Encrypted Mempools

Originally posted to Substack.

Introduction

Encrypted mempools are powerful tools in addressing MEV and censorship. There are a variety of related schemes which may be used in isolation or combined with others.

This post provides an overview of the design space largely built aroundJustin Drake's presentations on the topic (Columbia, Amsterdam, Flashbots Roast). I'm guessing most of you had a difficult time processing them on the first watch (or maybe I'm just slow, unclear), so I tried to break them down simply then expand a bit.

The basic idea - allow users to submit encrypted transactions. Block producers commit to these transactions before

decrypting them:

1. User encrypts and broadcasts transaction
2. Encrypted transaction is committed to
3. Transaction is decrypted
4. Transaction is executed (note: commit, decrypt, and execute can potentially be a single slot to preserve UX)

This looks to address two major problems:

- MEV

- E.g., I shouldn't be able to frontrun a trade if I can't see it

- Censorship

- I shouldn't be able to censor a transaction unless I decide to exclude all encrypted transaction (which is hopefully prohibitively expensive if many transactions are encrypted)

One key property we need here is guaranteed decryption withoutany dependency on the user. There are several ways to achieve this:

1. In-flight

Trust somebody. Privately send data to some party. That party can decrypt and see the data, but they promise not to reveal it publicly until it's been committed to on-chain. This is how products like Flashbots Protect work (and MEV-Share to start).

2. Trusted Hardware

You could leverage trusted hardware. The plaintext can be operated on within the trusted hardware, but it's not publicly decrypted until after the transactions have been committed. The most notable example is Flashbots' potential use of SGX in SUAVE.

3. Threshold Encryption/Decryption (TED)

A committee can jointly force decryption of the ciphertext. A "threshold" of signers are required to do this. For example, your "threshold" could be that 2/3 of the validator set are required to agree to decrypt. TED has been researched in several different domains, including:

- Ethereum L1 - Shutterized Beacon Chain

- Shutter Network - Ethereum L2s

- Arbitrum - Chainlink's FSS
- Osmosis & Anoma - Ferveo

Shutterized Beacon Chain

Taking a closer look at the Shutterized Beacon Chain proposal, a subset of validators ("keypers") produce public/private key pairs for using distributed key generation (DKG). Everyone can see the public key, but the private key is split into shares amongst the committee. A threshold of the key shares (e.g., ⅔) is required to reconstruct the private key and decrypt.

Users would be allowed to optionally encrypt transactions which get included on-chain before their content is decrypted and executed. Block producers create blocks with plaintext transactions (executed immediately) and encrypted transactions (scheduled for a future block height).

After block n

-1 is produced and attested to, keypers should generate and release the decryption key for block n

. This key allows for decryption of the transactions scheduled for block n

. Block n

must include the decryption key to be considered valid. The post state of the block is computed by executing the encrypted transactions scheduled for that block (in their set order) before

executing the plaintext transactions included in that block.

Downsides

TED has several downsides though:

Lack of accountability -

Malicious behavior here is not attributable or reportable

unlike traditional security assumptions (e.g., equivocation is slashable). Keypers could decrypt immediately, and you have no way to attribute it. Even if you assume it's incentive compatible for validators to behave (which is questionable), three-letter agencies outside the validator set can and likely will exert influence on them.

Strong honest majority assumption

- A dishonest committee could create problems such as choosing never to decrypt certain messages, or secretly decrypting messages they're not supposed to (e.g., at the behest of government agencies, or selfishly to conduct MEV extraction). In most instances, blockchains are effectively relying on an economically rational majority (i.e., the punishment of slashing generally outweighs the economic incentive to be maliciously double-spend, etc.). With threshold encryption, the assumption is shifted to a truly honest and altruistic majority (though you may argue the social implications outweigh the cost).

Destabilizing to the protocol

- Essentially a mix of the above two points - TED meaningfully increases the incentives for validators to be malicious, and it can't be easily punished. This increases the incentive for centralization and malicious behavior.

Weakened liveness

- A higher t

of n

decryption threshold strengthens protocol safety (more difficult to misbehave). However, this directly weakens liveness. For example, a massive network outage cutting off half of validators could now halt the protocol. Particularly for Ethereum, which places a strong emphasis on "World War III" liveness (Gasper is liveness-favoring), this is an unacceptable tradeoff.

Worse UX

- TED can add varying levels of undesirable latency and cost depending on the implementation details.

Suboptimal value allocation

- Let's look at an oversimplified frontrunning example:

- The global mid-market price for 1 ETH = $1000
- If I send my trade to a vanilla mempool - I get $990 (searcher frontruns and backruns, netting $10)
- If I send my trade to a TED mempool - I get $995 (searcher can't frontrun, but I push the local spot price to get filled, creating a $5 arbitrage opportunity for searchers)
- I send my trade to an optimal[order flow auction] - I get $1000 (I don't get frontrun or

backrun. Or similarly, I do

get frontrun and/or backrun, but the searcher has to bid a $10 rebate in a competitive auction for the right to do so, achieving the same end result.)

In this scenario, you're leaving money on the table for someone else to take. An economically rational user should expose enough order information such that they are compensated nearly the full value of their order. They should not want to naively threshold encrypt their trade, hiding the value it offers (allowing someone else to capture it).

This is a hyper-idealized example, but you get the point. Sweeping the problem under the rug doesn't always address the problem. TED can be a useful tool, but it's not an MEV panacea. It's arguably better for CR in some scenarios.

Additionally, Ethereum specifically brings additional complexity. For example, its lack of instant finality (though this may change some day with [single-slot finality]) introduces [issues around reorgs] here. I believe the tradeoffs here outweigh the benefits, at least in L1 Ethereum's case. Increasing Ethereum's trust assumptions, increasing incentive to collude (without detection), and decreasing its liveness violate more fundamental properties than the benefits it may bring.

However, some of these tradeoffs may be more acceptable for L2s or other L1s than they would be for Ethereum L1.

4. Delay Encryption/Decryption (DED)

With delay encryption, you can encrypt information which is set to automatically decrypt after some amount of time. This is a sequential computation related to VDFs.

To practically implement DED, you'll need VDF ASICs. Luckily, the[Ethereum Foundation and Protocol Labs have been working on getting these built], recently getting the first test samples of chips built by[GlobalFoundries]. These VDF evaluators should be capable of extremely fast sequential computation. These VDF ASICs also work for time-lock puzzles and DED.

The VDFs would also require production of SNARKs. This could be done using GPUs, but ideally the EF would also like to produce the relevant ASICs for SNARKs here as well eventually.

Overall, you're no longer trusting a committee here, so the security assumptions are generally preferred. However, needing specialized hardware isn't ideal, and the required delays may add latency to your system. More interesting, you could combine the best of TED + DED using...

5. Witness Encryption/Decryption (WED)

WED is powerful, but before you get too excited - it's not coming anytime soon. This is extra fancy stuff that's many years away if it happens. But it's cool and helpful to understand, so I'll cover it briefly.

WED allows for any arbitrary witness to force decryption of the ciphertext. For example, your rule could be "only decrypt once Ethereum has finalized the block including the encrypted payload," and it will only decrypt with a proof of this. You could imagine requiring a SNARK for basically any complicated statement you want fulfilled.

WED then is effectively a generalization of TED or DED. You could even build a hybrid of the two. For example, the designated witness could be a proof that either:

- Happy case

- the threshold committee has signed off on decryption, or

- Backup

- the requisite time has passed, fallback to delay decryption

This provides a backup in case the threshold committee is unresponsive, so you won't lose liveness. This also allows you to be more conservative in parameterizing your m of n threshold assumption (i.e., you could require every committee member or nearly all of them to sign off).

- Happy case

- optimal UX with instant confirmation by the entire committee

- Backup

- latency and UX suffers temporarily, but liveness is retained

In practice, WED would likely be verifying a SNARK for whatever statement you want. For example:

- TED

- You could have a SNARK proving the statement "I have the requisitem

out of n

signatures" to force the decryption.

- DED

- You could use a VDF and do the sequential computation (which is very expensive). Then at the end of your computation, you have a short proof. Then you take that short proof and wrap it in a SNARK (to make it even shorter), and feed that in to force decryption.

Homomorphisms

For each of these encryption schemes (threshold, delay, and witness), you can have homomorphisms. That is, it's possible to create threshold FHE, delay FHE, and witness FHE schemes that enable arbitrary operations on ciphertext.

The in-flight and trusted hardware (e.g., SGX) solutions can also simulate the same functionality - operating on private data. That's because in both of these cases, the trusted third party or hardware has access to the plaintext itself to operate on. In an ideal world, we'd like to remove these trust assumptions though, relying on hardened cryptography alone.

We could imagine an example given:

- m1

= transaction 1

- m2

= transaction 2

- f(x) = build a block

Then all of the five solutions could replicate the following functionality:

Keep this ability in mind as we think about how to hide transaction metadata and build optimal blocks with encrypted inputs.

Each of the five options here could in theory be used for the solutions below requiring "homomorphisms."

Metadata

Metadata is data that provides information about other data, but not the content of that data (such as the text of a message or the image itself). For our purposes, transaction metadata could include the sender's IP address, nonce, gas payment, etc.

Ideally, we would want to hide as much metadata as possible. Otherwise, it could leak information which allows outsiders to reason about your transaction (potentially giving them enough information to try to frontrun you, censor you etc.). However, we need metadata to verify that a transaction is valid, pays the requisite fees, isn't spam, etc. You don't want to commit to an encrypted transaction if it's actually invalid.

So can we have our cake and eat it too? Let's walk through some solutions for shielding various metadata.

1. IP Address - Tor

Pretty straightforward - people could use services like Tor to broadcast privately and shield their IP address.

2. Signature - SNARK

You need to know at the p2p level that an encrypted transaction is valid, and in particular that it has a valid signature. We can use a SNARK to prove that the signature is valid, and this would accompany each encrypted transaction. The SNARK has three important parts:

Let's unpack that. The user wants to prove that their tx ciphertext is valid, it corresponds to the tx plaintext, and its signature is valid. To prove that any signature is valid, you need the public key associated with the account to verify it against. So:

1. You look up the Ethereum state root (public).
2. You provide the sender pubkey Merkle proof (private) which corresponds to this state root. This proves their public key.
3. To verify the signature, you verify the sender pubkey Merkle proof against the state root. You prove with a SNARK that your sender pubkey Merkle proof is valid and your signature is valid.

3. Gas Payment - SNARK

You need to prove that the sender of an encrypted transaction has enough ETH to pay for it. We can again use a SNARK.

It's a similar process, but now you provide a Merkle path to the state root which proves the sender balance

(instead of the sender public key

). You verify the Merkle path and verify that the balance is sufficient to pay the required gas with a SNARK.

The minimum gas to just do the basics like deserialize a transaction, check its balance, and check the signature can be the minimum 21,000 gas. You could have a rule that the sender always has to pay this minimum.

But you also need to have the gas limit for the actual transaction itself when it comes to time of execution, and this gas limit can also be encrypted. So when you go to actually execute the transaction itself, you would check if the sender actually has sufficient balance to pay the full gas limit for the transaction at that time:

- If yes - run the transaction and issue a refund at the end
- If no - abort transaction, and the sender just pays the 21,000 gas

4. Sender & Nonce - SNARK

A nonce is a value equal to the number of transactions sent from an address (or the number of contract-creations made by an account). A user's nonce increments by one for each transaction from their address.

One reason we have nonces today is so that users can't spam the mempool (DoS attack) with a bunch of transactions to increase their likelihood of being executed. Nodes will only look at one transaction with that nonce and drop the others.

For an encrypted transaction, you'll prove that its nonce is the previous nonce + 1 with a SNARK. However, this could be vulnerable to the above DoS attack if done naively - nodes will no longer be able to tell that many encrypted transactions have the same nonce.

To preserve this anti-DoS property, we would like to add a "replay tag." This uniquely tags transactions, so you can no longer spam a ton of transactions for one address. To create this tag, you could simply hash (nonce, private key). Because both the nonce and private key are fixed, different transactions would have the same replay tag if the user tries to send both with the same nonce and private key.

Lastly, you may want to retain some level of flexibility. Let's say gas prices have moved, and you reasonably want to rebroadcast your transaction at a higher price. To alleviate this, you could hash (nonce, private key, slot). Now, you would be able to adjust your transaction once per slot. You are limited to sending one transaction per address per slot.

5. Size - Homomorphisms

Naively encrypting a plaintext of sizen

would create a ciphertext of size n

. Even this size may be enough to probabilistically figure out certain transactions. We could alleviate this by "padding" - adding 0s to the plaintext before encryption, rounding up the number of bits to the next power of 2 (e.g., add three 0s to a ciphertext with 5 bits).

In the following examples:

- White = 0s for padding
- Other Colors = actual transaction data

When you build your block, you could concatenate these ciphertexts, but this has two problems:

- Imperfect packing

- Extra 0s mean you have to pay for extra data availability to get them on chain

- Imperfect privacy

- Maybe even the power of 2 size is enough information. The distribution of transaction size is unclear, so certain sizes may still reveal enough information probabilistically.

A fancier solution involves "clipping" away the padding:

1. Pad every transaction to the maximum size (16 here) and encrypt them homomorphically
2. Apply a function over the ciphertext to pack the plaintext efficiently, removing unnecessary padding

Now you get optimal packing, and all encrypted transactions look the same.

By padding to the "maximum" size this could literally be setting the maximum to what the actual block size limit translates to (i.e., if 30mm gas translates to roughly x

kB). This would mean padding each transaction to that size of x

kB. Alternatively, you could set a smaller more practical size of y

kB if say 99.9% of transactions are normally within this bound.

Similarly, this "clipping" could be simulated using trusted hardware instead of FHE as discussed earlier. For example, you could run SGX and receive encrypted transactions to your trusted enclave. Within the enclave, the data can be operated on

in the clear to clip the unnecessary padding to densely pack the block. The data is then encrypted and shipped back out, which is followed by one of the forced decryption methods (threshold, delay, witness).

Note that the output should be a certain fixed size in either case, so you may have a small amount of padding left.

Bundle Selection - Homomorphisms & Disjoint State Access Lists

So let's say we solved everything above. Block producers are able to optimally pack everything into a block in regard to size

. But we still have another problem - can block producers optimally pack everything in regard to economics

? We want decentralized block building without throwing away value. Otherwise, it won't be competitive with the centralized status quo.

Solution 1 - FHE EVM

The straightforward brute force answer - run the EVM entirely in FHE. This would allow decentralized building of the optimal block which captures the most value for the block producers. They select and order the transactions which maximize value.

Unfortunately, that's not so easy. The costly part of an FHE circuit is its "depth" (i.e., the number of gates in the longest path). In particular, its multiplicative

depth is what really matters. As an example, the transaction "clipping" to remove padding described earlier is a rather "shallow" circuit (i.e., it's pretty simple). Use cases such as this may be computationally feasible within 2-3 years with hardware acceleration.

However, running the entire EVM using FHE requires an incredibly deep and complex circuit. We're finally starting to make some meaningful progress with zkEVM circuits (which is orders of magnitude more computationally intensive than running a standard EVM), but a FHE EVM is even further out (think more like a decade).

Once again, note that it's also feasible to simulate this functionality with trusted hardware (e.g., as SGX in SUAVE will do).

Solution 2 - State Access Lists

Alternatively, we could have a more limited use of FHE coupled with access lists. This is far more achievable than running a full FHE EVM - it's a much shallower circuit again. In this setup, transaction bundles could include an encrypted state access list along with their bid. This specifies exactly which part of the state their bundle will access.

Now, the block producers' homomorphic circuit just has to pick the highest bids for bundles with disjoint state access lists (i.e., no bundles touch overlapping state).

Note that this is not strictly optimal. For example, it's possible that the second highest bid (T2

) in a block touches the exact same state as the highest bid (T1

). In this access list paradigm, the block producer would throw that second transaction away. However, there is a possibility that both T1

and T2

are both valid and do not make the other invalid even though they touch the same state. In this scenario, the entire FHE EVM (or centralized block production as today) would capture more value.

Although it's not perfect, from what we see today this shortcut should be close enough to optimal. The vast majority of MEV would still be captured using disjoint access lists, as clashing examples such as the one above are quite rare.

6. Timestamp - Homomorphisms

You could possibly even try to hide the leakage of the existence of transactions by allowing validators to make "dummy" transactions, so the mempool always appears full. Validators would also need a SNARK proving they're a validator allowed to create these dummy transactions. When there's a spike in "real" transactions, validators will broadcast less dummy

transactions. When activity drops, then validators would issue more dummy transactions.

Assuming everything is encrypted using FHE, you'd be able to detect a "dummy flag" added to the respective transactions. You'll know to ignore these transactions when packing the actual block.

State Diffing - FHE EVM

Quick background - one advantage of ZKRs is that they don't need to post full transaction data on-chain. It suffices for them to post the "state diff(erence)" between state updates. Conversely, optimistic rollups must post on-chain the full data in case there is arbitration of a fraud proof. This lower data requirement is a cost saver for ZK-rollups, as they consume less resources from their data availability layer.

Bringing this into our conversation here - you would ideally like to not lose this benefit in the context of encrypted mempools. You want to get encrypted mempools, and still preserve the ability for ZK-rollups to only post state diffs. The answer here - run an FHE zkEVM. Sounds cool, but again you might be waiting a bit on this one.

One downside is that posting state-diffs alone could make rollup full nodes less responsive. For example, in[pure-fork choice rule rollups], rollup full nodes can finalize their view of the rollup immediately as the DA layer finalizes and be assured of validity. If you're only posting state diffs (not full transaction data), even full nodes will require ZK proof submission before they can confirm their view of the rollup. As things stand today - these proofs take a while to generate, but scalable DA will hopefully be very cheap soon. So there's some tradeoff here.

Conclusion

Encrypted mempools are a fascinating and promising design space. Unfortunately, practical challenges still exist. For example, you may just shift certain problems elsewhere in the stack. Wallets could censor you at the source preventing transaction encryption/routing, they could submit payloads to frontrun/backrun your trades (or sell the right), etc. One possible answer is for users to create transactions locally without needing a service provider.

The point is that nothing here is a silver bullet, but they're likely an important part of the solution if designed well. They can offer improvements to the current situation with varying efficacy and trust assumptions.