

leanSeal: A Committee-Free Encrypted Mempool from Hashes Alone

Vitalik Buterin¹

Quang Dao²

Justin Drake¹

¹Ethereum Foundation

vitalik@ethereum.org, justin.drake@ethereum.org

²Carnegie Mellon University

qvd@andrew.cmu.edu

October 2026

Abstract

We present leanSeal, a simple encrypted mempool construction that needs neither a committee nor a trusted setup, relying only on a hash function such as SHA-256. Encrypted mempools let block builders order transactions before reading them, which prevents front-running and sandwich attacks based on their contents. Existing designs either trust a committee to open the transactions afterwards, or let each sender open its own, in which case the sender can withhold its key when the ordering is unfavorable. In leanSeal, the sender opens its own transaction but also attaches the chained-hash time-lock puzzle of Mahmoody, Moran and Vadhan (CRYPTO 2011), which anyone can solve to recover the key after a fixed amount of sequential computation. Withholding the key therefore cannot keep the transaction hidden, provided that it was sealed correctly.

For our exact encoding, we prove in the random-oracle model that even a classical attacker with about the hash rate of the whole Bitcoin network recovers the key of a 30-second puzzle early with probability at most about 2^{-117} . Against an attacker that computes one chained SHA-256 every 3.5 ns, such a puzzle takes about 21 s to build on a laptop, ahead of time, and adds 152 to 512 bytes to a transaction. Two limitations follow from the time-lock fallback: a sender that withholds its key can still delay or cancel execution, which a deposit must deter, and transactions that are never included are eventually disclosed. We describe how a threshold committee can remove either one, depending on how its key is combined with the puzzle's.

Contents

1	Introduction	3
1.1	Our results	4
2	Sealing Transactions with leanSeal	5
2.1	The flow	5
2.2	The timing condition	6
2.3	Choosing the delay	6
2.4	A proof of concept	7
3	The Puzzle	8
3.1	Notation and model	8
3.2	Building and opening	8
4	Security	9
4.1	Assumptions	9
4.2	The delay theorem	10
4.3	The hardware assumption	11
4.4	Attacks and link width	11
5	Implementation and Costs	12
5.1	Measurements	12
5.2	What a delay costs	12
5.3	The number of chains	13
6	Limitations and Mitigations	13
6.1	Excluded transactions are also disclosed	13
6.2	Withholding: disclosure is forced, execution is not	14
6.3	Other requirements	16
7	Comparison with Other Encrypted-Mempool Designs	16
7.1	The design space	16
7.2	Composing leanSeal with a threshold committee	18
7.3	Settings where leanSeal is practical	19
8	Limits, Related Work and Open Questions	19
8.1	Limits of hash-only constructions	19
8.2	Related work	20
8.3	Open questions	20
	Acknowledgments	21
	AI Assistance	21
A	Proof of the Delay Theorem	23
B	Estimating the Hardware Speed	25

1 Introduction

In an encrypted mempool [PP26], block builders order transactions before they can read them, which removes their ability to front-run or sandwich a transaction based on its contents. The central question is who opens the transactions afterwards. Current designs give three answers, each at a cost.

- **A committee** holds a threshold decryption key and opens what was ordered, as in Shutter [Shu]. Privacy then rests on no threshold of its members colluding, and liveness on enough of them being online.
- **A trusted execution environment** opens transactions inside hardware whose vendor and side channels must be trusted.
- **The sender** opens its own transaction after the ordering, as in sealed transactions and the self-decryption path of LUCID [EFM26]. No other party is trusted, but the sender sees the ordering first and can withhold its key when the ordering is unfavorable, which gives it a free option.

Hybrid designs [WM25] let either the sender or a committee open, which removes the free option while the committee is online and the committee’s liveness problem while the sender is honest. Their failure case remains a dishonest sender while the committee is offline.

We present leanSeal, a fallback for self-decryption that needs no committee: no designated party has to be online or trusted for a withheld transaction to be disclosed. The sender opens its own transaction in the normal case, but it attaches a time-lock puzzle that anyone can solve to recover the key after a fixed amount of sequential computation, about 30 seconds on Ethereum. If the puzzle is well formed and the transaction is encrypted under its key, a sender that withholds its key can still delay or cancel the transaction’s execution, but cannot keep the transaction hidden. The reveal and the puzzle together form a timed commitment [BN00]. The puzzle uses only a hash function such as SHA-256: it needs no trusted setup and no hardware trust, and its idealized version is secure against quantum openers.

The puzzle is the chained-hash puzzle of Mahmoody, Moran and Vadhan [MMV11], in which the sender computes as many hashes as an opener, but in parallel (Figure 1). They also proved that no puzzle built only from hashes can make the sender’s work grow more slowly than the delay, so the cost of leanSeal falls on the sender. This paper examines whether that cost is acceptable for sealing transactions (Table 1), and what an encrypted mempool built on the puzzle can and cannot provide.

Table 1: The reference puzzle at a glance: a 30-second lock of $n = 16$ chains of 2^{29} SHA-256 steps with 24-byte links. Times are on an Apple M4 MacBook Air (Section 5); building and forced opening are half our measurements of the one-minute puzzle ($m = 2^{30}$, run with 32-byte links and the same chain work), since both are linear in the delay.

	Value	Basis
Earliest opening by an attacker	30 s	ASSUMED: 3.5 ns per chained hash
Opening when the sender reveals the key, the normal case	12.0 ms	MEASURED: command line, end to end
Forced opening without the sender, one laptop core	3 min 21 s	MEASURED, halved
Building the puzzle, before the transaction exists	21 s	MEASURED, halved: all CPU cores
Added to a transaction, on-demand or stockpiled puzzle	512 or 152 B	exact

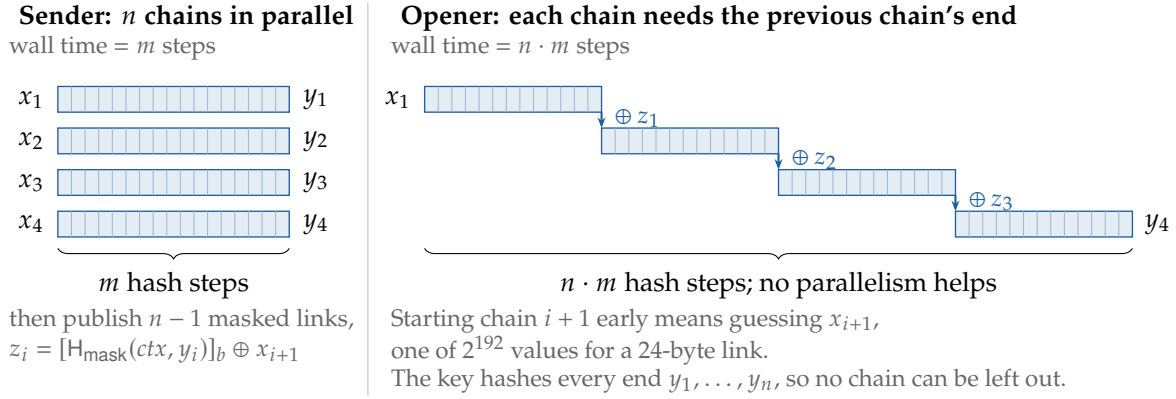


Figure 1: The same n chains, arranged two ways in time (drawn for $n = 4$). The sender knows every link and runs the chains side by side. An opener must run them one after another, because each chain's starting link is hidden under the end of the previous chain. The puzzle consists of the n links, so its size grows with n and not with the delay.

1.1 Our results

1. **A design.** We describe how leanSeal serves as the decryption path of an encrypted mempool (Section 2). It uses the same data as LUCID and the hybrid design, a ciphertext bound to a commitment to its key, so its puzzle fits them as a fallback that needs no third party.
2. **A concrete security proof.** In the random-oracle model, a classical attacker derives the key even one oracle round early with probability at most about 2^{-117} for a 30-second lock against about the hash rate of the whole Bitcoin network (Section 4). Turning rounds into seconds needs one hardware parameter, the fastest time for one chained SHA-256, which we set at 3.5 ns, 3.7 times faster than the fastest published design that we know of.
3. **Practical costs.** A 30-second puzzle, enough when the order is fixed within a slot, takes about 21 s to build on a laptop CPU and can be built before the transaction exists (Section 2.3 chooses the delay per use case). A sender that keeps a stock can use single-chain puzzles, which add 152 bytes to a transaction (Section 5).
4. **An analysis of the trade-offs.** The puzzle guarantees eventual disclosure, not timely execution. Withholding can still delay or cancel execution, so a deposit forfeited on a missed reveal, which the reveal deadline makes observable, must deter it. A malformed puzzle is not detected, since no efficient check is known, so the guarantee covers senders that build their puzzles honestly, including those that later go offline or change their minds, but not a sender that plans to withhold from the start. A time lock also reveals transactions that are never included; we argue that this holds for every design whose fallback opening is input-free, and show how validity windows, private submission and a committee limit it (Section 6). We compare leanSeal with threshold, TEE, hybrid and delay-encryption designs (Section 7).
5. **An implementation:** a specified format with a Rust implementation, a second implementation in Python, test vectors and benchmarks, and a prototype encrypted mempool on an EVM chain (Section 2.4) [lea26]. In the prototype, a sandwich attack costs a plaintext swap 9.25%, while the same swap, sealed, executes at the fair price, because the pool refuses direct trades from the swap's submission until its batch executes, including a trade placed with a key read from a pending reveal; a withheld key is forced open.

Neither the chained-hash puzzle nor the use of timed commitments against front-running is new (Section 8). We are not aware of a prior design that uses a hash-only timed commitment as the fallback for self-decryption, without a committee. Beyond the design, our contributions

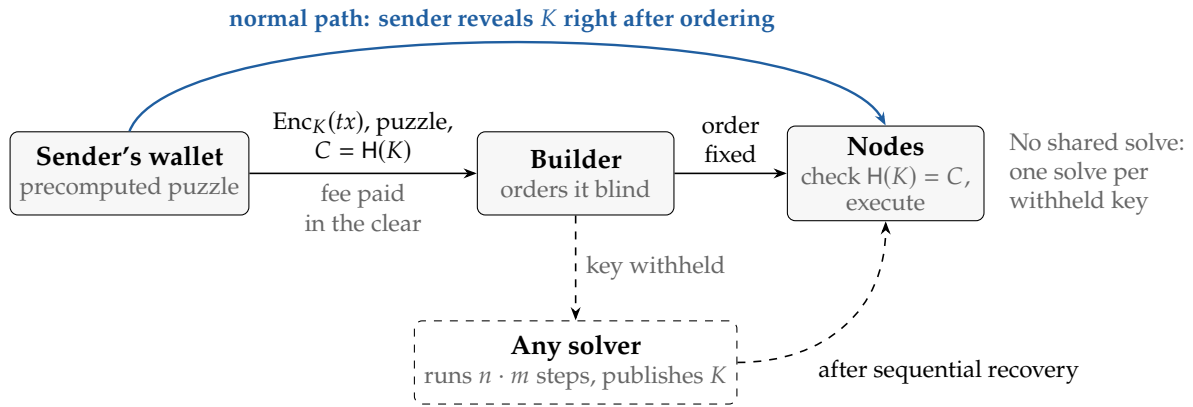


Figure 2: Commit–reveal with a time-lock fallback. When senders reveal their keys, no puzzle is solved. For a well-formed puzzle, withholding the key cannot prevent disclosure, but it can still delay or cancel execution.

are a concrete bound for an exact encoding, measured costs against an explicit attacker, and an analysis of what such a fallback can and cannot provide in an encrypted mempool.

2 Sealing Transactions with leanSeal

This section treats the puzzle as a black box and describes how it supports an encrypted mempool. The sender runs `Build` to obtain a puzzle Z , which contains a commitment C , and a key K . Anyone can check a claimed key against C with one hash. Anyone can also recover K from Z , without the sender, with N sequential hashes, which takes at least $T = N\ell$ seconds on the fastest hardware. Sections 3 and 4 construct the puzzle and prove this. Table 2 lists the properties an encrypted mempool may want and what leanSeal provides; no known design provides all of them, and Section 7 compares the alternatives.

2.1 The flow

leanSeal is used as a timed commitment [BN00]; Figure 2 shows the flow.

1. **Submit.** The sender takes a puzzle from its stock and encrypts its signed transaction under K . It sends the sealed transaction together with a plaintext envelope: the account that pays, a nonce, a gas limit, a fee and a deposit (Section 6.3).
2. **Order.** The builder orders sealed transactions without reading them. The ordering commits to the hash of each whole sealed transaction, not only to C : once K is known, anyone could seal a different transaction under the same puzzle.
3. **Reveal.** Once the ordering is fixed, the sender publishes K . Nodes check it against C , decrypt in milliseconds and execute the transaction in its committed position. No puzzle is solved.
4. **Fallback.** If the sender has not revealed K by the reveal deadline, its deposit is forfeited, and anyone can solve the puzzle and publish K . We call a puzzle *well formed* if its chains lead to the key committed in C and its transaction is encrypted under that key. A well-formed puzzle therefore opens without its sender, but its transaction can only run late or not at all.

A ciphertext bound to a commitment to its key is the structure that recent Ethereum designs already use. In LUCID (EIP-8184) [EFM26], a sealed transaction’s ticket goes on chain and carries a commitment $\text{keccak256}(k_{\text{dem}})$ to the key and a hash of the ciphertext envelope, which itself travels off chain. The key is released by a key publisher of the sender’s choice, possibly the sender itself, and the publication fee is paid only when decryption succeeds. In the hybrid design of

Table 2: Properties of an encrypted mempool, and what leanSeal provides.

Property	leanSeal
Contents hidden until the order is fixed	yes, for T after exposure, under the hardware assumption
Every included transaction opens	yes, without the sender or anyone else, if the puzzle is well formed, which is not checked (Section 6.2)
No free cancellation after seeing the order	no: withholding still delays or, with expiry, cancels execution; a deposit must deter it (Section 6.2)
Excluded transactions stay private	no: they open after T like any other (Section 6.1)
Sender, fee and size hidden	no; the envelope is public and the size can be padded
Trusted parties or setup	none
Post-quantum	plausibly; proved for the idealized puzzle only (Section 4.4)
Cost	152 to 512 bytes per transaction, sender work proportional to T paid in advance, no puzzle solving in the normal case (Section 5)

Wagner and Ma [WM25], the sender puts $\text{Enc}_k(tx)$ and $H(k)$ on chain and gives a committee a threshold encryption of k , so that either the sender or the committee can open. leanSeal fits both as a fallback that needs no third party: the sender is its own key publisher, and the puzzle replaces the committee as the party that discloses the key when the sender does not.

Inside LUCID, only the puzzle is used. It travels in the envelope’s opaque header with K as k_{dem} ; its SHA-256 commitment C is separate from the ticket’s keccak commitment, and the cipher stays ChaCha20-Poly1305. The puzzle provides eventual disclosure only: executing a transaction whose key is disclosed minutes late needs an additional protocol rule, which LUCID does not have.

The reveal should be due as soon as the ordering is fixed. In LUCID, the builder’s bid commits to the tickets of slot n , keys are released once the beacon block carrying the bid is attested and must be seen before the payload-timeliness committee votes, and the decrypted transactions execute at the top of block $n + 1$, before any ordinary transaction. A short window shortens the delay the puzzle must provide and limits what a sender learns before deciding whether to reveal (Section 6.2).

2.2 The timing condition

The attacker’s clock starts at *exposure*, t_{exp} , the first moment anyone but the sender holds the puzzle. The deadline t_{order} is the moment the transaction’s position can no longer change. The contents stay hidden until the deadline if

$$t_{\text{order}} - t_{\text{exp}} < N \cdot \ell_{\text{attacker}},$$

with a margin for propagation. A forced opening finishes $N \cdot \ell_{\text{solver}}$ after a solver starts, which on a laptop core is minutes after the deadline.

If the attacker is k times faster than assumed, the delay shrinks k -fold. Once it falls below $t_{\text{order}} - t_{\text{exp}}$, the contents can be read before the order is fixed: privacy before ordering is lost, and content-dependent MEV becomes possible again for any party that can solve the puzzle in time.

2.3 Choosing the delay

The delay should be the smallest that covers the time from exposure to the fixed order, $t_{\text{order}} - t_{\text{exp}}$. Building, forced opening, the solvers’ load and the window in which a withholding sender holds an option all grow linearly with T ; only the size does not. The attacker speed already carries a margin of about 3.7 over the fastest published design (Section 4.3), so we take T about twice the

time to fix the order, with a floor of 15 s and rounded up. [Table 3](#) applies this to the settings of [Section 7.3](#). We use $T = 30$ s as the reference: it covers a puzzle that is sent only to the slot's builder, or published after a commitment, and ordered within one 12-second slot.

The last row shows why leanSeal needs an order that is fixed early. Waiting for today's finality makes every cost sixty times larger than in the reference setting, so a protocol either accepts that a reorganization exposes revealed transactions ([Section 6.3](#)) or relies on fast finality, which brings the delay back to the reference range. Public gossip remains affordable for a wait of one or two slots but not longer, so the puzzle should not wait in a public mempool.

2.4 A proof of concept

We implemented this flow as an EVM contract, using EIP-7702 and transient storage, with off-chain sender, keeper and solver roles [[lea26](#)]. Each block's sealed transactions form a batch, and batches execute strictly in order from one queue. On submission, the contract checks each envelope's structure against the specification and caps the gas per transaction and per batch. The sender reveals its key within two blocks; anyone then executes the batch in order. The contract checks each key against its commitment, decrypts the envelope with SHA-256 only and calls the sender's account. Only accounts delegated to the contract's own account code may submit, and the delegation is checked again before the call, so the call that runs is the committed one. The queue advances past a transaction only if the executor supplied its full gas, and a failing or redelegated account cannot block it.

Two further rules make the committed order binding for a market. First, a guarded pool refuses direct trades from the submission of a sealed transaction until its batch has executed. A key is public as soon as its reveal is pending, before the reveal executes, so a lock that started at the reveal would let whoever reads the pending key trade ahead of it. While sealed transactions keep arriving, the pool therefore trades only through them. Second, sealed calls are restricted to the pool, so a sealed transaction ordered ahead of another has parameters fixed before any reveal. Otherwise it could read state that the attacker wrote after the reveals and adapt to the victim's plaintext. General programmable transactions need execution immediately after key release with nothing in between, which only a protocol such as LUCID provides.

A missed reveal burns most of the deposit: the solver who posts the key receives a smaller bounty, which the sender, who knows its key, could claim itself, so the net penalty is the deposit minus the bounty. The deposit prices only the missed reveal: a sender that reveals on time and then revokes its approval or moves its funds makes its own trade fail without penalty, a cancellation that also only immediate execution removes. A forced opening publishes the key and pays the bounty, but the transaction does not run. Its key may already have been public: a builder that reads a pending reveal and keeps it out of the two blocks makes the ticket miss while holding the key, and could trade around the transaction if it ran later. Such a builder can still make an honest sender forfeit its deposit, so the window must exceed the number of consecutive blocks that one builder controls.

On a local chain with 2-second blocks, a searcher watching the mempool sandwiched a plaintext swap, which lost 9.25% against the fair price. The same swap, sealed, executed at the fair price: the searcher read the key from the pending reveal, decrypted the swap and placed a trade ahead of the reveal in the same block, and the pool rejected it. With its key withheld, a demonstration puzzle of 2^{29} steps, 1/16 of the reference work, was solved on one core in 13 s; the key was published, and the swap did not run.

The keeper splits a batch into transactions within the gas cap of EIP-7825. In the contract's unit tests, with a 128-byte test puzzle and 256-byte swaps, submitting costs 194,000 to 270,000 gas, revealing about 74,500, executing a batch of two swaps about 357,000, and a forced opening about 67,000. These are contract-call costs; calldata and intrinsic costs come on top and grow with the envelope.

Table 3: The delay per setting. Building uses all cores of the M4 Air (0.70 s per second of delay); forced opening uses one of its cores (6.7 times the delay).

Setting	Time to fix the order	Delay T	Building	Forced opening
Rollup with a sequencer	1–2 s	15 s	11 s	1.7 min
Ethereum with enshrined PBS; puzzle sent to the builder, or published after a commitment	up to one slot, 12 s	30 s	21 s	3.3 min
Sealed-bid auction; bid committed, then posted near the close	about one slot	30 s	21 s	3.3 min
Public gossip, waiting w slots	$12(w + 1)$ s	$24(w + 1)$ s	$17(w + 1)$ s	$2.7(w + 1)$ min
Order fixed only at finality, Ethereum today	about 13 min	30 min	21 min	3.3 h

3 The Puzzle

A *hash chain* starts from a value and hashes it many times in a row. Each step needs the previous output, so even unlimited parallel hardware must compute the steps one after another. A single chain of N steps would serve as a lock, except that its builder also needs N sequential steps. The construction of Mahmoody, Moran and Vadhan [MMV11] avoids this: the sender builds n short chains of m steps at the same time and links them so that an opener can only run them in order. The opener’s delay is $N = nm$ steps; the sender’s wall time is m steps on n cores.

3.1 Notation and model

The notation follows the specification (SPEC.md in the repository [lea26]). Integers are encoded big-endian, $a \parallel b$ is concatenation, $[v]_b$ is the first b bytes of v , and $\oplus$ is bitwise XOR. A *labelled hash* is $\text{TH}(\text{label}, x) = \text{SHA256}(\text{u8}(\text{label}) \parallel \text{label} \parallel x)$, and we write $H_{\text{start}}, H_{\text{mask}}, H_{\text{acc}}, H_{\text{key}}, H_{\text{commit}}$ for labelled hashes with the labels `leanseal/v1/start`, `leanseal/v1/mask` and so on. The *chain step* for chain i is

$$F_i(s) = \text{SHA256}(s \parallel \text{salt} \parallel \text{be32}(i)),$$

where *salt* is 16 random bytes drawn for each puzzle. Its input is exactly 52 bytes, so one step is one SHA-256 compression, and no other hash in the format has an input of that length.

A puzzle has n chains of m steps and b -byte links; the specification bounds the parameters, in particular $m \leq 2^{48}$. The reference parameters are $n = 16$, $m = 2^{29}$ and $b = 24$: $N = 2^{33}$ steps, a 30-second delay at 3.5 ns per step, and a 452-byte puzzle. The measurements of Section 5 use the one-minute puzzle, with $m = 2^{30}$.

We prove the delay in the random-oracle model [MMV11]: each chain step and each labelled hash is an independent random function that anyone can query, and an attacker is charged only for its queries. Time after publication is counted in *oracle rounds*: all queries in a round are chosen before any answer from that round arrives. A bound in rounds therefore holds however much parallel hardware the attacker has; turning rounds into seconds takes the hardware assumption of Section 4.

3.2 Building and opening

Algorithms 1 and 2 give the two algorithms, following Sections 4 and 5 of the specification with the byte encodings left out. The context *ctx* is a 32-byte encoding of the version, b , n , m and the salt. Start hashes include the context and the chain index, and chain steps the salt and the index. Mask, key and commitment hashes include the context; the accumulator begins at the context and absorbs the chain ends in order.

Algorithm 1 Build(n, m, b), run by the sender

```
1: salt  $\leftarrow$  16 random bytes;  $x_1, \dots, x_n \leftarrow b$  random bytes each
2: ctx  $\leftarrow (1, b, n, m, \text{salt})$ 
3: for  $i = 1, \dots, n$  in parallel do
4:    $s_i \leftarrow H_{\text{start}}(\text{ctx}, i, x_i)$ 
5:    $y_i \leftarrow F_i^m(s_i)$   $\triangleright m$  sequential steps
6: for  $i = 1, \dots, n - 1$  do
7:    $z_i \leftarrow [H_{\text{mask}}(\text{ctx}, y_i)]_b \oplus x_{i+1}$   $\triangleright$  hide the next link
8:  $a \leftarrow \text{ctx}$ 
9: for  $i = 1, \dots, n$  do  $a \leftarrow H_{\text{acc}}(a, y_i)$ 
10:  $K \leftarrow H_{\text{key}}(\text{ctx}, a)$ ;  $C \leftarrow H_{\text{commit}}(\text{ctx}, K)$ 
11: return puzzle  $Z = (\text{ctx}, C, x_1, z_1, \dots, z_{n-1})$  and key  $K$ 
```

Algorithm 2 Open(Z), run by anyone

```
1: parse  $Z$  as  $(\text{ctx}, C, x_1, z_1, \dots, z_{n-1})$ 
2:  $x \leftarrow x_1$ ;  $a \leftarrow \text{ctx}$ 
3: for  $i = 1, \dots, n$  do  $\triangleright$  in sequence
4:    $y \leftarrow F_i^m(H_{\text{start}}(\text{ctx}, i, x))$   $\triangleright m$  sequential steps
5:    $a \leftarrow H_{\text{acc}}(a, y)$   $\triangleright$  lines 5 and 6 share one oracle round
6:   if  $i < n$  then  $x \leftarrow [H_{\text{mask}}(\text{ctx}, y)]_b \oplus z_i$   $\triangleright$  unhide  $x_{i+1}$ 
7:  $K \leftarrow H_{\text{key}}(\text{ctx}, a)$ 
8: if  $H_{\text{commit}}(\text{ctx}, K) \neq C$  then return  $\perp$   $\triangleright$  malformed puzzle
9: return  $K$ 
```

The sender knows every link, so it runs the n chains of [Algorithm 1](#) in parallel. Each link after the first is published only masked, under a hash of the end of the chain before it. The key is a hash of all chain ends in order, and the commitment C lets anyone check a claimed key with one hash. An opener without the key has only x_1 : it runs chain 1, uses its end to unhide x_2 , runs chain 2, and so on, nm steps in sequence ([Figure 1](#)). The only way to start a chain early is to guess its link. The sender encrypts the transaction under K with HMAC-SHA256 ([Section 5](#)) and keeps the puzzle and key until it needs them.

Four choices in the encoding matter for security. Every hash depends on its puzzle’s context or salt, and chain starts and steps also name their chain, so work on another puzzle does not help. Masks are hashes of chain ends, not the ends themselves, so learning a link says nothing about the chain before it. The key depends on every chain’s end, so a guessed link cannot be confirmed until the opener’s own chains reach it. Finally, the puzzle has $68 + nb$ bytes regardless of the delay, which depends only on N .

4 Security

Secrecy until the deadline rests on two assumptions about SHA-256 and two about the attacker’s hardware. This section states them, proves the delay in the random-oracle model, and estimates the one hardware parameter that the proof cannot supply.

4.1 Assumptions

A1. No shortcut. The chain step and each labelled hash behave like independent random oracles on their whole inputs. We instantiate them heuristically with SHA-256; a chain step is a

single compression. If this fails, the delay is lost and the hash must be replaced, which the versioned format allows.

A2. Speed. During the delay, no party computes one chain step in less than ℓ . An error here shrinks the delay in proportion.

A3. Budget. The attacker computes at most R hashes per second in total. An error here multiplies the bound of [Theorem 1](#) by the same factor; for the reference puzzle, any factor up to 2^{52} keeps it below 2^{-64} .

A4. Encryption. HMAC-SHA256 is a pseudorandom function.

4.2 The delay theorem

Model. The chain steps $F_1, \dots, F_n$ and the labelled hashes of [Algorithms 1](#) and [2](#) are independent random oracles ([Section 3.1](#)). The attacker is a classical randomized algorithm $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ with unbounded computation. $\mathcal{A}_0$ runs before the puzzle exists and makes Q_{pre} queries; its whole state passes to $\mathcal{A}_1$, which receives the puzzle and makes Q_{on} queries. Let $Q = Q_{\text{pre}} + Q_{\text{on}}$. Time after publication is counted in oracle rounds: all queries in a round are chosen before any answer from that round arrives.

Honest schedule. Computing each chain's mask and accumulator in the same round, Open derives K in oracle round

$$T_{\text{key}} = n(m + 2) + 1$$

and returns it, checked against C , one round later. Build takes $m + n + 3$ rounds: one for the starts, m for the chains, n for the accumulator, with the masks alongside, and one each for K and C . Let $P = n(m + 3) + 1$, the number of oracle answers Open uses, C included.

Theorem 1 (Classical attackers, random-oracle model). *Let $(Z, K) \leftarrow \text{Build}(n, m, b)$, with randomness independent of the oracles. For every attacker $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ as above,*

$$\Pr[\mathcal{A}_1 \text{ outputs } K \text{ within fewer than } T_{\text{key}} \text{ oracle rounds}] \leq \frac{Q_{\text{on}}}{2^{8b}} + \frac{Q_{\text{pre}}}{2^{128+8b}} + \frac{QP + P^2 + 1}{2^{256}},$$

and the first term is absent when $n = 1$.

The bound in seconds. Assumptions A2 and A3 turn oracle rounds and queries into time. Write $T = nm\ell$, and let the attacker make $Q_{\text{on}} = R \cdot T$ queries after publication. Against $R = 10^{21}$ queries per second, about the hash rate of the whole Bitcoin network ([Section 4.3](#)), the bound is about 2^{-117} for a 30-second lock and 2^{-97} for a one-year lock. Precomputation changes this only slightly: a full year of queries at the same rate before a 30-second puzzle is published adds about 2^{-128} . With 24-byte links the first term, the chance of guessing a link, dominates; for a 30-second lock the full-width terms are smaller by a factor of about 2^{31} . At a fixed delay the bound depends on the number of chains only weakly, through $P \approx N$, except that the first term is absent when $n = 1$. The bound covers every speed-up, however small: except with this probability, the attacker does not derive the key even one oracle round before the honest opener.

Scope. The threshold is the round in which K is derived, not the round in which Open returns: an attacker that skips the check against C finishes one round before Open does, with certainty. The precomputation term is needed even when $n = 1$, where the only link is public: an attacker that guesses the salt and the link in advance and computes the key opens the puzzle in zero rounds with probability $2^{-(128+8b)}$. The theorem concerns recovery of the key. The encryption layer is intended to hide the contents of equal-length messages under A1 and A4, but a reduction

from distinguishing envelopes to the events of the proof, which must also count queries on the derivation of the encryption and authentication keys, nonce collisions and several envelopes under one puzzle, has not been written. The bound is classical; extending it to this instantiation against quantum attackers is outside the scope of this paper.

[Appendix A](#) gives the proof. It follows the structure of Pietrzak and Reyzin’s soundness proof for proofs of sequential work [PR26].

4.3 The hardware assumption

The attacker’s total hash rate can be estimated from deployed hardware: the Bitcoin network computes about 10^{21} hashes per second (October 2026), and we take $R = 10^{21}$. A query is one compression, while a Bitcoin hash costs two, since it hashes twice and the first block of the header is computed once; counted in compressions, the network makes about $2 \cdot 10^{21}$ queries per second, which costs one bit: 2^{-116} instead of 2^{-117} for a 30-second lock. Mining hardware computes many independent hashes at once, of block headers rather than salted chains, and says little about how fast one hash can follow another, so it bounds R , not ℓ .

The speed ℓ is the assumption that most needs hardware evidence. [Table 4](#) collects what is known. Relative to the fastest published design that we know of, a synthesized 7 nm core at 13.0 ns per hash [BCF⁺20], we assume an *advantage* of $A \approx 3.7$, that is, $\ell = 3.5$ ns: 6.6 times faster than one core of the M4 laptop we measured. Some advantage over the published core is certain, since it is a general-purpose design that registers every round and does not exploit the constant part of a chain step’s input; [Appendix B](#) gives a circuit model for a chip built for chains. We do not claim 3.5 ns is a lower bound. The theorem holds for any value of ℓ the reader adopts, and every cost in [Section 5](#) scales linearly with $1/\ell$.

Sensitivity. Only A1 fails abruptly. An attacker k times faster than assumed opens a 30-second puzzle in $30/k$ seconds; restoring the margin costs k times more sender work and no extra bytes. An attacker with k times the budget multiplies the bound by k , which for the reference puzzle stays below 2^{-64} unless k exceeds 2^{52} . Slower honest hardware costs time, never secrecy. The secrecy is also temporary: a sealed transaction becomes public in any case, so only hardware that exists during its delay matters, and ℓ can be repriced as hardware improves.

4.4 Attacks and link width

- **Link guessing.** An attacker can guess a hidden link x_k and run chains $k, k + 1, \dots$ while its own earlier chains are still running. This is the first term of the theorem; with 24-byte links each guess is right with probability 2^{-192} . Because the key needs every chain’s end, a guess cannot be checked against C until the attacker’s own chains reach link k .
- **Precomputation.** Every hash on the honest path depends on the puzzle’s fresh salt and a fresh link, so work done before publication or on another puzzle does not help, except as counted by the theorem’s second and third terms.
- **Quantum attackers.** Afshar et al. [ACH⁺23] prove that the idealized chained puzzle keeps its delay against quantum openers in the quantum random-oracle model. Carrying this over to leanSeal’s exact encoding has not been done; our bound is classical.
- **Early access.** The attacker’s clock starts when it first holds the puzzle. Puzzles built in advance must stay private until use.
- **Message substitution.** A claimed key is checked against C with one hash and the tag then authenticates the transaction. But once K is known, anyone can seal a different transaction under the same puzzle, so an ordering must bind the hash of the whole sealed transaction ([Section 2.1](#)).

Table 4: Time per chained SHA-256 hash, from measurements to models.

Source	Time per hash	Basis
Apple M4, one core	23.2 ns	MEASURED: leanSeal benchmark
AMD EPYC 9554P (Zen 4), one core	37.4 ns	MEASURED: leanSeal benchmark, SHA-NI
Published SHA-256 core, 7 nm [BCF ⁺ 20]	13.0 ns	SYNTHESIS: 67 cycles at 5.15 GHz; not a fabricated chip
Custom round circuit, 3–5 nm	2.9–9.7 ns	MODEL: Appendix B ; no such chip has been built
Assumed attacker	3.5 ns	the design point of this paper ($A \approx 3.7$)

The link width b enters the online guessing term $Q_{\text{on}}/2^{8b}$ and the preprocessing term. With $n = 16$ and the attacker budget above, 16-byte links (324-byte puzzle) give 2^{-53} for a 30-second lock, 24-byte links (452 bytes) give 2^{-117} , and 32-byte links (580 bytes) give 2^{-148} ; for a one-year lock, which needs at least 33 chains to respect the format’s limit $m \leq 2^{48}$, the three give about 2^{-33} , 2^{-97} and 2^{-108} . We use 24 bytes, which keeps the bound near 2^{-96} or below for delays up to a year. With $n = 1$ there is no hidden link, so the online guessing term vanishes; the preprocessing term still depends on the salt and link entropy.

5 Implementation and Costs

leanSeal implements [Section 3](#) as a versioned format with a written specification, a Rust implementation and command-line tool, and an independent Python implementation that agrees with it on every test vector [lea26]. The solver saves its progress, so an interrupted solve can be resumed, and it can solve several puzzles at once, one per core. The Rust code depends only on `getrandom` and `signal-hook`, uses the ARMv8 and x86 SHA-256 instructions when present, and is tested with known answers, round trips, property tests over 480,000 mutated encodings and fuzzing of every decoder. The specification and an earlier C implementation were reviewed over several rounds by an AI model; no human cryptographer has reviewed the code or the format.

Encryption. Encryption uses only SHA-256. For a fresh 16-byte nonce, subkeys k_{enc} and k_{mac} are labelled hashes of K and the nonce; the keystream is $\text{HMAC}(k_{\text{enc}}, \text{be64}(j))$ for block j , and an HMAC tag under k_{mac} covers every preceding byte. A sealed transaction of L bytes takes $128 + nb + L$ bytes: the $68 + nb$ -byte puzzle, 8 bytes of framing, the nonce and length (20 bytes) and the tag (32 bytes). One puzzle can seal several messages under different nonces; they all open together.

5.1 Measurements

[Table 5](#) gives the measurements on two Apple laptops: a fanless MacBook Air with an M4 (10 CPU cores) and a MacBook Pro with an M4 Max (16 CPU cores). GPU figures come from a separate Metal benchmark of the chain step; the tool does not yet build puzzles on a GPU.

5.2 What a delay costs

Four formulas describe the scheme. The delay is $T = N\ell$. Building takes about N/R_s , where R_s is the sender’s total hash rate. The puzzle takes $68 + nb$ bytes. Opening without the key takes N times the opener’s own time per step: 3.3 minutes on one laptop core for a 30-second puzzle.

The sender’s cost is therefore proportional to the delay: about 0.70 seconds of a laptop’s time, with all its cores busy, per second of delay (0.097 on the M4 Max GPU, amortized over a

Table 5: Measurements. The measured puzzle is a one-minute lock with $n = 16$, $m = 2^{30}$ and 24-byte links; the 30-second reference puzzle ($m = 2^{29}$) takes half these times to build and to open without the key. The M4 Air building and opening runs used 32-byte links, with the same chain work. Opening with the key is the command-line tool’s end-to-end time, including process start and file input and output.

Measured	M4 Air	M4 Max
One chain step, one CPU core	23.2 ns	24.1 ns
Total CPU hash rate, independent chains	461 M/s	867 M/s
Total GPU hash rate, many chains in flight	0.71 G/s	2.96 G/s
Building the one-minute puzzle on the CPU	42.1 s	30.1 s
Opening it without the key, one core	6 min 42 s	–
Opening a sealed 110-byte transaction with the key	12.0 ms	–

batch that takes hours). For constructions from a random oracle, this work cannot be reduced asymptotically [MMV11] (Section 8.1). Two observations make this cost acceptable. First, a puzzle does not depend on the transaction it will seal, so a wallet can keep a stock and build more while idle or charging; the building time is then a cost, not a delay. A laptop CPU running all day builds about 2,800 30-second puzzles (Section 5.3). Second, the delay only needs to cover the time until the ordering is fixed (Section 2.2); halving the delay halves all the work and leaves the puzzle size unchanged.

5.3 The number of chains

The delay depends only on N ; the number of chains n sets how many cores can work on one puzzle at once, and the size. A sender that builds a puzzle when its transaction is ready needs one chain per core it wants to use: $n = 16$ builds a 30-second puzzle in about 21 s on the M4 Air and 15 s on the M4 Max, half the measured times for a one-minute puzzle. A sender that builds a stock obtains its parallelism from building many puzzles at once instead (five one-minute 16-chain puzzles built together on the M4 Air finished in 187.6 s, or 37.5 s amortized per puzzle, about 19 s per 30-second puzzle), so it can use a single chain per puzzle. Table 6 compares the options.

With $n = 1$, sealing a 110-byte transfer gives 262 bytes rather than 622, and the guessing term of Theorem 1 vanishes. The daily figure above is projected from a single run, which was not archived and was made on a machine that was not idle: ten single-chain puzzles built at once on the M4 Air, one per core, took 75 s for 2^{31} steps each. A stock builder could interleave two single-chain puzzles per core, as the builder already does with chains, to keep the 1.5-fold gain of two SHA-256 pipelines per core.

6 Limitations and Mitigations

6.1 Excluded transactions are also disclosed

A time lock opens whether or not its transaction is included. If a builder receives a sealed transaction and leaves it out, the builder, or anyone else holding it, can read it after the delay. A threshold committee that decrypts only included transactions keeps excluded ones private, which matters because an excluded transaction can still reveal an intent to trade. The secrecy leanSeal gives is temporary in every case: it suits protection against front-running, not the lasting confidentiality that shielded transfers need.

This is not specific to leanSeal. Call a fallback opening *input-free* if it takes only the sealed transaction and auxiliary data that is available whether or not the transaction is included, with no later input or secret release that depends on inclusion.

Table 6: 30-second puzzles ($N = 2^{33}$, 24-byte links) for different numbers of chains. The times are half those measured for one-minute puzzles, except for $n = 4$, which is projected from the measured time per step.

Chains n	Puzzle	Added to a transaction	Building on a 10-core laptop
16	452 B	512 B	on demand, about 21 s
4	164 B	224 B	on demand, about 50 s
1	92 B	152 B	from a stock: one per 30 s, about 2,800 a day

Claim 1 (informal). *In any scheme whose fallback opening is input-free, whoever can open an included sealed transaction without the sender can open it equally well when it is not included.*

Since inclusion is not an input to the opening, the opening cannot depend on it. Revealing a transaction if and only if it is included therefore needs either a party that can refuse to open, such as a committee, a TEE or the sender, or an opening that takes a later inclusion witness as input, as witness encryption with a blockchain clock does [LJKW18]. The second class is outside the claim; we know of no practical construction in it, and the later input must come from some party or from consensus. Naraghi draws the same three-way distinction among the inputs of a forced opening [Nar26]. This disclosure is inherent in the guarantee: a puzzle protects against a withholding sender only because the sender has released it before the ordering is fixed. Four measures delay the disclosure, narrow its audience and reduce its value.

- **Delay.** An excluded transaction opens a delay after exposure, not at exposure as in today’s public mempool; by then, most opportunities to front-run it have passed.
- **Expiry.** A short validity window in the signed transaction means that by the time an excluded transaction can be read, it can no longer be executed. What leaks is the intent. The sender can also cancel early by spending the nonce.
- **Fewer holders.** Sent only to the parties that can include it in the current slot, the builder or the inclusion-list committee, rather than gossiped, an excluded transaction is readable only by them, and exposure starts close to t_{order} . Broadcasting only a hash of the sealed transaction and publishing the rest once its slot is reserved [Her26, Nar26] also starts exposure close to t_{order} , and a transaction whose commitment is never included is never exposed; but the sender can then decline to publish, which the deposit must also cover (Section 6.2).
- **A committee.** Encrypting under a key derived from both a leanSeal key and a committee’s key keeps excluded transactions private while the committee is honest (Section 7.2).

6.2 Withholding: disclosure is forced, execution is not

With plain commit–reveal, the sender holds a free option: it reveals its key if the ordering is favorable and withholds it otherwise. The time lock turns withholding into delay, so withholding cannot keep a well-formed puzzle’s transaction hidden. Execution, however, cannot wait at the committed position for minutes of solving without stalling every later transaction. A late-opened transaction must therefore execute later or be dropped, and with a tight deadline, a late execution that reverts amounts to a cancellation. Executing it later is safe only if no one could read its key earlier: a builder that keeps a pending reveal out until the deadline holds the key, and could trade around the late execution, so our prototype drops it (Section 2.4). Withholding thus still lets the sender delay or cancel execution, and a deposit must deter that choice. The reveal deadline makes a missed reveal observable; the time lock separately guarantees disclosure.

- **A deposit forfeited on a missed reveal.** A missed reveal is visible on chain. The deposit is a plaintext field, since the protocol cannot see the transaction’s value; LUCID’s partly refunded top-of-block fee [EFM26] is one such rule. The penalty must be irrecoverable: the sender

knows its own key, so any part of the deposit paid to whoever publishes the key is a part the sender can claim back. A forfeitable fee is also much weaker against an attacker with two or more commitments ahead of its victim (fee inversion [Nar26], Section 7). Capping at one the commitments that may execute ahead of a transaction, or making release decisions blind (below), avoids this.

- **The value of the option.** During the reveal window the sender sees the ordering and the market, but not the other sealed transactions. Its option is roughly an at-the-money option over the window, worth about $0.4 \sigma \sqrt{\Delta t}$ of the traded value: about 0.015%, or \$1.50 on a \$10,000 trade, for an asset with 60% annual volatility and a 12-second window. If a late-opened transaction may still execute, the option lasts until the forced opening instead: about 0.02% if fast solvers open the reference puzzle in 30 seconds and 0.06% for the 3.3 minutes one laptop core takes.
- **Sizing the deposit.** These are values of an unpenalized option in a volatility-only model, not sufficient deposits. A deposit charged only when the option is exercised raises the exercise threshold but leaves a residual option: a deposit equal to the option's value still leaves more than half of it. The ordering itself also carries information that the model omits. Sizing the deposit therefore needs a target exercise probability or a bound on the gain from withholding [Nar26], over the whole window in which the option lasts.
- **The last revealer.** A sender that sees the others' reveals before its own deadline decides with more information. Releasing reveals simultaneously is an additional protocol requirement; a party that merely collects plaintext keys could leak them early. Blind simultaneous release [Nar26] is one mechanism: right after ordering, each releaser, here each sender, escrows its key under a timed pad that outlasts a window of one or two seconds, so every decision to reveal is fixed before any key is public. It needs a committee to fix the set of timely escrows, funded fast solvers and a slashable deposit, and escrows by end users remain an open problem (Section 7).

Applications in which order within a batch does not matter avoid this problem. In a sealed-bid or uniform-price batch auction, settlement can wait until every bid is open, so a withheld bid is forced open and still binds its bidder, provided puzzles are checked to be well formed and bidders' funds are escrowed. This is the classical use of timed commitments [BN00], and leanSeal suffices on its own here once puzzles can be checked.

What the puzzle adds to commit-reveal. With the same deposit and deadline, plain commit-reveal already makes a missed reveal observable and penalized, so in a general mempool the deposit, not the puzzle, deters withholding. The puzzle adds three things, each conditional on a well-formed puzzle. First, a withheld transaction becomes public, so withholding no longer keeps the sender's intent private: a sender cannot submit several sealed orders and reveal only the profitable ones without disclosing the others. Second, a sender that goes offline after submitting, or whose key publisher fails, does not lose its transaction: anyone can recover it, and a protocol that executes late openings can still run it. Third, where settlement can wait for every opening, as in the auctions above, a withheld commitment still binds.

Malformed puzzles. A sender could build a puzzle whose key matches C but whose chains lead elsewhere, or encrypt its transaction under a key other than the committed one; either way, the solved key does not open the transaction. Because no efficient check of puzzles is known, a sender that intends to withhold from the start can do so, and then none of the three additions holds. Against such a sender the deposit is the only deterrent, exactly as in commit-reveal, and it needs no separate treatment: the penalty is triggered by the missed reveal, which is observable, not by the failed solve, which is not cheap to verify. What is lost is the guarantee that such a

transaction becomes public. A protocol that needs that guarantee must check puzzles, and a proof much cheaper to check than the chains themselves is an open problem ([Section 8.3](#)).

6.3 Other requirements

Finality. The deadline t_{order} is when the position can no longer change. Today an Ethereum block can be reorganized until it is finalized, about 13 minutes later, far beyond a 30-second lock; after a reorganization, a revealed transaction can be front-run in the replacement block. A protocol can accept this once a block has its attestations, so that a reorganization degrades only the affected transactions to today's exposure. With the faster finality planned for lean consensus, t_{order} shrinks to seconds and a 30-second lock covers it with a wide margin.

Admission. A sealed transaction cannot be checked before it is opened, so, as in other encrypted mempools, its plaintext envelope is signed and checked as usual and its fee is charged whether or not the opened transaction is valid. The number of open puzzles per sender should be capped.

Solvers. Opening a withholding sender's transaction within about the delay needs a chip as fast as the assumed attacker; a laptop core takes 3.3 minutes for the reference puzzle. If every one of the roughly 265 transactions per slot in our mainnet sample had to be forced open, this would occupy about 660 attacker-speed circuits or 4,400 laptop cores; at a forced-opening rate of 1%, about 44 cores, before redundancy and failed solves. With deposits in place, withholding should be rare. A published key can be copied, so the reward cannot go to whoever posts it first: a commit-reveal of the key bound to the solver's address, or a designated solver per slot, prevents this race. Whatever funds the reward, a withholding sender can claim it by publishing its own key, so the cost of withholding is the forfeited deposit minus every reward the sender can recover, and it is this difference that must deter withholding ([Section 6.2](#)).

Size and gas. Sealing adds 152 bytes with stockpiled single-chain puzzles and 512 with the reference puzzle. In 20 consecutive mainnet blocks (22 October 2026; 5,300 transactions; reproducible with `bench/eth-sample.py`), the median transaction was 179 bytes, so sealing makes it 1.8 or 3.9 times larger. As calldata at 16 gas per nonzero byte the added bytes cost about 2,400 or 8,200 gas, against 21,000 for a plain transfer, and about 6,100 or 20,500 for a transaction that pays the calldata floor of EIP-7623, 40 gas per nonzero byte. The ordering needs only the 32-byte hash of each sealed transaction; the rest need only remain available until it is opened, for which a blob, retained for about 18 days, suffices.

Sharing puzzles. One puzzle can cover all of a sender's transactions that should open together. It cannot be shared with unrelated senders: everyone who encrypts under a key knows it. Sharing the cost of opening across senders is delay encryption, which hashes cannot provide when the setup is cheap ([Section 8.1](#)). A sender without sufficient hardware can instead outsource building to several services, and encrypt under a hash of all their keys, which stays secret if any one service is honest. Opening, however, needs every service's puzzle to be well formed, and the transaction carries all of them, so its size and the solvers' work grow with the number of services.

7 Comparison with Other Encrypted-Mempool Designs

7.1 The design space

Every encrypted mempool must answer who opens a transaction once it is ordered. [Table 7](#) compares the answers; the survey of Passerat-Palmbach [[PP26](#)] and the critique of Garimidi,

Bonneau and Heimbach [GBH25] discuss them at length.

The table exhibits a trade-off. Every design in the table that keeps excluded transactions private has a party that can refuse to open, and that party is also the one that can fail to open; [Claim 1](#) says this is forced for input-free fallbacks. Among the designs compared here with an input-free fallback, leanSeal is the only one with no setup and a plausible post-quantum argument. Relative to the other designs, it has three costs: bytes (152 to 512 per transaction, against a 32-byte commitment), the sender’s computation, and minutes rather than seconds for a forced opening.

DecryptChain [AK26] also removes the fixed committee. It keeps a decryption committee but opens its membership through public proof-of-work in each epoch, so that a compromised committee can be replaced and committed transactions are eventually decrypted. In leanSeal, no committee is involved: the fallback is the sender’s own puzzle, which anyone can solve.

The closest work is Naraghi’s analysis of reveal optionality [Nar26], which appeared while this paper was being written. Three of its results bear on leanSeal.

- **A timing bound.** Its Theorem 2 covers every forced opening whose inputs are public at broadcast: privacy until the order is fixed and opening by the execution deadline are compatible only if the attacker is at most an explicit ratio of protocol times faster than the honest solvers, a ratio below two once a ciphertext waits a slot in a public mempool and must execute in the next. leanSeal is consistent with this because the sender’s reveal is the timely path; the forced opening may finish minutes later, at the cost of late or no execution ([Section 6.2](#)). With 200 s from the fixed order to the execution deadline, about one laptop core’s time for the reference puzzle, a 4-second ordering delay and a 12-second wait, the bound allows an attacker 13.5 times faster than honest solvers starting at broadcast, or 12.5 if they start once the order is fixed; our assumed attacker is 6.6 times faster than a laptop core.
- **Fee inversion.** Pricing a missed reveal with a forfeitable fee, as LUCID does, is much weaker when two commitments may execute ahead of the victim: the attacker can arrange to forfeit the large fee only when its guess about the victim is right, so once its best guess has probability above $2/(F + 1)$, where $1/F$ is the share of the fee a revealed transaction pays, deterrence needs a fee of at least half its gain. This applies to leanSeal’s deposit as well ([Section 6.2](#)). Under BSR, below, the single-commitment fee suffices again, because every reveal decision is fixed before any key is public.
- **Blind simultaneous release (BSR).** BSR confines a time lock to the decision window. Right after the order is fixed, each releaser (a key publisher, a committee member, or a sender holding a fallback key) escrows its release value under a timed pad, a precomputed iterated hash, within a window of one or two seconds. After the window, values are revealed or forced open in time for the batch to execute at the top of the next payload, and a releaser whose escrow does not open its commitment is slashed. Privacy before ordering comes from the underlying scheme, not from the pad. On a ticket-first design with twelve-second slots, forced opening in time needs funded solvers within a factor of 3 to 3.5 of the attacker, or 7 to 8 with a one-second window.

BSR and leanSeal solve different problems. leanSeal keeps a transaction private before ordering without a committee or key publisher and guarantees that it is eventually disclosed; BSR makes release decisions simultaneous, which removes the option to decide after seeing others’ reveals. They could compose if a leanSeal sender acted as its own releaser and escrowed K in the BSR window, which would supply the simultaneous release that leanSeal leaves open ([Section 6.2](#)); Naraghi lists such end-user escrows as open, because of the deposit’s capital cost and the pad’s precomputation. A leanSeal puzzle could serve as the pad: one that covers a 2-second window against our assumed attacker builds in about 1.4 s on the M4 Air, which bears on the precomputation barrier, but opens in about 13 s on one laptop core, so it still needs BSR’s funded solvers.

Table 7: Who opens a sealed transaction, and what each answer rests on. “Excluded” says whether a transaction that is never included stays private.

Design	Privacy before ordering rests on	Opening guaranteed if	Excluded	Setup, PQ
Private order flow	the builder or relay	the builder includes it	if trusted	none; n/a
Commit-reveal, LUCID self-decryption [EFM26]	nothing	the sender chooses to	private	none; yes
Threshold committee: Shutter [Shu], Aptos [Apt26], batched schemes [CGPW24]	no threshold of members colluding	a threshold is online	private with per-transaction keys [CGPP24]	DKG or setup; no
Hybrid [WM25]	no threshold colluding	the sender <i>or</i> a threshold is online	private	as threshold
TEE builder	the vendor and physical security [TEE25]	the operator is online	private	attestation; n/a
RSA time lock with proofs [Rad22]	the hardware estimate	always; proofs check the puzzle	opens after T	RSA modulus; no
Delay encryption [BD21]	the hardware estimate	one solve per round	opens after T	trusted setup; no
Blind simultaneous release [Nar26], a layer on the above	the underlying scheme	the releaser escrows in time, and the attacker is at most 3 (2-s window) or 7 (1-s window) times faster than funded solvers	as the underlying scheme	as the underlying scheme; pads plausibly
leanSeal	the hardware estimate	the sender reveals <i>or</i> the puzzle is well formed	opens after T	none; plausibly
leanSeal OR committee	both	the sender, a threshold <i>or</i> the puzzle	opens after T	as threshold

Our analysis also supplies concrete figures that Naraghi’s leaves open. His pad is a single hash chain whose sequentiality is taken from the random-oracle model, without a concrete bound; a leanSeal puzzle with $n = 1$ has this form, and [Theorem 1](#) gives it one. Our hardware model ([Section 4.3](#)) bears on the hardware gap that he lists as open: against our assumed attacker, a factor of 3 to 3.5 requires honest solvers at 10 to 12 ns per hash, faster than one Apple M4 core (23.2 ns) or the Xeon core Naraghi measures (70 ns) and close to the fastest published design (13.0 ns); a factor of 7 to 8 admits a laptop core. These figures rest on our estimate, not on a measurement.

7.2 Composing leanSeal with a threshold committee

A time lock and a committee fail in different ways: the time lock relies on a hardware estimate and leaks excluded transactions, and the committee relies on no threshold of its members colluding and on enough of them being online. The two can be combined in two ways. Let K_{tl} be a leanSeal key and K_{th} a key that a threshold committee releases for included transactions.

- **AND.** Encrypt under $H(K_{tl}, K_{th})$. Reading the transaction needs both keys, so the contents stay hidden while the committee is honest *or* the delay has not passed. With an honest committee, excluded transactions are never revealed. With a colluding committee, the puzzle still keeps the contents hidden for the delay, so the committee cannot read transactions before ordering them. Opening needs both the committee and the puzzle (or the sender’s reveal), so a committee that goes offline blocks it.

- **OR.** Encrypt the transaction’s key twice, once under each. Either key opens the transaction, so opening needs the committee *or* the puzzle: a committee that withholds keys or goes offline cannot block the opening. Privacy needs the committee to be honest *and* the delay not to have passed.

Table 8 summarizes this. In both, the sender’s own reveal remains the fast path. The OR form has the structure of the hybrid designs of Section 7.1, with leanSeal as a fallback that requires no trusted party; the AND form uses leanSeal as a floor for privacy under a committee.

7.3 Settings where leanSeal is practical

leanSeal is practical where three conditions hold: the order is fixed within seconds of exposure, a deposit is acceptable, and a delay of minutes is tolerable for a withholding sender’s transaction. Three settings satisfy them.

- **The self-decryption path of LUCID or a hybrid.** Here leanSeal serves as a fallback that requires no trusted party. In the OR form it removes the hybrid design’s failure case, a withholding sender while the committee is offline, at the cost of 92 to 452 bytes of puzzle per transaction. Inside LUCID it uses only the puzzle, in the envelope’s header; executing a transaction disclosed late needs a protocol rule LUCID does not have.
- **Sealed-bid and batch auctions.** Settlement can wait for every bid to open, so a withheld bid is forced open and still binds its bidder. leanSeal suffices on its own once puzzles can be checked (Section 8.3) and bids are escrowed.
- **Rollups and chains with fast finality,** where a sequencer or a fast-finality protocol fixes the order within seconds, so a lock of 15 to 30 seconds leaves a wide margin.

It is impractical for transactions that wait several slots in a public mempool. The clock starts at exposure, so a two-slot wait exhausts a 30-second lock; broadcasting only a commitment and publishing the puzzle once the slot is reserved [Nar26], or sending it only to the slot’s builder, removes the wait. It is also impractical wherever excluded transactions must stay private and no committee is acceptable: an input-free fallback cannot do both (Claim 1), and openings that take later inclusion witnesses, which are outside this claim, are not practical today.

8 Limits, Related Work and Open Questions

8.1 Limits of hash-only constructions

The delay is limited by the sender’s work. Mahmoody, Moran and Vadhan [MMV11] proved that if building a puzzle takes N hash calls, an attacker with unlimited parallelism can open it in about N oracle rounds. The chained puzzle needs about N rounds, so it is asymptotically optimal among random-oracle constructions: no hash-only puzzle can make the sender’s work grow more slowly than the delay. The attack makes an enormous number of queries, so the result limits what a proof can promise rather than giving a cheap way to open puzzles.

Delay encryption. Suppose a public setup costs S hash calls, after which senders encrypt to a slot with E calls each and one long computation per slot opens everything. Then running the setup and encrypting is itself a puzzle that costs $S + E$ calls, so by the result above it opens in about $S + E$ rounds: with a cheap setup, the delay is limited by the cost of one encryption. A long setup is not covered by this argument, but if each slot then has a unique, quickly checkable key, computing it is a verifiable delay function, which no hash-only construction can give [MSW20, GRY25, AAK⁺26]. No method is known that opens independent puzzles for less than one full solve each.

Table 8: Composing leanSeal (time lock, TL) with a threshold committee (TH).

	Contents hidden while	Opening guaranteed if	Excluded transactions
TL alone	delay not passed	puzzle well formed	revealed after the delay
TH alone	committee honest	committee online	private, if the committee opens only included ones
TL AND TH	committee honest <i>or</i> delay not passed	committee online <i>and</i> puzzle well formed	private while the committee is honest
TL OR TH	committee honest <i>and</i> delay not passed	committee online <i>or</i> puzzle well formed	revealed after the delay

Size. Each chain the sender runs in parallel needs its own hidden link, so a chained puzzle that allows the sender g -fold parallelism has about gb bytes. Whether a hash-only puzzle can be smaller is open.

8.2 Related work

Rivest, Shamir and Wagner [RSW96] introduced time-lock puzzles from repeated squaring in RSA groups, which need a trusted setup or class groups and are broken by quantum computers. Later algebraic constructions trade one of these drawbacks for another. Post-quantum time-lock puzzles from lattices [AMZ24] build in time logarithmic in the delay after preprocessing, but rest on circular small-secret LWE and have not been engineered for practical use. Delay encryption from isogenies [BD21] opens a whole slot with one computation but needs a trusted setup and is broken by quantum computers. Batch-solvable puzzles [DGM24] use pairings, and tlock [GMR23] encrypts to a future round of a threshold beacon.

Hash-only puzzles avoid these assumptions. Mahmoody, Moran and Vadhan [MMV11] gave the chained-hash puzzle, and Afshar et al. [ACH⁺23] showed that it keeps its delay against quantum openers in the quantum random-oracle model. Proofs of sequential work [MMV13, CP18] and verifiable delay functions [BBBF18] are related notions with different requirements. The chained-hash idea has been implemented several times, by Branwen [Bra11], Johnson [Joh13], Todd [Tod14] and EtherDream [Eth24], without a concrete bound or an explicit attacker model.

Timed commitments [BN00] are the pattern leanSeal follows in the mempool, and their use against front-running is not new. Time-Capsule [DE20] gives multi-party timed commitments based on discrete logarithms, with puzzle difficulty set by a proof-of-work blockchain, so that operators cannot front-run committed transactions; Ambrona, Beunardeau and Toledo [ABT23] list front-running among the applications of their timed commitments. Multiparty delay encryption [KAJF23] keeps Ethereum transactions encrypted until execution using time-lock puzzles, assuming one honest attestation aggregator per slot. In leanSeal, the fallback is the sender’s own puzzle and needs only a hash function. Encrypted-mempool designs are compared in Section 7.

8.3 Open questions

1. **The speed of chained SHA-256.** Every parameter is set relative to ℓ . A circuit-level estimate by hardware designers would give the assumed advantage of 3.7 an empirical basis.
2. **Checking puzzles.** A proof over all nm chain steps would cost far more than building the puzzle, and it is not known whether a cheaper check, such as spot-checking the chains or the links between them, can suffice. Without such a check, a solver that finds a malformed puzzle could demonstrate it with a bisection game over checkpoints of the chains; Naraghi assumes a similar mechanism, without analysis, for the pads of blind simultaneous release [Nar26].

3. **A quantum bound for the exact encoding.** Extending the analysis of Afshar et al. [ACH⁺23] to truncated links, the accumulator and the commitment would give a quantum counterpart of Theorem 1.
4. **Mechanism design.** Open problems include deposit sizes that remain effective under fee inversion [Nar26], the payment of solvers, the simultaneous release of keys and rules for executing transactions that are opened late, each best evaluated within a protocol such as LUCID.

Acknowledgments

We thank Benedikt Wagner and Justin Thaler for helpful discussion and feedback.

AI Assistance

Claude Opus 5.5, GPT-6 Astra and GPT-5.6 Sol were used extensively in this work, under the authors' direction. Claude Opus 5.5 wrote the implementations (the earlier C implementation, the Rust implementation and the Python reference), the benchmarks and the proof-of-concept contracts, drafted the proof of Theorem 1, and helped prepare and edit the manuscript. GPT-6 Astra reviewed all of this work: it audited the specification, the security argument and the C implementation over nine rounds, and reviewed the Rust implementation, the proof, the proof of concept and this paper. GPT-5.6 Sol audited the whole repository, and Claude Opus 5.5 audited the proof of concept; the audit reports, and the changes made in response, are in the repository [lea26]. The authors take full responsibility for the accuracy and correctness of the paper and for any issues in the implementation.

References

- [AAK⁺26] H. Abusalah, K. Azari, C. Kamath, E. Tairi, and M. von Consbruch. Impossibility of VDFs in the ROM: The complete picture. In *Advances in Cryptology – EUROCRYPT 2026*, 2026. Cryptology ePrint Archive, Paper 2025/1773.
- [ABT23] M. Ambrona, M. Beunardeau, and R. R. Toledo. Timed commitments revisited. Cryptology ePrint Archive, Paper 2023/977, 2023.
- [ACH⁺23] A. Afshar, K.-M. Chung, Y.-C. Hsieh, Y.-T. Lin, and M. Mahmoody. On the (im)possibility of time-lock puzzles in the quantum random oracle model. Cryptology ePrint Archive, Paper 2023/932, 2023.
- [AK26] N. Alhaddad and A. Kavousi. DecryptChain: A permissionless proof-of-work encrypted mempool. Cryptology ePrint Archive, Paper 2026/1195, 2026.
- [AMZ24] S. Agrawal, G. Malavolta, and T. Zhang. Time-lock puzzles from lattices. In *Advances in Cryptology – CRYPTO 2024*, 2024. Cryptology ePrint Archive, Paper 2025/047.
- [Apt26] Aptos Foundation. AIP-144: Encrypted mempool. Aptos Improvement Proposal, 2026.
- [BBBF18] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *Advances in Cryptology – CRYPTO 2018*, 2018. Cryptology ePrint Archive, Paper 2018/601.
- [BCF⁺20] L. Baldanzi, L. Crocetti, F. Falaschi, M. Bertolucci, J. Belli, L. Fanucci, and S. Saponara. Cryptographically secure pseudo-random number generator IP-core based on SHA2 algorithm. *Sensors*, 20(7):1869, 2020.

- [BD21] J. Burdges and L. De Feo. Delay encryption. In *Advances in Cryptology – EUROCRYPT 2021*, 2021. Cryptology ePrint Archive, Paper 2020/638.
- [BN00] D. Boneh and M. Naor. Timed commitments. In *Advances in Cryptology – CRYPTO 2000*, 2000.
- [Bra11] G. Branwen. Time-lock encryption. Essay, 2011.
- [CGPP24] A. R. Choudhuri, S. Garg, J. Piet, and G.-V. Policharla. Mempool privacy via batched threshold encryption: Attacks and defenses. *USENIX Security*, 2024.
- [CGPW24] A. R. Choudhuri, S. Garg, G.-V. Policharla, and M. Wang. Practical mempool privacy via one-time setup batched threshold encryption. Cryptology ePrint Archive, Paper 2024/1516, 2024.
- [CP18] B. Cohen and K. Pietrzak. Simple proofs of sequential work. In *Advances in Cryptology – EUROCRYPT 2018*, 2018. Cryptology ePrint Archive, Paper 2018/183.
- [DE20] Y. Doweck and I. Eyal. Multi-party timed commitments. arXiv:2005.04883, 2020.
- [DGM24] J. Dujmovic, R. Garg, and G. Malavolta. Time-lock puzzles with efficient batch solving. In *Advances in Cryptology – EUROCRYPT 2024*, 2024. Cryptology ePrint Archive, Paper 2023/1582.
- [Dra18] J. Drake. VDF-based RNG with linear lookahead. Ethereum Research forum, July 2018.
- [EFM26] A. Elowsson, J. Florentine, and J. Ma. EIP-8184: LUCID encrypted mempool. Ethereum Improvement Proposal, draft, 2026.
- [Eth24] EtherDream. Timelock online. Browser implementation; 2024 design with PBKDF2: <https://raw.githubusercontent.com/EtherDream/timelock/e087ed1da8702244a8359097bd26ce2aca57f770/README.md>, 2024.
- [GBH25] P. Garimidi, J. Bonneau, and L. Heimbach. On the limits of encrypted mempools. a16z crypto, 2025.
- [GMR23] N. Gailly, K. Melissaris, and Y. Romainier. tlock: Practical timelock encryption from threshold BLS. Cryptology ePrint Archive, Paper 2023/189, 2023.
- [GRY25] Z. Guan, A. Riazanov, and W. Yuan. Breaking verifiable delay functions in the random oracle model. In *Advances in Cryptology – CRYPTO 2025*, 2025. Cryptology ePrint Archive, Paper 2024/766.
- [Her26] G. Herold. Towards encrypted mempools from threshold IBE without batching. Ethereum Research forum, September 2026.
- [HS03] D. Harris and I. Sutherland. Logical effort of carry propagate adders. In *Asilomar Conference on Signals, Systems and Computers*, 2003.
- [Joh13] D. Johnson. timelock. Go implementation, 2013.
- [KAJF23] A. Khajepour, H. Akbarinodehi, M. Jahanara, and C. Feng. A mempool encryption scheme for Ethereum via multiparty delay encryption. Cryptology ePrint Archive, Paper 2023/1612, 2023.
- [LCV07] Y. K. Lee, H. Chan, and I. Verbauwhede. Iteration bound analysis and throughput optimum architecture of SHA-256 (384, 512) for hardware implementations. In *Information Security Applications (WISA)*, 2007.

- [lea26] leanSeal: Specification, implementation, tests, audit reports and benchmark logs. <https://github.com/leanEthereum/leanSeal>, 2026.
- [LJKW18] J. Liu, T. Jager, S. A. Kakvi, and B. Warinschi. How to build time-lock encryption. *Designs, Codes and Cryptography*, 86, 2018. Cryptology ePrint Archive, Paper 2015/482.
- [MMV11] M. Mahmoody, T. Moran, and S. Vadhan. Time-lock puzzles in the random oracle model. In *Advances in Cryptology – CRYPTO 2011*, 2011. Harvard DASH copy.
- [MMV13] M. Mahmoody, T. Moran, and S. Vadhan. Publicly verifiable proofs of sequential work. In *4th Innovations in Theoretical Computer Science Conference (ITCS)*, 2013.
- [MSW20] M. Mahmoody, C. Smith, and D. Wu. Can verifiable delay functions be based on random oracles? In *47th International Colloquium on Automata, Languages, and Programming (ICALP)*, 2020. Cryptology ePrint Archive, Paper 2019/663.
- [Nar26] Aria Naraghi. Reveal optionality in encrypted mempools: Separation, pricing, and blind simultaneous release with post-quantum time-locks. Manuscript, summarized on the Ethereum Research forum, October 2026.
- [NCTI21] S. Nikolić, F. Catthoor, Z. Tőkei, and P. Ienne. Global is the new local: FPGA architecture at 5nm and beyond. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021. Table 5.
- [PP26] J. Passerat-Palmbach. SoK: Encrypted mempools through the MEV lens. In *Advances in Financial Technologies (AFT 2026)*, volume 395 of *LIPICs*, 2026.
- [PR26] K. Pietrzak and L. Reyzin. Optimal proofs of sequential work. Cryptology ePrint Archive, Paper 2026/2569, 2026.
- [Rad22] Radius. MEV-resistant ZK-rollups with practical VDE (PVDE). Ethereum Research forum, 2022.
- [RSW96] R. Rivest, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, MIT Laboratory for Computer Science, 1996.
- [Shu] Shutter Network. The road towards an encrypted mempool on Ethereum L1.
- [SSM18] V. Suresh, S. Satpathy, and S. Mathew. Bitcoin mining hardware accelerator with optimized message digest and message scheduler datapath. US patent application 2018/0089642 A1 (Intel), 2018.
- [TEE25] TEE.fail: Breaking trusted execution environments via DDR5 memory bus interposition, 2025.
- [Tod14] P. Todd. timelock. Software, with an announcement of a funded puzzle on the bitcoin-development mailing list, June 2014: <https://www.mail-archive.com/bitcoin-development@lists.sourceforge.net/msg05547.html>, 2014.
- [WM25] B. Wagner and J. Ma. Hybrid encrypted mempools. Ethereum Research forum, 2025.

A Proof of the Delay Theorem

Proof of Theorem 1. We compare the real experiment with an ideal one in which the values Open computes are drawn only when the attacker reaches them, show that the ideal experiment is sequential, and bound the probability that the two differ.

Path values. For each chain i , let $y_{i,0} = s_i = H_{\text{start}}(\text{ctx}, i, x_i)$ and $y_{i,k} = F_i(y_{i,k-1})$ for $k = 1, \dots, m$, so that $y_{i,m} = y_i$. Let $\mu_i = H_{\text{mask}}(\text{ctx}, y_i)$ for $i < n$, $a_0 = \text{ctx}$, $a_i = H_{\text{acc}}(a_{i-1}, y_i)$, $K = H_{\text{key}}(\text{ctx}, a_n)$ and $C = H_{\text{commit}}(\text{ctx}, K)$. These P values are the *path values*. Each path value v is the answer on one *path input* $u(v)$. The *predecessors* of v are the path values in $u(v)$, at most two; in addition, μ_{i-1} is a predecessor of s_i for $i \geq 2$, since $x_i = z_{i-1} \oplus [\mu_{i-1}]_b$.

Let $\rho(v)$ be the oracle round in which Open receives v :

$$\rho(s_i) = (i-1)(m+2) + 1, \quad \rho(y_{i,k}) = \rho(s_i) + k, \quad \rho(\mu_i) = \rho(a_i) = \rho(y_i) + 1, \quad \rho(K) = T_{\text{key}}.$$

Every path value v other than s_1 satisfies $\rho(v) = 1 + \max_w \rho(w)$, the maximum taken over its predecessors w .

The two experiments. In the real experiment R , the oracles are random functions sampled lazily; $\mathcal{A}_0$ runs, Build runs, and $\mathcal{A}_1$ receives Z . The ideal experiment G uses the same lazily sampled oracles, but Build makes no queries. It draws the salt and x_1 as before, and draws $z_1, \dots, z_{n-1}$ and C uniformly. A query of $\mathcal{A}_1$ is *legitimate* if it equals $u(v)$ for a path value v whose predecessors were all revealed before the query was submitted, that is, in earlier rounds; for s_i with $i \geq 2$ this defines $x_i = z_{i-1} \oplus [\mu_{i-1}]_b$. G answers a legitimate query with v , drawing v uniformly the first time (C was drawn already), and marks v revealed at the end of the round; a repeated legitimate query receives the same v . It answers every other query, and all queries of $\mathcal{A}_0$, from the lazily sampled oracles. When $\mathcal{A}_1$ stops, G draws the path values never revealed, which fixes the remaining links.

Drawing in advance. G reads a path value only after revealing it, and C only to publish it: deciding whether a query is legitimate uses only the values of revealed predecessors, and uses x_i only once μ_{i-1} is revealed. So G has the same distribution if all its randomness is drawn at the start: the salt, x_1 and $z_1, \dots, z_{n-1}$; an independent uniform value $v(v)$ for every path value v , with $v(C)$ published as C ; and lazily sampled oracles H' . We use this form below. The path values are the $v(v)$, and they determine the path inputs: $u(s_1)$ from the salt and x_1 , $x_i = z_{i-1} \oplus [v(\mu_{i-1})]_b$ for $i \geq 2$, and every other $u(v)$ from the values of the predecessors of v .

Claim 2. In G , no path value v is revealed before oracle round $\rho(v)$.

Proof of the claim. By induction on $\rho(v)$; for s_1 , with $\rho = 1$, there is nothing to show. A value is revealed only by a legitimate query, which is submitted after all its predecessors are revealed. Answers arrive at the end of a round, so if a predecessor w is revealed in round $\rho(w)$ or later, v is revealed in round $\rho(w) + 1$ or later; hence in round $1 + \max_w \rho(w) = \rho(v)$ or later. $\diamond$

Claim 3. Let **Bad** be the event, in G , that (a) two path values are equal, or (b) a query that is not legitimate equals a path input, evaluated with the path values as finally drawn. Then the probability that $\mathcal{A}_1$ outputs K early in R is at most the probability that it does so in G plus $\Pr_G[\text{Bad}]$.

Proof of the claim. We build R from the randomness of G , drawn in advance.

Links. Set $x_{i+1} = z_i \oplus [v(\mu_i)]_b$ for $i < n$. Since z_i is uniform and independent of everything else, the links $x_2, \dots, x_n$ are uniform and independent of the salt, x_1 , v and H' , as Build draws them.

Oracle. Define H by following the honest path in order of ρ (ties in a fixed order): the input $u(v)$ of each path value v receives the answer $v(v)$, unless an earlier path value has the same input; every other input receives the answer of H' . Each path input is determined by the salt, the links and earlier answers, and each new path input receives its own answer $v(v)$, drawn independently and uniformly. By the usual lazy-sampling argument, H is a uniformly random oracle independent of the salt and the links, so running $\mathcal{A}_0$, Build and $\mathcal{A}_1$ with H is the experiment R . Because H is a single function, an answer that $\mathcal{A}_0$ obtains and that Build or $\mathcal{A}_1$ later obtains again is the same answer.

Commitment and puzzle. Suppose (a) fails. Then the path inputs are distinct: inputs to different oracles are different points, the starts carry their chain index, and two equal inputs to one oracle

contain equal path values. So H answers every path input $u(v)$ with $v(v)$, and the values that Build computes are the $v(v)$. Its commitment is $H(u(C)) = v(C)$, the C that G publishes, and its masked links are $[v(\mu_i)]_b \oplus x_{i+1} = z_i$. The two puzzles are equal.

Queries. Off (a), H agrees with H' at every input that is not a path input. Compare the two runs query by query. A query of $\mathcal{A}_0$ is never legitimate; if it is not a path input, it receives the answer of H' in both experiments. A legitimate query of $\mathcal{A}_1$ equals some $u(v)$ and receives $v(v)$ in both, also when it is repeated. A query of $\mathcal{A}_1$ that is not legitimate and is not a path input receives the answer of H' in both. By induction on the queries, the transcripts, and so the sets of revealed values, agree up to the first query that is not legitimate and equals a path input. That query is event (b); in particular, a path input that $\mathcal{A}_0$ queried, whose answer Build would reuse, is event (b). So the outputs of the two experiments agree unless Bad happens. $\diamond$

Claim 4. *The probability of Bad in G is at most*

$$\Pr_G[\text{Bad}] \leq \frac{Q_{\text{on}}}{2^{8b}} + \frac{Q_{\text{pre}}}{2^{128+8b}} + \frac{QP + P^2}{2^{256}}.$$

Proof of the claim. In G , a query depends only on the values revealed before it was submitted, on the puzzle and on answers of H' . A path value other than C that is not revealed when a query is submitted is therefore independent of that query. So is $x_i = z_{i-1} \oplus [v(\mu_{i-1})]_b$ for $i \geq 2$ while μ_{i-1} is not revealed, and so are the salt and every link for every query of $\mathcal{A}_0$. The commitment C is public, but it is no predecessor of any path value and is independent of the others. Nothing is conditioned on: each bound below is the probability that a fixed query equals a value independent of it.

- Event (a): there are at most P^2 pairs of independent uniform 32-byte values, so its probability is at most $P^2/2^{256}$.
- Event (b), queries to H_{start} : a query names one index i and equals $u(s_i)$ only if it contains x_i . A query of $\mathcal{A}_1$ with $i = 1$ is legitimate whenever it equals $u(s_1)$. One with $i \geq 2$ that is not legitimate was submitted while μ_{i-1} was not revealed, so it contains x_i with probability 2^{-8b} ; this case needs $n \geq 2$. A query of $\mathcal{A}_0$ must also contain the salt, so it succeeds with probability $2^{-(128+8b)}$.
- Event (b), all other queries: a query that is not legitimate but equals $u(v)$ contains, in a 32-byte field, a predecessor of v that was not revealed when it was submitted. That happens with probability at most 2^{-256} for each of the at most P path inputs, so at most $P/2^{256}$ per query.

Summing over the Q_{on} and Q_{pre} queries gives the claim. $\diamond$

Conclusion. In G , if $\mathcal{A}_1$ stops before round T_{key} , then K is not revealed by Claim 2, so $v(K)$ is independent of $\mathcal{A}_1$'s output, and the output equals it with probability 2^{-256} ; $C = v(C)$ is independent of $v(K)$ and tells $\mathcal{A}$ nothing about it. Claims 3 and 4 add $\Pr_G[\text{Bad}]$. $\square$

The proof follows the structure of Pietrzak and Reyzin's soundness proof for proofs of sequential work (their Lemmas 6 and 9) [PR26].

B Estimating the Hardware Speed

This appendix gives the reasoning behind assumption A2 and the advantage factor A of Section 4.3.

How SHA-256 is computed in hardware. Assumption A2 can be examined against the structure of SHA-256. One SHA-256 compression is 64 rounds, and each round needs the exact output of the previous one. The new state words are sums of up to five 32-bit terms. Some of them, such as the round constant and the message word, can be added ahead of time, but the next round applies rotations and bitwise functions to the exact bits of the result. Every round must therefore finish at least one 32-bit addition with full carry propagation, plus a few levels of logic. Lee, Chan and Verbaauwhede [LCV07] show that this much is also enough: with carry-save adders and retiming, the loop through one round needs one carry-propagate addition, one carry-save layer and the choice function Ch.

Pipelining raises throughput but not the latency of one chain, since each step starts from the previous step's output. Unrolling removes the registers between rounds, so the relevant floor is the one for logic alone. Hence ℓ is roughly 64 times the shortest critical path of one round, measured over many unrolled rounds.

The circuit model. The model row of Table 4 quantifies this argument. Its unit is the FO4, the delay of an inverter driving four copies of itself. It estimates the round loop of Lee et al. at 10–17 FO4: 7–12 for a fast 32-bit adder [HS03] and 3–5 that we assume for the carry-save layer and Ch. Predicted FO4 delays for 3–5 nm processes are 4.3–4.7 ps [NCTI21]. Then 64 rounds take

$$64 \times 10 \times 4.3 \text{ ps} = 2.75 \text{ ns} \quad \text{to} \quad 64 \times 17 \times 4.7 \text{ ps} = 5.11 \text{ ns},$$

and with 0.1–0.3 ns for the final addition and startup, 2.85–5.41 ns. Our allowance of 1 to 1.8 times for wires, cell choice and registers gives 2.9–9.7 ns, that is, $A \approx 1.3$ –4.5.

As a cross-check, Intel's mining patent describes a state update 13 logic levels deep [SSM18]; if each level cost one FO4, which is optimistic, a hash would take 3.6–3.9 ns. A process shrink by itself gives only $A \approx 1.3$. At 3.5 ns a round has 55 ps, about 12 FO4, so the design point needs a highly optimized custom datapath and sits near the aggressive end of the model. The model's lower end is a conditional estimate, not a physical bound, and no published design that we know of comes close to it.

The precedent of verifiable delay functions. Work on verifiable delay functions faced the same question and also answered it with a stated factor: Drake proposed $A_{\max} = 10$ as a conservative parameter for RSA-based VDFs, hoping that public ASICs would bring it to about 5 [Dra18]. Here $A = 10$ means $\ell = 1.3$ ns, about 4.5 FO4 per round, outside the model but not ruled out. Section 8.3 lists a careful estimate as the most important open question.