Shutter Network: Private Transactions from Threshold Cryptography

Stefan Dziembowski[1], Sebastian Faust[2], and Jannik Luhn[3]

[1] University of Warsaw, EnParties Sp.z.o.o
[2] Technical University Darmstadt
[3] Shutter Network

Abstract. With the emergence of DeFi, attacks based on re-ordering transactions have become an essential problem for public blockchains. Such attacks include front-running or sandwiching transactions, where the adversary places transactions at a particular place within a block to influence a financial asset's market price. In the Ethereum space, the value extracted by such attacks is often referred to as miner/maximal extractable value (MEV), which to date is estimated to have reached a value of more than USD 1.3B. A promising approach to protect against MEV is to hide the transaction data so block proposers cannot choose the order in which transactions are executed based on the transactions' content. This paper describes the cryptographic protocol underlying the Shutter network. Shutter has been available as an open-source project since the end of 2021 and has been running in production since Oct. 2022.

1 Introduction

Public blockchains like Ethereum [28] process and store transaction data in a decentralized and transparent way, providing users with open access to financial services. However, in 2020, Daian et al. [10] showed that the public availability of transaction data in public blockchains poses a significant privacy concern for users, which they coined by the term *miner extractable value (MEV)*[4]. Essentially, MEV describes the value that a block proposer can extract from a batch of transactions by re-ordering the batch or by inserting/removing certain transactions. For instance, a well known MEV attack is *front-running*, where a block proposer sees a user's intent to buy an asset and then places its own buy order just before the user's transaction to take advantage of the price movement caused by the user's transaction. As a consequence the block proposer makes a risk-free profit at the expense of the user.

Countermeasures Against MEV Attacks. In the literature, two main types of countermeasures against MEV attacks have been proposed [29]. The first class of countermeasures tries to "democratize" MEV by ensuring all block proposers have the same capability of extracting MEV [22]. Essentially, this guarantees that MEV becomes an additional fee paid by the users to the proposers to ensure the system's overall stability. A second approach is ensuring that proposers have limited control over the order of transactions. This can be done using time-based order fairness, where the consensus algorithm ensures that transactions are processed in the order they are received [21]. Unfortunately, some impossibility results to what extent such time-based order fairness can be guaranteed have been shown [21]. Moreover, such schemes are typically incompatible with the existing blockchain ecosystems as they require significant changes to how the consensus is done. An alternative approach to limit the control of block proposers is to hide the transaction content such

[4] The term was later changed to *maximal extractable value*.

that block proposers have to decide on an order without knowing the transaction content. There are multiple cryptographic techniques to achieve such content-agnostic ordering. This includes techniques such as multiparty computation (MPC), where the processing of transactions is run by a committee of servers via MPC [5]. Another popular approach is to rely on a commit-and-reveal approach (e.g., see [29] for an overview). The commit-and-reveal approach works in two phases. In the first phase, users commit to their transactions, where the content of the transaction is hidden inside the commitment. Once the transaction order is fixed, the content of the transaction gets revealed, which allows the system to execute the transactions.

The commit-and-reveal approach can be instantiated using various cryptographic techniques. The two most popular are timed cryptography (e.g., leveraging time-lock encryption) [16], and threshold cryptography. Timed-cryptography-based systems do not rely on an honest majority assumption but require the maintainers of the system to execute some heavy computation. Threshold cryptography, on the other hand, requires that a majority of the system's maintainers are honest. On the positive side, however, it relies on more standard cryptographic techniques and thus can be much more efficient. Moreover, it eliminates practical issues such as estimating the exact complexity of solving the time-lock puzzle. The Shutter network that we present in this paper leverages threshold cryptography. In the following, we will give a high-level overview of the underlying cryptography of Shutter. We emphasize that the rest of this document is not intended as a full academic paper but to disseminate to the community the main cryptographic ideas behind Shutter. There have recently been several proposals that leverage similar ideas to what is currently implemented by Shutter [4, 12, 23]. As in Shutter, these works rely on identity-based encryption to decrypt transactions "as a bundle". In addition, they offer a more detailed security analysis of the techniques used in this paper.

1.1 Applications of the Shutter protocol

The Shutter system was originally designed to protect against MEV attacks on public blockchains and is already used for that purpose on the Gnosis Chain [5]. We write more about concrete, practical settings in this case in Section 4.3. However, it can also be used for other purposes. In fact, it is already used for *shielded voting for DAOs:* In an election, votes must typically remain private. With Shutter shielded voting, the content of a vote remains private. Currently, Shutter is already live on the Snapshot[6] voting platform. We believe that also other applications can benefit from Shutter. One example is *decryption based on condition*, where users may encrypt messages that are revealed only when a certain condition is met (e.g., after some time has passed or when a certain event has happened). Another example is *censorship protection on Layers 1 and 2:* When transaction content is encrypted, miners cannot censor certain transactions (and can also not be forced to do so).

2 High-level idea of the Shutter protocol

The general idea of Shutter is that the system users encrypt the content of their transactions, and the encrypted transactions are decrypted all at once. The protocol involves a set of *users* $\{\mathsf{U}_1, \ldots, \mathsf{U}_u\}$, and a set of so-called *keypers* $\{\mathsf{K}_1, \ldots, \mathsf{K}_n\}$. Moreover, the parties have access to a public ledger L, which supports smart contracts, e.g., Ethereum. We assume that a majority of the

[5] https://docs.gnosischain.com/shutterized-gc/
[6] https://snapshot.mirror.xyz/yGz91njKbw-sXsnAT6RkoMzPwvuddZritz37h1OWO8o

keypers are honest. The protocol starts with a setup phase when system parameters are generated in a distributed way by the keypers. These parameters consist of a master public key mpk (posted on the ledger) and the master secret key msk that is never revealed to any protocol participant. The master key msk exists in the system in an implicit way only: it is shared (see Sec. 3.3 for an introduction to secret sharing) between all the keypers (the share of each keyper K_j is denoted as msk_j), see also Fig. 1.

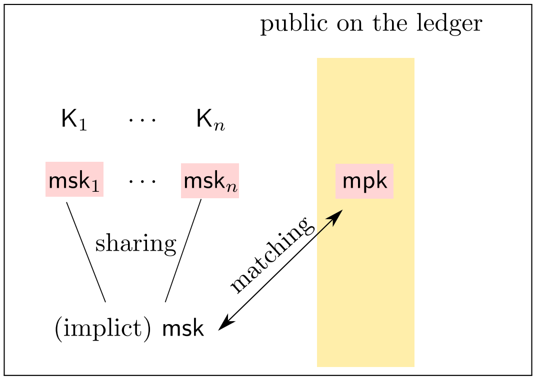


Fig. 1. Setup.

Then, the protocol runs in eons (indexed by numbers $i := 1, 2, \ldots$). In each eon i the users encrypt their transactions for this eon. To simplify the description, we assume that each user U has only one transaction T in each eon; the generalization to a larger number of transactions is straightforward. This encryption requires only local computation, taking as input mpk and the eon index i (no interaction with other parties is needed). Each encrypted transaction $C := \mathsf{Post}(\mathsf{mpk}, T, i)$ is posted on the ledger. This is depicted in Fig. 2.

The encrypted transactions of the users are publicly opened on the ledger at the end of each eon. This process involves the keypers and the ledger. By "publicly opened" we mean that the information on the ledger suffices to decrypt T in an efficient way (using hashing only). More precisely, for each encrypted transaction C, the keypers publish on the ledger a short information σ (where typically $|\sigma| \ll |C|$) such that later T (corresponding to C) can be quickly computed (using a function denoted Read) from (C, σ). This is done in order to minimize the amount of information sent to the ledger (an alternative approach would be to require that each decrypted transaction appears on the ledger in plaintext, but this can be costly for longer transactions). Technically, the

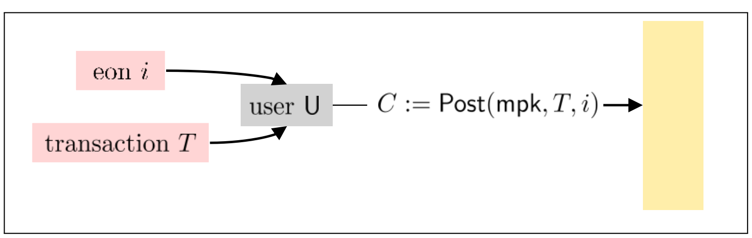


Fig. 2. Posting encrypted transactions on the ledger.

process of opening the transactions consists of (a) the keypers jointly (and interactively) computing the *ith eon's secret key* $\mathsf{sk}^{(i)}$, (b) using $\mathsf{sk}^{(i)}$ each keyper computes the σ value corresponding to each C. The correct value σ is decided by keypers voting on the ledger. This is depicted on Fig. 2

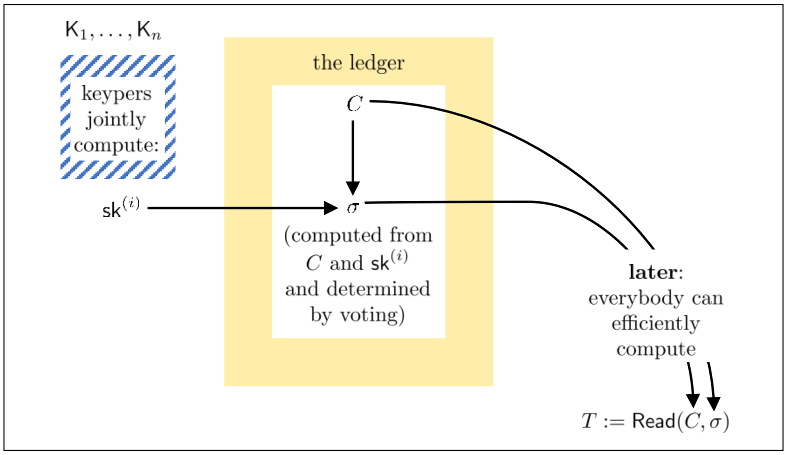


Fig. 3. Public opening of a ciphertext C.

The Shutter protocol is constructed using ID-based threshold cryptography [6, 14, 17]. Thanks to this, users do not need to interact with any other party while computing C (except for sending a message to the ledger).

3 Technical details

This section contains the technical details of Shutter. We start with the specification of the protocol properties (this is done in Sec. 3.1) and then describe the communication model (in Sec. 3.2). Sec. 3.3 contains cryptographic preliminaries. The actual construction is presented in Sec. 3.4.

3.1 Protocol specification

Let us now provide more details about the Shutter specification in addition to what was presented in Sec. 2. When we say that in some algorithm, a party P_i outputs a *private output*, we mean that this output is given only to P_i. If an algorithm's output is publicly available, then we say it is a *public output*. The protocol consists of the following algorithms:

Setup: A randomized algorithm DistrSetup executed jointly by all the keypers. We assume that it takes as input a security parameter 1^κ and the number of keypers n. The output of the DistrSetup is as follows:

- public output: master public key mpk posted on the ledger, and
- private output of each keyper K_i: a secret key msk_i.

Posting transactions in eon i: A randomized algorithm $\mathsf{Post}(\mathsf{mpk}, T, i)$ (where $i \in \mathbb{Z}_{>0}$ and $T \in \{0,1\}^*$) is a transaction) executed by a user in eon i. As a result, an encrypted transaction T is posted on the ledger. Denote this ciphertext with C.

Opening the transactions in eon i: A deterministic algorithm Open executed jointly by all the keypers. The algorithm looks at the ledger and for every C that appeared there in the previous phase publishes σ – a string that together with C can be used to efficiently compute T that corresponds to C using algorithm denoted "Read" as

$$T := \mathsf{Read}(C, \sigma) \tag{1}$$

We could simplify this procedure and just have $\sigma = T$ (in which case "computing T from σ is trivial: simply $\mathsf{Read}(C, \sigma) = \sigma$.). The need for the above definition comes from the fact that this will allow us to optimize the amount of data on the ledger.

We assume that these algorithms are executed in "phases": first, in the *setup phase*, the keypers execute the DistrSetup algorithm, then the eons are executed, each of them consisting of a *posting phase* and *opening phase*. We assume that the phases do not overlap in time. In particular, in each eon, the posting phase ends before the opening phase starts. We now have the following condition that Shutter has to satisfy:

Correctness: Suppose an honest user U_j posted a transaction T in eon i resulting in the following values appearing on the ledger: C (in the posting phase) and σ (in the opening phase). Then the $\mathsf{Read}(C, \sigma) = T$.

Additionally, we have the following security properties:

Secrecy: Each transaction T posted in eon i remains secret until the opening phase of this eon starts (the only thing that leaks to the adversary is the length of T).

Non-malleability: Suppose an honest user posted a transaction T in eon i. Then in this eon no dishonest user can successfully post a transaction T' that is related in a non-trivial way to T. In other words, each such T' is either independent of T, or it is equal to it (but, e.g., it *cannot* be equal to T with all the bits flipped). Again, this does not apply to the length of T: we allow a dishonest party to choose T' of length that is related to the length of T.

Commitment: Once a user posts a transaction T on the ledger, she cannot delete it or modify it. It will be opened by the keypers regardless of U_j's willingness to help.

The non-malleability property is a bit subtle: note that we *do* allow the malicious parties to "re-post" a transaction T of an honest party (in the same eon). This cannot be prevented since a malicious party can always simply take C (that corresponds to T) and post $C' := C$ on the ledger. Hence, the users have to take care to make such copying harmless. For example, a user U_j can add her identifier "U_j" to a transaction and post (U_j, T) (plus a counter, if we allow users to post more than one transaction per eon). Note that the non-malleability propery implies that a malicious user U_k (for $k \neq j$) *cannot* take C (that decrypts to (U_j, T)) and "maul it" to some other C' that decrypts to (U_k, T).

An error in the implementation discovered by Choudhuri et al. [8]. As discovered by [8], the initial implementation of Shutter was vulnerable to malleability attacks. This was because it did not follow the description presented in this document due to a programming error. More precisely the value of r computed in Eq. (2) (see page 7) was computed just by hashing σ (not (σ, T)). We are very grateful to the authors of [8] for spotting it and notifying the Shutter programming team and us according to the best practices of coordinated vulnerability disclosure.

3.2 Communication between the parties

Before we proceed to the description of Shutter, let us present the details of the communication model. The keypers are connected by pairwise secure channels. The keypers and the users have access to the ledger (they can read and post transactions on it). We assume that the keypers can run a broadcast protocol between themselves. Whenever we say that a keyper *broadcasts* some value v to other keypers, it means that she initiates this broadcast protocol with his input v. We assume that it is always clear who broadcasts a given message (i.e., it contains an identifier of the keeper who sent it and is signed by her). In our implementation (see Sect. 4.4), we use Tendermint (see https://tendermint.com) for broadcast.
Another technique that we use is voting on the main chain. Suppose each keyper locally computes some value v as a deterministic function of the publicly-available data (i.e. data on the main chain or data broadcast by the keypers). Then, the keypers can inform the blockchain about this value by sending it to the contract and voting on it. For completeness, the voting procedure is described in Fig. 4

The voting procedure

Let t be the maximal number of corrupt parties. A contract on the main chain has a function vote that takes two parameters:

- vote_id – a unique identifier of the particular voting procedure and
- a value v.

Each time a K_i calls vote with parameters $(\mathsf{vote\_id}, v)$ from a keyper K_i the function does the following:

1. If K_i already called vote with the same parameter vote_id then this call is ignored.
2. Otherwise:
   (a) If no keyper made a call with parameter $(\mathsf{vote\_id}, v)$ before: the contract stores $(\mathsf{vote\_id}, v)$ with a label $i := 1$
   (b) Otherwise (i.e. if $\mathsf{vote}(\mathsf{vote\_id}, v)$ was called before by another keyper): the contract increases the label of $(\mathsf{vote\_id}, v)$ by 1. If, as a result, this label exceeds t then this function changes this label to "agreed" and ignores any further calls of vote with parameter vote_id.

Fig. 4. The voting procedure

3.3 Preliminaries

In this paper, we use standard cryptographic notions like the *random oracle model* and *signature schemes* (see, e.g., [20]). Below, we describe the main cryptographic tools we use in Shutter (bilinear maps, identity-based encryption, and secret sharing).

Bilinear maps. We use the notation for bilinear maps from [6] (page 7), with the exception that the elements P and Q that are paired come from two groups that can be different (denoted $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively). Throughout this document, $\mathbb{G}_1$ and $\mathbb{G}_2$ are additive groups, i.e., its neutral element is 0, its operation is "+", and applying n times this operation to a group element P is denoted as "$n \times P$". Furthermore, let $\mathbb{G}_T$ be a multiplicative group, i.e., its neutral element is 1, its operation is "$\cdot$", and applying n times this operation to a group element P is denoted as "P^n".

We assume that both groups have a prime order q. Moreover, we let $\hat{e}$ be a *bilinear map*, i.e., it is a poly-time computable function $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ that satisfies the following conditions:

1. For all $(P, Q) \in \mathbb{G}_1 \times \mathbb{G}_2$ and all $a, b \in \mathbb{Z}$ we have that

$$\hat{e}(aP, bQ) = \hat{e}(P, Q)^{ab},$$

2. Map $\hat{e}$ does not send all pairs (P, Q) to identity in $\mathbb{G}_T$.

We assume that the *Bilinear Diffie-Hellman (BDH) problem* is hard for $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e})$ (see [6, 15]). In our implementation (see Sect. 4.4), we instantiate these abstract objects with the ones proposed in [7].

Identity-based encryption. To encrypt transactions, we use a CCA-secure scheme (instead of a simpler CPA-secure one). This is because the encryption scheme that we use has to be *non-malleable* [11], as otherwise, a malicious user could break the non-malleability of Shutter (i.e. post a transaction T' that is a function of a transaction T posted by an honest user in the same eon, see Sec. 3.1). Luckily, non-malleability is implied by CCA security (see, e.g., [2]).

Let κ be the security parameter. Let $(\mathsf{Setup}, \mathsf{Extract}, \mathsf{Encrypt}, \mathsf{Decrypt})$ an Identity-Based-Encryption (IBE) scheme secure against an adaptive Chosen Ciphertext Attack (CCA) FullIdent from [6] — see Def. 2.1 (in [6]) for the definition, and Sec. 4.2 (in [6]) for the construction. The only difference between our construction and the one of [6] is that we assume that the identities are elements of the set of natural numbers $\mathbb{N}$ (instead of binary strings). This small choice makes the IBE scheme more compatible with our application.

In the sequel: $H_1 : \mathbb{N} \to \mathbb{G}_1$, $H_2 : \{0,1\}^* \to \{0,1\}^\kappa$, $H_3 : \{0,1\}^* \to \mathbb{Z}_q$ and $H_4 : \{0,1\}^* \to \{0,1\}^\kappa$ are hash functions. Function H_4 will be used as a pseudorandom function to encrypt transactions in the "counter mode" (see below). This choice is due to the limitations of the programming language Solidity used by Ethereum. The scheme is described below:

Setup: Generate a (master secret key, master public key) pair $(\mathsf{msk}, \mathsf{mpk}) \in \mathbb{Z}_q^* \times \mathbb{G}_2$ as follows: take a random $\mathsf{msk} \leftarrow\$ \mathbb{Z}_q^*$ and set $\mathsf{mpk} := \mathsf{msk} \times P$.

Extract: For a party with identity $i \in \mathbb{N}$ let her private key be equal to $\mathsf{sk}_Q := \mathsf{msk} \times H_1(i)$.

Encrypt: To encrypt transaction T to a party with identity Q, divide T into m blocks: $T = T_1 || \cdots || T_m$ (where the length of each T_i is equal to the length of the output of a hash function H_4). Sample $\sigma \leftarrow\$ \{0,1\}^\kappa$ and let

$$r := H_3(\sigma, T). \qquad (2)$$

Then compute

$$C_1 := r \times P$$

and

$$C_2 := \sigma \oplus H_2\left((\hat{e}(Q, \mathsf{mpk}))^r\right),$$

and

$$C_3 := (H_4(\sigma||1) \oplus T_1, \ldots, H_4(\sigma||m) \oplus T_m)$$

(note that this is essentially the counter mode of encryption; see, e.g., [20]). Finally, define the ciphertext as:

$$\mathsf{Enc}^{\mathsf{ID}}(\mathsf{mpk}_Q, T) := (C_1, C_2, C_3),$$

Decrypt: To decrypt a ciphertext $(C_1, C_2, (C_3^1, \dots C_3^m))$ with a secret key sk_Q do as follows. Reject all ciphertexts that do not have a form

$$(C_1, C_2, C_3) \in \mathbb{G}_2 \times \{0,1\}^\kappa \times (\{0,1\}^\kappa)^* .$$

Then let

$$\sigma := C_2 \oplus H_2(\hat{e}(\mathsf{sk}_Q, C_1)), \tag{3}$$

and

$$T := (C_3^1 \oplus H_4(\sigma||1))|| \cdots ||C_3^m \oplus H_4(\sigma||m)) \tag{4}$$

and

$$r := H_3(\sigma, T).$$

Reject the ciphertext if $C_1 \neq r \times P_2$. Otherwise output

$$\mathsf{Dec}^{\mathsf{ID}}(\mathsf{sk}_Q, (C_1, C_2, (C_3^1, \dots C_3^m))) := T.$$

Distributed Boneh-Franklin encryption. We now outline the distributed Boneh-Franklin identity-based encryption scheme, which was originally sketched in [6] (see "Distributed PKG" on page 22) and then described in more detail in [18, 17]. For consistency with the rest of the paper, we assume that the parties who run the protocol are also called *keypers* and denoted $\mathsf{K}_1, \dots, \mathsf{K}_n$.

We use Shamir's (t, n)-threshold secret sharing scheme ($\mathsf{share} : \mathbb{Z}_q \to \mathbb{Z}_q^n, \mathsf{reconstruct} : \mathbb{Z}_q^{t+1} \to \mathbb{Z}_q$) [26], in order to ensure that $t+1$ keypers are needed to reconstruct the master secret key msk, and any set of at most t keypers has no information about msk. We assume that every keyper K_i is assigned a unique point $x_i \in \mathbb{Z}_q \setminus \{0\}$ and let $x_0 := 0$. Recall that in Shamir's secret sharing, a secret s is shared by selecting a random polynomial φ such that $\varphi(0) = s$, and each keyper K_i holds $\varphi(i)$. For completeness, the full description of Shamir's secret sharing appears Appx. A.1.

The *distributed Boneh-Franklin identity-based scheme* is a pair ($\mathsf{DistrSetup}, \mathsf{DistrExtract}, \mathsf{Encrypt}, \mathsf{Decrypt}$) of distributed algorithms, where $\mathsf{DistrSetup}$ is the *distributed key generation* algorithm, and $\mathsf{DistrExtract}$ is the distributed key extraction algorithm, and $\mathsf{Encrypt}$ and $\mathsf{Decrypt}$ are as in the standard Boneh-Franklin scheme (see Sect. 3.3). $\mathsf{DistrSetup}$ procedure takes no input. At its end, each honest keyper K_i outputs a value $\mathsf{msk}_i \in \mathbb{Z}_q$ such that msk_i's are a valid sharing of some secret msk. In addition to this, a vector $(\mathsf{mpk}, \mathsf{mpk}_1, \dots, \mathsf{mpk}_n)$ is made public, where

$$\begin{aligned}\mathsf{mpk} &:= \mathsf{msk} \times P_2 \\ \mathsf{mpk}_i &:= \mathsf{msk}_i \times P_2 \quad \text{for } i = 1, \dots, n.\end{aligned}$$

In the construction of the $\mathsf{DistrSetup}$ procedure, we use the fact that Shamir's secret sharing is homomorphic (i.e. a sum of shares of some secrets $s^{(1)}, \dots, s^{(n)}$ yields sharing of $s^{(1)} + \cdots + s^{(n)}$)

More precisely, in this phase, the keypers use a distributed key generation protocol over $\mathbb{Z}_q$ (see, e.g., [17, 18]) that results in

- a master secret key $\mathsf{msk} \in \mathbb{Z}_q$ that is shared using the Shamir's (t, n)-threshold scheme described above (where $t > n/2$ is some threshold) with each K_i holding a share msk_i, and
- a master public key $\mathsf{mpk} = \mathsf{msk} \times P_2 \in \mathbb{G}_2$ that is posted on the ledger and a vector $(\mathsf{mpk}_1, \dots, \mathsf{mpk}_n)$ that is known to all the keypers (we can think of each mpk_i as a "commitment" of K_i to msk_i). There is a consensus among the keypers about the value of this vector.

For the details, the reader may consult [6, 17, 18], or Fig. 12 in the appendix (see page 19). Below, we sketch the main idea of this procedure. It uses a *Feldman Verifiable Secret Sharing (VSS)* protocol denoted FeldmanVSS (see [13, 14]). For completeness, a full description of Feldman VSS is presented in Fig. 11 in the appendix (page 18). This protocol allows a dealer $D \in \{\mathsf{K}_1, \ldots, \mathsf{K}_n\}$ to share a secret value $s \in \mathbb{Z}_q$ between all the other keypers. At the end, each keyper K_i learns a share s_i of s. More precisely, we are guaranteed that there exists a polynomial φ of degree at most t such that for each honest keyper, we have that $s_i = \varphi(x_i)$. Moreover, if the dealer is honest, then $\varphi(0) = s$. In addition to this, a vector $(\pi_0, \ldots, \pi_n)$ is made public, were

$$\pi_0 := s \times P_2$$
$$\pi_i := s_i \times P_2 \quad \text{for } i = 1, \ldots, n.$$

Given these values, the keypers can reconstruct the shared secret s using Lagrange polynomial interpolation. Moreover, VSS's "verifiability" feature allows parties to check the consistency of the shares using the vector $(\pi_0, \ldots, \pi_n)$.

Remark. We notice that Shutter currently relies on a simple DKG, whose performance can easily be improved by relying on more advanced techniques (e.g., KZG commitment [19]).

3.4 The protocol

Setup. The parties run the DistrSetup protocol resulting in values mpk_i and msk_i (see above).

Posting transactions in eon i. In order to post a transaction $T \in \{0,1\}^*$ in eon i a user U_j performs a procedure $\mathsf{Post}(\mathsf{mpk}, T, i)$ depicted on Fig. 5.

Opening the transactions in eon i. This phase is executed at the end of an eon by the keypers. It proceeds as in Fig. 6.

$\mathsf{Post}(\mathsf{mpk}, T, i)$

1. Let $(C_1, C_2, C_3) := \mathsf{Enc}^{\mathsf{ID}}(\mathsf{mpk}, H_1(i), T)$
2. Post (C_1, C_2, C_3) on the ledger.

Fig. 5. Posting an encrypted transaction T on the blockchain in eon i.

Reading T. As a result of the opening procedure for each (C_1, C_2, C_3) that was posted during the "encrypting transactions" phase, the corresponding σ value (see Eq. (4)) is posted on the ledger (or it is decided that this ciphertext is invalid). Everybody can now decrypt T using the Read algorithm defined in Fig. 7.

Open(i)

3. Each keyper K_j proceeds as follows:
   (a) K_j broadcasts to all the keypers a value $Q_j^{(i)} := \mathsf{msk}_j \times H_1(i)$.
   (b) K_j waits to receive the "Q" values from the other keypers. For each such value $Q_k^{(i)}$ received from K_k the keyper K_j checks if it satisfies the following equation:

$$\hat{e}(Q_k^{(i)}, P_2) = \hat{e}(H_1(i), \mathsf{mpk}_k).$$

   K_j stops waiting once she receives $t+1$ values (including her own) that satisfy the above equation.
   Assume that the first $t+1$ values $Q_j^{(i)}$'s that satisfied the above equation check are $Q_{\ell_1}^{(i)}, \ldots, Q_{\ell_{t+1}}^{(i)}$. After receiving them each keyper K_j computes $\mathsf{sk}_j^{(i)}$ as

$$\mathsf{sk}_j^{(i)} := \lambda_{\ell_1} Q_{\ell_1}^{(i)} + \ldots + \lambda_{\ell_{t+1}} Q_{\ell_{t+1}}^{(i)}. \quad (5)$$

   (where the λ_j's are the Lagrange coefficients, see Sec. 3.4).
   (c) Now, for each (C_1, C_2, C_3) that was posted in this eon, the keyper K_j runs the decryption procedure $\mathsf{Dec}^{\mathsf{ID}}(sk_j^{(i)}, (C_1, C_2, C_3))$ (see Sec. 3.3), and:
      i. if the decryption procedure rejects this ciphertext, then send a vote "(C_1, C_2, C_3) is invalid" to the ledger,
      ii. otherwise send a vote "the key corresponding to (C_1, C_2, C_3) is σ" (where σ is computed in Eq. (3))

   (see Sec. 3.2 for the description of the voting procedure).
4. For each C_1: once the vote is finished all the parties accept that k that obtained σ votes is the valid decryption of C_1 (or, if $t+1$ votes say that the ciphrtext is invalid, then they accept that it is invalid).

Fig. 6. Opening the transactions from eon i.

Adding and removing keypers The basic protocol presented above can be extended to add or remove the keypers. The simplest way to do it is to simply re-initialize the system parameters. This should work as follows. First, the old set of keypers votes on the ledger if they want to add/remove some keyper. If the vote passes (i.e., it gets the majority of votes), then $\mathsf{DistrSetup}$ is executed by an updated set of keypers. Note that this can be "batched" to avoid performing the setup procedure too often. More precisely, the set of keypers can be updated once a month (say), with keypers removed/added simultaneously. Observe also that the new public key needs to be posted on the ledger (and the users need to be aware of this fact to update mpk locally).

Read(σ, C_3)

Let $C_3^1, \ldots, C_3^m$ be the blocks of C_3. Output

$$(C_3^1 \oplus H_4(\sigma||0))|| \cdots ||C_3^m \oplus H_4(\sigma||m)).$$

Fig. 7. Reading transactions

Concrete instantiation

We must be careful about identifying concurrent protocol sessions and different eons. To this end, in the real-life implementation, we require the following:

1. Each session of Shutter is parametrized by a unique identifier. Each message sent between the parties or sent by the parties to the ledger is labeled with this identifier. Moreover, the contract on the ledger knows this identifier. Messages are accepted as coming from a given session only if they have the right identifier.
2. Additionally: each message sent or broadcast in eon i is labeled with i. Only the messages with this label are accepted in a given eon.
3. When we say that K_i broadcasts a message to all the keypers this includes also K_i sending a message to herself. Of course, this can be implemented completely locally.

4 Analysis

This section argues that the Shutter protocol from Sec. 3.4 satisfies the protocol specification. We first start with arguing about correctness (in Sec. 4.1) and then about security (in Sec. 4.2).

4.1 Correctness

Correctness follows from the following facts.

Lemma 1. *Consider an execution of* DistrSetup*. Let* $\mathsf{mpk}, \mathsf{mpk}_1, \ldots, \mathsf{mpk}_n, \mathsf{msk}_1, \ldots, \mathsf{msk}_n$ *be the outputs of this execution. Let* $\{\mathsf{K}_{a_1}, \ldots, \mathsf{K}_{a_h}\}$ *be the set of honest keypers (we have* $t+1 \leq h \leq n$*). Then* $\mathsf{msk}_{a_1}, \ldots, \mathsf{msk}_{a_h}$ *are points on polynomial* ϕ *of degree at most* t*. Moreover*

$$\text{for every } j \text{ it holds that } \mathsf{mpk}_{a_j} = \mathsf{msk}_{a_j} \times P_2, \tag{6}$$

and

$$\mathsf{mpk} = \mathsf{msk} \times P_2, \tag{7}$$

where msk *is the "implicit master secret key"* msk*, i.e.:* $\mathsf{msk} := \phi(0)$*.*

The above lemma will be proven in an extended version of this document. We also have the following standard facts whose proofs appear in Appx. B.

Lemma 2. *If* K_j *is honest then* $\hat{e}(Q_j^{(i)}, P_2) = \hat{e}(H_1(i), \mathsf{mpk}_j)$ *holds.*

Lemma 3. *If a user* U *is honest, and at least* $t+1$ *keypers are honest, then the protocol never halts, and the value of* σ *computed while calculating* (C_1, C_2, C_3) *is equal to* σ *computed in the opening phase of* (C_1, C_2, C_3)*.*

4.2 Security

Let us now sketch the security argument. Assume the BDH assumption holds. First of all, note that by Theorem 5.1 of the extended (Eprint) version of [17], we get that our protocol, when viewed as a threshold IBE protocol, is IND-ID-CCA-secure against any poly-time adversary that corrupts at most t keypers. The only differences are as follows. Firstly, the identities of the parties are taken from the set of natural numbers (not strings), but this is only a syntactic difference. Secondly, the message T (denoted "M" in [17]) is computed as $T := C_3 \oplus H_4(\sigma)$, while we compute it as $T := (C_3^1 \oplus H_4(\sigma||0))||\cdots||C_3^m \oplus H_4(\sigma||m))$ (see Eq. (4)). This is ok, since a function $H(\sigma) := H_4(\sigma||0)||\cdots||H_4(\sigma||m)$ can be viewed as a random oracle, assuming that H_4 is a random oracle.

This implies that the transactions remain secret until eon's private key is reconstructed (this proves the "secrecy" of our scheme). "Non-malleability" follows from the fact that every IND-ID-CCA-secure scheme is non-malleable (see, e.g., [2]). Finally, "commitment" comes from the fact that no user can change the ciphertext once it is posted on the ledger.

4.3 Applications

The Shutter protocol can be used in different settings (Setting A, B, and C, which we describe below). In all cases, the keypers provide a public key, the users encrypt transactions for different eons, the sequencer [7] collects encrypted transactions in batches, and the keypers publish eon secret key shares once the batch for a certain eon is fixed.

Setting A: Traditional exchange In this setting, the sequencer is a stock exchange. Users are traders, and trades are executed in the order of batches once the batch is decrypted. The users are protected from front-running by an exchange or other traders.

Setting B: On-Chain sequencer Here, the sequencer is a smart contract. Users send on-chain transactions with encrypted payload to the sequencer contract. The encrypted transactions are first only scheduled for execution and the actual execution happens through another transaction when the decryption key becomes available. Transactions are front-running protected from other transactions passing through the sequencer (but not other on-chain transactions as they can front-run the decryption transaction), so the DEX that only accepts transactions coming from the sequencer is safe from frontrunners

Setting C: Layer 1 blockchain/sidechain/roll-up Here, the sequencer is the block production mechanism of the chain itself. All transactions in the chain are front-running protected.

4.4 Implementation

To demonstrate the practicality of the protocol, we built a production-ready implementation of Shutter in the on-chain setting. It is freely available as open source [25].

The code is written primarily in Go, with the exception of the smart contracts in Solidity and an exemplary web interface in HTML and JavaScript. We instantiated the Shutter protocol using optimal ate pairings on a 256-bit Barreto-Naehrig curve [24] using a library by Cloudflare [9]. Network messages are encoded using Go's gob package [3] and signed using ECDSA signatures as used in Ethereum [28].

[7] In this section, we denote the entity that determines the execution order of transactions by the sequencer.

The keyper nodes communicate with each other on a customized Tendermint [27] blockchain. The chain enables keypers to achieve consensus over which messages have been received and which have not. As a side-effect, this choice greatly reduces implementation complexity as the peer-to-peer networking code can be reused. The keypers act as validators of the Tendermint chain, ensuring their messages cannot be censored as long as at least ⅔ of them are honest. This introduces no additional security assumptions if Shutter's threshold parameter is chosen accordingly.

The implementation uses an Ethereum-compatible blockchain for transaction sequencing and execution. We divide the sequence of blocks on this chain in eons of configurable length. To each eon, an eon key pair is assigned and the keypers generate the corresponding decryption key once the eon's end block is reached.

During its life-cycle, Shutter transactions pass through three smart contracts deployed on the underlying chain: The batcher contract, the executor contract, and the target contract. To send a transaction, users first pick an eon and encrypt their payload with the corresponding encryption key. They then send the ciphertext wrapped in a standard Ethereum transaction (including some metadata) to the batcher contract. Here, its arrival is logged if the batch does not already exceed a size limit and the corresponding eon has neither already ended nor is too far into the future.

Whenever an eon ends, the keypers will publish their share of the corresponding decryption key on the Tendermint chain. They also listen for shares of their peers and, once they have acquired enough of them, combine them to get the decryption key. They then decrypt all transactions that have been submitted to the batcher contract for this eon, sign the result, and publish the signature as a vote on the Tendermint chain. One keyper is selected to submit the decrypted transactions to the executor contract for execution. If they are unavailable, they are replaced by the next in line after a timeout, and so on.

The executor contract automatically passes the decrypted transactions to the target contract, which stands for the actual application using the system (e.g., a decentralized exchange). It is at liberty how to interpret the data, but in most cases will do some form of authentication as well as replay protection.

In case a keyper submits a wrong result, they can be challenged to produce votes of their peers showing that they acted on their behalf. If they are unable to, they can be punished by freezing a deposit they were required to make earlier. This form of verifying signatures "optimistically", i.e., only in case of misbehavior, is more efficient than doing it in every single execution step and is an acceptable security tradeoff in many settings.

Furthermore, in an ideal setup, the keypers would only have to provide key shares and not be involved in the decryption process at all, leaving it entirely to the underlying blockchain. However, currently the resulting high gas costs make this infeasible. Proposed changes to the Ethereum Virtual Machine [1] will likely change this.

4.5 Evaluation

We deployed our implementation to the Ethereum Goerli testnet with 21 keypers that we hosted, a threshold of 7, and an eon length of 10 blocks (2.5 minutes). A publicly available web interface allows anyone to submit transactions. The system has been operational from April to August 2021. It processed a total of 344 submitted transactions and generated about 100000 eon decryption keys.

There were brief periods of downtime due to maintenance, and on two occasions, keyper nodes ran out of funds needed to pay transaction fees. After resupplying the nodes and a brief catch-up period, the system returned to functioning normally. In addition to the public deployment, we performed a set of benchmarks in order to be able to quantify the performance of the protocol and the implementation. The examined properties are the efficiency of the contracts as well as the key generation protocol during both setup and operational phases.

Gas is the unit in which Ethereum-compatible blockchains measure the resources a transaction consumes. It translates directly to the fee users have to pay; thus, applications have to be gas-efficient to be practical. In our implementation of the Shutter protocol in the on-chain setting, gas costs arise mostly in two places: The batcher contract when a user submits an encrypted transaction and the executor contract when a keyper executes a batch of now decrypted transactions.

Fig. 8 shows the gas usage per transaction for batches of different size, assuming each transaction has a size of 100 bytes. The cost of adding a transaction to a batch is constant at about 80000 units. The cost of executing a batch increases with the number of transactions in it, but does so sublinearly so that the per-transaction cost quickly becomes negligible and the batching cost dominates.

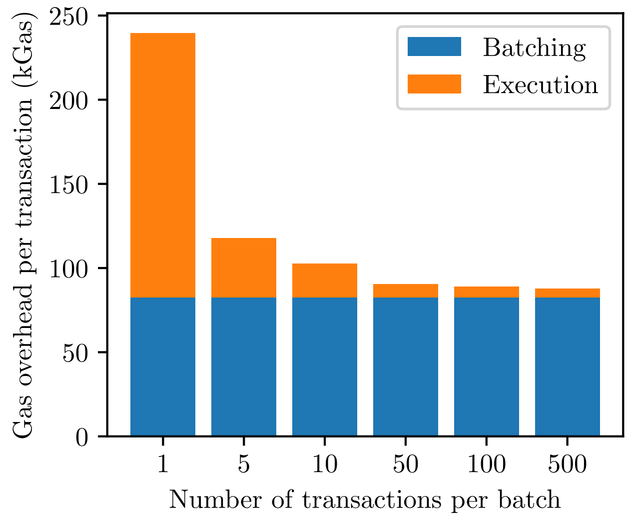


Fig. 8. Gas usage of batching and execution for different batch sizes, assuming a constant transaction size of 100 bytes.

The network traffic during the setup phase is examined in Fig. 9. It grows quadratically with the size of the keyper set. The contribution of by messages sent directly between nodes outweighs the broadcast, but the difference becomes only significant for large networks. In total, the resulting bandwidth requirements are reasonable even for consumer-grade network connections.

The last property of interest is the latency of eon key generation. The results are shown in Fig. 10. Subsecond latencies with an average of ca. $0.4s$ are achieved, with no noticeable dependence on the size of the keyper set, demonstrating the protocol's scalability.

References

1. Eip-2537: Precompile for bls12-381 curve operations. Available: https://eips.ethereum.org/EIPS/eip-2537., February 2020. Ethereum Improvement Proposals, no. 2537,.

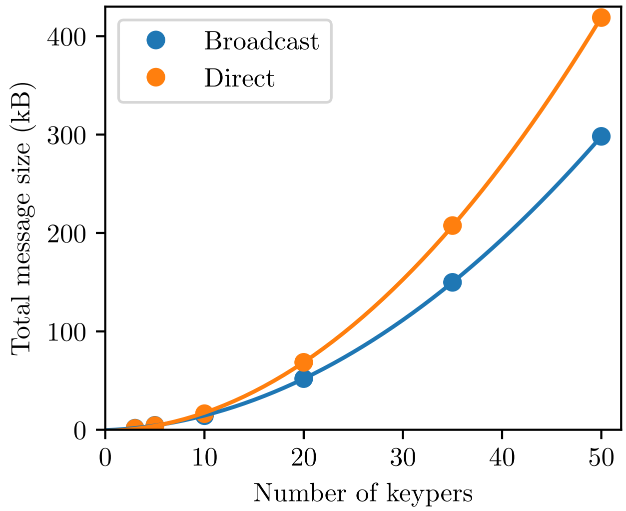


Fig. 9. Network traffic during setup phase with a quadratic fit. Shown is the sum of the sizes of all DKG messages, broken up into those sent between individual peers ("direct") and those addressed all of them ("broadcast").

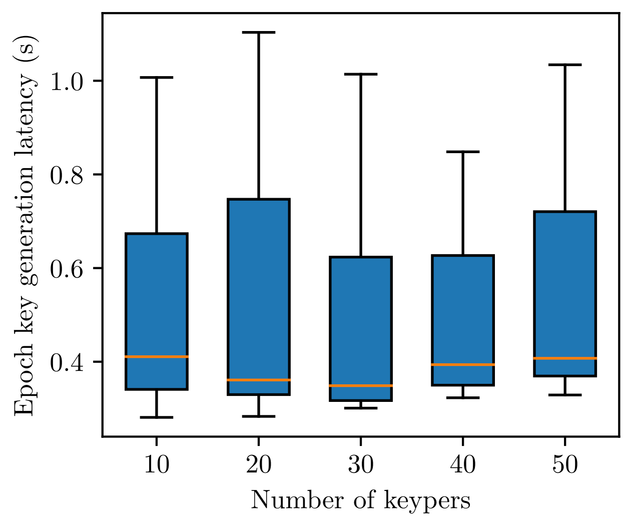


Fig. 10. Latency of eon key generation. The orange line marks the median of each measurement. The box encompasses the inter quartile range, and the whiskers all measured values excluding outliers.

2. Nuttapong Attrapadung, Yang Cui, David Galindo, Goichiro Hanaoka, Ichiro Hasuo, Hideki Imai, Kanta Matsuura, Peng Yang, and Rui Zhang. Relations among notions of security for identity based encryption schemes. In José R. Correa, Alejandro Hevia, and Marcos A. Kiwi, editors, *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings*, volume 3887 of *Lecture Notes in Computer Science*, pages 130–141. Springer, 2006.
3. The Go Authors. Go gob package. Available: https://pkg.go.dev/encoding, August 2021.
4. Leemon Baird, Pratyay Mukherjee, and Rohit Sinha. i-tire: Incremental timed-release encryption or how to use timed-release encryption on blockchains? In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 235–248. ACM, 2022.
5. Carsten Baum, Bernardo David, and Tore Kasper Frederiksen. P2DEX: privacy-preserving decentralized cryptocurrency exchange. In Kazue Sako and Nils Ole Tippenhauer, editors, *Applied Cryptography and Network Security - 19th International Conference, ACNS 2021, Kamakura, Japan, June 21-24, 2021, Proceedings, Part I*, volume 12726 of *Lecture Notes in Computer Science*, pages 163–194. Springer, 2021.
6. Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. *SIAM J. Comput.*, 32(3):586–615, 2003.
7. Vitalik Buterin and Christian Reitwiessner. Eip-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. Available: https://eips.ethereum.org/EIPS/eip-197., February 2017. Ethereum Improvement Proposals, no. 197,.
8. Arka Rai Choudhuri, Sanjam Garg, Julien Piet, and Guru-Vamsi Policharla. Mempool privacy via batched threshold encryption: Attacks and defenses, 2023. manuscript.
9. Cloudflare. bn256. Available: https://github.com/cloudflare/bn256., January 2017.
10. Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 910–927. IEEE, 2020.
11. Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM J. Comput.*, 30(2):391–437, 2000.
12. Nico Döttling, Lucjan Hanzlik, Bernardo Magri, and Stella Wohnig. Mcfly: Verifiable encryption to the future made practical. *IACR Cryptol. ePrint Arch.*, page 433, 2022.
13. Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 427–437. IEEE Computer Society, 1987.
14. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptol.*, 20(1):51–83, 2007.
15. Antoine Joux. A one round protocol for tripartite diffie-hellman. *J. Cryptol.*, 17(4):263–276, 2004.
16. Fredrik Kamphuis, Bernardo Magri, Ricky Lamberty, and Sebastian Faust. Revisiting transaction ledger robustness in the miner extractable value era. In Mehdi Tibouchi and Xiaofeng Wang, editors, *Applied Cryptography and Network Security - 21st International Conference, ACNS 2023, Kyoto, Japan, June 19-22, 2023, Proceedings, Part II*, volume 13906 of *Lecture Notes in Computer Science*, pages 675–698. Springer, 2023.
17. Aniket Kate and Ian Goldberg. Distributed private-key generators for identity-based cryptography. In Juan A. Garay and Roberto De Prisco, editors, *Security and Cryptography for Networks, 7th International Conference, SCN 2010, Amalfi, Italy, September 13-15, 2010. Proceedings*, volume 6280 of *Lecture Notes in Computer Science*, pages 436–453. Springer, 2010. Extended version available at https://eprint.iacr.org/2009/355.
18. Aniket Kate, Yizhou Huang, and Ian Goldberg. Distributed key generation in the wild. *IACR Cryptol. ePrint Arch.*, 2012:377, 2012.
19. Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2010.
20. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* CRC Press, 2014.
21. Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 451–480. Springer, 2020.
22. Flashbots Ltd. Flashbots. Available: https://www.flashbots.net/., February 2023.

23. Peyman Momeni, Sergey Gorbunov, and Bohan Zhang. Fairblock: Preventing blockchain front-running with minimal overheads. *IACR Cryptol. ePrint Arch.*, page 1066, 2022.
24. Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *Progress in Cryptology–LATINCRYPT 2010: First International Conference on Cryptology and Information Security in Latin America, Puebla, Mexico, August 8-11, 2010, proceedings 1*, pages 109–123. Springer, 2010.
25. Shutter Network. Shutter github repository. Available: https://github.com/shutter-network/shutter., August 2020.
26. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
27. Tendermint. Tendermint v0.34.0. Available: https://github.com/tendermint/tendermint/releases/tag/v0.34.0., November 2020.
28. Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
29. Sen Yang, Fan Zhang, Ken Huang, Xi Chen, Youwei Yang, and Feng Zhu. Sok: Mev countermeasures: Theory and practice. December 2022.

A Background

In this section, for the sake of completeness, we provide some background on standard cryptographic tools.

A.1 Shamir's secret sharing

Shamir's (t, n)-*threshold secret sharing scheme over* $\mathbb{Z}_q$ is a pair of functions ($\mathsf{share} : \mathbb{Z}_q \to \mathbb{Z}_q^n, \mathsf{reconstruct} : \mathbb{Z}_q^{t+1} \to \mathbb{Z}_q$) [26]. Function $\mathsf{share} : \mathbb{Z}_q \to \mathbb{Z}_q^n$ is a function defined as:

$$\mathsf{share}(x) := (y_1, \ldots, y_n),$$

where

$$(y_1, \ldots, y_n) := (\varphi(x_1), \ldots, \varphi(x_n)) \text{ and } \varphi \text{ is a random polynomial of degree} \leq t \text{ such that } \varphi(0) = x. \tag{8}$$

Function $\mathsf{reconstruct}$ takes as input a set $\{\mathsf{K}_{j_1}, \ldots, \mathsf{K}_{j_{t+1}}\}$ keypers (of size $t + 1$) and a sequence $(y_{j_1}, \ldots, y_{j_{t+1}})$ (where each y_j is defined in Eq. 8), and outputs x computed by interpolating the polynomial φ in point 0. More precisely, we compute x as the following linear combination of the y_j's.

$$x = \lambda_1 y_1 + \cdots \lambda_{t+1} y_{t+1}, \tag{9}$$

where the λ_j's are the Lagrange coefficients.

B Proofs of lemmas from the main body

Proof (Proof of Lemma 2). We have

$$\begin{aligned} \hat{e}(Q_j^{\mathsf{Id}}, P_2) &= \hat{e}(\mathsf{msk}_j \times H_1(i), P_2) && (11) \\ &= \hat{e}(H_1(i), \mathsf{msk}_j \times P_2) && (12) \\ &= \hat{e}(H_1(i), \mathsf{mpk}_j), && (13) \end{aligned}$$

where Eqs. (11) and (13) follow from the fact that K_j is honest, and in (12) we used the blinearity of $\hat{e}$.

FeldmanVSS(D, s)

1. The dealer D select a random polynomial $\varphi(x) = \sum_{j=0}^{t} c_i \cdot x^j$ over $\mathbb{Z}_q$ of degree at most t such that $\varphi(0) = s$.
2. The dealer sends to each $\mathsf{K}_i \in \{\mathsf{K}_1, \ldots, \mathsf{K}_n\}$ the value $s_i := \varphi(x_i) \in \mathbb{Z}_q$.
3. The dealer broadcasts to all keypers the sequence

$$(\gamma_0, \ldots, \gamma_t) := (c_0 \times P_2, \ldots, c_t \times P_2) \in \mathbb{G}_2^{t+1}$$

   (recall that P_2 is the generator of $\mathbb{G}_2$).
   If a correctly formatted sequence is not broadcast then each keyper decides that D is corrupt and ends the protocol with private output $0 \in \mathbb{Z}_q$ and public output $(0, \ldots, 0) \in \mathbb{G}_2^n$. Otherwise proceed to the next step.
4. Every keyper calculates a sequence $(\pi_0, \ldots, \pi_n) \in \mathbb{G}_2^n$ where each π_i is computed as

$$\pi_i := \sum_{j=0}^{t} (x_i^j \bmod q) \times \gamma_j$$

   (recall that we assumed that $x_0 = 0$)
5. For each $\mathsf{K}_i \in \{\mathsf{K}_1, \ldots, \mathsf{K}_n\}$ execute the following in parallel
   (a) K_i checks if

$$\pi_i = s_i \times P_2.$$

   If this check passes then K_i sets his private output s_i and public output to $(\pi_0, \ldots, \pi_n)$. Normally this will be her output at the end of this protocol. However, if the dealer is corrupt, this output can still change (see below), hence K_i does not end the protocol yet.
   If $\pi_i \neq s_i \times P_2$ (or if K_i did not receive correctly formatted messages from D) then K_i broadcast an accusation against D to all the keypers (e.g. this can be a special message (accuse, D)).
   (b) D has to reply to this accusation by broadcasting to all the keypers the value s_i (that she sent to K_i is Step 2).
   We now do the following.
      i. If D sends a correctly formatted value s_i such that $\pi_i = s_i \times P_2$ then K_i sets his private output to s_i and public output to $(\pi_0, \ldots, \pi_n)$.
      ii. Otherwise each keyper K_j assumes that D is corrupt and ends the protocol with private output $0 \in \mathbb{Z}_q$ and public output $(0, \ldots, 0) \in \mathbb{G}_2^n$.

Fig. 11. Feldman Verifiable Secret Sharing procedure in which a *dealer* $D \in \{\mathsf{K}_1, \ldots, \mathsf{K}_n\}$ shares her *secret* $s \in Z_q$.

Proof (Proof of Lemma 3). First, observe that by Lemma 2 we get that the protocol never halts. We have that σ computed while calculating (C_1, C_2, C_3) is equal to $(\hat{e}(H_1(i), \mathsf{mpk}))^r$ and σ computed in the corresponding opening phase is equal to $\hat{e}(\mathsf{sk}^{(i)}, C_1)$. Hence, what remains is to show the following.

$$\hat{e}(\mathsf{sk}^{(i)}, C_1) = (\hat{e}(H_1(i), \mathsf{mpk}))^r \,. \tag{14}$$

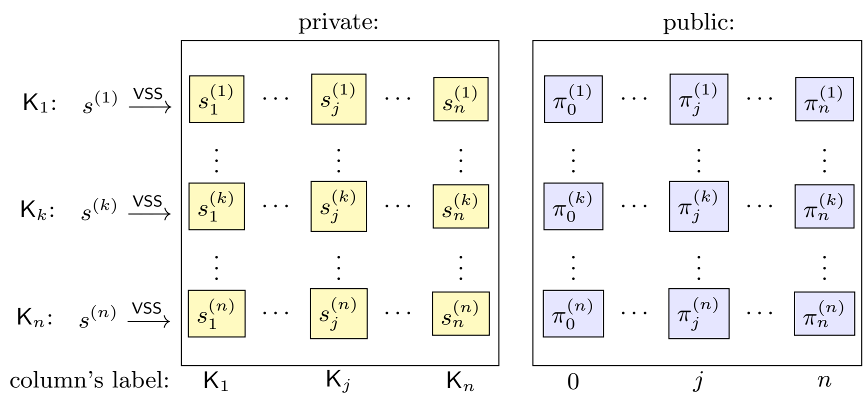
We show Eq. (14) as follows (below, λ_i's are Lagrange coefficients, see Sec. 3.4):

$$\begin{aligned} & \hat{e}(\mathsf{sk}^{(i)}, C_1) \\ &= \hat{e}(\mathsf{sk}^{(i)}, r \times P_2) && (15) \\ &= \hat{e}(\lambda_1 \times Q^{\mathsf{Id}}_{\ell_1} + \ldots + \lambda_{t+1} \times Q^{\mathsf{Id}}_{\ell_{t+1}}, r \times P_2) && (16) \\ &= \hat{e}(\lambda_1 \times Q^{\mathsf{Id}}_{\ell_1}, r \times P_2) \cdot \cdots \cdot \hat{e}(\lambda_{t+1} \times Q^{\mathsf{Id}}_{\ell_{t+1}}, && (17) \end{aligned}$$

Distributed key generation $\mathsf{Setup}(1^\kappa, n)$

1. For each $\mathsf{K}_j \in \{\mathsf{K}_1, \ldots, \mathsf{K}_n\}$ do the following in parallel:
   (a) Choose $s^{(k)} \leftarrow\$ Z_q$.
   (b) Execute $\mathsf{FeldmanVSS}(\mathsf{K}_k, s^{(k)})$ (see Fig. 11, page 18). For each $\mathsf{K}_i \in \{\mathsf{K}_1, \ldots, \mathsf{K}_n\}$ let $s_i^{(k)}$ be the private output of K_i and let $(\pi_0^{(k)}, \ldots, \pi_n^{(k)})$ be the public output.

   Note that we now have a quadratic number of variables, that can be depicted as follows.



   where each keyper K_j receives values that are in her "s" (yellow) column, and the "π" (blue) values are public.
2. Once all the parallel executions above are finished, each K_j computes her private output msk_j of as

$$\mathsf{msk}_j := s_j^{(1)} + \cdots + s_j^{(n)} \bmod q$$

   (note that on the picture above this corresponds to summing the values in the K_j's yellow column). The public output of each keyper is equal to $(\mathsf{mpk}_1, \ldots, \mathsf{mpk}_n) \in \mathbb{G}_2^n$ and $\mathsf{mpk} \in \mathbb{G}_2$, where each mpk_j is computed as:

$$\mathsf{mpk}_j := \pi_j^{(1)} + \cdots + \pi_j^{(n)}$$

   (i.e.: it is the sum of the values in jth blue column), and mpk is calculated as:

$$\mathsf{mpk} := \pi_0^{(1)} + \cdots + \pi_0^{(n)}.$$

Fig. 12. Distributed Key Generation protocol. Above 1^κ is the security parameter and n is the number of keypers.

$\mathsf{DistrExtract}(\mathsf{Id})$ for a party P_{Id} where $\mathsf{Id} \in \{0,1\}^*$

Each keyper K_j proceeds as follows:

1. K_j sends to P_{Id} a value $Q_j^{\mathsf{Id}} := \mathsf{msk}_j \times H_1(i)$.
2. P_{Id} waits to receive the "Q" values from the other keypers. For each such value Q_k^{Id} received from K_k the party P_{Id} checks if it satisfies the following equation:

$$\hat{e}(Q_k^{\mathsf{Id}}, P_2) = \hat{e}(H_1(i), \mathsf{mpk}_k).$$

P_{Id} stops waiting once she receives $t+1$ values that satisfy the above equation. Assume that the first $t+1$ values Q_j^{Id}'s that satisfied the above equation check are $Q_{\ell_1}^{\mathsf{Id}}, \ldots, Q_{\ell_{t+1}}^{\mathsf{Id}}$. After receiving them P_{Id} computes $\mathsf{sk}_j^{\mathsf{Id}}$ as

$$\mathsf{sk}_j^{\mathsf{Id}} := \lambda_{\ell_1} Q_{\ell_1}^{\mathsf{Id}} + \ldots + \lambda_{\ell_{t+1}} Q_{\ell_{t+1}}^{\mathsf{Id}}. \tag{10}$$

(where the λ_j's are the Lagrange coefficients).

Fig. 13. Extraction of a secret key for a party P_{Id} with identifier Id .

$$\begin{aligned}
& r \times P_2) && (18)\\
&= \hat{e}(\lambda_1 H_1(i), r \times \mathsf{mpk}_1) \cdot \cdots \cdot && (19)\\
& \hat{e}(\lambda_{t+1} H_1(i), r \times \mathsf{mpk}_j) && (20)\\
&= \hat{e}(\lambda_1 H_1(i), r \times \mathsf{msk}_1 \times P_2) \cdot \cdots \cdot \hat{e}(\lambda_{t+1} H_1(i), && (21)\\
& r \times \mathsf{msk}_{t+1} \times P_2) && (22)\\
&= \hat{e}(\mathsf{msk}_1 \times \lambda_1 \times H_1(i), r \times P_2) \cdot \cdots \cdot && (23)\\
& \hat{e}(\mathsf{msk}_{t+1} \times \lambda_{t+1} \times H_1(i), r \times P_2) && (24)\\
&= \hat{e}((\mathsf{msk}_1 \times \lambda_1 + \cdots + \mathsf{msk}_{t+1} \times \lambda_{t+1}) && (25)\\
& H_1(i), r \times P_2)) && (26)\\
&= \hat{e}(\mathsf{msk} \times H_1(i), r \times P_2)) && (27)\\
&= (\hat{e}(H_1(i), \mathsf{msk} \times P_2)))^r && (28)\\
&= (\hat{e}(H_1(i), \mathsf{mpk}))^r, && (29)
\end{aligned}$$

where we use bilinearity of $\hat{e}$ in Eqs. (18), (24), (26), and (28). In Eq. (15) we used the fact that $C_1 = rP_2$ (see Step 2 on Fig. 5). In Eq. (16) we used the formula from Eq. (5) for $\mathsf{sk}^{(i)}$. In Eq. (20) we used Lemma 2, and in (22) — the fact that $\mathsf{mpk}_j = \mathsf{msk}_j \times P_2$. Eq. (27) follows from the fact that λ_i's are Lagrange coefficients for this secret sharing scheme (cf. Eq. (9))). Finally Eq. (29) follows from the fact that $\mathsf{mpk} = \mathsf{msk} \times P_2$ (see Lemma 1).