ETHTID: Deployable Threshold Information Disclosure on Ethereum

Oliver Stengele, Markus Raiber, Jörn Müller-Quade, Hannes Hartenstein
Institute of Information Security and Dependability (KASTEL)
Karlsruhe Institute of Technology
Karlsruhe, Germany
{oliver.stengele, markus.raiber, joern.mueller-quade, hannes.hartenstein}@kit.edu

***Abstract*—We address the Threshold Information Disclosure (TID) problem on Ethereum: An arbitrary number of users commit to the scheduled disclosure of their individual messages recorded on the Ethereum blockchain if and only if all such messages are disclosed. Before a disclosure, only the original sender of each message should know its contents. To accomplish this, we task a small council with executing a distributed generation and threshold sharing of an asymmetric key pair. The public key can be used to encrypt messages which only become readable once the threshold-shared decryption key is reconstructed at a predefined point in time and recorded on-chain. With blockchains like Ethereum, it is possible to coordinate such procedures and attach economic stakes to the actions of participating individuals. In this paper, we present ETHTID, an Ethereum smart contract application to coordinate Threshold Information Disclosure. We base our implementation on an existing smart contract, ETHDKG, and optimise functionality and costs to fit the TID use case. While optimising for cost savings, we show that the security of the underlying cryptographic scheme is still maintained. We evaluate how the execution costs depend on the size of the council and the threshold and show that the presented protocol is deployable with a council of more than 200 members with gas savings of 20–40% compared to ETHDKG.**

***Index Terms*—distributed key generation, threshold encryption, smart contract, time-lock cryptography**

I. Introduction

In this paper, we make use of threshold cryptography to achieve *Threshold Information Disclosure* (TID) on Ethereum where mutual trust cannot be assumed and protocol violations need to be discouraged. Threshold Information Disclosure provides a way for arbitrarily many users to coordinate the disclosure of individually-held pieces of information: Each user commits to the disclosure of their information if and only if the information of all other users is disclosed as well. Before such a disclosure, each user should only be privy to their own piece of information. Such a functionality is useful to allow users to generate and record their submissions independently from each other with the assurance that no user can prevent the disclosure of any particular submission. Applications where independent submissions are useful include scientific evaluation of experimental raw data, security audits of software by independent groups, and sealed-bid auctions where the secrecy of losing bids is not required.

This work was supported by funding of the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL).

To accomplish such a coordinated information disclosure, an asymmetric key pair should be generated in such a way that a public encryption key becomes available immediately, but the corresponding decryption key is threshold-shared among a council that is tasked with recovering and publishing said key at a predefined point in the future. Submissions can then be encrypted and published in the present but only become collectively readable once the decryption key is published. In the meantime, neither submitters nor individual members of the council can decrypt any submissions.

The functionality described above is challenging to achieve on public blockchain systems like Ethereum due to a core aspect of their functionality: It is vital that user-generated information spreads quickly through their global peer-to-peer networks in order to be recorded on their corresponding blockchains. This very requirement makes it difficult for multiple parties to coordinate the release of information: One party has to go first without knowledge of subsequent submissions and other parties can wait, observe the submissions of others, and potentially change their submission accordingly. Individual "commit and reveal" schemes exist, e.g. based on a cryptographic hash function, to nullify any ordering advantages, but they introduce the possibility of a party refusing to reveal their commitment, which may be unacceptable depending on the use case. In order to enable an arbitrarily large number of parties to commit to the coordinated public release of submissions to a blockchain, delegation mechanisms for both, keeping a secret as well as coordination of key management, are therefore needed. To preserve the notion of decentralisation inherent in public blockchain systems, such mechanisms can not rely on any trusted third party or centralised coordinator.

With ETHDKG, Schindler et al. [1], [2] demonstrated that a distributed key pair generation for the BLS signature scheme [3] can be coordinated and recorded with an Ethereum smart contract. The core of their construction is the well-known distributed key generation by Feldman [4]. The TID use case differs in two significant ways: First, the asymmetric key pair is *only* intended for the encryption of user-defined information to keep it secret until disclosure; and second, one explicitly requires a scheduled reconstruction of the threshold-shared secret key to facilitate this disclosure. Based on the TID use case and the corresponding cryptographic schemes, our goal is to optimise ETHDKG to save on deployment and execution

costs while extending it with the functionality necessary to orchestrate the scheduled key reconstruction. In particular, we show that biasing attacks as described by Gennaro et al. [5], [6], against which ETHDKG employs a countermeasure by Neji et al. [7], are of no concern in our case and we can therefore simplify the overall protocol to save costs.

The main contributions of this paper are as follows:

- An analysis of the coordination efforts for distributed key generation, verifiable secret sharing, and scheduled reconstruction to achieve Threshold Information Disclosure.
- A use-case oriented examination of cryptographic schemes with the goal of minimising deployment and execution costs of the implementation.
- An argumentation that our main optimisation maintains the security of the underlying cryptographic scheme.
- ETHTID, an Ethereum smart contract implementation to achieve Threshold Information Disclosure.
- An evaluation of ETHTID with regard to deployment and execution costs to determine its deployability and limits.

An extended version of this paper is available on arXiv [8]. The remainder of this paper is structured as follows: We derive general requirements for a decentralised TID coordinator and give a system overview in Section II before we address related work in Section III. In Section IV, we review the essential building blocks we make use of and give a brief rationale for their selection before we describe our implementation in Section V. We thoroughly evaluate the implementation in Section VI and discuss the results and open issues in Section VII. With Section VIII, we conclude the paper.

II. Overview and Problem Statement

For keeping a secret in a decentralised fashion, we assume a *council* of n members tasked with generating a master public key *mpk* such that the corresponding secret key *msk* can be recovered after a predefined point in time r through the cooperation of council members. In particular, the following conditions need to be fulfilled: The master secret key *msk* is not available to any party before time r but eventually becomes available to all parties (after time r). Note that both conditions refer to parties both actively participating in the generation, sharing, and reconstruction phases (i.e. council members) as well as user parties that use *mpk* and *msk*. Whether or not these conditions hold, however, depends entirely on the behaviour of council members. Thus, the trust assumption of TID is that the council does not reconstruct the master secret key *msk* before time r. An incentive scheme like the one presented by Yakira et al. [9], coupled with the assumption that council members act rationally, can encourage correct behaviour.

In broad strokes, TID proceeds as follows: The council performs a distributed key generation of *msk* with each council member contributing an individual secret, making *mpk* available in the process. At the predefined time for disclosure, council members cooperate to recover and publish *msk*. Users of TID can publish their submissions after encrypting them with *mpk* and they will become readable once *msk* is published. To avoid single points of failure, we require each council member to share their individual secret through a verifiable secret sharing scheme (VSS) [4], [10] based on threshold cryptography, so the secret can be recovered by fewer than n council members. Therefore, council members have to broadcast and store certain pieces of information to hold each other accountable. Thus, in addition to performing any necessary verification to arbitrate disputes, a coordination functionality is needed to fulfil the following requirements:

1) The coordination functionality stores and provides data from council members, enabling them to execute the VSS protocol and detect misbehaviour.
2) The coordination functionality performs verification given previously stored data and submitted evidence.
3) The coordination functionality receives, holds, and redistributes assets according to the outcome of verification.
4) The coordination functionality has access to a public form of timekeeping to enforce a schedule.

The coordination functionality of TID should be implemented in a decentralised fashion, i.e. as a blockchain-based smart contract. In addition to the ability to perform verification and distribute assets accordingly, these blockchain-based systems include a timekeeping mechanism in the form of block height.

With the general structure and requirements in mind, we can now outline a system overview as well as the corresponding workflow. Figure 1 shows the parties and their interactions.

Setup and registration phase: An *initiator* triggers the entire procedure. While instantiating the smart contract, the initiator sets parameters, provides a financial incentive for participation, and sets a schedule for the remaining phases. Council members register with the smart contract (selection of council member is discussed in Section VII).

Share distribution and dispute phase: Once authorised, the council members perform a distributed key generation and submit data to the contract for mutual accountability. This data simultaneously protects members against false accusations

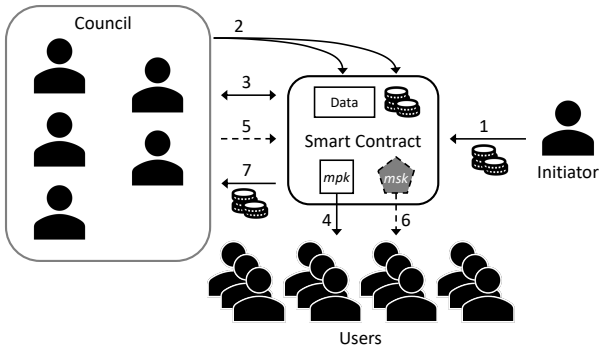

Figure 1. System overview. (1) Initiator deploys smart contract with parameters and incentives. (2) Council members register with the contract by submitting data and a deposit. (3) Council members communicate through the smart contract to generate the public encryption key *mpk* and establish a sharing of the secret decryption key *msk*. In case of misbehaviour, members submit a dispute to the contract to enforce punishments. (4) Users can obtain *mpk*. (5) At a codified time, council members submit data to the contract to enable the reconstruction of *msk*. (6) Users and the general public can obtain the decryption key *msk*. (7) Based on their behaviour, council members receive a reward in addition to their collateral and are released from their obligation.

and convicts them in conjunction with submitted evidence in case of misbehaviour. Alongside this verification data, council members may also need to send a security deposit to the contract that can be destroyed or redistributed in case they misbehave. Once all council members are registered with the contract, they can privately exchange data through the smart contract to establish a sharing of *msk*. Council members that remain inactive after their registration can be disqualified based on the codified schedule. Similarly, members who perform an invalid sharing of their secret are caught by the VSS scheme and can be disqualified after the smart contract verifies submitted evidence on-demand.

Reconstruction phase: Once the predefined time for reconstruction r has come, members of the council are incentivised to submit data necessary for the reconstruction of *msk* to the contract. If enough members cooperate, *msk* can be recovered via the previously established secret sharing. A member can then post *msk* to the contract and both users and the general public can obtain it to decrypt all previously published messages, thus achieving a coordinated disclosure. Lastly, the members of the council can reclaim their deposit in addition to their share of the reward for their service.

The problem addressed in this paper is the selection and implementation of cryptographic primitives as well as the implementation of the coordination functionality in Ethereum such that TID becomes actually deployable with respect to costs and block sizes.

III. Related Work

Coordinating the release of information has previously been tackled in two distinct ways, either through time-lock puzzles or with trusted third party custodians in either centralised or decentralised fashion. When comparing the work presented here with previous approaches, it is helpful to consider the number of "sending" parties that can commit to the timed release of information without disclosing it to each other and to how many other "receiving" parties the information is disclosed to.

Time-lock puzzles [11]–[13] bear similarities to the TID use case in that they can "send information into the future". The receiver of a puzzle is required to do some inefficient but feasible computation that is expected to take at least a certain amount of time to solve it. The time it takes a recipient to solve the puzzle and recover the decryption key can only be estimated by the sender. Additionally, the recipient must exert computational effort to eventually receive the encrypted information. These schemes generally constitute a "one-sender-to-one-recipient" timed release functionality with low timing accuracy but without a trusted third party.

Another solution to timed-release encryption makes use of a trusted party that releases decryption keys at the right time [11], [14]–[16]. To circumvent a single point of failure that a trusted party poses, it is possible to securely distribute the task to multiple parties, such that it is sufficient that some amount of them behave honestly. Some of these works allow for multiple recipients with more or less favourable scalability or setup procedures, but lacking a distributed key setup they only support a timed release for individual senders.

Benhamouda et al. [17] present a concept that allows the Algorand blockchain itself to act as a long-term secret keeper through a randomised sequence of anonymous committees. They list functionalities like the one we present here as future achievable goals of their concept. While their approach is more general and possibly covers the TID functionality, our approach is practically feasible on an established blockchain. However, their "cryptographic sortition" approach to forming committees may be transferable to the TID concept as a countermeasure to Sybil attacks. It will be interesting to see the future developments of Algorand and its capabilities.

In 2018, two projects appeared independently of each other within the Ethereum ecosystem that combined threshold secret sharing with Ethereum smart contracts: Kill Cord[1] and Kimono[2]. Both use Shamir's secret sharing [18] to fragment a decryption key and entrust the shares to Ethereum nodes which are incentivised via smart contracts to only post their shares at a certain time. In both cases, the initiator of the protocol generates the key and acts as a trusted dealer, making both systems "one-to-many" compared to the TID construction which achieves a coordinated "many-to-many" release.

IV. Essential Building Blocks

When implementing the coordination functionality as a smart contract on a public blockchain, it is important to minimise operational costs. These are (1) broadcast information which may be stored on the blockchain entirely or partially and (2) computations performed by the smart contract, mostly during optional verification. The obvious candidates for encryption schemes are the ElGamal and RSA cryptosystems. We chose ElGamal since it can be used over elliptic curves, thus achieving short public and secret keys and more importantly, it is easy to generate a distributed public key for it. Accordingly, we base our work on the distributed key generation protocol for ElGamal due to Feldman [4], which itself is based on the threshold secret sharing scheme by Shamir [18]. It is important to note that, while we actually employ this primitive on an elliptic curve, we will use the more common notation of multiplicative groups here.

A. Distributed Key Generation and Threshold Secret Sharing

Our goal is for the council to generate an ElGamal public key for which the corresponding secret key is $(t+1, n)$-secret shared among them. We first briefly recall Shamir's threshold secret sharing scheme [18]: To share a secret s, a random polynomial p of degree t with $p(0) = s$ is drawn and each party i is given $p(i)$ as its share. Since p has degree t, it is possible to uniquely reconstruct p, and thus find s, given $t+1$ such shares, but t shares reveal no information at all about s. Handling data related to these polynomials is at the very core of the TID coordinator.

[1] https://github.com/nomasters/killcord

[2] https://medium.com/@pfh/kimono-trustless-secret-sharing-using-time-locks-on-ethereum-8e7e696494d

To generate a threshold shared ElGamal key, we make use of a variant of the Joint-Feldman DKG [4] similar to the one by Pedersen [19], which we briefly recall. One important difference between these VSS protocols in theory and our practical implementation is the use of private communication channels between members. In order for the TID coordinator to efficiently arbitrate disputes, all communication between members must flow through it. We therefore adopt from ETHDKG [1] a symmetrical encryption scheme for the confidential exchange of information between members and a non-interactive zero-knowledge proof and verification scheme as a part of dispute resolution. The end result is that a dispute can be completed non-interactively in a single transaction.

We can now review the cryptographic procedures at the core of the TID implementation. Let $\mathbb{G}$ be a cyclic group of prime order p with generator g for which the decisional Diffie-Hellman problem is hard. It is easy to generate an ElGamal public key where each member holds an additive share of the secret key and all n council members have to cooperate to reconstruct it: Each party i chooses $sk_i \in \mathbb{Z}_p$, calculates $pk_i \coloneqq g^{sk_i}$ and broadcasts pk_i. The master public key is then $mpk \coloneqq \prod_{i=1}^{n} pk_i = g^{\sum_{i=1}^{n} sk_i}$ and the secret key $msk \coloneqq \sum_{i=1}^{n} sk_i$ is (n, n)-secret shared among all members. This is quite beneficial as it allows executing the distributed key generation and threshold sharing simultaneously. With this construction, it is also possible to disqualify members for misbehaviour and simply discard their pieces of *msk* and *mpk* as outlined below.

To achieve a $(t+1, n)$-threshold secret sharing of the (not yet reconstructed) master secret key *msk*, each member individually $(t+1, n)$-shares its contribution sk_i among all n council members using Shamir secret sharing. We will use the term *shadow* for shares of sk_i, while reserving the term *share* for shares of the master secret key *msk*. Since Shamir secret sharing is additively homomorphic, each member can combine all valid shadows that they received into a single share of the master secret key. To prevent malicious members from sending inconsistent shadows during this phase, Joint-Feldman [4] employs the following consistency checks: When members pick a random polynomial $p_i = sk_i + \sum_{k=1}^{t} a_{i,k} X^k$, they also compute verification values $A_{i,k} = g^{a_{i,k}}$ and broadcast $\{A_{i,k}\}_{k=1}^{t}$ in addition to pk_i. They privately send to member j the shadow $u_{i,j} = p_i(j)$ and treat $p_i(i)$ as their own shadow. When member j receives a shadow $u_{i,j}$ from member i, they check whether it agrees with the broadcast polynomial:

$$g^{u_{i,j}} \stackrel{?}{=} pk_i \prod_{k=1}^{t} A_{i,k}^{j^k} = g^{sk_i + \sum_{k=1}^{t} a_{i,k} j^k} = g^{p_i(j)} \qquad (1)$$

If this check fails, member j reveals the invalid shadow $u_{i,j}$ to the coordinator and member i is excluded if the complaint is valid. Notice that the coordinator needs access to the verification values pk_i and $\{A_{i,k}\}_{k=1}^{t}$ of the accused member i in order to evaluate Equation (1) as part of a dispute. Rather than storing these values on-chain, we adopt the strategy of ETHDKG to store a hash of the previous broadcast and let the accusing member resubmit it. Since all valid shadows lie on the same degree t polynomial, reconstruction from any $t+1$ shadows will yield sk_i.

Finally, each party i adds up all n shadows $u_{i,j}$ it has, where $n-1$ shadows were received from other parties and one comes from evaluating its own polynomial, to get a share s_i of the polynomial $p = \sum_{i=1}^{n} p_i$, for which it holds that $p(0) = \sum_{i=1}^{n} p_i(0) = \sum_{i=1}^{n} sk_i = msk$. Since polynomial p is of degree t, *msk* can be recovered by a group of $t+1$ cooperating members through Lagrange interpolation. With any fewer shares, no useful information of *msk* can be derived. Note that a council member can submit *msk* and the coordinator can verify it against the master public key *mpk* rather than performing any interpolation on-chain.

To transmit shadows confidentially over the coordinator, Schindler et al. [1] employ a symmetric key encryption scheme inspired by the Diffie-Hellman key exchange [20] and ElGamal encryption [21]. As part of their registration, every council member i submits a public encryption key $\hat{pk}_i = g^{\hat{sk}_i}$ to the coordinator while keeping $\hat{sk}_i$ to themselves. Note that $\hat{sk}_i$ is distinct from sk_i, the contribution of member i to the master secret key. Every council member i can then compute a symmetric encryption key for another council member j as $k_{ij} = \hat{pk}_j^{\hat{sk}_i} = \hat{pk}_i^{\hat{sk}_j} = g^{\hat{sk}_i \hat{sk}_j}$. With this key, shadows can then be encrypted before being broadcast via the coordinator.

In order to file a dispute, a council member j reveals to the coordinator the symmetric encryption key k_{ij} used by member i whom he accuses of sending an incorrect shadow. To prove the correctness of this key without revealing $\hat{sk}_j$, Schindler et al. [1] employ a non-interactive zero knowledge proof and verification scheme that we adopt without any changes. For the purpose of this paper, it suffices to know that unfounded disputes with an incorrect decryption key are not accepted by the coordinator. By resubmitting the broadcast message of the accused member as part of the dispute, the coordinator can decrypt the shadow in question, check its validity against the verification values of the sender i with Equation (1), and disqualify member i if the shadow is found invalid.

B. Ethereum

We assume general familiarity with Ethereum [22] including transactions, the gas cost mechanism, and the underlying blockchain and peer-to-peer infrastructure. The components of Ethereum that are most crucial for our implementation are *precompiled contracts* for the elliptic curve operations that we rely on. The term is rather misleading since they are not actually smart contracts that were compiled and stored at specific addresses on the blockchain, but rather optimised implementations of certain functions external to the EVM that are called as if they were contracts. We rely on two precompiled contracts for addition and scalar multiplication on the Barreto-Naehrig (BN) [23] curve as defined by EIP-196[3], which were deployed with the Byzantium hardfork in 2017. At the end of 2019, the gas costs for both of these contracts

[3]https://eips.ethereum.org/EIPS/eip-196

were substantially reduced per EIP-1108[4], which was deployed as part of the Istanbul hardfork. At the time of writing, the amount of gas that can be spent within a single block is approximately $15\,000\,000$, any transaction that exceeds this limit is practically impossible to execute. This gas limit per block can be influenced by miners and changes over time.

V. ETHTID

In this section, we describe ETHTID, an Ethereum smart contract for a decentralised Threshold Information Disclosure. We base our work on ETHDKG [1], [2] but optimise (simplify) and extend the implementation for the TID use case.

The goal of ETHDKG was to establish a group BLS signature [3], whereas we only need an ElGamal key pair for encryption and eventual decryption. The first optimisation is rather straightforward: We can do without the bilinearity of the BN curve. The main benefit is that the ETHTID contract does not need to perform any pairing checks, thus saving costs. The reason we still use the BN curve is that it is currently the only elliptic curve available on Ethereum. Implementing a simpler curve in Solidity, in lieu of a precompiled contract, would be prohibitively expensive.

The second optimisation is more intricate but also more significant in terms of cost savings. Gennaro et al. [5], [6] thoroughly examined how an attacker can bias the result of a distributed key generation by entering under the guise of multiple identities and then selectively denouncing some of them. The core issue is that all information necessary to compute the result of the protocol, *mpk* in our case, is known before the last opportunity to disqualify participants. In ETHDKG [1], Schindler et al. employed a countermeasure by Neji et al. [7] that involved an additional round of broadcasts and an optional reconstruction in case any party did not perform this broadcast. Instead of preventing biasing attacks in this way, we can simply accept them. This is because they do not impact the security of the ElGamal encryption scheme: Assume an attacker waits and observes all pk_i from honest parties, computing $\tilde{mpk} = \prod pk_i$. The attacker may then choose any value b and force the resulting public key of the protocol to be $mpk = \tilde{mpk} \cdot g^b$. Whatever attack an adversary can perform against this biased public key *mpk* can also be performed against the unbiased public key $\tilde{mpk}$. Given $\tilde{mpk}$ and a cipher text $\tilde{c} = (\tilde{c}_1, \tilde{c}_2) \coloneqq (g^r, \tilde{mpk}^r \cdot m)$, and given b, this cipher text can be transformed to the biased public key *mpk*: $c = (c_1, c_2) \coloneqq (\tilde{c}_1, \tilde{c}_2 \cdot \tilde{c}_1^b) = (g^r, \tilde{mpk}^r \cdot g^{rb} \cdot m) = (g^r, (\tilde{mpk} \cdot g^b)^r \cdot m)$ which is a valid cipher text under $mpk = \tilde{mpk} \cdot g^b$. Simply put, an attacker gains no advantage from biasing the ElGamal key pair, so we can simplify the TID protocol to save costs. It is crucial to note that this optimisation is only viable because the key pair in question is meant to be used solely for ElGamal encryption.

With these optimisations and the extension of a scheduled key recovery, ETHTID proceeds in five phases: *Setup*, *Registration*, *Share Distribution*, *Dispute* (if needed), and *Reconstruction*. We first consider an execution without misbehaving parties before examining different cases of misbehaviour and the contract's ability to handle them.

A. Execution without Misbehaviour

First, the *initiator* deploys the ETHTID contract by submitting a corresponding transaction to the Ethereum peer-to-peer network. As a part of the deployment transaction, the initiator also sets the threshold t as a fraction of the council members registered by the end of the next phase. The initiator also sets the schedule for the subsequent phases at this point, most crucially the time for the coordinated recovery r, via block heights of the Ethereum blockchain. All of these settings are write-once and cannot be changed afterwards. In practice, the initiator would also supply the contract with a reward to incentivise correct participation.

During the *Registration* phase, council members call REGISTER() and submit a public key to be used for the confidential exchange of shadows. For the sake of simplicity, we use a first-come-first-served method here, but more intricate approaches could be used in practice, notably to defend against Sybil attacks. Depending on the incentive scheme, council members would also submit a deposit as part of their registration.

With the beginning of the *Share Distribution* phase, registration is no longer possible. Each registered council member i draws a random individual secret sk_i, embeds it into their random individual polynomial $p_i = sk_i + \sum_{k=1}^{t} a_{i,k} X^k$ of degree t, and generates verification values as described in Section IV-A: $pk_i = g^{sk_i}, \{A_{i,k} = g^{a_{i,k}}\}_{k \in \{1,\dots,t\}}$. Member i encrypts the shadow $u_{ij} = p_i(j)$ via a one-time pad based on the symmetric encryption key k_{ij} (see Section IV-A) using a cryptographic hash function $\mathrm{H}(k_{ij}||j)$. This way, the shadows u_{ij} and u_{ji} are not encrypted with the same one-time pad but both recipients are still able to decrypt them. Each member then calls DISTRIBUTE_SHARES() to broadcast both the encrypted shadows for other members $\{\overline{u_{ij}}\}_{j \neq i}$ as well as the verification values from above. The smart contract stores a hash of the encrypted shadows and verification values for a possible dispute. In this way both the (n, n) and $(t+1, n)$-sharing of *msk*, as described in Section IV-A, are performed in a single step. Note, however, that council members who broadcast invalid shadows can still be disqualified via the *Dispute* phase that we examine closer in the next section.

With the end of the Dispute phase, all remaining council members can combine their shadows to generate their threshold share s_i of the master secret key *msk*. Council members are meant to keep both s_i and their individual secret sk_i private until the scheduled time for disclosure and only then broadcast both via SUBMIT_SECRET(). As soon as $t+1$ members do this, anyone can compute the master secret key *msk* via Lagrange interpolation and submit it to the contract via SUBMIT_MSK() to conclude the functional part of the protocol. It is worth noting that the smart contract performs no checks on sk_i or s_i as part of SUBMIT_SECRET(). We argue that neither of these checks are necessary in practice, since the council is collectively under pressure to produce the master secret key in order to earn their reward and reclaim their security deposit.

[4]https://eips.ethereum.org/EIPS/eip-1108

B. Misbehaviour Detection and Dispute Handling

The first general class of misbehaviour that smart contracts handle rather easily is inactivity. If a member is expected to call a function but fails to do so within a certain amount of time, designated by block height in Ethereum, a smart contract can notice the passing of a predefined deadline during a transaction and act accordingly. There are two opportunities for a council member to be inactive in ETHTID: during the Share Distribution and the Reconstruction phase. If a council member registers but fails to call DISTRIBUTE_SHARES(), they would lose their security deposit and the protocol proceeds without them. In the case that too many council members remain inactive in this way, the entire protocol would have to be restarted with a new council. Due to the established threshold sharing of *msk*, up to $n-t-1$ council members can refrain from calling SUBMIT_SECRET() without consequence, as the remaining members are still able to complete the reconstruction. Consequently, if one more member remains inactive during this phase, *msk* cannot be recovered. In practice, this outcome would be discouraged via an incentive scheme [9].

The more interesting case of misbehaviour revolves around the validity of shadows broadcast via DISTRIBUTE_SHARES(), which ETHTID inherits from ETHDKG [1]. Based on the verification values that a council member must broadcast alongside the encrypted shadows, other members can verify the correctness of all received and decrypted shadows via Equation (1). If a member j detects an invalid shadow, they should call SUBMIT_DISPUTE() and resubmit the broadcast of the accused member i along with the corresponding symmetric encryption key k_{ij} and a zero-knowledge correctness proof. The ETHTID contract verifies the integrity of the broadcast via the previously stored hash, verifies the correctness proof, decrypts the shadow in question and checks its validity via Equation (1). If the shadow is indeed invalid, the accused member is disqualified from the remaining protocol and their security deposit could be destroyed or redistributed. It is worth noting that unfounded disputes have no consequences other than the execution costs for the accusing member. Similarly, a single valid dispute suffices to disqualify a misbehaving member, regardless of how many invalid shadows they may have sent. In such a case, only the first dispute would involve the costly verification of a shadow, whereas any subsequent disputes against the same offending member would be recognised as moot and not incur any significant cost.

VI. Evaluation

We now examine the deployment and execution costs of the proposed construction. We performed this evaluation with Ganache[5] (v2.13.2) and compiled the smart contract with solc (v0.5.17). At the time of writing, the active Ethereum hard fork was Muir Glacier, so we set our local development blockchain accordingly. Recall that the Istanbul hard fork, which preceded Muir Glacier, included gas cost reductions for the precompiled contracts that we use for elliptic curve operations. We adapted the accompanying Python application of ETHDKG [1] in conjunction with its smart contract. The Python application serves to check the smart contract operations for correctness and automate the gas cost evaluation.

Table I
Cost of functions independent of threshold t and number of council members n. Conversion rates: USD 2109.29 per ETH, ETH 25.13×10^{-9} per gas.

Function	Gas	USD
Contract Deployment	1 881 722	99.74
REGISTER()	106 407	5.64
SUBMIT_SECRET()	25 196	1.34
SUBMIT_MSK()	52 225	2.77

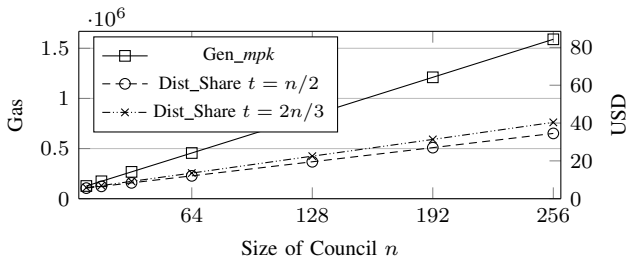

Figure 2. Execution costs of GENERATE_MPK() (independent of t) and DISTRIBUTE_SHARES() with thresholds of $t = \lceil n/2 \rceil - 1$ and $t = \lceil 2n/3 \rceil - 1$. Conversion rates: USD 2109.29 per ETH, ETH 25.13×10^{-9} per gas.

Using the above setup, we compiled and deployed our contract with thresholds of $t = \lceil n/2 \rceil - 1$ and $t = \lceil 2n/3 \rceil - 1$ before executing the protocol described in Section V with a varying number of council members and recording the gas costs for individual transactions. Since we observed very regular costs, we capped the evaluation at $n = 256$.

While gas costs are constant, barring any improvements to the contract or gas price altering hard forks like Istanbul, monetary costs are subject to market forces and network utilisation and, thus, fluctuate over time. For a better intuition, we report costs in both gas and USD, using the daily average exchange rates of 1st July 2021 as reported by Etherscan[6]: USD 2109.29 per Ether and a gas price of ETH 25.13×10^{-9}.

Table I shows the execution costs of functions that are independent of the threshold t and council size n. Recall that SUBMIT_SECRET() is merely a broadcast of two values without any checks and SUBMIT_MSK() only verifies that the submitted master secret key is consistent with the previously generated master public key.

Figure 2 shows the execution costs for the DISTRIBUTE_SHARES() broadcast per council member as well as the execution cost of GENERATE_MPK(). These costs of DISTRIBUTE_SHARES() scale in both the size of the council n and the threshold t since the broadcast consists of $n-1$ encrypted shadows and $t+1$ verification values. Even though only one verification value and a hash over the broadcast payload is persistently stored on-chain, sending all this data

[5]https://www.trufflesuite.com/ganache

[6]https://etherscan.io

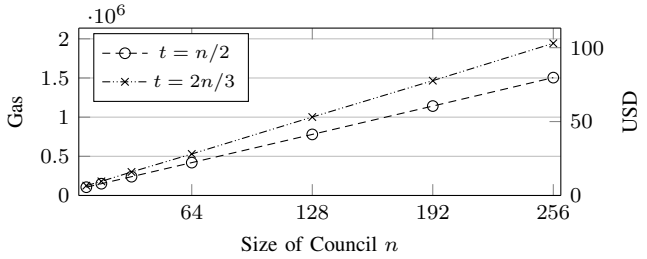

Figure 3. Execution costs of SUBMIT_DISPUTE() with threshold ratios of $t = \lceil n/2 \rceil - 1$ and $t = \lceil 2n/3 \rceil - 1$. Conversion rates: USD 2109.29 per ETH, ETH 25.13×10^{-9} per gas.

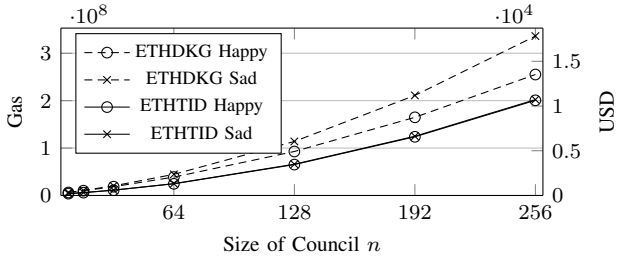

Figure 4. Total execution costs of ETHDKG and ETHTID with $t = \lceil n/2 \rceil - 1$. Happy case: No misbehaviour. Sad case: One incorrect share distribution and dispute and only $t+1$ members complete each protocol. Conversion rates: USD 2109.29 per ETH, ETH 25.13×10^{-9} per gas.

with a transaction still incurs costs. The costs for GENERATE_MPK() are independent of the threshold since it only involves combining the pk_i of all n qualified council members.

Figure 3 shows the execution costs for a valid SUBMIT_DISPUTE() transaction. The brunt of these costs are caused by the evaluation of Equation (1), which scales with threshold t as it determines the degree of the sharing polynomials. Since the DISTRIBUTE_SHARES() broadcast of an offending council member must be resubmitted as part of the dispute, the council size n has a very slight influence as well. To see this, compare the costs of $n = 192, t = \lceil 2n/3 \rceil - 1 = 127$ of 1 465 837 gas and $n = 256, t = \lceil n/2 \rceil - 1 = 127$ of 1 504 658 gas. Since the dispute mechanism is unchanged compared to ETHDKG [1], this evaluation incidentally also shows the effect of the cost reductions to the elliptic curve operations that were part of the Istanbul hard fork.

Based on our measurements, we can determine lower bounds for the deployment and execution costs as depicted in Figure 4 for $t = \lceil n/2 \rceil - 1$ with the costs for $t = \lceil 2n/3 \rceil - 1$ being 1–14% higher. Note that these bounds represent a happy case without any disputes and where each phase is completed with the minimally necessary transactions: One contract deployment, n calls of REGISTER() and DISTRIBUTE_SHARES(), one call of GENERATE_MPK(), $t+1$ calls of SUBMIT_SECRET(), and one call of SUBMIT_MSK().

To demonstrate the amount of gas saved by our adaptations, we evaluated both ETHDKG and ETHTID on the Muir Glacier hard fork in two scenarios: A happy case where everything goes as planned and no council member misbehaves or becomes inactive, similar to the description in the previous paragraph; and a sad case where one council member distributes invalid shares and all but the minimally required $t + 1$ council members become inactive after the distribution of shares. It is important to note that ETHDKG lacks the functionality necessary for a scheduled reconstruction of *msk*. However, the added deployment costs should be minimal and our measurements of Table I show that the added execution costs are very low as well. The costs for SUBMIT_MSK() would be higher by a constant amount for ETHDKG since an additional pairing check would be necessary but it would still not present a significant increase to the overall costs.

Figure 4 shows a direct comparison of the total costs of ETHDKG and ETHTID and Figure 5 illustrates the relative

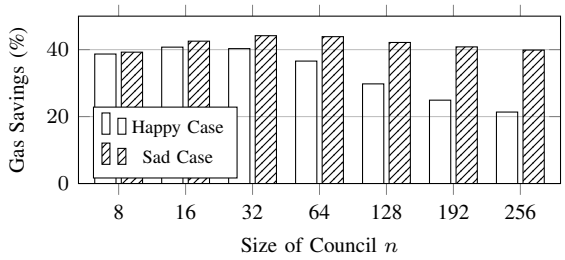

Figure 5. Relative cost savings of ETHTID compared to ETHDKG. Happy case: No misbehaviour. Sad case: One incorrect share distribution and dispute and only $t + 1$ members complete each protocol.

gas savings. It is very clear that the additional broadcast and optional reconstruction phases of ETHDKG for the biasing countermeasure due to Neji et al. [7] are the main sources for gas savings, as demonstrated by the difference between happy and sad case. Nevertheless, even in the happy case, we observe gas savings of 20–40%. It is also noteworthy how the happy and sad cases for ETHTID show very similar costs. This is due to the only difference in execution being a call of SUBMIT_DISPUTE() to disqualify a member who performed an invalid share distribution.

VII. Discussion and Future Work

First and foremost, we can deduce from the results of Section VI that ETHTID is indeed deployable on Ethereum and able to execute the distributed key generation and threshold sharing by Feldman [4] as well as a coordinated reconstruction. With up to 256 council members, all transactions stay far below the current block gas limit of 15 000 000. Currently, the ETHTID contract is only able to run the TID protocol once. With adjustments, it could be made reusable, which would allow an amortisation of deployment costs. However, in order to reuse prior council member registrations, the one-time pad construction for the distribution of shadows would have to be altered, as it would currently weaken with every reuse.

It is important to discuss both the assumptions we base our constructions on as well as the freedoms it provides in its applications. Mainly, the selection of council members is a linchpin that can render all efforts meaningless if not done with the utmost care. Recently, proof-of-work-based Sybil defences that could be applicable to the TID concept have been

developed [24] that are currently being refined and improved [25]. An alternative to the first-come-first-served selection method we used for simplicity would be a manual preselection of council members by the initiator. Similarly, a form of "reputation and collateral" system that spans across multiple instantiations of the TID protocol could be used to build trust in participants over time. When scheduling a TID execution, it is important to account for network congestion and be lenient with deadlines. After the setup and share distribution, the most crucial and time-sensitive transaction is the publication of the master secret key. We described the exchange of shares for reconstructing the secret key as on-chain transactions, but the council can in fact use any communication channel to recover the secret key and then publish it on-chain and on time.

Intertwining cryptography and economics presents opportunities to expand existing attacker models, where participants either adhere to the protocol completely or are under the full control of an attacker. Similarly, the goal of an attacker is usually quite singular in these models, be it extracting a secret or distinguishing between two messages, to name two examples. With economic incentives in a practical setting, both the states of participants and the goals of attackers become more varied. For example, council members could follow the protocol but look to sell their secrets and shares to an attacker to maximise their profit. Attackers could also look to cause as much disruption as possible for a given budget they are willing to lose. Handling such scenarios without giving council members or users perverse incentives is the main challenge when adapting an incentive scheme for the presented protocol.

Lastly, we look towards future developments in Ethereum and possible improvements to our implementation they could present. Currently, only the Barreto-Naehrig curve [23] is supported via precompiled contracts, which would not have been our first choice if more suitable alternatives were available. More precompiled contracts for elliptic curve cryptography are currently being discussed[7], which would include curves that are more suitable to our use case like secp256k1 or Ed25519.

VIII. Conclusion

In this paper, we presented ETHTID, an Ethereum smart contract that acts as a coordinator and arbiter of conflict for a distributed key generation, threshold sharing, and coordinated reconstruction to facilitate Threshold Information Disclosure in a context of mutual distrust. We demonstrated the deployability of the construction experimentally and provided measurements to estimate the overall execution costs based on council size and threshold for disclosure. With ETHTID providing a functionality that Ethereum does not offer innately, namely the coordinated disclosure of arbitrary data by mutually distrustful parties, new applications may become possible, particularly when it comes to publishing records from multiple parties independently. While our results are generally positive, we also highlight areas where both the tools available in Ethereum as well as their application can be improved.

[7]https://eips.ethereum.org/EIPS/eip-1962

References

[1] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, "ETHDKG: Distributed key generation with Ethereum smart contracts." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 985, 2019.

[2] ——, "Distributed key generation with Ethereum smart contracts," in *CIW'19: Cryptocurrency Implementers' Workshop*, 2019.

[3] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *Int. Conf. Theory and Appl. of Cryptographic Techn.* Springer, 2003, pp. 416–432.

[4] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *28th Annu. Symp. Found. of Comput. Sci.*, 1987, pp. 427–438.

[5] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," in *Int. Conf. Theory and Appl. of Cryptographic Techn.* Springer, 1999, pp. 295–310.

[6] ——, "Secure applications of Pedersen's distributed key generation protocol," in *Cryptographers' Track at the RSA Conf.* Springer, 2003, pp. 373–390.

[7] W. Neji, K. Blibech, and N. Ben Rajeb, "Distributed key generation protocol with a new complaint management strategy," *Secur. and Commun. Networks*, vol. 9, no. 17, pp. 4585–4595, 2016.

[8] O. Stengele, M. Raiber, J. Müller-Quade, and H. Hartenstein, "ETHTID: Deployable threshold information disclosure on Ethereum," *arXiv preprint arXiv:2107.01600*, 2021.

[9] D. Yakira, I. Grayevsky, and A. Asayag, "Rational threshold cryptosystems," *arXiv preprint arXiv:1901.01148*, 2019.

[10] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Annu. Int. Cryptology Conf.* Springer, 1991, pp. 129–140.

[11] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," MIT, USA, Tech. Rep., 1996.

[12] W. Mao, "Timed-release cryptography," in *Sel. Areas in Cryptography*, ser. LNCS, S. Vaudenay and A. M. Youssef, Eds. Berlin, Heidelberg: Springer, 2001, pp. 342–357.

[13] M. Mahmoody, T. Moran, and S. Vadhan, "Time-lock puzzles in the random oracle model," in *Annu. Cryptology Conf.*, ser. LNCS, P. Rogaway, Ed. Berlin, Heidelberg: Springer, 2011, pp. 39–50.

[14] M. Bellare and S. Goldwasser, "Verifiable partial key escrow," in *Proc. 4th ACM Conf. Comput. and Commun. Secur.*, ser. CCS '97. Zurich, Switzerland: ACM, 1997, pp. 78–91.

[15] G. Di Crescenzo, R. Ostrovsky, and S. Rajagopalan, "Conditional oblivious transfer and timed-release encryption," in *Int. Conf. Theory and Appl. of Cryptographic Techn.*, ser. LNCS, J. Stern, Ed. Berlin, Heidelberg: Springer, 1999, pp. 74–89.

[16] J. Cathalo, B. Libert, and J.-J. Quisquater, "Efficient and non-interactive timed-release encryption," in *Int. Conf. Inf. and Commun. Secur.*, ser. LNCS, S. Qing, W. Mao, J. López, and G. Wang, Eds. Berlin, Heidelberg: Springer, 2005, pp. 291–303.

[17] F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin, "Can a public blockchain keep a secret?" in *Theory of Cryptography Conf.* Springer, 2020, pp. 260–290.

[18] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[19] T. P. Pedersen, "A threshold cryptosystem without a trusted party," in *Workshop Theory and Application of Cryptographic Techn.*, ser. LNCS, D. W. Davies, Ed. Berlin, Heidelberg: Springer, 1991, pp. 522–526.

[20] E. Rescorla, "RFC2631: Diffie-Hellman key agreement method," IETF, Tech. Rep., 1999.

[21] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Inf. Theory*, vol. 31, no. 4, pp. 469–472, 1985.

[22] V. Buterin, "A next-generation smart contract and decentralized application platform," *white paper*, 2014.

[23] P. S. L. M. Barreto and M. Naehrig, "Pairing-friendly elliptic curves of prime order," in *Sel. Areas in Cryptography*, B. Preneel and S. Tavares, Eds. Berlin, Heidelberg: Springer, 2006, pp. 319–331.

[24] D. Gupta, J. Saia, and M. Young, "Peace through superior puzzling: an asymmetric Sybil defense," in *IEEE Int. Parallel and Distrib. Process. Symp.* IEEE, 2019, pp. 1083–1094.

[25] ——, "ToGCom: an asymmetric Sybil defense," *arXiv preprint arXiv:2006.02893*, 2020.