehand, and sell them to fund the transaction fees at the desired inclusion time.

The first type of gas token [88–90] was implemented by abusing Ethereum's *gas refund mechanism*, where several operations had negative gas costs. The principle was to deliberately expend gas on storing arbitrary data when the *gas-price* is low, and later, when the *gas-price* rises, delete that data for a gas refund. This method was inefficient, as only about a third of the spent gas is refunded. Moreover, the August 2021 Ethereum upgrade [127] broke this mechanism by changing the refund policy [91, 92]. In contrast, LedgerHedger does not rely on Ethereum's internal implementation, and hence applies to a wide range of systems. Moreover, its overhead is three orders of magnitude less than the hedged amount for practical parameter values.

Another approach for implementing gas tokens is pegging them to the gas value, e.g., *uGAS* [87], and Pitch Lake [128]. uGAS tokens have month-granularity expiration dates, and their expiration value is set according to an *oracle* [129] – another contract that, by external measures, feeds the median gas price of all Ethereum transactions. Users can deposit and release cryptocurrency to mint and destroy uGAS tokens, respectively. The required cryptocurrency amount, deposit duration and withdrawal availability all depend on a set of variables such as the oracle-reported price and the token availability in the managing contract. Moreover, user deposits may be confiscated in a so-called *liquidation* if their deposit value falls below a certain threshold. Protocols of this kind are susceptible to various attacks and manipulations [130–135], in particular to taking advantage of the oracle [136–147]. Furthermore, setting the oracle measured time period is nontrivial – short periods make it easy to manipulate, but long periods result with the reported value being inaccurate.

In contrast, LedgerHedger does not rely on oracles, and is conducted solely among the two interacting parties, removing the ability to affect its state through the aforementioned manipulations. LedgerHedger also enables arbitrary choice of the target time frame.

Traditional financial hedging instruments typically rely on either cash settlement or the delivery of goods [71, 148]. The latter kind of gas tokens, as mentioned previously, leans on cash settlement. This method, however, necessitates knowledge of the price, typically sourced from oracles, which are vulnerable to manipulation. In contrast, LedgerHedger paves the way for a novel category of oracle-less gas tokens, which can use it for transferring gas allocations. We leave the exploration of this direction for future work.

The August 2021 [127] update to the Ethereum network applied *Ethereum Improvement Proposal (EIP)* 1559 [83], changing the transaction fee mechanism. EIP1559, along with other work [77–82], attempts to ease transaction issuers estimation of the required fee solely for the next block; they do not apply (or claim to apply) to further blocks.

Aside from benign price fluctuations, previous work shows the fee market is susceptible to *congestion attacks* [12,149,150]. These create multiple transactions that artificially increase the market price, congesting the network, resulting with time-sensitive transactions being delayed. A similar impact can arise from *censorship attacks* [151], where a malicious party prevents transaction from being confirmed. Such attacks are executed by manipulating miners' incentives, often through methods like bribes [152–155].

LedgerHedger is capable of withstanding such attacks or benign market spikes of any magnitude by incorporating a sufficiently high *Seller* collateral, thereby ensuring future confirmations at predetermined prices, even in far-future scenarios. Moreover, LedgerHedger's functionality remains unaffected by updates such as EIP1559, underlining its resilience and versatility in a rapidly-evolving domain.

7 Conclusion

We introduce LedgerHedger, a blockchain smart contract for confirming a future transaction of *Buyer* for a predetermined fee by *Seller*. We analyze fee variability and prove that fulfilling the contract is an SPE for a wide range of practical parameters. We implement LedgerHedger as a smart contract for Ethereum, deploy it, and demonstrate its efficacy and low gas overhead compared to common gas requirements.

LedgerHedger is directly applicable to secure smart contracts executed over Ethereum and similar systems, resolving the prevalent issue of unjustified reliance on fee stability.

Acknowledgments

This research was supported by Avalanche Foundation and by IC3.

References

1. Buterin, V.: A next generation smart contract & decentralized application platform (2013), https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf/
2. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)
3. Yakovenko, A.: Solana: A new architecture for a high performance blockchain v0. 8.13. Whitepaper (2018)
4. Rocket, T.: Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. Available [online] (2018)
5. Rocket, T., Yin, M., Sekniqi, K., van Renesse, R., Sirer, E.G.: Scalable and probabilistic leaderless bft consensus through metastability. arXiv preprint arXiv:1906.08936 (2019)
6. Binance: Binance smart chain (2020), https://github.com/binance-chain/whitepaper/blob/master/WHITEPAPER.md
7. coinmarketcap.com: Cryptocurrency market capitalizations (2022), https://coinmarketcap.com/, accessed: 2022-01-10
8. Decker, C., Wattenhofer, R.: Information propagation in the Bitcoin network. In: IEEE P2P. Trento, Italy (2013)
9. Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Sirer, E.G., et al.: On scaling decentralized blockchains. In: International conference on financial cryptography and data security. pp. 106–125. Springer (2016)
10. Venugopal, S.: Users raise a stink over sunflower farmers nft for gas fee spikes on polygon (2022), https://ambcrypto.com/users-raise-a-stink-over-sunflower-farmers-nft-for-gas-fee-spikes-on-polygon/
11. Frangella, E.: Crypto black thursday: The good, the bad, and the ugly (2020), https://medium.com/aave/crypto-black-thursday-the-good-the-bad-and-the-ugly-7f2acebf2b83
12. Munro, A.: Fomo3d ethereum ponzi game r1 ends as hot play outmaneuvers bots (2018), https://www.finder.com.au/fomo3d-ethereum-ponzi-game-r1-ends-as-hot-play-outmaneuvers-bots
13. ConsenSys: The inside story of the cryptokitties congestion crisis (2018), https://media.consensys.net/the-inside-story-of-the-cryptokitties-congestion-crisis-499b35d119cc
14. Shevchenko, A.: Here are the best and worst times of the day to use ethereum (2021), https://cointelegraph.com/news/here-are-the-best-and-worst-times-of-the-day-to-use-ethereum
15. Sigalos, M.: Ethereum had a rough september. here's why and how it's being fixed (2021), https://www.cnbc.com/2021/10/02/ethereum-had-a-rough-september-heres-why-and-how-it-gets-fixed.html
16. Hake, M.R.: Fees threaten ethereum's perch as king of nfts, https://www.nasdaq.com/articles/fees-threaten-ethereums-perch-as-king-of-nfts-2021-10-11
17. Analytics, D.: Average gas price per day for the last 30 days (2022), https://dune.xyz/queries/7898/15742, accessed: 2022-01-13
18. Hoenicke, J.: Johoe's mempool statistics (2021), https://jochen-hoenicke.de/queue/#ETH,all,fee, accessed: 2022-01-13
19. coinmarketcap.com: Loopring market cap (2022), https://coinmarketcap.com/currencies/loopring/, accessed: 2022-01-10

20. coinmarketcap.com: Hermez market cap (2022), https://coinmarketcap.com/currencies/hermez-network/, accessed: 2022-01-10
21. Möser, M., Eyal, I., Sirer, E.G.: Bitcoin covenants. In: Financial Cryptography and Data Security (2016)
22. McCorry, P., Möser, M., Ali, S.T.: Why preventing a cryptocurrency exchange heist isn't good enough. In: Cambridge International Workshop on Security Protocols (2018)
23. Bryan Bishop: Bitcoin vaults with anti-theft recovery/clawback mechanisms, https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2019-August/017231.html
24. Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., Knottenbelt, W.: Xclaim: Trustless, interoperable, cryptocurrency-backed assets. In: 2019 IEEE S&P (2019)
25. Herlihy, M.: Atomic cross-chain swaps. In: Proceedings of the 2018 ACM symposium on principles of distributed computing (2018)
26. Malavolta, G., Moreno-Sanchez, P., Schneidewind, C., Kate, A., Maffei, M.: Anonymous multi-hop locks for blockchain scalability and interoperability. In: NDSS (2019)
27. van der Meyden, R.: On the specification and verification of atomic swap smart contracts. In: IEEE ICBC (2019)
28. Miraz, M.H., Donald, D.C.: Atomic cross-chain swaps: development, trajectory and potential of non-monetary digital token swap facilities. Annals of Emerging Technologies in Computing (AETiC) Vol (2019)
29. Zie, J.Y., Deneuville, J.C., Briffaut, J., Nguyen, B.: Extending atomic cross-chain swaps. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology (2019)
30. Xue, Y., Herlihy, M.: Hedging against sore loser attacks in cross-chain transactions. arXiv preprint arXiv:2105.06322 (2021)
31. Maxwell, G.: The first successful zero-knowledge contingent payment, https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/
32. Campanelli, M., Gennaro, R., Goldfeder, S., Nizzardo, L.: Zero-knowledge contingent payments revisited: Attacks and payments for services. In: Proceedings of the 2017 ACM CCS (2017)
33. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: European Symposium on Research in Computer Security (2016)
34. Fuchsbauer, G.: Wi is not enough: Zero-knowledge contingent (service) payments revisited. In: Proceedings of the 2019 ACM CCS (2019)
35. Bursuc, S., Kremer, S.: Contingent payments on a public ledger: models and reductions for automated verification. In: European Symposium on Research in Computer Security (2019)
36. Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts. White paper pp. 1–47 (2017)
37. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016)
38. Decker, C., Wattenhofer, R.: A fast and scalable payment network with Bitcoin Duplex Micropayment Channels. In: Stabilization, Safety, and Security of Distributed Systems - 17th International Symposium (2015)
39. Green, M., Miers, I.: Bolt: Anonymous payment channels for decentralized currencies. In: Proceedings of the 2017 ACM CCS (2017)

40. McCorry, P., Möser, M., Shahandasti, S.F., Hao, F.: Towards bitcoin payment networks. In: Australasian Conference on Information Security and Privacy (2016)
41. Miller, A., Bentov, I., Bakshi, S., Kumaresan, R., McCorry, P.: Sprites and state channels: Payment networks that go faster than lightning. In: Financial Cryptography and Data Security (2019)
42. Dziembowski, S., Faust, S., Hostáková, K.: General state channel networks. In: Proceedings of the 2018 ACM CCS (2018)
43. Dziembowski, S., Eckey, L., Faust, S., Malinowski, D.: Perun: Virtual payment channels over cryptographic currencies. IACR ePrint (2017)
44. Tsabary, I., Yechieli, M., Manuskin, A., Eyal, I.: Mad-htlc: because htlc is crazy-cheap to attack. In: 2021 IEEE Symposium on Security and Privacy (SP) (2021)
45. Wadhwa, S., Stöter, J., Zhang, F., Nayak, K.: He-htlc: Revisiting incentives in htlc. Cryptology ePrint Archive (2022)
46. Garoffolo, A., Kaidalov, D., Oliynykov, R.: Zendoo: a zk-snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains. In: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS). pp. 1257–1262. IEEE (2020)
47. Buterin, V.: The dawn of hybrid layer 2 protocols (2019), available online
48. McCorry, P., Buckland, C., Yee, B., Song, D.: Sok: Validating bridges as a scaling solution for blockchains. Cryptology ePrint Archive (2021)
49. Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S.M., Felten, E.W.: Arbitrum: Scalable, private smart contracts. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 1353–1370 (2018)
50. Optimism: Optimism website (2021), https://www.optimism.io/
51. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. IACR Cryptol. ePrint Arch. **2018**, 46 (2018)
52. Starkware: Starkware website (2021), https://starkware.co/
53. Hermez: Scalable payments. decentralised by design, open for everyone. (2020), https://hermez.io/hermez-whitepaper.pdf
54. Wang, D., Zhou, J., Wang, A., Finestone, M.: Loopring: A decentralized token exchange protocol. URL https://github. com/Loopring/whitepaper/blob/-master/en_whitepaper. pdf (2018)
55. Labs, M.: Matter labs website (2021), https://matter-labs.io/
56. Sguanci, C., Spatafora, R., Vergani, A.M.: Layer 2 blockchain scaling: A survey. arXiv preprint arXiv:2107.10881 (2021)
57. Thibault, L.T., Sarry, T., Hafid, A.S.: Blockchain scaling using rollups: A comprehensive survey. IEEE Access (2022)
58. Cong, L.W., He, Z., Li, J.: Decentralized mining in centralized pools. The Review of Financial Studies **34**(3), 1191–1235 (2021)
59. Chatzigiannis, P., Baldimtsi, F., Griva, I., Li, J.: Diversification across mining pools: Optimal mining strategies under pow. arXiv preprint arXiv:1905.04624 (2019)
60. Jiang, S., Li, Y., Wang, S., Zhao, L.: Blockchain competition: The tradeoff between platform stability and efficiency. European Journal of Operational Research **296**(3), 1084–1097 (2022)
61. Chen, X., Papadimitriou, C., Roughgarden, T.: An axiomatic approach to block rewards. In: Proceedings of ACM AFT (2019)
62. Lewenberg, Y., Bachrach, Y., Sompolinsky, Y., Zohar, A., Rosenschein, J.S.: Bitcoin mining pools: A cooperative game theoretic analysis. In: Proceedings of the

2015 International Conference on Autonomous Agents and Multiagent Systems (2015)
63. Wang, C., Chu, X., Qin, Y.: Measurement and analysis of the bitcoin networks: A view from mining pools. In: 2020 6th International Conference on Big Data Computing and Communications (BIGCOM). pp. 180–188. IEEE (2020)
64. Yaish, A., Zohar, A.: Correct cryptocurrency asic pricing: Are miners overpaying. arXiv preprint arXiv:2002.11064 (2020)
65. Arrow, K.J.: The theory of risk aversion. Essays in the theory of risk-bearing pp. 90–120 (1971)
66. Pratt, J.W.: Risk aversion in the small and in the large. In: Uncertainty in economics, pp. 59–79. Elsevier (1978)
67. Karatzas, I., Shreve, S.E., Karatzas, I., Shreve, S.E.: Methods of mathematical finance, vol. 39. Springer (1998)
68. Cochrane, J.: Asset pricing: Revised edition. Princeton university press (2009)
69. Shreve, S.: Stochastic calculus for finance I: the binomial asset pricing model. Springer Science & Business Media (2005)
70. Simon, C.P.: Mathematics for economists. Norton & Company, Inc (1994)
71. Hull, J.C.: Options futures and other derivatives. Pearson Education India (2003)
72. Kawai, M.: Spot and futures prices of nonstorable commodities under rational expectations. The Quarterly Journal of Economics **98**(2), 235–254 (1983)
73. Narayanan, A., Bonneau, J., Felten, E., Miller, A., Goldfeder, S.: Bitcoin and cryptocurrency technologies: a comprehensive introduction (2016)
74. Dziembowski, S., Eckey, L., Faust, S.: Fairswap: How to fairly exchange digital goods. In: ACM CCS (2018)
75. Asgaonkar, A., Krishnamachari, B.: Solving the buyer and seller's dilemma: A dual-deposit escrow smart contract for provably cheat-proof delivery and payment for a digital good without a trusted mediator. In: IEEE ICBC (2019)
76. McCorry, P.: any.sender, transactions made simple (2020), https://medium.com/anydot/any-sender-transactions-made-simple-34b36ba7519b
77. Pierro, G.A., Rocha, H., Tonelli, R., Ducasse, S.: Are the gas prices oracle reliable? a case study using the ethgasstation. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE). pp. 1–8. IEEE (2020)
78. Werner, S.M., Pritz, P.J., Perez, D.: Step on the gas? a better approach for recommending the ethereum gas price. In: Mathematical Research for Blockchain Economy, pp. 161–177. Springer (2020)
79. Valson, P.: Transaction fee estimations: How to save on gas? (2020), https://medium.com/@pranay.valson/transaction-fee-estimations-how-to-save-on-gas-part-2-72f908b13d67
80. Turksonmez, K., Furtak, M., Wittie, M.P., Millman, D.L.: Two ways gas price oracles miss the mark. In: 2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS). pp. 1–7. IEEE (2021)
81. Liu, F., Wang, X., Li, Z., Xu, J., Gao, Y.: Effective gasprice prediction for carrying out economical ethereum transaction. In: 2019 6th International Conference on Dependable Systems and Their Applications (DSA). pp. 329–334. IEEE (2020)
82. Mars, R., Abid, A., Cheikhrouhou, S., Kallel, S.: A machine learning approach for gas price prediction in ethereum blockchain. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). pp. 156–165. IEEE (2021)
83. Roughgarden, T.: Transaction fee mechanism design for the ethereum blockchain: An economic analysis of eip-1559. arXiv preprint arXiv:2012.00854 (2020)

84. Lavi, R., Sattath, O., Zohar, A.: Redesigning bitcoin's fee market. In: The World Wide Web Conference (2019)
85. Basu, S., Easley, D., O'Hara, M., Sirer, G.: Stablefees: A predictable fee market for cryptocurrencies. Work. Pap. (2020)
86. Lotem, A., Azouvi, S., Zohar, A., McCorry, P.: Sliding window challenge process for congestion detection. In: Financial Cryptography and Data Security (2022)
87. Degenerative: ugas token (2021), https://web.archive.org/web/20220119144017/https://docs.degenerative.finance/
88. Breidenbach, L., Daian, P., Tramer, F.: Gas token (2018), https://gastoken.io/
89. 1inch Network: 1inch introduces chi gastoken (2020), https://blog.1inch.io/1inch-introduces-chi-gastoken-d0bd5bb0f92b
90. Nadler, M.: A quantitative analysis of the ethereum fee market: How storing gas can result in more predictable prices (2020)
91. Vitalik Buterin, M.S.: Eip-3529: Reduction in refunds (2021), https://eips.ethereum.org/EIPS/eip-3529
92. Benson, J.: Ethereum london hard fork to make some tokens worthless (2021), https://decrypt.co/77345/ethereum-london-hard-fork-make-some-tokens-worthless
93. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), http://www.bitcoin.org/bitcoin.pdf
94. Poon, J., Dryja, T.: The Bitcoin Lightning Network, http://lightning.network/lightning-network.pdf
95. Tsabary, I., Eyal, I.: The gap game. In: ACM CCS (2018)
96. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on computing (1988)
97. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ecdsa). International journal of information security **1**(1), 36–63 (2001)
98. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: A composable treatment. In: Annual International Cryptology Conference (2017)
99. Carlsten, M., Kalodner, H., Weinberg, S.M., Narayanan, A.: On the instability of bitcoin without the block reward. In: Proceedings of the 2016 ACM CCS (2016)
100. Allen, S., Čapkun, S., Eyal, I., Fanti, G., Ford, B.A., Grimmelmann, J., Juels, A., Kostiainen, K., Meiklejohn, S., Miller, A., et al.: Design choices for central bank digital currency: Policy and technical considerations. Tech. rep., National Bureau of Economic Research (2020)
101. Fung, B.S., Halaburda, H.: Central bank digital currencies: a framework for assessing why and how. Available at SSRN 2994052 (2016)
102. Fama, E.F.: Random walks in stock market prices. Financial analysts journal **51**(1), 75–80 (1995)
103. Kendall, M.G., Hill, A.B.: The analysis of economic time-series-part i: Prices. Journal of the Royal Statistical Society. Series A (General) **116**(1), 11–34 (1953)
104. Reeve, T.A., Vigfusson, R.J.: Evaluating the forecasting performance of commodity futures prices. FRB International Finance Discussion Paper (1025) (2011)
105. Bowman, C., Husain, A.M., et al.: Forecasting commodity prices: Futures versus judgment. March (2004)
106. Walker, H.M., Helen, M.: De moivre on the law of normal probability. Smith, David Eugene. A source book in mathematics. Dover (1985)
107. Papoulis, A., Pillai, S.U.: Probability, random variables, and stochastic processes. Tata McGraw-Hill Education (2002)

108. etherscan.info: Ethereum average gas price chart (2021), https://etherscan.io/chart/gasprice
109. Massey Jr, F.J.: The kolmogorov-smirnov test for goodness of fit. Journal of the American statistical Association **46**(253), 68–78 (1951)
110. Thomopoulos, N.T., Thomopoulos, N.T., Philipson: Probability Distributions. Springer (2018)
111. Levy, H.: Stochastic dominance: Investment decision making under uncertainty. Springer (2015)
112. Kimball, M.S.: Standard risk aversion. Econometrica: Journal of the Econometric Society pp. 589–611 (1993)
113. Chiappori, P.A., Paiella, M.: Relative risk aversion is constant: Evidence from panel data. Journal of the European Economic Association **9**(6), 1021–1052 (2011)
114. Dyer, J.S., Sarin, R.K.: Relative risk aversion. Management science **28**(8), 875–886 (1982)
115. Outreville, J.F.: Risk aversion, risk behavior, and demand for insurance: A survey. Journal of Insurance Issues pp. 158–186 (2014)
116. Cicchetti, C.J., Dubin, J.A.: A microeconometric analysis of risk aversion and the decision to self-insure. Journal of political Economy **102**(1), 169–186 (1994)
117. Conlon, T., Cotter, J., Gençay, R.: Commodity futures hedging, risk aversion and the hedging horizon. The European Journal of Finance **22**(15), 1534–1560 (2016)
118. Haushalter, D.: Why hedge? some evidence from oil and gas producers. Journal of Applied Corporate Finance **13**(4), 87–92 (2001)
119. Traeger, C.P.: Why uncertainty matters: discounting under intertemporal risk aversion and ambiguity. Economic Theory **56**(3), 627–664 (2014)
120. Ellsberg, D.: Risk, ambiguity, and the savage axioms. The quarterly journal of economics pp. 643–669 (1961)
121. etherscan.io: Optimism 10.2m gas transaction (2021), https://etherscan.io/tx/0x90ebd9630d98d5b0a186eec4c2382c296e5f41e828da910d76a53ab72ffe30e8
122. Ohad Barta: Zk roll-up gas consumption (2021), https://twitter.com/OhadBarta/status/1463875770196049931
123. etherscan.io: Ether transaction fees, https://etherscan.io/chart/transactionfee
124. Benchimol, J.: Risk aversion in the eurozone. Research in Economics **68**(1), 39–56 (2014)
125. Chernoff, H., et al.: A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. The Annals of Mathematical Statistics (1952)
126. Kallsen, J., Muhle-Karbe, J.: Option pricing and hedging with small transaction costs. Mathematical Finance **25**(4), 702–723 (2015)
127. Foundation, E.: Ethereum london hard fork (2021), https://ethereum.org/en/history/#london
128. Network, O.: Pitch lake. Available at SSRN 4123018 (2022)
129. UMA: How uma solves the oracle problem (2020), https://docs.umaproject.org/oracle/econ-architecture
130. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: 2020 IEEE S&P
131. Zhou, L., Qin, K., Torres, C.F., Le, D.V., Gervais, A.: High-frequency trading on decentralized on-chain exchanges. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 428–445. IEEE (2021)
132. Qin, K., Zhou, L., Gervais, A.: Quantifying blockchain extractable value: How dark is the forest? arXiv preprint arXiv:2101.05511 (2021)

133. Zhou, L., Qin, K., Cully, A., Livshits, B., Gervais, A.: On the just-in-time discovery of profit-generating transactions in defi protocols. arXiv preprint arXiv:2103.02228 (2021)
134. Qin, K., Zhou, L., Livshits, B., Gervais, A.: Attacking the defi ecosystem with flash loans for fun and profit. In: International Conference on Financial Cryptography and Data Security. pp. 3–32. Springer (2021)
135. Wang, Z., Qin, K., Minh, D.V., Gervais, A.: Speculative multipliers on defi: Quantifying on-chain leverage risks. In: Financial Cryptography and Data Security (2022)
136. Tjiam, K., Wang, R., Chen, H., Liang, K.: Your smart contracts are not secure: Investigating arbitrageurs and oracle manipulators in ethereum. In: Proceedings of the 3rd Workshop on Cyber-Security Arms Race. pp. 25–35 (2021)
137. Gupta, M.: All twaps are subject to manipulation (2021), https://tinyurl.com/5xv2nnpj
138. Paleko: The bzx attacks explained (2020), https://www.palkeo.com/en/projets/ethereum/bzx.html
139. Eskandari, S., Salehi, M., Gu, W.C., Clark, J.: Sok: Oracles from the ground truth to market manipulation. arXiv preprint arXiv:2106.00667 (2021)
140. Todd, R.: Synthetix suffers oracle attack, more than 37 million synthetic ether exposed (2019), https://www.theblockcrypto.com/linked/28748/synthetix-suffers-oracle-attack-potentially-looting-37-million-synthetic-ether
141. Qin, K., Zhou, L., Gamito, P., Jovanovic, P., Gervais, A.: An empirical study of defi liquidations: Incentives, risks, and instabilities. arXiv preprint arXiv:2106.06389 (2021)
142. Perez, D., Werner, S.M., Xu, J., Livshits, B.: Liquidations: Defi on a knife-edge. In: International Conference on Financial Cryptography and Data Security. pp. 457–476. Springer (2021)
143. Salehi, M., Clark, J., Mannan, M.: Red-black coins: Dai without liquidations. In: International Conference on Financial Cryptography and Data Security. pp. 136–145. Springer (2021)
144. Sardon, A.: Zero-liquidation loans: A structured product approach to defi lending. arXiv preprint arXiv:2110.13533 (2021)
145. Analytica, C.: Mev attack – just-in-time liquidity (2021), https://twitter.com/ChainsightA/status/1457958811243778052
146. Zhao, M.: Yield farming is a misnomer (2021), https://twitter.com/FabiusMercurius/status/1454513434209312772
147. Wang, Y., Li, J., Su, Z., Wang, Y.: Arbitrage attack: Miners of the world, unite! In: Financial Cryptography and Data Security (2022)
148. Lioui, A., Poncet, P.: Dynamic asset allocation with forwards and futures. Springer Science & Business Media (2005)
149. Harris, J., Zohar, A.: Flood & loot: A systemic attack on the lightning network. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. pp. 202–213 (2020)
150. Mizrahi, A., Zohar, A.: Congestion attacks in payment channel networks. In: International Conference on Financial Cryptography and Data Security. pp. 170–188. Springer (2021)
151. Wahrstätter, A., Ernstberger, J., Yaish, A., Zhou, L., Qin, K., Tsuchiya, T., Steinhorst, S., Svetinovic, D., Christin, N., Barczentewicz, M., et al.: Blockchain censorship. arXiv preprint arXiv:2305.18545 (2023)

152. Bonneau, J.: Why buy when you can rent? bribery attacks on bitcoin-style consensus. In: International Conference on Financial Cryptography and Data Security. pp. 19–26. Springer (2016)
153. McCorry, P., Hicks, A., Meiklejohn, S.: Smart contracts for bribing miners. In: Financial Cryptography and Data Security: FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers 22. pp. 3–18. Springer (2019)
154. Judmayer, A., Stifter, N., Zamyatin, A., Tsabary, I., Eyal, I., Gazi, P., Meiklejohn, S., Weippl, E.R.: Pay-to-win: Incentive attacks on proof-of-work cryptocurrencies. IACR Cryptol. ePrint Arch. **2019**, 775 (2019)
155. Winzer, F., Herd, B., Faust, S.: Temporary censorship attacks in the presence of rational miners. In: 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 357–366. IEEE (2019)
156. Ethereum: Solidity language (2020), https://github.com/ethereum/solidity
157. Foundation, E.: Smart contract wallets (2022), https://docs.ethhub.io/using-ethereum/wallets/smart-contract-wallets/
158. di Angelo, M., Salzer, G.: Wallet contracts on ethereum–identification, types, usage, and profiles. arXiv preprint arXiv:2001.06909 (2020)
159. Griffith, A.T.: Ethereum meta transactions (2022)
160. OpenZeppelin: Ecdsa solidity library (2022), https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/cryptography/ECDSA.sol
161. Goerli: Ethereum goerli test network (2018), https://goerli.net/
162. Blockchair: Blockchain explorer, analytics and web services (2021), blockchair.com
163. Watson, J.: Strategy: an introduction to game theory (2002)
164. Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. Proceedings of the ACM on Programming Languages **2**(POPL), 1–28 (2017)
165. Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: Surviving out-of-gas conditions in ethereum smart contracts. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–27 (2018)
166. Cecchetti, E., Yao, S., Ni, H., Myers, A.C.: Compositional security for reentrant applications. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1249–1267. IEEE (2021)
167. Shoham, Y., Leyton-Brown, K.: Multiagent systems: Algorithmic, game-theoretic, and logical foundations (2008)
168. Osborne, M.J., Rubinstein, A.: A course in game theory (1994)
169. Bernheim, B.D.: Rationalizable strategic behavior. Econometrica: Journal of the Econometric Society (1984)
170. Rosenthal, R.W.: Games of perfect information, predatory pricing and the chain-store paradox. Journal of Economic theory (1981)
171. Fudenberg, D., Tirole, J.: Game theory, 1991. Cambridge, Massachusetts (1991)
172. Myerson, R.: Game theory: Analysis of conflict harvard univ. Press, Cambridge (1991)
173. Selten, R.: Spieltheoretische behandlung eines oligopolmodells mit nachfrageträgheit: Teil i: Bestimmung des dynamischen preisgleichgewichts. Zeitschrift für die gesamte Staatswissenschaft/Journal of Institutional and Theoretical Economics (1965)

174. Van Damme, E.: Strategic equilibrium. Handbook of game theory with economic applications (2002)
175. Cerny, J.: Playing general imperfect-information games using game-theoretic algorithms. Ph.D. thesis, PhD thesis, Czech Technical University (2014)
176. Aumann, R.J.: Backward induction and common knowledge of rationality. Games and Economic Behavior (1995)
177. Kamiński, M.M.: Backward induction: Merits and flaws. Studies in Logic, Grammar and Rhetoric (2017)
178. Roughgarden, T.: Algorithmic game theory. Communications of the ACM (2010)

A Implementation

To demonstrate the practicality of LedgerHedger, we implement it as an Ethereum smart contract, and deploy it on a test network. In this section, we describe the implementation and deployment details, and present measured gas overheads.

Appendix G reviews an alternative implementation, based on a recent Ethereum improvement proposal [83,127]. Appendix D reviews possible modifications, concerning user experience and overheads. These include enabling *Buyer* to withdraw the tokens earlier in case *Seller* is unresponsive, and reducing function overheads by using constant, predefined parameters.

Ethereum smart contracts are written in the Solidity smart contract programming language [156]; we bring the code in Appendix H.

Design Our implementation follows the *smart contract wallet (SCW)* [76,157,158] design pattern. This design enables customizing the retrieval of the contract tokens, which, for LedgerHedger, is done only through the *Apply*, *Exhaust*, and *Recoup* functions.

Additionally, this design enables decoupling the transaction *issuer* (i.e., the party that pays the transaction fees) from the transaction *signer* (the party that creates the transaction). This, in turn, enables having one party, *Seller*, use her gas allocation (or pay the transaction fees) to confirm a transaction by the other party *Buyer*, using so-called *meta transactions* [159].

We implemented LedgerHedger to be reusable for *Buyer*, that is, it is deployed once, and then can be used to create new instances over and over again. This amortizes the deployment gas requirements, which are higher than other operations [2].

Function Implementations The implementation of *Initiate*, *Accept* and *Recoup* is straightforward, based on Alg. 1.

In the *Exhaust* function, the only novel element is the gas exhaustion through null operations. We implement this by looping sufficiently many times to ensure the exhausted gas matches its target. Our implementation results in a difference between the target g_{alloc} and actual consumed gas of up to 120 gas units – 4 orders of magnitude lower than practical values of g_{alloc}.

Finally, the contract pinnacle, the *Apply* function, is implemented using the aforementioned meta-transaction mechanism. It accepts a meta transaction, issued by *Seller*, verifies it is signed by *Buyer*, and then executes it. The signature verification is performed using a prevalent Ethereum cryptographic library [160]. Note this requires *Buyer* to create her transaction $tx_{payload}$ in a format fitting this design.

EIP1559 Compatibility Recall the payment for $tx_{payload}$ confirmed by LedgerHedger is *payment*, and it does not need to pay an additional fee. In principle, we could have had $tx_{payload}$ offer no fee, and let *Seller* confirm it as an ordinary transaction. However, Ethereum's EIP1559 [83,127] requires that all transactions in a block pay a minimal, *base fee*. Our implementation is compatible with EIP1559 since $tx_{payload}$ is a meta transaction, and *Seller*'s transaction that invokes the *Apply* pays the required base fee.

Deployment and Gas Costs We deploy LedgerHedger on the Ethereum Goerli test network [161], and invoke all its functions. We bring the transaction identifiers in Appendix I.

We initiate the contract using the *Initiate* function three times, and conclude it differently after each initiation.

The first initiation consumed $g_{init} = 117e3$ gas. We then concluded the contract using the *Recoup* function, consuming $g_{done} = 57.3e3$ gas.

The second initiation consumed $g_{init} = 37.4e3$ gas, followed by an invocation of the *Accept*, consuming $g_{accept} = 50e3$ gas, and then an invocation of *Exhaust*, consuming $g_{alloc} + g_{done} = 3.021e6$ gas. Using a local profiler, we found that $g_{done} = 21e3$, aligned with this experiment's chosen $g_{alloc} = 3e6$ value.

Finally, we initiated the contract for the third time, consuming $g_{init} = 37.4e3$ gas, again invoked *Accept* for $g_{accept} = 50e3$ gas. Then, we invoked the *Apply* function on an arbitrary meta-transaction that we created, consuming $g_{pub} + g_{done} = 2.668e6$ gas. Again, using a local profiler, we find that $g_{done} = 12e3$.

Note that the first initiation required $2.5X$ gas compared to the second and third initiations. This discrepancy is due to Ethereum operations consuming gas as a function of their state changes, e.g., setting a value to an unassigned variable is more gas-consuming than assigning a value to an already-assigned one. The first initiation higher costs can therefore be considered as part of the deployment.

To conclude, our LedgerHedger implementation incurs an (amortized) overhead of $g_{init} = 37.4e3$ gas on *Buyer*, and $g_{accept} + g_{done} = 62e3$ gas on *Seller* in the desired execution. These are 3 orders of magnitude lower than a representative example of an applicable hedging use-case of $g_{alloc} = 10e6$ gas [121].

B Gas Allocation Assurances

As mentioned (§2.2), we consider *Seller* to have a gas allocation of g_{alloc} in the required block interval. This modeling trivially fits ledger systems where the system validators (miners) are chosen in advance, such as planned Central Bank Digital Currencies (CBDCs) [100, 101].

We now show this modeling also applies to systems where miners are chosen probabilistically. We begin by first considering practical parameters, showing that *Seller* manages to create a block with overwhelming probability. Conservatively, consider a short interval of a one hour (cf., Optimistic roll-ups like Optimism [50] and Arbitrum [49] that use week-long intervals). For Ethereum, in one hour interval there are about 240 blocks, and the probability that a 10% miner would fail to create any block in that interval is $(1-0.1)^{240} \approx 10^{-11}$. A 5% miner would reach the same probability in about two hours. These values mean failing to find a single block is expected to occur only once in a few million years. We emphasize that in a probabilistic system we do not expect a miner to reserve all her expected future blocks, i.e., miners will retain margins of their reservations.

Finally, we emphasize that a *Seller* does not need to create a block by herself to begin with, as she can have the $tx_{payload}$ confirmed (the action denoted by a_{apply}) by paying the required *gas-price*, regardless of her block-creation capabilities and regardless of random events occurring or not. Moreover, all of *Seller*'s possible interactions with LedgerHedger do not require *Seller* creating a block by herself, and therefore can all be performed even by non-mining entities. It immediately follows that any mining or non-mining *Seller* can simply use the aforementioned transaction-fee mechanism to fulfill the contract as required.

C Price-Prediction-Model Validation

We compare Ethereum past *gas-price* measurements with a normal distribution, validating the random walk prediction model (§2.3).

First, we use Blockchair [162] to obtain measurements of Ethereum's blocks for September 2021, chosen arbitrarily. During this period, about 200K blocks (numbered 13136427 to 13330089) were created, for which we consider the *gas-price* as the ratio of the total paid fees and the total consumed gas (while ignoring empty blocks).

Then, we find the *gas-price* difference between each two consecutive blocks; the hypothesis is that these differences follow a normal distribution, i.e., they are each independently drawn from $N\left(\mu, \sigma^2\right)$, for some μ and σ^2 values.

To mitigate effects of long-lasting trends (e.g., *gas-price* increases at US daytime, where there is generally higher volume of trade and therefore higher demand), we split our samples to batches of 20 blocks, corresponding to an expected time period of 5 minutes. For each batch we numerically find μ and σ^2 values that maximizes the *p-value* for the Kolmogorov–Smirnov test [109], i.e., values of μ and σ^2 that maximize the probability that the *gas-price* change is drawn from $N\left(\mu, \sigma^2\right)$. We present histogram of the resultant *p-values* (significance levels) in Figure 5.

Figure 5 shows that, indeed, *gas-price* fluctuations for most of the examined batches can be modeled as drawn from a normal distribution with high probabil-

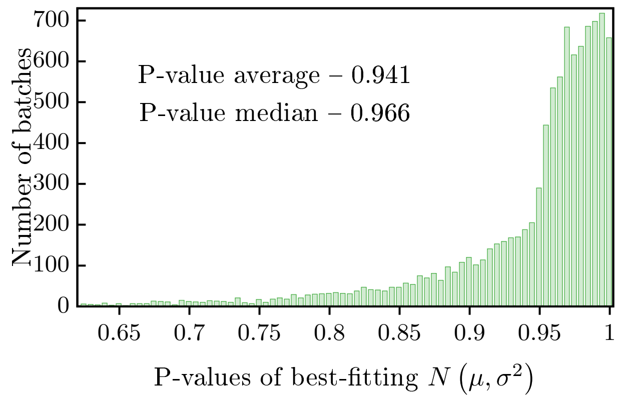


Fig. 5: Kolmogorov–Smirnov test p-values for September 2021 Ethereum blocks and normal distributions.

ity, thus justifying the *gas-price* random walk model. Specifically, 99.8% of the batches are normally distributed with significance level of at least 0.5, 90.4% of batches are normally distributed with significance level of at least 0.85, and 66% of the batches are normally distributed with significance level of at least 0.95. Additionally, we note the average p-value is 0.941, and the median is 0.966, both indicating statistical significance that the samples were drawn from a normal distribution, verifying the hypothesis.

Finally, we consider the found normal distribution parameters μ and σ^2, presented (excluding a few outliers) in Figure 6.

Figure 6 shows the vast majority of batches are best-fitted with $\mu \approx 0$ and relatively low σ^2 values. Indeed, 98% of the examined batches are best-fitted with $\mu \in [-1, 1]$ and $\sigma^2 \leq 5$.

Repeating this analysis for different batch sizes (10, 40 and 80) yields similar results. We thus conclude that the random walk model describes with statistical significance the *gas-price* changes over the sampled period, and that each step has little drift, if any, and low variance.

D Modifications

We present a few modifications to LedgerHedger that might be of practical interest. These focus on user experience in case of unintended usage, e.g., enabling *Buyer* to get the contract tokens earlier in case of no *Seller* accepting the contract. We also present a few modifications for reducing the contract overhead.

Enabling earlier refunds from a declined contract First, one can consider a modification the *Recoup* function requires to be invoked after b_{acc} instead of during $[b_{start}, b_{end}]$. This allows *Buyer* to withdraw tokens from a declined contract at an earlier stage.

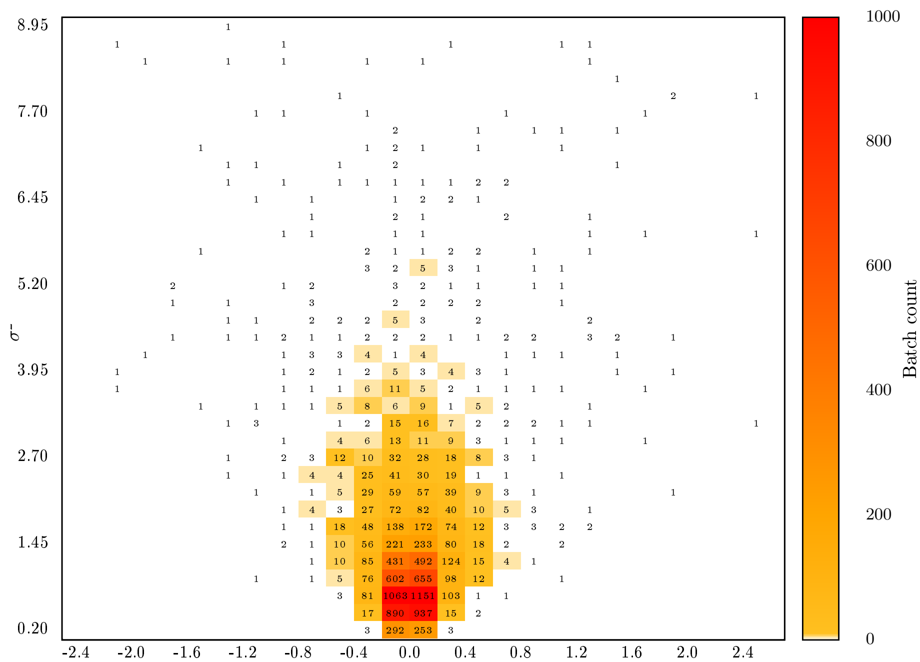


Fig. 6: Best-fitted μ and σ^2 values for September 2021 Ethereum blocks.

Note that initiating a contract that will not be accepted is not an SPE – *Buyer* pays the initiation fees, and then later either forfeits her tokens or pays additional fees to withdraw them (Eq. 8).

Enabling refund from a non-depleted contract Additionally, we can change the *Recoup* function to accept invocations after b_{end} if *Seller* accepted the contract, but then ignored it.

This allows *Buyer* to withdraw tokens in case *Seller* crashed. Similarly to the previous refund modification, initiating a contract that will be refunded is not an SPE.

Higher ε values Setting $\varepsilon = 1$ suffices to incentivize *Seller* to prefer confirming $tx_{payload}$ (assuming she meets the g_{alloc} quota). *Buyer* setting higher values for ε further improves this incentive, even in the presence of *Seller* having exogenous considerations for excluding $tx_{payload}$.

Setting $\varepsilon = 0$ Setting $\varepsilon = 0$ means *Buyer* has to pay (a single token) less for $tx_{payload}$. This, however, means *Seller* has the same benefit from confirming $tx_{payload}$ and from exhausting the contract (see Table .1). This change might

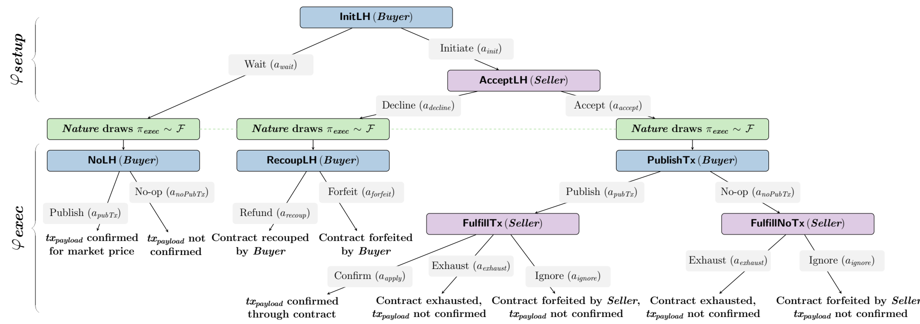


Fig. 7: Γ game states, actions, and conclusion.

be suitable if *Seller* is expected to prefer the former due to an exogenous consideration or due to being benign.

Hard coding block intervals Our implementation takes as parameters b_{start} and b_{end}, indicating the block interval for transaction confirmation or contract exhaustion. However, this requires storing two values, and storing data is a rather costly operation [2]. Instead, one can create a contract with a hard coded interval length, and take only b_{start} as a parameter. This still enables enforcing the engagement interval, but requires storing one less variable.

E Game Definition and Analysis

The need of *Buyer* to confirm a future transaction, *Seller* having a future gas allocation, and the existence of LedgerHedger contract, all give rise to a game played by *Buyer* and *Seller*. The game, denoted by Γ, begins when the blockchain is at the block preceding b_{init}, and progresses with the players taking *actions*.

We present the possible *game states* and *actions* (§E.1), and consider player *strategies* (§E.2). We then specify the solution concept (§E.3): We consider a *subgame perfect equilibrium (SPE)*, capturing the dynamic, turn-based nature of the game. We continue to express the equilibrium strategy as a function of the distribution, the utility functions, and the contract parameters, and prove Theorem 1, showing there are scenarios where engaging and fulfilling the contract is an SPE (§E.4).

E.1 States and Actions

The game takes place during two *phases*. The first phase, denoted by φ_{setup}, describes the creation of blocks b_{init} to b_{acc}. The second phase, denoted by φ_{exec}, describes the creation of blocks b_{start} to b_{end}.

The *game state* comprises the player tokens, the contracts they possibly engage with, their published transactions, the current phase, and the *gas-price*. Figure 7 summarizes the game progress.

Broadly speaking, *Buyer* and *Seller* can set a LEDGERHEDGER contract at the game start for φ_{exec}, and then execute it. Alternatively, *Buyer* and *Seller* can wait for φ_{exec}, and then *Buyer* can publish $tx_{payload}$ as any other transaction for confirmation, and *Seller* can use her gas allocation to confirm any transaction.

We ignore nonsensical, obviously dominated or unrelated actions [163] such as either party sharing her private key, *Seller* not using her gas allocation, or either player publishing unrelated transactions. We assume both parties initially have sufficiently many tokens to support the following actions.

The value of π_{setup} is known to *Buyer* and *Seller* at the game beginning. However, the value of π_{exec} is drawn by *Nature* from $\mathcal{F}$ just before φ_{exec} starts. After the players publish and confirm transactions for φ_{exec} the game is concluded.

The game starts in state *InitLH* (in φ_{setup}), where *Buyer* can choose to initiate a LEDGERHEDGER instance (action a_{init}), and choose its parameters. She incurs the initiation cost $g_{init} \cdot \pi_{setup}$, deposits the payment $payment + \varepsilon$, and the game transitions to state *AcceptLH*. Alternatively, she can choose to refrain from initiating (action a_{wait}), incurring no costs, and the game transitions to game state *NoLH*.

Game state *NoLH* (in φ_{exec}) takes place after *Nature* draws $\pi_{exec} \sim \mathcal{F}$. In this state, *Buyer* can pay the *gas-price* π_{exec} to have $tx_{payload}$ confirmed (action a_{pubTx}), incurring the fee cost $g_{alloc} \cdot \pi_{exec}$, but have $tx_{payload}$ confirmed. Alternatively, she can do nothing, incurring no costs, but receiving no reward. *Seller* sells her g_{alloc} gas for the *gas-price* π_{exec}, earning $g_{alloc} \cdot \pi_{exec}$.

In game state *AcceptLH* (φ_{setup}) *Seller* chooses whether to accept the LEDGERHEDGER instance (action a_{accept}). To accept, *Seller* publishes a transaction that invokes the *Accept* function, deposits the *col* collateral tokens, and incurs a cost of $g_{accept} \cdot \pi_{setup}$. The game then transitions to state *PublishTx*. Alternatively, she can decline by simply ignoring it ($a_{decline}$), leading to *RecoupLH*.

Game state *RecoupLH* is in φ_{exec}, after *Nature* draws $\pi_{exec} \sim \mathcal{F}$. *Buyer* can choose to withdraw her deposited $payment + \varepsilon$ tokens from the declined LEDGERHEDGER instance (action a_{recoup}), incurring the withdrawal transaction fee cost $g_{done} \cdot \pi_{exec}$. If not, she can simply ignore it (action $a_{forfeit}$), forfeiting the $payment + \varepsilon$ tokens. As in *NoLH*, *Buyer* can also publish $tx_{payload}$, and *Seller* can also confirm other transactions; the former costs *Buyer* $g_{alloc} \cdot \pi_{exec}$ tokens, but has $tx_{payload}$ confirmed, and the latter rewards *Seller* with $g_{alloc} \cdot \pi_{exec}$.

Game state *PublishTx* is in φ_{exec}, after *Nature* draws $\pi_{exec} \sim \mathcal{F}$. Here *Buyer* can publish transactions for *Seller* to confirm using the contract's *Apply* function. These transactions do need not to further incentivize a miner to confirm them, hence offer no fee. However, *Buyer* can publish multiple transactions for *Seller* to choose from, and *Seller* is clearly incentivized to consider only the transaction requiring the least gas. So, we consider the following two cases. First, *Buyer* chooses not to publish a transaction at all (action $a_{noPubTx}$), incurring no costs, leading to *FulfillNoTx*. Alternatively, *Buyer* publishes $tx_{payload}$ (action a_{pubTx}), leading to the *FulfillTx* state.

In game states *FulfillNoTx* and *FulfillTx* (φ_{exec}) *Seller* can choose to invoke the contract's *Exhaust* function (action $a_{exhaust}$). This transfers $payment + col$

tokens to *Seller*, but requires g_{alloc} for the null operations and g_{done} gas for the remaining operations (verification, token transfer, etc.). Note this action exceeds the g_{alloc} quota of *Seller*, requiring *Seller* to pay fees for g_{done}, resulting in an incurred cost of $g_{done} \cdot \pi_{exec}$. Action $a_{exhaust}$ results with $tx_{payload}$ not confirmed, so *Buyer* can have it included by paying the *gas-price*.

Alternatively, *Seller* can choose to ignore the contract (action a_{ignore}), receiving no tokens but incurring no additional costs. Action a_{ignore} results with *Seller* not using her gas, which she can sell for the *gas-price* of π_{exec}. It also results with $tx_{payload}$ not being confirmed through the contract, so *Buyer* can pay the current *gas-price* π_{exec} to have it confirmed.

Finally, in $\mathit{FulfillTx}$, *Seller* can choose to invoke the contract's *Apply* function, using the published transaction $tx_{payload}$. This rewards *Seller* with $payment + \varepsilon + col$ tokens, but requires $g_{pub} + g_{done}$ gas, resulting in an incurred cost of $((g_{pub} + g_{done}) - g_{alloc}) \cdot \pi_{exec}$.

Note 1. We assume that *Seller* verifies the execution of $tx_{payload}$ and is content with its results (as in, e.g., [164–166]). Namely, *Seller* verifies $tx_{payload}$ does not terminate the contract nor transfer away its funds.

Note 2. LEDGERHEDGER works whether *Seller* is a miner or not: If she is a miner she can use some of her block's gas to confirm $tx_{payload}$ and get the contract tokens, forfeiting other transactions that pay the market price (cost of loss-of-opportunity); if she is not, she can confirm $tx_{payload}$ and get the contract tokens by publishing a transaction that pays the *gas-price* to a miner (cost of the transaction fee). In both cases, the cost is identical, resulting in a similar game-theoretic analysis.

As in the NoLH and the $\mathit{RecoupLH}$ states, if $tx_{payload}$ is not confirmed by *Seller* as part of the contract (i.e., if *Seller* plays $a_{exhaust}$ or a_{ignore}), then *Buyer* can pay to have $tx_{payload}$ included at market price, resulting with $tx_{payload}$ confirmed and a cost of $g_{alloc} \cdot \pi_{exec}$. Any of these actions concludes the game.

E.2 Strategy

Each player has a *strategy*, mapping each game state to an action. The action space for *Buyer* comprises which transactions to publish and when to do so. For *Seller*, it comprises which transactions to publish, when to publish them, and which transactions to confirm using her allotted gas.

We denote by $\bar{s}$ a *strategy profile*, comprising the strategies of *Buyer* and *Seller*. We denote $\bar{s}(\mathit{state}) = a$ if the player's strategy in the profile $\bar{s}$ dictates playing action a in game state state. We say a player *follows* strategy profile $\bar{s}$ if at each game state she chooses to play her strategy's mapped action.

E.3 Solution Concept

The sequential nature of Γ lends itself to the definition of *subgames*, each capturing the possible extensions starting from a specific state. We denote by $\Gamma^{\mathit{player}}_{\mathit{state}}$

the subgame starting at state state where $player \in \{Buyer, Seller\}$ is to take an action. The game begins with the initial subgame $\Gamma^{Buyer}_{\mathsf{InitLH}} = \Gamma$.

We can therefore define the wealth and utility of each player starting in a subgame as follows. Let *Buyer* and *Seller* follow a strategy profile $\bar{s}$ in subgame $\Gamma^{player}_{\mathsf{state}}$, and let *Nature* draw *gas-price* π_{exec}. We denote the resultant wealth of *Buyer* and of *Seller* by $W_{Buyer}(\pi_{exec}, \mathsf{state}, \bar{s})$ and by $W_{Seller}(\pi_{exec}, \mathsf{state}, \bar{s})$, respectively. We denote the utility of *Buyer* by $U_{Buyer}(W_{Buyer}(\pi_{exec}, \mathsf{state}, \bar{s}))$ and of *Seller* by $U_{Seller}(W_{Seller}(\pi_{exec}, \mathsf{state}, \bar{s}))$, or simply $U_{Buyer}(\mathsf{state}, \bar{s})$ and $U_{Seller}(\mathsf{state}, \bar{s})$ for succinctness. We denote the expected utility of *Buyer* and *Seller* when they follow strategy profile $\bar{s}$ starting in $\Gamma^{player}_{\mathsf{state}}$, over the distribution $\mathcal{F}$, by $\mathbb{E}[U_{Buyer}(\mathsf{state}, \bar{s})]$ and by $\mathbb{E}[U_{Seller}(\mathsf{state}, \bar{s})]$, respectively.

We focus on rational *Buyer* and *Seller* that strive to maximize their expected utility. We assume the players' utility functions, their utility-maximizing tendencies, and the game state are all common knowledge. So, the defined game is of *perfect information* [167,168].

We are interested in a strategy profile that is a *subgame perfect equilibrium (SPE)* [163,169–175]. Intuitively, this means that at any stage of the game both players are content with the action defined in the strategy profile. Formally, an SPE is a strategy profile where no player can increase her utility by deviating in any subgame, considering the other player's reaction to such deviation, i.e., Nash equilibrium at every subgame.

We are interested in finding conditions in which the SPE, denoted by $\bar{s}_{\text{spe}}$, results with *Buyer* initiating the contract, *Seller* accepting it, *Buyer* publishing $tx_{payload}$ with $g_{pub} = g_{alloc}$, and *Seller* confirming it.

The common method for finding $\bar{s}_{\text{spe}}$ is using *backward induction* [169,176–178], applicable in perfect information and finite games. The analysis begins at the subgames comprising only the last action (e.g., subgames $\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$ and $\Gamma^{Seller}_{\mathsf{FulfillTx}}$), where the SPE is found by directly comparing the utility from the different possible actions. Then, considering the last player chooses that utility-maximizing action, the second to last subgames are analyzed (e.g., subgame $\Gamma^{Buyer}_{\mathsf{PublishTx}}$). This process is repeated recursively until the initial subgame $\left(\Gamma^{Buyer}_{\mathsf{InitLH}} = \Gamma\right)$ is analyzed. We move forward to finding $\bar{s}_{\text{spe}}$ in Γ.

E.4 SPE Expressions

We start by expressing the SPE for an initiated and accepted contract, and then address the initiation and acceptance.

Fulfilling an Initiated and Accepted Contract The subgame describing possible interactions with the initiated and accepted contract is $\Gamma^{Buyer}_{\mathsf{PublishTx}}$, which is played after *Nature* had already drawn π_{exec}. Therefore, any choice of action in $\Gamma^{Buyer}_{\mathsf{PublishTx}}$ and the subsequent subgames results in deterministic wealth for both *Buyer* and *Seller*.

It follows that maximizing the expected utility (by choosing preferable actions) is the same as maximizing the utility. Additionally, since utility functions are monotonic, maximizing the utility is equivalent to maximizing wealth.

Following this observation, we compare the resultant wealth of each action in $\Gamma^{Buyer}_{\mathsf{PublishTx}}$ and the subsequent subgames, presenting a condition on π_{exec}, by which the $\bar{s}_{\text{spe}}$ action is decided.

Throughout the analysis, we assume $\varepsilon = 1$, that is, a single token (we consider different ε values in Appendix D).

Towards the upcoming resultant wealth analysis, recall that in the subgames preceding $\Gamma^{Buyer}_{\mathsf{PublishTx}}$, *Buyer* already incurred a cost of $g_{init} \cdot \pi_{setup} + \mathit{payment} + \varepsilon$ for initiating the contract, and *Seller* incurred a cost of $g_{accept} \cdot \pi_{setup} + \mathit{col}$ for accepting the contract.

The available actions in $\Gamma^{Buyer}_{\mathsf{PublishTx}}$ are a_{pubTx}, leading to $\Gamma^{Seller}_{\mathsf{FulfillTx}}$, or $a_{noPubTx}$, leading to $\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$. We begin by considering these two subgames, and present their analysis summary in Table 1.

Subgame $\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$ In the $\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$ subgame *Seller* plays either $a_{exhaust}$ or a_{ignore}.

Playing $a_{exhaust}$ results with *Seller* exhausting the contract's gas, rewarding *Seller* with $\mathit{payment} + \mathit{col}$ at the incurred cost of $g_{done} \cdot \pi_{exec}$. Alternatively, playing a_{ignore} results with *Seller* forfeiting the contract tokens, but selling her gas for the *gas-price*, that is, a reward of $g_{alloc} \cdot \pi_{exec}$.

It follows $a_{exhaust}$ is preferred over a_{ignore} if $\mathit{payment} + \mathit{col} - g_{done} \cdot \pi_{exec} > g_{alloc} \cdot \pi_{exec}$, and the resultant wealth of *Seller* in this subgame is therefore

$$\begin{aligned} W_{Seller}\left(\pi_{exec}, \mathsf{FulfillNoTx}, \bar{s}_{\text{spe}}\right) = {} & w^{init}_{Seller} - g_{accept} \cdot \pi_{setup} \\ & + \max(\mathit{payment} - g_{done} \cdot \pi_{exec}, g_{alloc} \cdot \pi_{exec} - \mathit{col}) . \end{aligned} \tag{1}$$

Regardless of the action *Seller* chooses, *Buyer* can pay the *gas-price* π_{exec} for her transaction inclusion. The cost for that is $g_{alloc} \cdot \pi_{exec}$ with a reward of w^{exo}. This is profitable as long as $w^{exo} > g_{alloc} \cdot \pi_{exec}$, resulting with

$$\begin{aligned} W_{Buyer}\left(\pi_{exec}, \mathsf{FulfillNoTx}, \bar{s}_{\text{spe}}\right) = {} & w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - \mathit{payment} - \varepsilon \\ & + \max\left(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0\right) . \end{aligned} \tag{2}$$

Subgame $\Gamma^{Seller}_{\mathsf{FulfillTx}}$ In the $\Gamma^{Seller}_{\mathsf{FulfillTx}}$ subgame *Seller* plays either a_{apply}, $a_{exhaust}$ or a_{ignore}.

Playing either $a_{exhaust}$ or a_{ignore} results with the same wealth as playing them in $\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$. However, playing a_{apply} includes the published $\mathit{tx}_{payload}$ transaction with its gas requirement g_{pub}, resulting with a reward of $\mathit{payment} + \varepsilon + \mathit{col}$. However, it also results with a cost of $g_{done} \cdot \pi_{exec}$, and an additional $(g_{pub} - g_{alloc}) \cdot \pi_{exec}$; note the latter is positive if $g_{alloc} < g_{pub}$, that is, if $\mathit{tx}_{payload}$ exceeds the agreed quota g_{alloc}, or negative if $\mathit{tx}_{payload}$ under-utilizes it, leaving gas for *Seller* to sell, and thus netting a positive reward.

Comparing a_{apply} and $a_{exhaust}$, we get a_{apply} is preferred if $\varepsilon > (g_{pub} - g_{alloc}) \cdot \pi_{exec}$. As $\varepsilon = 1$, $\pi_{exec} > 0$, and $(g_{pub} - g_{alloc}) \cdot \pi_{exec}$ is a number of tokens (i.e., an integer), this inequality holds if $g_{pub} \leq g_{alloc}$. Similarly, comparing a_{apply} and a_{ignore} results with the former yielding more tokens if $\pi_{exec} < \frac{payment+col+\varepsilon}{g_{pub}+g_{done}}$.

The resultant wealth of *Seller* in this subgame is therefore

$$W_{Seller}\left(\pi_{exec}, \mathsf{FulfillTx}, \bar{s}_{\mathrm{spe}}\right) = w^{init}_{Seller} - g_{accept} \cdot \pi_{setup} + \max\bigl(g_{alloc} \cdot \pi_{exec} - col, payment - g_{done} \cdot \pi_{exec}, payment + \varepsilon - (g_{done} + g_{pub} - g_{alloc}) \cdot \pi_{exec}\bigr)\,. \quad (3)$$

If *Seller* chooses not to confirm $tx_{payload}$, then *Buyer* can pay the *gas-price* π_{exec} for her transaction inclusion. The cost for that is $g_{alloc} \cdot \pi_{exec}$ with a reward of w^{exo}. This is preferred as long as $w^{exo} > g_{alloc} \cdot \pi_{exec}$, resulting with

$$W_{Buyer}\left(\pi_{exec}, \mathsf{FulfillTx}, \bar{s}_{\mathrm{spe}}\right) = w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + \begin{cases} w^{exo}, & \pi_{exec} < \frac{payment+col+\varepsilon}{g_{pub}+g_{done}} \text{ and } g_{pub} \leq g_{alloc} \\ \max\left(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0\right), & \text{otherwise} \end{cases}. \quad (4)$$

We are now ready to consider the $\Gamma^{Buyer}_{\mathsf{PublishTx}}$ subgame.

Subgame $\Gamma^{Buyer}_{\mathsf{PublishTx}}$ In this subgame, *Buyer* chooses whether to publish $tx_{payload}$, and with what gas requirement g_{pub}. We present the following lemma, providing an upper bound for *gas-price* π_{exec} such that *Buyer* is strictly incentivized to publish $tx_{payload}$ with $g_{pub} = g_{alloc}$:

Lemma 1. *If* $\pi_{exec} < \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}}$ *then* $\bar{s}_{spe}\left(\mathsf{PublishTx}\right) = a_{pubTx}$*, satisfying* $g_{pub} = g_{alloc}$.

Intuitively, *Buyer* publishing a transaction with gas consumption $g_{pub} > g_{alloc}$ disincentivizes *Seller* to confirm it. But, by definition, the transaction of *Buyer* yields no value to her if $g_{pub} < g_{alloc}$, resulting with the optimal gas consumption being $g_{pub} = g_{alloc}$. Additionally, meeting the π_{exec} bound results with *Seller* confirming the published transaction, incentivizing *Buyer* to publish it to begin with.

Proof (Lemma 1). In the $\mathsf{PublishTx}$ subgame, *Buyer* chooses if to publish $tx_{payload}$ or not. Additionally, if she chooses to publish $tx_{payload}$ then she also decides what its gas consumption g_{pub} is.

Publishing $tx_{payload}$ (a_{pubTx}) leads to subgame $\Gamma^{Seller}_{\mathsf{FulfillTx}}$. If so, her resultant wealth is $W_{Buyer}\left(\pi_{exec}, \mathsf{FulfillTx}, \bar{s}_{\mathrm{spe}}\right) = w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + w^{exo}$ if $\pi_{exec} < \frac{payment+col+\varepsilon}{g_{pub}+g_{done}}$ and $g_{pub} \leq g_{alloc}$, and $w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + \max\left(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0\right)$ otherwise (Eq. 4).

Alternatively, not publishing a transaction ($a_{noPubTx}$), leads to subgame $\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$. This results with wealth $W_{Buyer}\left(\pi_{exec}, \mathsf{FulfillNoTx}, \bar{s}_{\mathrm{spe}}\right) = w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + \max\left(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0\right)$ (Eq. 2).

Let us take note that $w^{exo} > 0$, $\pi_{exec} > 0$ and $g_{alloc} > 0$. Therefore, we get that $w^{exo} > \max(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0)$. Subsequently, considering all the aforementioned options, the wealth of *Buyer* is maximized when $\pi_{exec} < \frac{payment+col+\varepsilon}{g_{pub}+g_{done}}$ and $g_{pub} \leq g_{alloc}$.

With that, let us consider the value of g_{pub}. First, setting $g_{pub} > g_{alloc}$ violates the mentioned condition, as *Seller* will not confirm $tx_{payload}$.

And, setting $g_{pub} < g_{alloc}$ is also unfavorable, as $g_{pub} \geq g_{alloc}$ is required to receive the w^{exo} tokens to begin with. Thus, publishing a transaction that requires exactly $g_{pub} = g_{alloc}$ is the preferred action.

When $g_{pub} = g_{alloc}$, we get the condition for the preferable outcome is simply $\pi_{exec} < \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}}$, which is exactly the condition mentioned in the lemma. □

Following Lemma 1, if the *gas-price* satisfies $\pi_{exec} < \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}}$ then *Seller* confirms $tx_{payload}$, and we get the resultant wealth of the $\Gamma^{Seller}_{\mathsf{FulfillTx}}$ subgame (see Eq. 3 and Eq. 4). However, if *gas-price* exceeds $\pi_{exec} > \frac{payment+col+\varepsilon}{g_{pub}+g_{done}}$ then *Seller* does not confirm $tx_{payload}$. In that case, *Seller* chooses between exhausting or ignoring the contract, and the resultant wealth is that of the $\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$ subgame (see Eq. 1 and Eq. 2). Therefore, we get

$$W_{Seller}(\pi_{exec}, \mathsf{PublishTx}, \bar{s}_{\rm spe}) = \begin{cases} W_{Seller}(\pi_{exec}, \mathsf{FulfillTx}, \bar{s}_{\rm spe}), & \pi_{exec} < \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}} \\ W_{Seller}(\pi_{exec}, \mathsf{FulfillNoTx}, \bar{s}_{\rm spe}), & \pi_{exec} \geq \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}} \end{cases}, \quad (5)$$

and

$$W_{Buyer}(\pi_{exec}, \mathsf{PublishTx}, \bar{s}_{\rm spe}) = \begin{cases} W_{Buyer}(\pi_{exec}, \mathsf{FulfillTx}, \bar{s}_{\rm spe}), & \pi_{exec} < \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}} \\ W_{Buyer}(\pi_{exec}, \mathsf{FulfillNoTx}, \bar{s}_{\rm spe}), & \pi_{exec} \geq \frac{payment+col+\varepsilon}{g_{alloc}+g_{done}} \end{cases}. \quad (6)$$

In conclusion, Lemma 1 presents the required conditions for the SPE to include the publication and confirmation of $tx_{payload}$. We now proceed to express the conditions for initiation and acceptance.

***Seller* Accepting** We start with analyzing the contract acceptance, that is, with subgame $\Gamma^{Seller}_{\mathsf{AcceptLH}}$. In this subgame, *Seller* can play a_{accept}, leading to subgame $\Gamma^{Buyer}_{\mathsf{PublishTx}}$, discussed in Lemma 1. She can also play $a_{decline}$, leading to subgame $\Gamma^{Buyer}_{\mathsf{RecoupLH}}$, which we analyze below.

Subgame $\Gamma^{Buyer}_{\mathsf{RecoupLH}}$ In the $\Gamma^{Buyer}_{\mathsf{RecoupLH}}$ subgame, *Buyer* plays either a_{recoup} or $a_{forfeit}$.

Playing a_{recoup} results with *Buyer* getting *payment* $+ \varepsilon$ and spending $g_{done} \cdot \pi_{exec}$ tokens. Alternatively, she can play $a_{forfeit}$, not getting or spending any tokens. She can also publish $tx_{payload}$ for $w^{exo} - g_{alloc} \cdot \pi_{exec}$. Either way, *Seller* gets $g_{alloc} \cdot \pi_{exec}$ for her gas allocation.

It follows a_{recoup} is preferred over $a_{forfeit}$ if $payment + \varepsilon > g_{done} \cdot \pi_{exec}$. The resultant wealth of *Seller* is

$$W_{Seller}\left(\pi_{exec}, \mathsf{RecoupLH}, \bar{s}_{\mathrm{spe}}\right) = w^{init}_{Seller} + g_{alloc} \cdot \pi_{exec}\ , \tag{7}$$

and of *Buyer* is

$$\begin{aligned} W_{Buyer}\left(\pi_{exec}, \mathsf{RecoupLH}, \bar{s}_{\mathrm{spe}}\right) &= w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} \\ &+ \max\left(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0\right) \\ &+ \max\left(-g_{done} \cdot \pi_{exec}, -payment - \varepsilon\right) \end{aligned} . \tag{8}$$

We are now ready to analyze the $\Gamma^{Seller}_{\mathsf{AcceptLH}}$ subgame.

Subgame $\Gamma^{Seller}_{\mathsf{AcceptLH}}$ Recall this is played in φ_{setup}, before π_{exec} is drawn, so *Seller* chooses the action that maximizes her expected utility.

She can either play a_{accept}, resulting with

$$\mathbb{E}\left[U_{Seller}\left(\mathsf{PublishTx}, \bar{s}_{\mathrm{spe}}\right)\right] = \int_{-\infty}^{\infty} U_{Seller}\left(\mathsf{PublishTx}, \bar{s}_{\mathrm{spe}}\right) \cdot \mathcal{F}_{\mathrm{pdf}}\left(\pi_{exec}\right) d\pi_{exec}\ , \tag{9}$$

or play $a_{decline}$, resulting with

$$\mathbb{E}\left[U_{Seller}\left(\mathsf{RecoupLH}, \bar{s}_{\mathrm{spe}}\right)\right] = \int_{-\infty}^{\infty} U_{Seller}\left(\mathsf{RecoupLH}, \bar{s}_{\mathrm{spe}}\right) \cdot \mathcal{F}_{\mathrm{pdf}}\left(\pi_{exec}\right) d\pi_{exec}\ . \tag{10}$$

Let us denote the *expected utility difference (EUD)* of *Seller* by

$$EUD_{Seller} = \mathbb{E}\left[U_{Seller}\left(\mathsf{PublishTx}, \bar{s}_{\mathrm{spe}}\right)\right] - \mathbb{E}\left[U_{Seller}\left(\mathsf{RecoupLH}, \bar{s}_{\mathrm{spe}}\right)\right]\ .$$

The following corollary therefore details the condition for *Seller* to accept the contract:

Corollary 1. *In* $\Gamma^{Seller}_{\mathsf{AcceptLH}}$*, if* $EUD_{Seller} > 0$ *then* $\bar{s}_{spe}\left(\mathsf{AcceptLH}\right) = a_{accept}$*, and if* $EUD_{Seller} \le 0$ *then* $\bar{s}_{spe}\left(\mathsf{AcceptLH}\right) = a_{decline}$*.*

Corollary 1 presents the contract acceptance condition, as discussed in Theorem 1. It also allows us to draft the expected utility of *Buyer* in $\Gamma^{Seller}_{\mathsf{AcceptLH}}$ in the following equation:

$$\mathbb{E}\left[U_{Buyer}\left(\mathsf{AcceptLH}, \bar{s}_{\mathrm{spe}}\right)\right] = \begin{cases} \mathbb{E}\left[U_{Buyer}\left(\mathsf{PublishTx}, \bar{s}_{\mathrm{spe}}\right)\right], & EUD_{Seller} > 0 \\ \mathbb{E}\left[U_{Buyer}\left(\mathsf{RecoupLH}, \bar{s}_{\mathrm{spe}}\right)\right], & EUD_{Seller} \le 0 \end{cases} . \tag{11}$$

***Buyer* Initiating** It remains to consider the conditions for contract initiation being an SPE for *Buyer*. The subgame describing this decision is $\Gamma^{Buyer}_{\mathsf{InitLH}}$, where *Buyer* decides whether to initiate the contract (a_{init}), leading to $\Gamma^{Seller}_{\mathsf{AcceptLH}}$, or to not initiate ($a_{wait}$), leading to $\Gamma^{Buyer}_{\mathsf{NoLH}}$.

Subgame $\Gamma^{Buyer}_{\mathsf{InitLH}}$ is also before *Nature* draws π_{exec}, so we compare the actions' expected utilities. Eq. 11 gives $\mathbb{E}\left[U_{Buyer}\left(\mathsf{AcceptLH}, \bar{s}_{\mathrm{spe}}\right)\right]$, the expected utility from playing a_{init}.

We now find $\mathbb{E}\left[U_{Buyer}\left(\mathsf{NoLH}, \bar{s}_{\mathrm{spe}}\right)\right]$, the expected utility from playing a_{wait}. For that, we first analyze the $\Gamma^{Buyer}_{\mathsf{NoLH}}$ subgame.

Subgame $\Gamma^{Buyer}_{\mathsf{NoLH}}$ In the $\Gamma^{Buyer}_{\mathsf{NoLH}}$ subgame, *Buyer* can pay $g_{alloc} \cdot \pi_{exec}$ to have $tx_{payload}$ confirmed, receiving w^{exo} tokens. We get

$$W_{Buyer}\left(\pi_{exec}, \mathsf{NoLH}, \bar{s}_{\mathrm{spe}}\right) = w^{init}_{Buyer} + \max\left(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0\right) \text{ , and}$$

$$\mathbb{E}\left[U_{Buyer}\left(\mathsf{NoLH}, \bar{s}_{\mathrm{spe}}\right)\right] = \int_{-\infty}^{\infty} U_{Buyer}\left(\mathsf{NoLH}, \bar{s}_{\mathrm{spe}}\right) \cdot \mathcal{F}_{\mathrm{pdf}}\left(\pi_{exec}\right) d\pi_{exec} \,. \quad (12)$$

We are finally ready to address the full game $\Gamma = \Gamma^{Buyer}_{\mathsf{InitLH}}$.

Subgame $\Gamma^{Buyer}_{\mathsf{InitLH}}$ Given $\mathbb{E}\left[U_{Buyer}\left(\mathsf{NoLH}, \bar{s}_{\mathrm{spe}}\right)\right]$ (Eq. 12) and $\mathbb{E}\left[U_{Buyer}\left(\mathsf{AcceptLH}, \bar{s}_{\mathrm{spe}}\right)\right]$ (Eq. 11), we denote the expected utility difference of *Buyer* by

$$EUD_{Buyer} = \mathbb{E}\left[U_{Buyer}\left(\mathsf{AcceptLH}, \bar{s}_{\mathrm{spe}}\right)\right] - \mathbb{E}\left[U_{Buyer}\left(\mathsf{NoLH}, \bar{s}_{\mathrm{spe}}\right)\right] \,.$$

The following corollary presents the condition for *Buyer* initiating the contract.

Corollary 2. *If* $EUD_{Buyer} > 0$ *then* $\bar{s}_{spe}\left(\Gamma^{Buyer}_{\mathsf{InitLH}}\right) = a_{init}$.

Corollary 2 shows the contract initiation condition, thus concluding the conditions for the SPE to be as detailed in Theorem 1.

It is now easy to see the correctness of Theorem 1. Take any distribution $\mathcal{F}$. By Lemma 1, setting *col* sufficiently high deterministically assures (or assures with high probability for an unbounded distribution) that if LedgerHedger is initiated and accepted, then *Buyer* publishes an adequate $tx_{payload}$ and *Seller* confirms it.

Corollary 1 and Corollary 2 both present conditions for the contract initiation and acceptance – conditions on preferring a predetermined payment over one that changes according to the drawn π_{exec}. Sufficiently risk-averse participants result with both of them preferring a predetermined contract over the drawn price uncertainty.

F Resultant Required Price for Linear Utility Functions

Recall Figure 4a shows that the required prices are fixed for the linear utility function, for both *Buyer* and *Seller*, for any considered $\mathcal{F}$. We thoroughly explain this result.

Broadly speaking, this holds due to $\Pr[\pi_{exec} < \pi_{bound}] = 1$, the linearity of the utility function, and the fact all considered distributions have the same mean value.

First, note that our parameter choice results in $\Pr[\pi_{exec} < \pi_{bound}] = 1$, where $\pi_{bound} = \frac{payment + col + \varepsilon}{g_{alloc} + g_{done}}$ (Lemma 1). So, we get $W_{Seller}\left(\pi_{exec}, \Gamma^{Buyer}_{PublishTx}, \bar{s}_{\text{spe}}\right)$ (Eq. 5) and $W_{Buyer}\left(\pi_{exec}, \Gamma^{Buyer}_{PublishTx}, \bar{s}_{\text{spe}}\right)$ (Eq. 6) are *linear* in π_{exec}. This is in contrast to parameter values where $0 < \Pr[\pi_{exec} < \pi_{bound}] < 1$, resulting in *piece-wise linear* functions of π_{exec}.

Following that, consider the linear utility Linear is also a linear function, so both $U_{Seller}\left(\Gamma^{Buyer}_{PublishTx}, \bar{s}_{\text{spe}}\right)$ and $U_{Buyer}\left(\Gamma^{Buyer}_{PublishTx}, \bar{s}_{\text{spe}}\right)$ are also linear in π_{exec}.

When considering the expected utility (e.g., Eq. 10), the integration is therefore of a linear function. Let us denote that function as $a\pi_{exec} + b$ for some constants a and b, and note that $\int_{-\infty}^{\infty} (a\pi_{exec} + b) \cdot \mathcal{F}_{\text{pdf}}(\pi_{exec})\, d\pi_{exec} = a\int_{-\infty}^{\infty} \pi_{exec} \cdot \mathcal{F}_{\text{pdf}}(\pi_{exec})\, d\pi_{exec} + b\int_{-\infty}^{\infty} \mathcal{F}_{\text{pdf}}(\pi_{exec})\, d\pi_{exec}$.

The result of the first integral $\int_{-\infty}^{\infty} \pi_{exec} \cdot \mathcal{F}_{\text{pdf}}(\pi_{exec})\, d\pi_{exec}$ is the distribution's mean value, which is equal for all our considered distributions. The result of the second integral $\int_{-\infty}^{\infty} \mathcal{F}_{\text{pdf}}(\pi_{exec})\, d\pi_{exec}$ is exactly 1, as $\mathcal{F}_{\text{pdf}}(\pi_{exec})$ is a probability density function.

So, we get that the expected utility from the $\Gamma^{Buyer}_{PublishTx}$ subgame is equal for all distributions. Similar considerations apply to the expected utility from the Γ^{Buyer}_{NoLH} subgame, resulting with these expected utility differences being constant across the distributions, as indicated by Figure 4a.

G Alternative Implementation

The recent implementation of EIP1559 [83,127] introduced a new `GASPRICE` opcode, opening doors to query the current gas price. This development enables us to create an economically equivalent alternative implementation of LedgerHedger.

Instead of relying on *Seller* to publish the transaction, this alternative approach allows LedgerHedger to mandate *Buyer* to publish the transaction, following which *Seller* reimburses *Buyer* for the incurred gas cost. Before the advent of this new opcode, smart contracts lacked the ability to retrieve the current gas price, thereby hindering the calculation of the reimbursement amount.

This alternative route suggests a streamlined implementation of LedgerHedger. It relinquishes the need for *Seller* to publish the transaction and involves no meta transactions. Essentially, it constitutes a straightforward future contract where *Buyer* is reimbursed for the transaction's gas cost at the contract's conclusion.

While this approach retains the same economic outcome and offers simplified implementation, it's not without drawbacks. Firstly, it necessitates that *Buyer* possess sufficient funds to cover the transaction's gas cost. Secondly, the new opcode's limitations restrict its application to a single block, as it doesn't support retrieving the gas price for any block other than the current one. This limitation reduces flexibility, preventing *Seller* from waiting for a more advantageous gas price or for a block that *Seller* has mined. Furthermore, since *Buyer* is responsible for publishing the transaction, she can overpay for the gas to ensure its inclusion. This would cause *Seller* to incur unnecessary loss. Thus, this alternative approach would require some additional mechanism to prevent this.

H LedgerHedger Solidity Implementation

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

struct MetaTx {
    uint256 nonce;
    address to;
    uint256 value;
    bytes callData;
}

enum State {
    INIT,
    REGISTERED,
    IDLE
}

contract GasFuture {
    uint256 public nonce;

    uint32 public startBlock;
    uint32 public endBlock;
    uint32 public regBlock;

    address public buyer;
    address public seller;
    uint256 public gasHedged;

    uint256 public collateral;
    uint256 public payment;
    uint256 public eps;

    State public status;

    constructor(address _owner) public {
        buyer = _owner;
```

```
        status = State.IDLE;
    }

    receive() external payable {}

    function init(
        uint32 _regBlock,
        uint32 _startBlock,
        uint32 _endBlock,
        uint256 _gasHedged,
        uint256 _col,
        uint256 _eps
    ) external payable {
        require(buyer == msg.sender, "Not owner");
        require(block.number <= _regBlock && _regBlock < _startBlock && _startBlock <= _endBlock, "block out of bound");
        // NOTE: Optionally let this be reinitiated if depeleted
        require(status == State.IDLE, "Contract already initialized");
        require(_gasHedged > 0, "Hedged amount can't be negative");
        require(_col >= 0, "Collateral can't be negative");
        require(_eps > 0, "Epsilon can't be negative");
        require(msg.value > eps, "Payment can't be negative");

        regBlock = _regBlock;
        startBlock = _startBlock;
        endBlock = _endBlock;

        gasHedged = _gasHedged;

        eps = _eps;
        payment = msg.value - eps;
        collateral = _col;

        status = State.INIT;
    }

    // The callers of the function sets themselves as the gasPayer
    function register() external payable {
        require(block.number <= regBlock, "Register block expired");
        require(status == State.INIT, "Contract not initialized");
        require(msg.value >= collateral, "Insufficient collateral provided");
        seller = msg.sender;
```

```
        status = State.REGISTERED;
    }

    function refund() external {
        require(block.number >= startBlock && block.number <= endBlock, "Block must be between start and end");
        require(status == State.INIT, "Contract must be only initiated");
        require(msg.sender == buyer, "Not owner");
        status = State.IDLE;
        buyer.call{ value: payment + eps }("");
        // the payment is sent to the buyer anyway
    }

    function execute(MetaTx memory _metaTx, bytes memory _sig) external {
        require(block.number >= startBlock && block.number <= endBlock, "Block must be between start and end");
        require(status == State.REGISTERED, "Contract not registered");
        require(msg.sender == seller, "Wrong seller");
        status = State.IDLE;
        verifyAndExecute(_metaTx, _sig);
        seller.call{ value: collateral + payment + eps }("");
        // the payment is sent to the seller anyway
    }

    function exhaust() external {
        require(block.number >= startBlock && block.number <= endBlock, "Block must be between start and end");
        require(status == State.REGISTERED, "Contract not registered");
        require(msg.sender == seller, "Wrong seller");
        loopUntil();
        status = State.IDLE;
        seller.call{ value: collateral + payment }("");
        // the payment is sent to the seller anyway
    }

    function verifyAndExecute(MetaTx memory _metaTx, bytes memory _sig)
                        public returns (bytes memory) {
        require(_metaTx.nonce == nonce, "Nonce incorrect");
        bytes32 metaTxHash = keccak256(abi.encode(_metaTx.nonce,
                        _metaTx.to, _metaTx.value, _metaTx.callData));
        address signer = ECDSA.recover(ECDSA.toEthSignedMessageHash(metaTxHash), _sig);
        require(buyer == signer, "UNAUTH");
```

```
            nonce++; // We increment the nonce regardless of success
            (bool _success, bytes memory _result) = _metaTx.to.call{
                                value: _metaTx.value }(_metaTx.callData);
            if (status == State.INIT) {
                require(address(this).balance >= payment + eps,
                                    "cannot spend locked funds");
            } else if (status == State.REGISTERED) {
                require(address(this).balance >= payment + eps + collateral,
                                    "cannot spend locked funds");
            }
            return _result;
        }

        function loopUntil() public {
            uint256 i = 0;
            uint256 times = (gasHedged - 23330) / 117;
            for (i; i < times; i++) {}
        }
}
```

I Goerli Test Network Deployment

Table 2 presents our deployment of LedgerHedger on the Goerli Ethereum test network. It lists the invoked contract function, the transaction identifiers, and the consumed gas.

We took the following approach to verify the gas overhead of *Apply* produced by our local profiler. We created another meta-transaction, and performed its operations both with and without the contract. The gas consumption difference is $12e3$, matching the local profiler measurement $g_{done} = 12e3$. Table 2 includes the relevant transaction identifiers for this experiment as well.

Algorithm 1: LEDGERHEDGER

Parameter : *acc*, *start*, *end*, block number operation ranges
Parameter : g_{alloc}, required gas
Parameter : *col*, the required collateral by *Seller*
Parameter : *payment*, payment for execution
Parameter : ε, additional payment for *successful* execution.
Global Variable: *current*, current block number
Variable : $status \leftarrow \perp$, contract status variable
Variable : $PK_{Seller} \leftarrow \perp$, public identifier of *Seller*
Variable : $PK_{Buyer} \leftarrow \perp$, public identifier of *Buyer*

Function *Initiate*($txIssuer, sentTokens; acc, start, end, g_{alloc}, col, \varepsilon$):
    Assert: $current \leq acc < start \leq end$
    Assert: $g_{alloc} > 0, col \geq 0, \varepsilon \geq 0, sentTokens \geq \varepsilon$
    Set $acc, start, end, g_{alloc}, col$ from inputs, $payment \leftarrow sentTokens - \varepsilon$
    $PK_{Buyer} \leftarrow txIssuer$
    $status \leftarrow$ initiated

Function *Accept*($txIssuer, sentTokens$):
    Assert: $current \leq acc$
    Assert: $status =$ *initiated*
    Assert: $sentTokens \geq col$
    $PK_{Seller} \leftarrow txIssuer$
    $status \leftarrow$ accepted

Function *Recoup*($txIssuer, sentTokens$):
    Assert: $start \leq current \leq end$
    Assert: $status =$ *initiated*
    Assert: $PK_{Buyer} = txIssuer$
    $status \leftarrow$ completed
    Send $payment + \varepsilon$ to PK_{Buyer}

Function *Apply*($txIssuer, sentTokens; tx_{provided}$):
    Assert: $tx_{provided}$ *was issued by* PK_{Buyer}
    Assert: $start \leq current \leq end$
    Assert: $status =$ *accepted*
    Assert: $PK_{Seller} = txIssuer$
    Execute the operations of tx_{provided}
    $status \leftarrow$ completed
    Send $payment + \varepsilon + col$ to PK_{Seller}

Function *Exhaust*($txIssuer, sentTokens$):
    Assert: $start \leq current \leq end$
    Assert: $status =$ *accepted*
    Assert: $PK_{Seller} = txIssuer$
    Perform null operations summing to g_{alloc} gas
    $status \leftarrow$ completed
    Send $payment + col$ to PK_{Seller}

Table 1: $\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$ and $\Gamma^{Seller}_{\mathsf{FulfillTx}}$ subgame summaries.

Subgame	**Condition**	$\bar{s}_{\mathbf{spe}}$ **Action**	$W_{\boldsymbol{Buyer}}$	$W_{\boldsymbol{Seller}}$
$\Gamma^{Seller}_{\mathsf{FulfillNoTx}}$	$\pi_{exec} < \frac{payment+col}{g_{alloc}+g_{done}}$	$a_{exhaust}$	$w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + \max(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0)$ (Eq. 2)	$w^{init}_{Seller} - g_{accept} \cdot \pi_{setup} + payment - g_{done} \cdot \pi_{exec}$ (Eq. 1)
	$\pi_{exec} > \frac{payment+col}{g_{alloc}+g_{done}}$	a_{ignore}	$w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + \max(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0)$ (Eq. 2)	$w^{init}_{Seller} - g_{accept} \cdot \pi_{setup} + g_{alloc} \cdot \pi_{exec} - col$ (Eq. 1)
$\Gamma^{Seller}_{\mathsf{FulfillTx}}$	$\pi_{exec} < \frac{payment+col+\varepsilon}{g_{pub}+g_{done}}$ $g_{pub} \leq g_{alloc}$	a_{apply}	$w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + w^{exo}$ (Eq. 4)	$w^{init}_{Seller} - g_{accept} \cdot \pi_{setup} + payment + \varepsilon - (g_{done} + g_{pub} - g_{alloc}) \cdot \pi_{exec}$ (Eq. 3)
	$\pi_{exec} < \frac{payment+col}{g_{alloc}+g_{done}}$ $g_{pub} > g_{alloc}$	$a_{exhaust}$	$w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + \max(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0)$ (Eq. 4)	$w^{init}_{Seller} - g_{accept} \cdot \pi_{setup} + payment - g_{done} \cdot \pi_{exec}$ (Eq. 3)
	$\pi_{exec} > \frac{payment+col}{g_{alloc}+g_{done}}$ $\pi_{exec} > \frac{payment+col+\varepsilon}{g_{pub}+g_{done}}$	a_{ignore}	$w^{init}_{Buyer} - g_{init} \cdot \pi_{setup} - payment - \varepsilon + \max(w^{exo} - g_{alloc} \cdot \pi_{exec}, 0)$ (Eq. 4)	$w^{init}_{Seller} - g_{accept} \cdot \pi_{setup} + g_{alloc} \cdot \pi_{exec} - col$ (Eq. 3)

Table 2: Ethereum Goerli Network Deployment and Gas Requirements.

Invocation	**Transaction Identifier**	**Consumed Gas**
Initiate	37d4a7332ad18753277c62b96f9e8b97 d2f59c7aa22126dd23fe6825c361743f	$g_{init} = 117e3$
Recoup	e8b69c4ae70f40e72e3a8df353c38e44 9c176d9a4d7aee86b073e3a3a6a55531	$g_{done} = 57.3e3$
Initiate	7a47b67e574b748105ef31f6ebed8990 c17a96f19ef01307779a6119edf2318f	$g_{init} = 37.4e3$
Accept	b5607e9c499279c7bd4b0abf2f3d212b b3c294c684d87678cd06dd5d049a6b26	$g_{accept} = 50e3$
Exhaust	c482ad2b3bfc1ca64b83e8fcdc29fe82 652ef7d839fc24323d035f8aba0b66b0	$g_{alloc} + g_{done} = 3.021e6$
Initiate	9fee96dcfedd8f94e5442c1d8d50c92e 40bcfbf27ae512f9e2e3b01e670b005f	$g_{init} = 37.4e3$
Accept	b0f3cd808d5ad637b94541f3519614dc 444d2c76eaf60e4917f32bfc57df6eb9	$g_{accept} = 50e3$
Apply	facb062758d24a2266b3e6d989ffe430 202fdc2f23f4f73a585945e132fe0d7b	$g_{pub} + g_{done} = 2.668e6$
Arbitrary *tx* without *Apply*	27b4ad41e814d432a6c3e060eee6c6e7 f7e8fdc615b904548dfd9387db79020a	$g_{pub} = 63e3$
Arbitrary *tx* with *Apply*	b8a45902b247cd812e784e940ed822c3 cf8155a732b09ced2823fc27265fb7e2	$g_{pub} + g_{done} = 75e3$