Mempool Privacy via Batched Threshold Encryption: Attacks and Defenses

Arka Rai Choudhuri, NTT Research; Sanjam Garg, Julien Piet, and Guru-Vamsi Policharla, University of California, Berkeley

https://www.usenix.org/conference/usenixsecurity24/presentation/choudhuri

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.
August 14–16, 2024 • Philadelphia, PA, USA
978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
Mempool Privacy via Batched Threshold Encryption: Attacks and Defenses

Arka Rai Choudhuri  
*NTT Research*

Sanjam Garg  
*UC Berkeley*

Julien Piet  
*UC Berkeley*

Guru-Vamsi Policharla  
*UC Berkeley*

Abstract
With the rising popularity of DeFi applications it is important to implement protections for regular users of these DeFi platforms against large parties with massive amounts of resources allowing them to engage in market manipulation strategies such as frontrunning/backrunning. Moreover, there are many situations (such as recovery of funds from vulnerable smart contracts) where a user may not want to reveal their transaction until it has been executed. As such, it is clear that preserving the privacy of transactions in the mempool is an important goal.

In this work we focus on achieving mempool transaction privacy through a new primitive that we term *batched-threshold encryption*, which is a variant of threshold encryption with strict efficiency requirements to better model the needs of resource constrained environments such as blockchains. Unlike the naive use of threshold encryption, which requires communication proportional to $O(nB)$ to decrypt $B$ transactions with a committee of $n$ parties, our batched-threshold encryption scheme only needs $O(n)$ communication. We additionally discuss pitfalls in prior approaches that use (vanilla) threshold encryption for mempool privacy.

To show that our scheme is concretely efficient, we implement our scheme and find that transactions can be encrypted in under 6 ms, independent of committee size, and the communication required to decrypt an entire batch of $B$ transactions is 80 bytes per party, independent of the number of transactions $B$, making it an attractive choice when communication is very expensive. If deployed on Ethereum, which processes close to 500 transaction per block, it takes close to 2.8 s for each committee member to compute a partial decryption and under 3.5 s to decrypt all transactions for a block in single-threaded mode.

1 Introduction
A prominent application of cryptocurrencies and blockchain technologies is in Decentralized Finance (DeFi) which aims to provide financial services through a decentralized network of parties. A significant amount of capital has been invested in this space with a peak value peak of over $250 billion in December 2021.\(^1\) This capital was drawn not just from retail investors but also from major institutional players.

The traditional financial system has well documented evidence [44] to prove that in the absence of regulations, market manipulation strategies will be employed by insiders who seek to financially exploit the average consumer. However, despite the large amounts of capital locked in the DeFi space, there is virtually no protections implemented for consumers against miners/builders who can not only see the details of all transactions submitted by clients to a *memory pool* (mempool) but also control the order in which transactions will be executed.

Although cryptocurrencies promise a fair and permissionless platform where all users are on a level playing field, the reality is far from this. Unsurprisingly, the DeFi space is ripe for exploitation by greedy (although arguably rational) miners who can frontrun/backrun/sandwich client trades to extract additional value from the asymmetric information available to them. This was first documented as “Miner Extractable Value” or MEV in [18]. Early work focused on devising the best strategies to implement MEV [1,26,61,65]. Many studies since have shown it to be a widespread problem in blockchains [14,32,37,51,60]. In particular, an estimated 200,000,000 USD were lost on Ethereum in 2021 alone, mostly benefiting miners [48]. In the case of trading on Decentralized Exchanges, the attacks mentioned above allow a miner to extract financial gains at the expense of the users.

Alternatively, consider the situation where a user discovers a bug in their smart contract that allows any party to drain all of its funds. A natural course of action is to update the smart contract with new code. But the diff between the old and new code could reveal the bug to any party with access to the mempool, allowing an attacker to exploit the bug before it is fixed. This class of attack is actually widespread, and has been\(^1\) https://cointelegraph.com/news/defi-needs-to-start-creating-real-world-value-if-it-wants-to-survive
observed and documented [52, 54]. In fact, with the recent advancements in Large Language Models, researchers have shown that fine-tuning can be carried out to identify bugs in smart contracts [20]. Using the same strategy, it is conceivable that an attacker could also automate this process. Finally, revealing the content of transactions allow malicious miners to selectively censor targeted groups of users or transactions.

Protecting the privacy of transactions in the mempool has been a subject of active discussion [5, 40, 46, 49, 53, 58] and threshold public key encryption [21, 23, 28] has emerged as the de facto solution to this problem. A threshold public key encryption system has a single public key, and distributes the private key among \( n \) decryption servers such that at least a threshold number (say \( k \)) are needed for decryption. At a high level, threshold encryption is used as follows. First, a committee - corresponding to the decryption servers - is set up such that the committee members are in possession of the (distributed) private key, and all users encrypt to the committee’s public key. Miners can only see encrypted transactions in the mempool and create blocks containing ciphertexts. After the block has been confirmed, the committee jointly decrypts all transactions that have been included and announces them to all nodes in the network.

Existing solutions are insufficient to solve the problem of mempool privacy for two main reasons. First, the proposed solutions fail to account for a malleability attack, where an adversary could create a related transaction ciphertext, submit that instead, and learn information about a transaction which has not been included on chain. This is a very realistic attack as most users don’t run their own nodes and instead rely on other untrusted nodes who submit transactions for them.

Second, the communication involved to decrypt ciphertexts is \( O(nB) \) instead of \( O(B) \) without privacy, where \( n \) is the committee size and \( B \) is the block size (number of transactions in a block). In a widely distributed network like blockchains, the additional latency introduced in an already time-critical system is untenable. We emphasize here that the metric of interest is not the bandwidth available on a node but the latency of the underlying broadcast/gossip network. In otherwords, it is not how fast a committee member can upload their partial decryptions to some server but how fast a message broadcast by a committee member can reach all nodes on the network, as all parties need to decrypt the ciphertexts. For concrete numbers, we refer to [22], which shows that each kilobyte over 20kB adds an 80ms delay to the time it takes for a message to reach a majority of nodes.

Deployed systems [58] have attempted to handle this by introducing a new encryption scheme similar to that of [24], where clients encrypt to an epoch, and the decryption key is the same for all ciphertexts in that epoch. However, this makes unacceptable compromises to privacy, as ciphertexts that are not included/withheld lose privacy despite not being confirmed transactions. We refer to this property as pending transaction privacy. We summarize a comparison against prior work in Table 1 and elaborate on other possible attack vectors in Section 3.

**Our Model.** To address the above two problems we introduce a new security notion of batched-threshold-encryption which protects the privacy of messages against adaptive chosen ciphertext attacks [7] and in addition demands that an entire batch of ciphertexts can be decrypted non-interactively, and with sub-linear communication. This translates to ensuring that(i) each member of the committee can locally compute partial decryption given only the batch of ciphertexts, and these can later be combined by any party (even outside the committee) to recover the messages; and (ii) the size of partial decryption is sub-linear in the batch size.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Comm.</th>
<th>Pending Tx privacy</th>
<th>Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thresh. Enc. [13, 25]</td>
<td>( O(nB) )</td>
<td>✓</td>
<td>DKG</td>
</tr>
<tr>
<td>Ferveo [5]</td>
<td>( O(nB) )</td>
<td>✓</td>
<td>DKG</td>
</tr>
<tr>
<td>McFly [24]</td>
<td>( O(n) )</td>
<td>✗</td>
<td>DKG</td>
</tr>
<tr>
<td>Shutter [58]</td>
<td>( O(n) )</td>
<td>✗</td>
<td>DKG</td>
</tr>
<tr>
<td><strong>This Work</strong></td>
<td>( O(n) )</td>
<td>✓</td>
<td>MPC + Epoch Setup</td>
</tr>
</tbody>
</table>

Table 1: A comparison of this work against prior work in terms of communication to decrypt a batch of \( B \) transactions and whether the scheme preserves the privacy of transactions which were submitted but may not have been included in the batch. DKG refers to a distributed key generation protocol and MPC refers to a (more expensive) multi-party computation protocol used during the setup phase and Epoch Setup refers to a per epoch setup that can be carried out well before the actual epoch takes place.

### 1.1 Our Contributions

In this work,

- We demonstrate the ineffectiveness of existing approaches in protecting the privacy of transactions in the mempool, highlighting the need for improved security definitions and constructions. In particular, we show that the Shutter Network [58] suffers from the issues highlighted in the prior section. (Section 3)

- We undertake a formal study of encrypted mempools and propose the notion of batched-threshold encryption, which we argue is the appropriate security notion of threshold encryption for mempool privacy. Additionally, our definitions demands efficiency properties that aim
to model the limited bandwidth available in large-scale distributed networks. (Section 6)

- We then provide the first concretely efficient construction satisfying the batched-threshold encryption definition. Importantly, to decrypt a batch of messages, the size of partial decryption is independent of the batch size; in fact, it is just one field element. The ciphertext size is 370 bytes for a 32 byte message, and only \(3 \times 100\times\) larger than the Cramer-Shoup threshold encryption scheme [12, 13], which does not satisfy necessary efficiency requirements. (Section 7). However, our scheme incurs a more expensive setup \(\approx 100\times\) along with a per epoch setup, which can be carried out well before the actual epoch. We believe that this can be a favorable trade-off when the number of transactions in a block is large, as our scheme reduces the communication during the time-sensitive phase (decryption) by several orders of magnitude. Concretely, for a block of 512 transactions and a committee size of 128, we reduce the communication by over 300\times. Moreover, this number will only grow larger as the number of transactions in a block increases. (Section 7)

- We created a Rust crate for our batched-threshold encryption scheme. It takes \(6\, \text{ms}\) to encrypt a ciphertext. For an Ethereum deployment, which processes \(500\, \text{transactions per block}\), it would take close to \(2.8\, \text{ms}\) to compute a partial decryption and \(<3.5\, \text{s}\) to reconstruct all messages. We emphasize that our scheme can take advantage of multi-threaded architectures and Ethereum nodes are likely to already have machines with multiple cores. (Section 9)

## 2 Our Techniques

We now describe the main ideas underlying our work. Before we describe our construction, we note that a common theme when describing the shortcomings of prior approaches was the lack of clearly defined security properties. Further, the security needs to be defined while ensuring that any solution we propose is agnostic to the system (or blockchain) it is deployed on. To this end, we define a new primitive called batched threshold public key encryption (bTPKE) whose properties (to be defined shortly) will prove sufficient in achieving mempool transaction privacy. We elaborate below.

### Batched Threshold Public Key Encryption

At a high level, this primitive specifies a public key for which some parties that are designated as servers are given (threshold) shares of the corresponding secret key. Users, who are not necessarily servers, are then allowed to encrypt to the public key such that: (i) a batch of ciphertexts are chosen by the servers to be decrypted; (ii) each server broadcasts a short batch decryption key (of size independent of \(B\)) that enables decryption of all ciphertexts included in the batch; and (iii) the contents the ciphertexts not included in the batch remain hidden.

Given the properties described for a bTPKE, it follows in a straightforward manner how one can utilize the primitive to realize the the guarantees for mempool privacy. Specifically, in the mempool privacy setting, the miners correspond to the bTPKE servers, and users encrypt transactions using the bTPKE encryption scheme with the guarantee that encrypted transactions that are not selected by miners in the batch hide the underlying transaction. Further, the efficiency properties of bTPKE ensure that this is indeed a feasible solution to guarantee mempool transaction privacy. Note that, as desired, the properties of the bTPKE are agnostic to how the batch of ciphertexts to be decrypted are decided, and thus work for any set of selected transactions.

In Section 6 we formalize the above described properties via an ideal functionality \(f_{\text{bTPKE}}\), which is motivated by the ideal functionality for threshold encryption schemes that are secure against chosen ciphertext attack [13]. The main difference in our setting is that our ideal functionality incorporates a notion of “epochs” such that users can encrypt messages to a specific epoch, and a batch of ciphertexts to be decrypted must all belong to the same epoch.

**Construction** For simplicity, in this overview, we will highlight the main ideas underlying our bTPKE construction in the setting that there is a single server that is honest. We defer the extension to the threshold setting with multiple servers to Section 7, where we use the threshold to emulate the single honest server. As a further simplification, we first describe a scheme that provides the weak form of security where privacy is guaranteed only against adversaries that only see ciphertexts (indistinguishability of chosen plaintext), but not their decryption. We will then describe the shortcomings of this construction, and then strengthen it to achieve the desired security guarantees.

Our construction is based on polynomial-commitment schemes, where such a scheme allows a user to generate a “commitment” \(c(x)\) to a polynomial \(p(x)\) such that the user at a later point can provide a short proof of opening with respect to \(c(x)\) given the polynomial \(p\) with the guarantee that \(p(x) = y\). We now describe the template of our bTPKE construction: (i) the public-key \(p\) will correspond to the commitment \(c(x)\) of an unspecified degree \(B\) polynomial, i.e. the polynomial will be determined by the transactions and the reader can think of \(c(x)\) as a commitment to an arbitrary polynomial that will be explained correctly later; (ii) each user will sample a pair \((x_i, y_i)\) and “encrypt” their message \(m_i\) to only be decrypted using the proof of opening of \((x_i, y_i)\) with respect to \(c(x)\); and (iii) finally, the server given \(B\) ciphertexts, each with their corresponding \((x_i, y_i)\) pair, uses a trapdoor to explain \(c(x)\) to be a commitment to a degree \(B\) polynomial \(p(x)\) such that for each \(i \in [B]\), \(p(x_i) = y_i\) - since a degree \(B\) polynomial is fully determined by \(B+1\) points, the server can simply broadcast a single pair \((0, p(0))\) which allows all-par-
ties to reconstruct \( p(X) \) via polynomial interpolation using the points \( \{(0, p(0)), (x_1, y_1), \ldots, (x_B, y_B)\} \). Once \( p(X) \) is reconstructed, as discussed from the properties of the polynomial commitment scheme, anyone with access to the polynomial \( p(X) \) can compute the proof of openings to \( (x_i, y_i) \) and thus decrypt the ciphertexts. We now fill in the details of the template using the KZG polynomial commitment scheme [39]. We first briefly describe the commitment scheme below.

In the KZG scheme, the commitment to a degree \( B \) polynomial \( p(X) \) given the common reference string \( \text{crs} = (g, g^2, g^3, \ldots, g^B) \) is simply the deterministic value \( \text{com} = g^{\text{crs}[\tau]} \). Here \( g \) is the generator of a multiplicative group \( \mathbb{G} \), and the committer is able to compute \( g^{\text{crs}[\tau]} \) given the crs, without knowledge of \( \tau \) - which is the trapdoor hidden from the committer. Given a pair \( (x, y) \) such that \( y = p(x) \), the proof of opening is a group element \( \pi = g^{\text{crs}[\tau]} \) that is a commitment to \( q \), which is a polynomial such that \( q(X) = (p(X) - y)/(X - x) \). If the group is appropriately equipped with a pairing operation \( e(\cdot, \cdot) \) (see Section 5 for details on groups and pairing operations), verification of the proof of opening is done by checking the equality \( e(\text{com}/g^\tau, g) = e(\pi, g^{\tau}/g^\tau) \). The binding property of the scheme guarantees that if \( \tau \) is hidden from the committer, for any pair \( (x', y') \) such that \( p(x') \neq y' \), one cannot find a group element \( \pi' \) that satisfies the aforementioned verification check.

Given the discussed properties from the KZG polynomial commitment scheme, specifically the verification check, we use ideas present in the construction of the Boneh-Franklin Identity Based Encryption (IBE) scheme [11] (also recently used in the context of encryption under signature verification [24, 45]). Specifically, the public key is the CRS \( (g, g^2, g^3, \ldots, g^B) \) and a random group element \( \text{com} = g^{\text{crs}[\tau]} \) treated as a “commitment” to a polynomial yet to be determined. The server is in possession of the trapdoors \( \alpha \) and \( \tau \). To encrypt a message \( M \) with respect to the public key \( \text{com} \), a user first samples a pair \( (x, y) \), and then computes the ciphertext as

\[
ct = (x, y, (g^\tau g^\pi)^\rho, H(e(\text{com}/g^\tau, g)^\rho \oplus M))
\]

where \( H \) is a hash function (modeled as a random oracle in our security proofs), and \( \rho \) is a randomly sampled value. Given \( B \) ciphertexts \( \{(x_i, y_i, c_{i,1}, c_{i,2})\}_{i \in [B]} \), the server explains the commitment com by interpolating the unique degree \( B \) polynomial \( p(X) \) using the \( B + 1 \) points \( (x_1, y_1), \ldots, (x_B, y_B) \) and \( (\tau, \alpha) \) (where we view \( \alpha = p(\tau) \) by treating \( \text{com} \) as a KZG commitment). The server then simply broadcasts the batch decryption key \( (0, p(0)) \).

To decrypt, any user in possession of the \( B \) ciphertexts \( \{(x_i, y_i, c_{i,1}, c_{i,2})\}_{i \in [B]} \) and \( (0, p(0)) \), performs the following steps in order: (i) reconstructs the polynomial \( p(X) \) using the \( B + 1 \) points in its possession; for each \( i \in [B] \), (ii) computes the proof of opening \( \pi_i \) with respect to \( \text{com} \) and the pair \((x_i, y_i)\); and (iii) compute \( m_i = H(e(\pi, c_{i,2})) \oplus c_{i,1} \). From the correctness of the KZG verification, it is easy to see that the decryption procedure does indeed reconstruct the \( B \) ciphertexts. Further, the server was only required to send 2 elements to enable the decryption of a batch of \( B \) ciphertexts.

Any ciphertext with \((\bar{x}, \bar{y})\) not selected as a part of the \( B \) ciphertexts will (with high probability) be such that \( p(\bar{x}) \neq \bar{y} \) if \((\bar{x}, \bar{y})\) is sampled randomly. Thus, by the binding property of the KZG commitment scheme, \( \text{com} \), given \( p(X) \), cannot be opened to the pair \((\bar{x}, \bar{y})\) ensuring that the ciphertext encrypted using \((\bar{x}, \bar{y})\) remains hidden.

First, an observant reader may note that in the use cases envisioned, the security argued above is insufficient since the adversary can “maul” ciphertexts to break any privacy guarantees upon obtaining the batch decryption key. For instance, an adversary can simply flip bits in the \( c_{i,2} \) component of ct, and perhaps more devastatingly, since the batch decryption key solely depends on the pair \((x, y)\), the adversary can execute a “copy attack” by copying the pair \((x, y)\) from an honest ciphertext to generate a new ciphertext such that given the batch decryption key, the adversary can decrypt the honest ciphertext even if it was not included in the batch of ciphertexts to be decrypted. As we will describe shortly in Section 3, prior works often fail to address these pitfalls.

Second, note that while the binding of the KZG scheme guarantees that ciphertexts with pairs \((\bar{x}, \bar{y})\) that won’t be included in the \( B \) ciphertexts cannot be decrypted, our use case requires a stronger property - namely that even given a ciphertext with a pair \((x, y)\) that will be included in the \( B \) ciphertexts, an adversarial user should not be able to decrypt the ciphertext (i.e. compute opening proof) until the server broadcasts the batch decryption key. This latter requirement requires the use of a variant of KZG that is randomized, and a subsequent strengthening of the ‘KZG assumption’ to argue security.

In the remainder of this section, we describe how we handle the first type of malleability issues, and defer details regarding the “randomized KZG” to the technical sections.

**Adding Non-Malleability** Adding non-malleability to ciphertexts corresponds closely to securing encryption scheme against chosen ciphertext attacks, and in this work we shall follow the same approach. To prevent an adversarial mauling of the \( c_{i,2} \) component, we use the folklore approach used in the context of adding non-malleability to El-Gamal encryption (see [57]) by additionally attaching a non-interactive proof of knowledge to our ciphertext \( ct \) proving knowledge of the encryption randomness \( \rho \) in the exponent. By assuming that the proof of knowledge extractor is an online extractor (see remark below), this approach prevents the malleability of the \((c_{i,1}, c_{i,2})\) component of \( ct \). This unfortunately doesn’t protect the aforementioned “copy attack” since the adversary can replace \((c_{i,1}, c_{i,2})\) with a fresh ciphertext while keeping \((x, y)\) the same. We handle this in a similar fashion by changing the way \( y \) is generated, and having the prover explain the

\[\text{FOOTNOTES:} \]
\footnote{For simplicity, we assume that the \( B \) \( x_i \) are all distinct.}
generation of y. Specifically, the user still samples x at random, but for y the user first samples a random exponent s, computes \( y = H(S) \) where \( S = g^x \) and then provide a non-interactive proof of knowledge of s such that \( S = g^s \). Thus our final ciphertext is of the form

\[
ct = (x, S, (g^y / g^s)^\rho, H(e(com / g^s, g)^\rho) \oplus M, \Pi_1, \Pi_2),
\]

where \( \Pi_1 \) and \( \Pi_2 \) correspond to the non-interactive proofs of knowledge discussed above. Intuitively, for any ciphertext such that \( \Pi_1 \) and \( \Pi_2 \) verify, the adversary is able to explain the y that is used, and the message that is encrypted. It turns out that this intuition can be formalized, and is sufficient to provide the security guarantees of our primitive. Note that the above description was intended to provide only the key ideas in our construction, and we refer the reader to Section 7 for the full details.

**Remark 1.** We note that for our implemented solution to be efficient, we use the non-interactive version of Schnorr protocol [55] to prove knowledge of discrete log. Unfortunately, it was observed in [57] that proving security of the folklore El-Gamal construction, even in the random oracle model, is problematic using the Schnorr protocol. At a very high level, this stems from the fact that the Schnorr protocol necessitates rewinding for extraction, which is incompatible with the decryption queries of an adversary mounting a chosen ciphertext attack (see [57] for details). This is in fact more broadly an issue with any proof of knowledge protocol that requires rewinding, and thus in our construction we assume the existence of a ‘straight-line extractor’ to circumvent this issue. We justify our instantiation of the Schnorr protocol by noting that in the Random Oracle Model in conjunction with the Algebraic Group Mode (AGM), Schnorr does indeed have straight line extraction and can be used to prove that the folklore ‘Schnorr signed El Gamal’ is chosen ciphertext secure [29]. The ideas in that work directly extend to our setting as well.

## 3 Pitfalls in prior work

In this section we describe a flaw in the Shutter Network [58] that allows an adversary to compromise transactions privacy by exploiting the malleability of the encryption scheme. We believe that this not only necessitates the need for a formal security model, but also informing the various choices in our construction.

The Shutter Network [58] uses its own construction of threshold encryption for mempool privacy. Indeed it is more efficient than other proposals in the space (such as Ferveo [5] or MEVade [50]), and has been integrated to work on top of Ethereum without requiring changes to the base protocol. However, we found a few flaws in its design and implementation that render it insecure in its current form.

We begin by describing shutter’s encryption scheme. We note that at the time of writing this paper, we were not able to find a whitepaper containing a specification of their scheme and had to infer details of their protocol from their GitHub repository.

Shutter works in epochs, each corresponding to a set of encrypted transactions. At the beginning of the protocol, an epoch-secret key \( sk_e \in \mathbb{F} \) is secret shared amongst the committee and they publish an epoch-public key \( pk_e = h^{sk_e} \). For each epoch, a unique epoch id \( eid \in \mathbb{G}_1 \) is broadcast to users. Then, users encrypt transactions as follows:

- \( C_1 = h^r \) for \( r \leftarrow \mathbb{F} \)
- \( C_2 \leftarrow H(e(id, pk_e)^r) \oplus M \). Here, \( M \) could be replaced with an encapsulation key which is in turn used to encrypt a long message.

The user then sends \((C_1, C_2)\) as the encryption of their transaction. In order to decrypt, the committee members each release shares of a private epoch key \( sk = e^{id^{sk_e}} \). Then, anyone can decrypt as \( M = H(e(sk, C_1)) \oplus C_2 \). We now describe some flaws with this design.

- First observe that ciphertexts can easily be malleed by flipping some bits, and they remain valid ciphertexts. Any party who intercepts a user’s transaction can modify the ciphertexts, have it be included on chain and learn the transaction without the transaction being included on chain. Thus, this already violates the IND-CCA2 requirements.

- Next, Shutter’s efficiency gains compared to Ferveo and MEVade come from sharing a decryption key for all transactions submitted during the same epoch: every transaction encrypted to that epoch can be decrypted once the key is released, regardless of whether the transaction was actually included on chain. This means transactions that are stuck in the mempool and don’t make it on chain before the end of the epoch will be revealed even though they won’t be executed. For the optimistic roll-up version of Shutter [59], a malicious collator (the entity in charge of selecting a batch of transactions) could withhold transactions from the batch, and still be able to decrypt them once the key is published.

## 3.1 Disclosure

Upon disclosing our findings to the Shutter team and the Cryptographers who designed Shutter’s protocol – Stefan Dziembowski (University of Warsaw and IDEAS NCBR) and Sebastian Faust (TU Darmstadt) – we received the following responses:

3https://github.com/shutter-network/shutter/blob/42b2338ed4970bc0df95fa6bd6b752b5?d8ce2f/shlib/shcrypto/encryption.go#L150
From the Shutter team:

We thank the authors for finding and disclosing the malleability vulnerability in our implementation, which we have fixed thereupon. We are aware of the lack of pending transaction privacy in our protocol, but consider it prohibitively difficult to exploit in the practical scenarios we are focusing on. This is because the fact that transactions are encrypted largely prevents targeted attacks, because modern blockchains and rollups usually include transactions very quickly, and because leaked transactions are prohibited from being included in future blocks. In one variant of our protocol intended for a different setting, we achieve pending transaction privacy, but at the cost of a communication overhead linear in the number of transactions. Nevertheless, we agree with the authors that achieving pending transaction privacy in an efficient manner is very valuable.

From the Cryptographers:

Thanks for looking at Shutter’s implementation and informing us about its issues! We address them below. About the “stuck in the mempool” problem - you are right - Shutter assumes that the transactions make it to the blockchain before the decryption key is published (how big of a problem it is depends on the concrete application).

About malleability: the formal cryptographic description of the Shutter protocol we prepared in 2021 did not suffer from this problem. In fact, we were well aware of the non-malleability requirement, so we based Shutter on non-malleable ID-based threshold encryption. The bug you found was introduced during Shutter programming. We take it as a lesson on the importance of interaction between the cryptographers and software developers during the entire system development process. The development team has fixed the issue in the meantime. We will also publish the Shutter description (as it was designed in 2021) in a paper that we will put on the Cryptology Eprint archive soon. Thanks again!

4 Related Work

4 Trusted-Execution. One could explore Trusted-Execution based solutions where the enclave contains the secret key of a public-key encryption scheme and clients can encrypt their transactions to this public key. The TEE then decrypts and publishes a block of transactions, which gets included on-chain. Given that over $1 billion USD of MEV has been extracted, and that TEEs have been repeatedly shown to be exploitable, it is not reasonable to expect that this approach is secure in the long-term.

Cryptographic. Here, clients use cryptography to hide the content of their transaction until it is included in a block i.e. execution is confirmed and ordering has been fixed. The three main approaches considered are timelock encryption [10], threshold encryption [5, 40, 46, 49, 58] and witness encryption [30].

In timelock encryption, transactions lose privacy if they are not included within a given window of time and also need wasteful computation to be carried out leading to additional delays.

Threshold encryption with adaptive chosen ciphertext security [12, 13], provides the required privacy guarantees but the communication is proportional to the number of transactions times the committee size for decryption, which is a bottleneck for scaling these systems.

Finally, witness encryption is typically considered to be impractical but a recent work by Döttling et al. [24] showed that you can build signature based witness encryption even for aggregate and threshold BLS signatures where a signature on some public message is the decryption key. Now users can encrypt transactions to a predictable block number and committee members can sign this block number. Thus it is guaranteed that a key to decrypt the ciphertext will appear (in the form of a signature on the block number) at some point in the future. Although this allows for encryption to the future, it is not clear how to ensure that transactions which are not included on-chain will not be decrypted.

In contrast, our work achieves the best of threshold encryption and signature based witness encryption as the communication to decrypt an entire block of transactions is independent of the block size and privacy of transactions that have not been included is still preserved.

Order-fairness. An alternate approach to reducing MEV involves attaining consensus amongst miners on the order of incoming transactions to the mempool. This method aims to limit transaction reordering, thus limiting a miner’s potential for extracting MEV. Themis [41] and Aequitas [42] illustrate the principle of decentralized order fairness. These methods, however, can only guarantee a weak form of order-fairness, dubbed batch order-fairness, which does not entirely eliminate MEV. A protocol suite for maintaining order fairness based on linearizability has been proposed in Wendy [43], although it only ensures coarse order-fairness and does not wholly rule out all possible reorderings.

---


7https://sgx.fail

8See https://explore.flashbots.net for flashbots statistics.

---
MEV-resistant DEX. Several strategies for redefining DEXs transaction processing model have been proposed. McMenamin et al. [47] and Heimbach et al. [36] offer game-theoretic defenses against MEV extraction through frequent batch auctions and setting slippage tolerance, respectively. Off-chain communication [17] and multi-party computation [4] have been suggested as means to eliminate arbitrage. A unified AMM to mitigate attacks is proposed by A2MM [64]. Despite improved guarantees, these models are more complex and potentially more vulnerable to smart contract bugs. Furthermore, they do not restrict MEV across different DEXs.

5 Preliminaries

We use \([n]\) to denote the set \(\{1, 2, \ldots, n\}\). The security parameter is denoted by \(\lambda \in \mathbb{N}\). A function \(f : \mathbb{N} \rightarrow \mathbb{N}\) is said to be polynomial if there exists a constant \(c\) such that \(f(n) \leq n^c\) for all \(n \in \mathbb{N}\), and we write \(\text{poly}(\cdot)\) to denote such a function. A function \(f : \mathbb{N} \rightarrow [0, 1]\) is said to be negligible if for every \(c \in \mathbb{N}\), there exists \(N \in \mathbb{N}\) such that for all \(n > N\), \(f(n) < n^{-c}\), and we write \(\text{negl}(\cdot)\) to denote such a function. A probability is noticeable if it is not negligible, and overwhelming if it is equal to \(1 - \text{negl}(\lambda)\) for some negligible function \(\text{negl}(\lambda)\).

A set \(S\), we write \(s \leftarrow S\) to indicate that \(s\) is sampled uniformly at random from \(S\). For a random variable \(\mathcal{D}\), we write \(d \leftarrow \mathcal{D}\) to indicate that \(d\) is sampled according to \(\mathcal{D}\). An algorithm \(\mathcal{A}\) is \(\text{PPT}\) (probabilistic polynomial-time) if its running time is bounded by some polynomial in the size of its input. For two ensembles of random variables \(\{\mathcal{D}_\lambda\}_{\lambda \in \mathbb{N}}, \{\mathcal{D}_\lambda\}_{\lambda \in \mathbb{N}}\), we write \(\mathcal{D}_0 \approx_c \mathcal{D}_1\) to indicate that for all \(\text{PPT} \ \mathcal{A}\), it holds that

\[
\Pr_{d \leftarrow \mathcal{D}_0} \left[ \mathcal{A}(d) = 1 \right] - \Pr_{d \leftarrow \mathcal{D}_1} \left[ \mathcal{A}(d) = 1 \right] \leq \frac{1}{2} + \text{negl}(\lambda).
\]

We use \(e\) to denote an efficiently computable non-degenerate bilinear pairing from the groups \((\mathbb{G}_1, \mathbb{G}_2)\) to a target group \(\mathbb{G}_T\) and use \(\mathbb{F}\) to denote the associated field. We will represent the Shamir secret sharing of an element \(x\) as \([x]\), with \(i\)-th share represented as \([x]_i\). We note that while we overload the previously discussed notation \([n]\), the notation for secret shares will be clear from the context.

The random oracle model. In the random oracle model (ROM), parties are given oracle access to some function \(H\) that is sampled uniformly at random from the space of all functions \(H : X \rightarrow \mathcal{Y}\), where \(X\) and \(\mathcal{Y}\) are finite non-empty sets. That is, parties can query their oracle on an input \(x \in X\) and receive in return \(H(x) \in \mathcal{Y}\). When proving security of a protocol in the random oracle model, the simulator \(\text{Sim}\) is able to “control” the oracle, observing queries made by the adversary and simulating responses.

Lagrange Polynomials. We will use \(L_i(x) = \Pi_{j=0, j \neq i}^{d-1} \frac{(x-x_j)}{(x_i-x_j)}\) to denote the \(i\)-th lagrange polynomial for polynomials of degree \(\leq d\) over a sufficiently large field, corresponding to a domain \(\Omega = \{x_0, x_1, \ldots, x_d\}\). These polynomials have the property that \(f(\tau) = \sum_{i=0}^{d-1} L_i(\tau) f(x_i)\) for any point \(\tau \not\in \Omega\) in the field.

Non-interactive Zero-Knowledge proofs. Let \(\mathcal{L}\) be an NP-language and \(\mathcal{R}\) the corresponding NP-relation. A Simulation Extractable-NIZK for \(\mathcal{R}\) consists of the following algorithms:

- **Setup**: Takes as input a security parameter, and outputs a common reference string \(\text{crs}\) and trapdoor \(td\).
- **Prove**: Takes as input \(\text{crs}\) and any pair \((x, \omega)\in \mathcal{R}\) and outputs a proof \(\pi\) for the statement \(x \in \mathcal{L}\).
- **Verify**: Takes as input \(\text{crs}, \pi\) and \(\pi\) and outputs a bit \(b\) indicating whether verification has passed or failed.
- **SimProve**: Takes as input \(\text{crs}\), trapdoor \(td\) and statement \(x\) and outputs a *simulated* proof \(\pi\).

We will drop the \(\text{crs}\) term wherever implicit. An SE-NIZK satisfies satisfies the following properties:

- **Perfect Completeness.** An SE-NIZK satisfies perfect completeness if for any \((x, \omega)\in \mathcal{R}\), we have
  \[
  \Pr_{(\text{crs}, td) \leftarrow \text{Setup}(1^\lambda)} \left[ \pi \leftarrow \text{Prove}(\text{crs}, x, \omega) \quad : \quad \text{Verify}(x, \pi) = 1 \right] = 1
  \]

- **Proof of knowledge (and soundness).** For every \(\text{PPT}\) adversary \(\mathcal{A}\), there is a \(\text{PPT}\) extractor \(\text{Ext}\) such that
  \[
  \Pr_{(\text{crs}, td) \leftarrow \text{Setup}(1^\lambda)} \left[ \pi \leftarrow \mathcal{A}(\text{crs}) \quad : \quad \text{Verify}(x, \pi) = 1 \quad (x, \omega) \not\in \mathcal{R} \right] \leq \text{negl}(\lambda)
  \]

- **Computational Zero-knowledge.** There exists a simulator \(\text{Sim}\) such that for all stateful distinguishers \(\mathcal{A}\) the following probabilities are equal:
  \[
  \Pr_{(\text{crs}, td) \leftarrow \text{Setup}(1^\lambda)} \left[ (x, \omega) \leftarrow \mathcal{A}(\text{crs}) \quad : \quad \text{Verify}(x, \pi) = 1 \quad (x, \omega) \in \mathcal{R} \right] \leq \text{negl}(\lambda)
  \]

- **Weak Simulation-Extractability.** For every \(\text{PPT}\) adversary \(\mathcal{A}\), there exists a \(\text{PPT}\) extractor \(\text{Ext}\) such that
  \[
  \Pr_{(\text{crs}, td) \leftarrow \text{Setup}(1^\lambda)} \left[ (\pi, w) \leftarrow \mathcal{A}(\text{SimProve}) \quad : \quad \text{Verify}(x, \pi) = 1 \quad \land (x, w) \not\in \mathcal{R} \right] \leq \text{negl}(\lambda)
  \]
where $A$ has oracle access to SimProve$(crs, td, \cdot)$, and $Q$ is a list of queries made by the adversary.

We will also demand that the proof is straight-line extractable. This means that the extractor can, on input a valid proof and (potentially) the list of random-oracle queries made by the adversary (prover), extract a witness with overwhelming probability without rewinding the prover.

6 Model and Definitions

To define the security and efficiency requirements of our batched-threshold encryption scheme we extend the IND-CCA2 threshold encryption definition from [13]. First we specify the syntax for batched-threshold encryption.

- **Setup**: $\{pk, (sk_1, \ldots, sk_n)\}$: On input threshold $t$ for $n$ parties, and batch size $B$, outputs the public key $pk$ along with secret keys for each party.

- **ESetup**: $\{epk, com((td_1, \ldots, td_n))\}$: An optional, per epoch setup, which has the same inputs as Setup and outputs strings $epk, com \in \{0, 1\}^*$ which is published and trapdoor information $(td_1, \ldots, td_n)$ for each member of the committee.

- **Enc**: $(pk, epk, com, m) \rightarrow ct$: Takes as input a message $m$, epoch setup $epk$, $com$, and a public key $pk$ and encrypts $m$ to output a ciphertext $ct$.

- **BatchDec**: $(ct_1, \ldots, ct_B, sk_i, td_i) \rightarrow d_i$: Takes as input $B$ ciphertexts $(ct_1, \ldots, ct_B)$, and a secret key $sk_i$ and outputs a partial decryption $d_i$ or outputs $\bot$.

- **Combine**: $(pk, epk, com((ct_1, \ldots, ct_B)), \{d_i\}_{i \in S}) \rightarrow (m_1, \ldots, m_B)$: Takes as input the public key $pk$, epoch setup $epk$, $com$, and $t + 1$ partial decryptions where $S \subseteq [n]$ and $|S| = t + 1$, and output messages $(m_1, \ldots, m_B)$ or $\bot$.

**Efficiency.** We impose various efficiency constraints on the above algorithms. Setup and ESetup must run in time $\text{poly}(n, t, B, \lambda)$. Enc must run in $O(1)$ time. BatchDec must be non-interactive i.e., each party computes it’s partial decryption independently. It must also run in time $\text{poly}(B, \lambda)$ and the size of each partial decryption must be sub-linear in the batch size ($o(B)$) and independent of number of parties. Finally, Combine must run in time $\text{poly}(B, n, \lambda)$.

**The ideal functionality.** We now describe an ideal functionality (Fig. 1) similar to [13] which models an idealized encryption service. Note that to closely model applications we have in mind, the interaction with the ideal functionality proceeds in epochs. Further, note that the batch of ciphertexts to be decrypted can be determined in an arbitrary manner.\footnote{In secure computation, this is often explicitly modeled by the environment. Since we do not discuss composability in this work, we refrain from formally defining the environment for improved readability, and implicitly use the fact that the environment determines the batch of ciphertexts to be decrypted.}

Note that the ideal functionality already incorporates the progression of the protocol through epochs by specifying epoch ids ($eid$). Any protocol that securely emulates this ideal functionality, does not leak information about the ciphertexts not decrypted in the epoch. The definition further ensures that there is no leakage of any long term secret information in the protocol across epochs. Indeed, the protocol we describe will be shown to securely emulate the ideal functionality.

![Figure 1: Ideal Functionality for Threshold Public-Key Encryption scheme.](image)

7 Construction

We build a concretely efficient scheme for batch-threshold encryption where the communication to decrypt an entire block of $B$ ciphertexts is independent of the number of transactions and only requires each member of the committee to send one field element. Both our ciphertexts and public key have constant size and do not grow with the batch size nor the number of parties in the committee. We present our construction assuming a trusted dealer that runs the Setup and
ESetup algorithms, and distributes necessary information to the committee. In Section 8 we briefly describe how one can use (specialized) multiparty computation (MPC) to emulate the trusted dealer using the committee.

Before we describe our construction we first recall the KZG polynomial commitment scheme [39]. Specifically, we use the randomized version of the KZG scheme, with the randomizing polynomial degree set to be 1. Intuitively, we do not require a higher degree randomizing polynomial because we use the polynomial commitment scheme in an ‘all or nothing’ manner - we only require the underlying polynomial in the commitment be hiding until a single opening proof is provided, which in our case will correspond to revealing the entire polynomial.

**KZG Commitment Scheme [39].** The setup consists of a crs = (g, g^2, g^3, ..., g^B, h, h^p, ˜h) for randomly sampled generators g 6= g E G, h 6= ˜h E G and dlogg g = dlogh h. The commitment to a polynomial of degree ≤ B is computed as com = g^r g^com[t], where r ∈ [2]. To open the polynomial at a point x in the domain Ω, the prover publishes π = (π = g^rτ, π = g^τ, g^τ, g^τ, g^τ) wherein (q(x) = (p(x) − y)) / (x − x). The proof is verified by checking that

\[ e(com / g^x, h) = e(π^1, h/h^p) · e(π^2, ˜h). \]

(1)

For convenience of notation we will separate the crs into two separate crs1 = (g, g^5, g^5, ..., g^B, h, h^p) and crs2 = (g, ˜h). We note that from the above description, to compute the opening to the polynomial it suffices to be in possession of the polynomial p, and g^τ, i.e. it is not required to know r to compute π. We will utilize this observation in our construction.

**Setup.** Now to build the desired batched-threshold encryption scheme we use the KZG commitment scheme as follows. The committee runs a setup procedure for a batch size B, and publishes crs1 = (g, g^τ, g^τ, ..., g^B, h, h^p). The committee also holds Shamir secret shares of the trapdoor – Lagrange coefficients (L_0(γ), L_1(γ), ..., L_B(γ)) on the domain Ω_L = {x_0, x_1, ..., x_B}, for some γ 6∈ Ω_L. Note that the batch size B is an upper limit and dummy ciphertexts can always be inserted to appropriately pad any batch of size < B.

For our concrete instantiation, whenever an FFT is to be carried out, we use domains that form a smooth multiplicative subgroup such as the roots of unity (Ω = {1, ω, ω^2, ..., ω^{B−1}}) in a prime field. However, for the Lagrange polynomials we use Ω_L = {1, ω, ω^2, ..., ω^{B−1}, τ}. Looking ahead, we aim to interpolate a polynomial over the points Ω ∪ {γ} in order to produce KZG openings over all points in Ω [27], but this can still be done in O(B log B) time as the domain is almost smooth. The idea is to first interpolate a quotient polynomial q(x) over the smooth subset Ω. Then we observe that the polynomial we are interested can be computed using O(B) multiplications as f(x) = q(x)/(x − γ) + f(γ).

**EpochSetup.** For each epoch, the committee samples a fresh crs2 = (g, ˜h) such that dlogg ˜g = dlogh ˜h and publishes a random group element com = g^β ˜g^γ for which they hold shares of α L_B(γ) and r.

**Remark 2.** We note that to compute α L_B(γ), EpochSetup requires the “secret state” L_B(γ) (dependent on τ sampled in Setup). When emulating the trusted dealer via an MPC, at the end of Setup, the committee members will have also possess shares of L_B(γ) which will allow for the subsequent computation of the shares of α L_B(γ) for each instance of EpochSetup via an MPC. This ensures distributed transfer of the secret state L_B(γ) between the two setups without affecting efficiency. In our trusted dealer description, we assume this state is maintained between the two setups.

**Encrypt.** Now to encrypt a message we rely on the fact that the opening proof π is hard to compute. Taking a closer look at the KZG verification equation, we observe that if the encryptor exponentiates the left hand side of Eq. (1) by p for any x ∈ Ω, x^p ∈ Ω, then this is a uniformly random element that can be used to mask the message. However, the committee must still be able to decrypt the ciphertext and we do this by providing hints = (h^p / h^τ)^p and h^p as part of the ciphertext.

**Decrypt.** To decrypt, it is sufficient to produce a proof π of opening the commitment to y^r at x. Here we note that the shares of the powers of τ, and α actually form a trapdoor that allows the committee to equivocate the commitment, and hence open it to any point (x^r, y^r) of their choice. As described in Fig. 2, the committee will choose a batch of transactions and decrypts them by equivocating the commitment according to the ciphertexts.

In more detail, a threshold number of parties in the committee agree on a batch of ciphertexts {c_1, ..., c_B} to be decrypted for some particular epoch. The committee then finds the unique degree-B polynomial p(x) satisfying the following constraints, where ṡ corresponds to the y value sampled during Encrypt:

- \[ \{ p(α^i) = τ_0 \}_{i=0}^{B-1} \]
- \[ p(τ) = α \]

and output p(γ), where com = g^α ˜g^γ is that epoch’s setup and γ 6∈ Ω_L. Given p(γ), the polynomial p(x) can be recovered, allowing anyone to generate the first term of opening proof π^1. We note that this can be computed non-interactively as p(γ) = \[ \sum_{i=0}^{B-1} p(x_i) L_i(γ) \] and the parties have shares of L_B(γ)p(τ) and {L_i(γ)}_{i=0}^{B-1} along with \[ \{ p(x_i) \}_{i=0}^{B-1} \] in the clear. Since Shamir secret sharing is linearly homomorphic, the committee can compute shares of p(γ) locally as

\[ [p(γ)] = \sum_{i=0}^{B-1} p(x_i) [L_i(γ)] + [L_B(γ)p(τ)]. \]
In addition, each member of the committee also sends shares of \( \hat{g} \) computed as \( \hat{g}^{\tau} \), which allows \( \tau \) to be recovered. Thus, reconstruction only requires one field element and one group element to be sent by each party. For simplicity of exposition, we use a trusted dealer in Fig. 2, to handle the Setup and EpochSetup phases. In Section 8 we discuss how the dealer can be emulated using secure multi-party computation.

CCA2 security. It is easy to see that simply masking the message with randomness leaves it susceptible to man-in-the-middle attacks. We handle this by making \( y^* \) the output of a one way function and having the encryptor attach a simulation-extractable NIZK proving knowledge of the pre-image of the one way function and the randomness \( \rho \) used, and the ciphertext is a tag in the NIZK. The one way function we use is based on the discrete log problem and the NIZK is a very efficient sigma protocol (proof of knowledge of discrete logarithm).

8 Emulating the Dealer

In this section we discuss how the committee can use secure multi-party computation to emulate the trusted dealer.

8.1 Setup

The Setup requires the committee to secret share the Lagrange coefficients \(- L_0(\gamma), L_1(\gamma), \ldots, L_{B-1}(\gamma)\) defined on the domain \( \Omega_L = \{1, \omega, \ldots, \omega^{B-1}, \tau\} \), for some \( \gamma \notin \Omega_L \). In addition they also publish \( crs_1 = (g, g^\gamma, g^{\gamma^2}, \ldots, g^{\gamma^B}, h, h^\tau) \). The MPC first samples a secret sharing of a random value \( \tau \) and computes it’s powers \( \tau^2, \tau^3, \ldots, \tau^{B-1} \) using standard techniques [19]. In fact, all terms can be computed in constant rounds using [3]. Given shares of the powers of \( \tau \), the parties can locally exponentiate their share and announce \( \{g^{\tau^i}\}_{i \in [B-1]} \). The crs can now be computed by carrying out Lagrange interpolation in the exponent.

We now describe an MPC protocol to distribute shares of the \( i \)-th Lagrange coefficient \( L_i(\gamma) \). Recall that for a domain \( \Omega = \{x_0, x_1, \ldots, x_B\}, L_i(\gamma) = \prod_{j \neq i} \frac{\gamma - x_j}{x_i - x_j} \). We separate this into two cases:

- \((x_i = \omega^k)\) Here, a secret value \( \tau \) only shows up once in the numerator and once in the denominator. Thus, this only requires computing one inversion and one multiplication of a secret shared value. Again, these can be done using standard techniques.

- \((x_i = \tau)\) Although this is not needed in the setup phase, it will be needed later in the epoch setup phase. This case is slightly trickier as \( \tau \) appears multiple times in the denominator. However, upon expanding the denominator in symbolic manner, the denominator is just a linear combination of the powers of \( \tau \). Since the parties already have a secret sharing of the powers of \( \tau \), they can locally compute a secret sharing of the \( i \)-th Lagrange coefficient \( L_i(\gamma) \) without revealing the secret share of \( \tau \).

Batched-Threshold Encryption

Parameters: A pairing friendly group \((G_1, G_2, G_T)\).

- Setup(\(\mathcal{P}, n, t, B\)): A trusted dealer runs the setup for NIZK and the commitment scheme and sets \( pk = (crs_{\text{NIZK}}, \text{ck}, crs_1 = (g, g^\gamma, g^{\gamma^2}, \ldots, g^{\gamma^B}, h, h^\tau)) \), and distributes shares of \( \{L_0(\gamma), L_1(\gamma), \ldots, L_{B-1}(\gamma)\} \) to the committee, setting for each \( i \in [n] \), \( sk_i = [L_i(\gamma)]_{\tau} \). Output (pk, \( \{sk_1, \ldots, sk_n\} \)).

- ESetup(\(\mathcal{P}, n, t, B\)): The trusted dealer samples \( \alpha, \delta, r \leftarrow \mathcal{F} \) and sets \( crs_2 = (\hat{g} = g^\delta, \hat{h} = h^\delta) \) and \( \text{com} = g^\delta \hat{g} \). They also secret share \( \text{ALG}(\gamma) \) (see Remark 2) and \( g^\gamma \) with the committee members. Output (epk, \( td_1, \ldots, td_n \)) where \( \text{epk} = crs_2 \) and for each \( i \in [n] \), \( td_i = (\{\text{ALG}(\gamma)_i, \gamma_i\}) \). Additionally commitments \( \{c_i = \text{Com}(\gamma_i, \eta_i)\}_{i \in [n]} \) are published and the opening \( \eta_i \) to \( c_i \) is given to the \( i \)-th committee member.

- Enc(\(pk, \text{epk}, \text{com}, m\)): To encrypt a message \( M \) for a particular epoch with \( \text{com} \) as the epoch setup, sample \( s \leftarrow \mathcal{F} \) and compute \( S = g^s \) and \( t_g = H(S) \) where \( t_g \in \mathcal{F} \). Finally, sample a random value \( \hat{x} \in \Omega \) and output the following as ct:

\[
\begin{align*}
\text{ct}(1) &= H(e(\text{com}/g^\delta, h)\rho) \oplus M \\
\text{ct}(2) &= (\hat{h}^\rho)^{\hat{x}} \\
\text{ct}(3) &= \hat{h}^\rho \\
\text{φ} &= \text{NIZK}(s, \rho | g^s = S \land \text{ct}(2) = h^{\rho(\tau - 4)} \land \text{ct}(3) \land \text{ct}(1) \land \hat{x}) \\
\text{BatchDec}((\text{ct}_0, \ldots, \text{ct}_{B-1}), (sk_0, \ldots, sk_{B-1}), (td_0, \ldots, td_{B-1})) : A threshold number of parties in the committee agree on a batch of ciphertexts \( \{\text{ct}_0, \ldots, \text{ct}_{B-1}\} \) such that every ciphertext has a valid proof \( \phi_i, e(g, \text{ct}_i^{(2)}) = e(g^{\gamma^s}, \text{ct}_i^{(3)}) \), and \( \{\text{ct}_0, \ldots, \text{ct}_{B-1}\} = \Omega \setminus \{\gamma\} \), where \( \text{ct}_i = (\text{ct}_i^{(1)}, \text{ct}_i^{(2)}, \text{ct}_i^{(3)}, \phi_i) \). The i-th party then outputs

\[
[p(\gamma)]_i = \sum_{j=0}^{B-1} p(x_j)[L_j(\gamma)]_i + [L_B(\gamma)p(\tau)]_i,
\]

and \( g^{[i]} \) along with a proof \( \phi_i = \text{NIZK}(\eta_i, [r_i] | c_i = \text{Com}([\gamma_i] \oplus \phi_i) \). Shares with \( \phi_i \) not verifying are discarded.

- Combine(\(pk, (\text{ct}_0, \ldots, \text{ct}_{B}), (d_0, \ldots, d_{B})\)): Recover the polynomial \( p(x) \) and \( g^\gamma \) (using error-correction for \( p(x) \) if necessary) and produce KZG proofs of opening \( \{\pi_i = (g^{\text{ct}_i^{(1)}}, g^{\text{ct}_i^{(2)}})^{B-1} \} \) for all points in \( \Omega \), where \( q_i(x) = (p(x) - \text{tg}_i)/(x - x_i) \) and \( t_g = H(S_i) \). Output

\[
\{M_i = \text{ct}_i^{(1)} \oplus H(e(\pi_i^{(1)}, \text{ct}_i^{(2)}) \cdot e(\pi_i^{(2)}, \text{ct}_i^{(3)}))^{B-1} \}.
\]

Figure 2: Protocol for batched-threshold encryption
compute shares of the denominator, effectively reducing to the case above.

8.2 Epoch Setup

The epoch setup is much more simple and can be carried out for multiple epochs simultaneously, which also reduces the communication [6]. The parties only need to establish sharings of random values $\alpha, \delta, r$, after which shares can be locally exponentiated with appropriate group elements and announced to recover $\text{com}, \tilde{g}, \tilde{h}$. Finally, they carry out one multiplication to obtain shares of $\alpha L_B(\gamma)$. To ensure that the correct shares of $g'$ are revealed during partial decryption, we publish commitments to shares of $r$, using a publicly verifiable secret sharing protocol [31, 35, 38, 56]. During batch decryption, we provide a proof that $[g']$ was computed correctly (if cheating is detected).

Although the MPC protocols described above are a straightforward application of prior work, we expect them to be reasonably efficient. We also note that an important direction for future work is to optimize these protocols by exploiting the structure of the function being evaluated.

9 Implementation and Evaluation

To evaluate the concrete performance of our batched-threshold encryption scheme, we implemented a performant version of our scheme in Rust. We use the arkworks [2] library for implementations of pairing-friendly curves and associated algebra and the merlin library to handle the Fiat-Shamir transform. Our code can be found at https://github.com/guruvamsi-policharla/batch-threshold-encryption.

Setup. All of our experiments for performance benchmarks Table 2 were run on a 2019 MacBook Pro with a 2.4 GHz Intel Core i9 processor and 16 GB of DDR4 RAM in single threaded mode. Next, for the committee churn experiments, we adapted the implementation from [33] to our setting and simulated a WAN on the Google Cloud Platform using the NetEm package with a delay time of 200 ms, jitter time of 20 ms and a rate of 10 mbit per second. For the committee nodes we used an e2-small instance with 2 GB RAM in single threaded mode.

Algebra. We use BLS12-381 as our pairing friendly curve and the BLAKE3 hash function as our random oracle.

Dealer. We implement the Setup and EpochSetup protocols using a trusted dealer for a committee of size 16. However, these can be replaced with efficient secure multi-party computation protocols. We note that the Setup protocol is run only once per committee and the EpochSetup can be carried out by the committee during downtime where they are not issuing partial decryptions.

Evaluation. We now evaluate the performance of our batched-threshold encryption scheme. In particular, we aim to answer three main questions and provide insights about bottlenecks in different parts of the protocol.

- How long does it take to encrypt a message? What is the size of corresponding ciphertexts?
- How long does it take for a committee member to compute partial decryptions and how does it vary with batch size? How does it vary with committee size?
- How long does it take to recover all messages using partial decryptions and how does it vary with batch size?

Encryption. As is evident from the protocol description Fig. 2, a constant number of operations are required to encrypt a message, independent of committee/batch size. Our experiments show that it takes less than 6 ms to encrypt a 32 byte message. The ciphertext consists of 1 $G_1$ element, 2 $G_T$ elements, 3 $F$ elements, a 2 byte description of $\tilde{x}$, and a string proportional to the message length (32 bytes), resulting in a total size of 370 bytes with group element compression. This is $\approx 3 \times$ larger than the threshold Cramer-Shoup CCA2 secure encryption scheme [12, 13] which consists of 4 group elements in a non-pairing friendly group (128 bytes) such as [9].

Partial Decryption. Here, each party in the committee checks a zero-knowledge proof, and then computes it’s partial decryption. We observe that over 99% of the cost comes from verifying the proofs of knowledge of discrete log and pairing checks needed for CCA2 security. This warrants the investigation of zero-knowledge proofs that can be verified faster in batches. Table 2 confirms that the time taken grows linearly in the size of the batch of transactions being decrypted. If our scheme was deployed on Ethereum, which processes roughly 500 transactions per block, it would take approximately 2.8 s to verify and compute a partial decryption. Each partial decryption is 80 bytes (one $F$ element and one $G_1$ element).

<table>
<thead>
<tr>
<th>Batch size</th>
<th>Partial decrypt (ms)</th>
<th>Reconstruct (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>41.5</td>
<td>41.9</td>
</tr>
<tr>
<td>32</td>
<td>173.4</td>
<td>165.0</td>
</tr>
<tr>
<td>128</td>
<td>678.11</td>
<td>781.4</td>
</tr>
<tr>
<td>512</td>
<td>2818.6</td>
<td>3472.2</td>
</tr>
</tbody>
</table>

Table 2: Benchmarks of time taken to compute partial decryptions and to reconstruct messages as the batch size grows.

Reconstruct. Here, any party (without secrets) can recover the messages given all the partial decryptions. The bottleneck here is the $O(B \log B)$ group operations, and $O(B)$ pairings
needed to produce all KZG opening proofs using \[27\]. We provide timings in Table 2. Again, if our scheme were to be deployed on Ethereum, it would cost less than 3.5 s to decrypt 512 transactions in single-threaded mode and this can be fully parallelized. We note that we implement an optimistic decryption protocol where all the shares received from the committee are valid. If certain shares are invalid, the reconstruction protocol will need to use a more expensive error correction protocol such as \[62\].

**Varying Committee Size.** Given that the only component that scales with committee size is the size of the Fast Fourier transform to reconstruct \(p(y)\) and \(g'\) in Fig. 2, increasing the committee size when batch size is large (512 say), does not have a noticeable impact on the reconstruction time which is dominated by the cost of producing KZG opening proofs.

**Incentive Mechanism.** Finding a good committee where a majority of parties behave honestly can be quite challenging. The Shutter network, which has deployed threshold encryption, aims to solve this through decentralized governance via a ShutterDao, where the committee members are chosen by stakeholders in the system. To ensure liveness, one can reward nodes and penalize malicious behavior via stake slashing. We note that this is similar to the case where if too many block validators go offline in Ethereum, the blockchain comes to a standstill.

**Denial of Service.** Unfortunately, transaction privacy comes with the additional challenge of potential denial of service attacks. We present a few approaches to handle this below. However, these approaches are application specific and should be addressed more carefully in future work that aims to deploy encrypted mempools.

A general strategy to prevent “spam” is to demand proofs from users that transactions are valid. These proofs can be quite efficient when designed for specific applications. Consider for instance, a DEX such as Uniswap on an L2, and the client wishes to exchange tokens. Here, it may be sufficient to encrypt just the amounts and token names along with a range proof showing that they indeed own enough tokens. Another option is to implement some kind of penalty for submitting invalid transactions, which raises the cost of launching a denial of service attack. Note that sender addresses need not be encrypted, which allows for “blacklisting” of repeat offenders.

**Remark 3.** Not all transactions in the mempool need privacy. For instance, a peer-to-peer transfer can be submitted in the clear to the mempool as the recipient would receive the same amount, irrespective of the ordering of the block. Since transactions with mempool privacy are more expensive they can be priced differently, and those with a need for privacy can opt in.

### 9.1 End-to-End Performance.

In this section, we compare the end to end performance of our batched-threshold encryption scheme with Ferveo \[5\], MEVade \[50\], McFly \[24\] and Shutter Network \[58\]. We consider the constant sized ciphertext scheme from McFly which relies on the committee holding shamir shares of the secret key. The other scheme has ciphertext size linear in the committee size, which is impractical in the blockchain setting.

We provide concrete performance figures in Table 3 for our work and contrast them with Ferveo \[5\], MEVade \[50\], McFly \[24\] and Shutter Network \[58\]. We also provide benchmarks on how expensive committee changes are in Table 4.

**Setup.** All schemes apart from ours, only need a random value to be secret shared at the beginning of the protocol. In addition, our protocol needs to carry out multiplication and inversions in MPC. To estimate the cost of this we point to prior work \[15\] which provides benchmarks for a protocol run over a WAN with 50 parties. The depth of our setup circuit is 3 as the powers of tau can be computed in depth 1 using \[3\], and the lagrange coefficients need an additional 2 rounds (one for inverse and one for multiplication). The number of gates is \(O(B)\) which is \(< 10000\) for a batch size of 512. From Table 5 of \[3\], we can see that a circuit of depth 20, and 1 million gates can be computed in under 3 minutes for the WAN configuration. This provides a very conservative upper bound on the setup protocol and the cost grows linearly with the number of parties. We note here that these protocols will abort when a party behaves maliciously. There are other guaranteed output delivery MPC protocols, albeit with more rounds but similar communication \[34\], that can identify misbehaving parties, allowing them to be penalized. One could run the protocols with abort, optimistically, and if parties abort, then they can run the guaranteed output delivery protocol.

**Epoch Setup.** We assume that the setup for many epochs is carried out at once. Each epoch requires one multiplication and 3 secret shared random values for \(\alpha, \delta\) and \(r\). However, in this case we additionally demand that the sharing of \(g'\) is a publicly verifiable secret sharing \[31, 35, 38, 56\] as malicious parties should not be able to cause the protocol to abort after execution. For this particular task, a 50 node network, \[38\] takes less than 18 seconds on a LAN. We note here that the communication is actually quite small (< 1 MB) and most of the time is spent on computation. Thus, we expect that one can also scale this to a WAN setting without incurring much more overhead. Moreover the epoch setups can be parallellized and carried out well before the actual epoch. It is expected that this number will come down with faster implementations of hidden order groups. This cost can be further amortized when generating many sharings using techniques from \[6\].

**Practical findings.** We used average transaction and block sizes, as well as selected a standard curve, to provide real estimates of the cost of our system. In particular, we analyze average transaction size increase, as well as the gas cost of
Table 3: Ciphertext size and total decryption broadcast size for \( n \) committee members, \( B \) transactions per block, in terms of \( s \) the size in bytes of the encapsulation key, and \( |G_1|, |G_2|, |G_T|, |F| \) the size in bytes of the representation of elements from each group and field. Concrete numbers obtained with the BLS12-381 curve, using the average transaction and block size on Ethereum in 2023. We use the ETH/USD rate on January 1, 2024 for conversions. The last two rows use a committee size of 128 and a block of 512 transactions to give real-world communication cost estimates.

Table 4: Time taken to change a committee with 64 parties using the protocol from [33] on a LAN and WAN network. Here Setup refers to the time taken to establish \( n/2 \) correlations used to "transfer" ownership of shamir secret shared values to a new committee. Refresh refers to the time taken to transfer one secret to the new committee.

**Committee churn.** When a party joins or leaves the committee, the old committee needs to securely "transfer" the secrets it holds to the new committee. This can be done using a dynamic pro-active secret sharing protocol such as [33]. We adapt their code to our setting and provide benchmarks in Table 4. Due to the structure of the protocol in [33], the setup phase, in one-shot, generates correlations required for the refresh of \( n/2 \) correlations. Even if we only need to refresh one secret, as is the case in prior work, such as Ferveo, we still have to pay the price of generating \( n/2 \) correlations during the setup phase. However, the refresh phase is the time taken to refresh one secret, but again this can be trivially parallelized across the total number of secrets that need to be refreshed. Extrapolating from Table 4, for a blocksize of 512, with a committee size of 64, tolerating 31 corruptions, and assuming each node can refresh 8 secrets in parallel (needs 8 threads), it would take less than 2 minutes on a WAN and under 4 minutes on a LAN.

Table 4: Time taken to change a committee with 64 parties using the protocol from [33] on a LAN and WAN network. Here Setup refers to the time taken to establish \( n/2 \) correlations used to "transfer" ownership of shamir secret shared values to a new committee. Refresh refers to the time taken to transfer one secret to the new committee.

<table>
<thead>
<tr>
<th>Network</th>
<th>Setup (s)</th>
<th>Refresh (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAN</td>
<td>6.24</td>
<td>0.22</td>
</tr>
<tr>
<td>WAN</td>
<td>8.21</td>
<td>1.5</td>
</tr>
</tbody>
</table>

In conclusion, we believe that our work represents an important step towards achieving mempool privacy in blockchains. Although the (epoch) setup is more expensive than prior work, we believe it offers a desirable tradeoff when the committee changes are infrequent viz. orders of magnitude less communication in the time-sensitive decryption step in exchange for a more expensive setup.

---

**Appendix:**

- **Our setup phase is more expensive than prior works,** who only need to establish one publicly verifiable secret sharing. But we conservatively expect this overhead to be under 100×. Moreover, our full setup is only run once for a given committee, and our per-epoch setup can be done in advance, and only requires communication between the committee members, where good network connections can be established. Whereas during decryption, the partial decryptions need to be broadcast to the entire network, via gossiping which is typically much slower. Thus, we optimize the latter at the expense of the former. The size of the decryption shares needed to reveal all transactions in a block is only 2% of the size of a block using batched-threshold encryption, versus 500% using Ferveo, making propagation to all nodes in the blockchain prohibitively slow.

- **During batched decryption,** each committee member broadcasts one group element from \( G_1 \) and one field element. This is roughly the same as McFly and Shutter’s (our message size is roughly double), but we additionally provide transaction privacy. Ferveo provides pending transaction privacy but the amount of information that needs to be broadcast is \( B \times \) larger. In a setting with \( n = 128 \) and \( B = 512 \) transactions, we only require a total broadcast of 10KB, versus 3MB for Ferveo. [22] showed that each kilobyte adds \( \approx 80 \) ms to the block propagation time. This implies that Ferveo adds an overhead of \( \approx 250 \) s for block propagation, whereas we add just 0.8 s.

- **Our ciphertext is slightly larger than prior works,** but still constant size. In practice, we require 400 bytes for our ciphertext, which represents $2 additional spent gas, while Ferveo costs $0.7, McFly $1.4, and Shutter $0.6.

---

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Ferveo/MEVade</th>
<th>McFly</th>
<th>Shutter</th>
<th>This Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ciphertext size</td>
<td>(</td>
<td>G_1</td>
<td>+</td>
<td>G_2</td>
</tr>
<tr>
<td>Block decryption broadcast size</td>
<td>( nB</td>
<td>G_1</td>
<td>)</td>
<td>( n</td>
</tr>
<tr>
<td>Change in transaction size</td>
<td>12%</td>
<td>23%</td>
<td>10%</td>
<td>30%</td>
</tr>
<tr>
<td>Ciphertext storage gas cost (USD)</td>
<td>15,120 ($0.7)</td>
<td>30,240 ($1.4)</td>
<td>13,440 ($0.6)</td>
<td>42,000 ($1.9)</td>
</tr>
<tr>
<td></td>
<td>Partial Decryptions</td>
<td>3MB</td>
<td>6KB</td>
<td>6KB</td>
</tr>
<tr>
<td></td>
<td>Block Size</td>
<td>500%</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>
10 Acknowledgments

We thank the USENIX Security Symposium reviewers for their valuable feedback. Research supported in part by DARPA under Agreement No. HR00112020026, AFOSR Award FA9550-19-1-0200, NSF CNS Award 1936826, and research gifts/awards by Visa Inc, BAIR Commons Meta Fund, Stellar Development Foundation, a J.P. Morgan Faculty Research Award, a Berkeley Center for Responsible, Decentralized Intelligence (RDI) Fellowship and a Bakar Fellows Spark Award.

References


A Security Proof

Due to space constraints, we defer the security proof to the full version of the paper [16]. Below, we introduce the $q$-SBDHT assumption (Definition 4) and the main theorem (Theorem 5).

We prove security with guaranteed output delivery against a fully malicious adversary corrupting up to $t < n/3$ members of the committee and any number of users creating ciphertexts. We note that it is also possible to upgrade the protocol to $t = n/2$ using standard techniques of publishing commitments to secret keys of parties and proving that partial decryptions were computed correctly. For instance, one could use the KZG commitment scheme itself and prove that the computation was carried out using a zkSNARK such as the Halo2/STARK [8] proof systems. Although this would make the decryption procedure more expensive, one can always optimistically decrypt without proofs and if it fails, then demand proofs. By introducing appropriate penalties such as slashing of stake, we can ensure that incorrect partial decryptions are rarely sent – similar to validators on Ethereum attesting to incorrect state transitions.

In order to highlight the core technical ideas without distractions, we focus on the $t = n/3$ case. In the process, we introduce a new assumption that is closely related to the assumption introduced in [39]. We call this the $q$-strong Bilinear Diffie Hellman Triple Assumption to indicate that there are additional triples given to the adversary beyond that in $q$-SDH and “Bilinear” to indicate that the Adversary needs to output a value in the target group.

Definition 4 ($q$-SBDHT Assumption). Let $a, b, c$ and $d$ be sampled uniformly at random from $\mathbb{F}$. Given as input a tuple $(g, g^a, g^{a^2}, \ldots, g^{a^n}, g^b, g^d, h^d, h^{ac}, h^{cd})$, for every probabilistic polynomial time algorithm $A$, 

$$\Pr[A(g, g^a, g^{a^2}, \ldots, g^{a^n}, g^b, g^d, h^d, h^{ac}, h^{cd}) \rightarrow e(g, h)^{bc}] \leq \text{negl}(\lambda).$$

We briefly discuss the need for the new assumption, and discuss why the “KZG assumption”, i.e. the $q$-strong Diffie Hellman ($q$-SDH) assumption does not suffice for our setting. Specifically, in the KZG polynomial commitment scheme, the $q$-SDH is used to argue evaluation binding, which states that even for a (potentially adversarially) chosen polynomial commitment, the adversary cannot produce two accepting evaluation proofs for two distinct evaluations at the same evaluation point, i.e. adversary cannot find a commitment $c$, a tuple $x, y, y’, \pi, \pi’$ with $y \neq y’$ such that: (i) $\pi$ attests that $y$ is a valid evaluation of $x$ with respect to $c$; and (ii) $\pi’$ attests that $y’$ is a valid evaluation of $x$ with respect to $c$. Whereas we require that for an honestly generated commitment, if the adversary does not know the polynomial used in the commitment, it cannot generate a valid proof for any pair $(x, y)$, even if $y$ was such that $p(x) = y$, where $p$ was the polynomial that was committed. We thus view the above assumption as a strengthening of $q$-SDH, necessary for our construction.

Theorem 5. The protocol in Fig. 2 securely emulates the $\mathcal{F}_{\text{TPKE}}$ ideal functionality (Fig. 1) in the dealer model against any static PPT adversary $A$ corrupting $< n/3$ parties, given any weakly simulation-extractable NIZK, and provided the $q$-SBDHT assumption holds in the programmable random oracle model.

Remark 6. We note that the $n/3$ threshold arises from the fact that adversarial committee members may send incorrect shares during the batch decryption procedure. One can also allow the adversary to corrupt $< n/2$ parties and prove that the protocol emulates the ideal functionality with abort. To achieve guaranteed output delivery in this setting we can use standard techniques (at the cost of concrete efficiency) of attaching proofs that all parties carried out their local computation correctly, thereby allowing us to successfully recover secrets from threshold secret sharing, even if $n/2 − 1$ corrupt parties behave maliciously and provide incorrect shares. In order to verify proofs of correct local computation, the statements need to be “public”. Proving correct computation over private shares is often done by broadcasting private shares to the committee by encrypting to the committee member in question, but as this is not relevant to our contribution, we ignore such details from this work and refer the reader to relevant multiparty computation works for more details.