vetKeys: How a Blockchain Can Keep Many Secrets

Andrea Cerulli[1], Aisling Connolly[1], Gregory Neven[*],
Franz-Stefan Preiss[1], Victor Shoup[1]
[1] DFINITY

April 30, 2023

Abstract

We propose a new cryptographic primitive called *verifiably encrypted threshold key derivation* (vetKD) that extends identity-based encryption with a decentralized way of deriving decryption keys. We show how vetKD can be leveraged on modern blockchains to build scalable decentralized applications (or *dapps*) for a variety of purposes, including preventing front-running attacks on decentralized finance (DeFi) platforms, end-to-end encryption for decentralized messaging and social networks (SocialFi), cross-chain bridges, as well as advanced cryptographic primitives such as witness encryption and one-time programs that previously could only be built from secure hardware or using a trusted third party. And all of that by secret-sharing just a single secret key...

1 Introduction

1.1 Privacy on Blockchains

Since its creation in 2008, Bitcoin [Nak08] has taken the world by storm. The financial world has come to accept cryptocurrencies as an investment class of their own and is still wrapping its head around the longer-term implications to the system of global finance. At the same time, the technology sector has witnessed an unseen spurt of innovation in decentralized computing, in particular, in blockchains, Bitcoin's main underlying technical ingredient.

Bitcoin is restricted to payments and is notoriously inefficient, processing just 7 transactions per second using the energy of a medium-sized country [Sch22]. Modern blockchains often make different trade-offs in terms of decentralization and efficiency to offer much more functionality at a fraction of the energy to process thousands of transactions per second. Rather than being restricted to payments, they usually support running arbitrary pieces of code, called *smart contracts* or *dapps* (short for decentralized applications), on the blockchain. The nodes that maintain the blockchain keep track of the dapps' state and apply user transactions that modify the state.

Many modern blockchains are Turing-complete, meaning that they can run arbitrary computations, ranging from small scripts to complete decentralized finance (DeFi) platforms, social networks (SocialFi), or games (GameFi). The vision that such decentralized, token-based applications will one day replace current Internet services is called *Web3*, a third iteration of the World Wide Web, after the first iteration enabled users to mainly consume static pages while the second that added user-contributed content.

[*]Work partially done while at DFINITY.

Security and privacy on blockchains. Blockchains are often touted to increase security for users, which is true to a certain extent. Their decentralized nature means that dapps can offer stronger availability and integrity than a centralized web application: rather than relying on a single provider to keep the service running, dapps remain online as long as a sufficiently large fraction of the diverse set of nodes are reachable. Moreover, because all dapp computations are performed in lockstep on many nodes at the same time, a faulty or malicious party cannot easily tamper with a dapp's state. Finally, users and transactions are typically authenticated by cryptographic keys instead of passwords, making impersonation and account compromise considerably harder to pull off.

The elephant in the room, however, is privacy. The same decentralization that helps integrity and availability is a disaster for privacy. As the state of dapps is replicated across tens, hundreds, or even thousands of nodes, dapps are generally unsuitable for storing sensitive data.

Encryption can help, but key management is complicated, more so than for authentication. Syncing encryption keys across devices is particularly difficult. An identity provider dapp can link multiple signature keys to a user so that any third-party dapp can accept signatures from a new device as soon as it's added to a user's profile. Data encrypted under the existing devices' encryption keys is not automatically legible to a new device, however. Moreover, hardware wallets and secure environments in consumer devices usually don't support encryption. Decryption keys can be held in web browsers' local storage, but are more exposed to malware there and do not survive cookie cleanups.

Solutions for blockchain privacy. Secret sharing [Sha79] seems like a natural fit to the problem: users could secret-share any sensitive information over the nodes in the network, guarded by a dapp that determines under which conditions the secret can be reconstructed. This may work for a handful of secrets or in a static, permissioned network, but, because all secrets have to be verifiably re-shared when membership changes, this doesn't scale to a setting with billions of secrets in a dynamic blockchain network.

Threshold public-key encryption (PKE) [SG98, RB94, CKPS01] is a much better fit: here, all nodes have a share of the decryption key of a public-key encryption scheme. Users encrypt sensitive information under the corresponding public key, and a dapp controls the conditions under which the nodes collaborate to decrypt a particular ciphertext. Nodes can send the decryption shares directly to the user or encrypted under a user-provided public key, so that the nodes don't learn the plaintext. When network membership changes, the nodes only have to re-share that single decryption key, independent of the number of encrypted ciphertexts.

While threshold PKE works for many scenarios, it is still sub-optimal for others, especially when many ciphertexts have to be decrypted all at once. Think, for example, of a secret-bid auction dapp, where at the end of each auction, all bids need to be decrypted so that the winner can be determined. Or think of a decentralized exchange (DEX) that prevents front-running by sequencing encrypted transactions and only decrypt them for execution when the order is fixed.[1]

In such applications, performing a threshold decryption for every single ciphertext is simply not feasible because of the sheer volume of ciphertexts that need to be decrypted.

A more scalable solution to the front-running problem was developed by Sekar [Sek22] and Gailly, Melissaris, and Romailler [GMR23], who implemented the threshold time-lock encryption scheme suggested by Boneh and Franklin [BF01] on top of the drand randomness beacon [dra]. At each round, the random beacon computes a threshold BLS signature [Bol03, BLS01] on the round number. Because BLS signatures are also decryption keys to the Boneh-Franklin identity-based encryption (IBE) scheme, users can encrypt transactions to a future round number, so that the nodes can decrypt all ciphertexts using the random beacon for that round. Front-running is prevented by fixing the execution order based on the encrypted transactions; only when the randomness beacon becomes known, the nodes decrypt the transactions for that round and execute them in the committed order.

[1]Front-running is actually a major problem on decentralized exchanges on Ethereum where it's known as *miner-extracted value* (MEV) and has cost users more than $600 million since 2020. [Gen22]

1.2 Our Contributions

We extend this idea to build a flexible and scalable solution to achieve user privacy on blockchains by means of a new cryptographic primitive that we call *verifiably encrypted threshold key derivation* (vetKD). Intuitively, vetKD can be seen as the key derivation of an IBE scheme, where the master secret is secret-shared among a set of servers that can assist users in deriving IBE keys by sending encrypted key shares to the users. The idea is that the shares are encrypted under a public key provided by the user, so that the servers do not learn the derived key. Nevertheless, servers can verify that a encrypted share indeed contains a valid key share, so that users are guaranteed to recover the correct key if they receive sufficiently many valid encrypted shares. We refer to the derived keys as *vetKeys*.

In this paper, we focus on vetKD schemes that apply to the Boneh-Franklin IBE scheme [BF01], meaning that that we are actually looking to build verifiably encrypted threshold BLS (vetBLS) signatures. We define a vetBLS ideal functionality in the universal composability (UC) model [Can01] and provide four different vetBLS protocols that securely realize it. The first two protocols, the simple vetBLS protocol $\pi_{\text{vetbls-sim}}$ and the zero-knowledge vetBLS protocol $\pi_{\text{vetbls-zkp}}$, are presented in Section 4 and have encrypted signature size and verification time linear in the threshold. We also present two aggregate vetBLS protocols $\pi_{\text{vetbls-agg1}}$ and $\pi_{\text{vetbls-agg2}}$ in Section 5 that have constant encrypted signature size and verification time, but that realize a slightly weaker version of the vetBLS functionality where the encrypted signature leaks some information about the BLS signature to the adversary. Whether such leakage can be tolerated depends on the application; in Sections 6 and 7, we prove that it is harmless for use as a verifiably encrypted threshold IBE, pseudo-random function (PRF), verifiably random function (VRF), or signature scheme. We note that these proofs extend to the non-threshold case, i.e., that the security of the IBE, PRF, VRF, or signature scheme are not affected if the adversary additionally sees verifiably encrypted BLS signatures using the (non-threshold) VES scheme of [BGLS03].

In theory, each dapp that wants to derive vetKeys could use its own vetBLS instance with an independently generated master key. The cost of re-sharing these master keys when the set of validator nodes changes quickly becomes prohibitive, however. We therefore analyze the security of a composed protocol in Section 8 that uses a single master key for all dapps and for all purposes listed above at the same time, and show that doing so is fine as long as the derivation identity domains are properly separated by including a sub-session identifier.

Finally, we describe the integration of vetKeys into the Internet Computer [DFI22], the blockchain created by the DFINITY Foundation. Somewhat surprisingly, the bottleneck turns out to be the verification time of encrypted key shares, rather than block space, so that it is worth considering variants of our schemes that make different trade-offs in this regard.

1.3 Applications

Equipping a blockchain with a vetKD interface enables a wide range of applications, even beyond the ones we already discussed.

End-to-end encryption. Our main use case is to enable a blockchain to host threshold-encrypted data in a way that scales to millions of users and billions of secrets, using just a single threshold-shared secret key. The BLS signatures that underlie the Boneh-Franklin IBE scheme are unique, making them immediately useful as symmetric keys.

Think, for example, of a secure file storage dapp: a user could use the BLS signature on their identity as the root secret under which they encrypt their files before storing them in the dapp. The dapp enforces that only the authenticated user is allowed to recover the root key, and hence decrypt the files. The nodes in the blockchain assist a user in recovering their root key, but never see that key or the content of the files.

More sophisticated access policies can be expressed as well. In a secure messaging dapp, the conversation between two users can be encrypted using the BLS signature on their pair of identities, to which only those users are given access by the dapp. An end-to-end encrypted decentralized social network (SocialFi) can let users encrypt posts using a key that is related to the post, e.g., the BLS signature on a unique identifier for the post. The SocialFi dapp then ensures that only the author, and the users that the post is shared with, get access to that key.

Blockchain-issued signatures and cross-chain bridges. Because the key derivation of an IBE scheme automatically yields a signature scheme [BF01], the resulting decryption keys can also be used as signatures issued by the blockchain. This is especially useful for blockchains that don't have a built-in certification feature enabling dapps to sign statements. It can also be used to efficiently bridge blockchains, e.g., to swap assets in DeFi application: a dapp on a first blockchain can verify signed statements issued by a second blockchain, without having to implement a complete light client of that second chain.

Verifiable randomness. Because of their uniqueness, BLS signatures can also act as a verifiable random function (VRF). Trusted, verifiable randomness is important for applications such as trustless online lotteries and casinos, fair decentralized games (GameFi), and selecting random features for non-fungible tokens (NFTs).

Dead man's switch. Journalists or whistleblowers can ensure that compromising information in their possession is automatically published if they were to become incapacitated. They can store the information in a dapp, encrypted under a BLS signature that the dapp automatically and publicly recovers when a certain amount of time passes after it has received an authenticated ping from its owner.

Secret-bid auctions and MEV protection. As described in the introduction, a vetKD-equipped blockchain can also cover use cases where many ciphertexts needs to be decrypted at the same time. In a secret-bid auction dapp, users can submit bids that are IBE-encrypted under an identifier of the auction, so that at the end of the auction, the dapp can decrypt all bids with a single vetKD evaluation.

A similar technique can be used to prevent front-running, also known as *miner-extracted value* (MEV), on a decentralized exchange (DEX). Users submit their transactions IBE-encrypted under a predictable batch identifier. The DEX orders the transactions in encrypted form and, when all transactions for a particular batch have been ordered, triggers the recovery of the decryption key for that batch and executes the decrypted transactions in the fixed order.

Note that all of the symmetric-encryption use cases listed above can be modified to encrypt using an IBE scheme instead of a symmetric-key encryption, thereby eliminating the need to perform a vetKD derivation for encryption. (Decryption, of course, still requires a vetKD evaluation.)

Time-lock encryption. Time-lock encryption [RSW96, LJKW18] enables a sender to encrypt a message "to the future," ensuring that it will get decrypted at a given time, but no earlier than that time. Existing solutions rely on centralized trusted parties, witness encryption (see next paragraph), or gradual release through puzzle solving. Time-lock encryption can be achieved via IBE [BF01] by letting a centralized authority release IBE decryption keys corresponding to the current time at regular intervals, and letting the sender IBE-encrypt its message using the desired decryption time as identity. The authority's functionality can be run in a dapp on a vetKD-equipped blockchain, eliminating the need for a trusted central party.

Witness encryption. A witness encryption scheme [GGSW13] for a language L with witness relationship R lets a sender encrypt a message to an instance $x \in L$ that can only be decrypted using a witness w such that $R(x, w)$. The only current implementations are based on indistinguishability obfuscation [BGI$^+$01], of which few instantiations are known based on well-founded assumptions [JLS21]. Witness encryption is almost trivial to implement on a vetKD-enabled blockchain: anyone can IBE-encrypt their message using the instance x as identity, while a witness-verifying dapp lets anyone who provides a valid witness w for x (or a valid zero-knowledge proof of knowledge of w, if it should remain private) to obtain the decryption key for x. The primitive may sound rather theoretical at first, but it actually covers quite practical use cases as it enables one to encrypt to any verifiable future event, e.g., the price of a stock going above or below a certain level, information escrow, or break-the-glass policies.

One-time programs. Another cryptographic primitive with few instantiations is one-time programs [GKR08] that can be executed only once on a single input, and that don't leak anything about the program other than the result of the computation. Their only currently known instances rely on trusted hardware [GKR08] or on witness encryption on a blockchain [GG17]. Given that witness encryption is easy to build on a vetKD-enabled blockchain, it should not come as a surprise that one-time programs are as well. The creator of the program garbles the circuit and IBE-encrypts the input wire keys, using the wire index and the value as the identity. A dapp assists users in recovering the IBE decryption corresponding to their input, making sure that only a single value for each wire is ever recovered.

1.4 Related Work

From a constructional point of view, the closest related work is the (non-threshold) verifiably encrypted signature (VES) scheme due to Boneh et al. [BGLS03]. Indeed, both of our aggregate vetBLS schemes can be seen as threshold variants of Boneh et al.'s VES scheme, mapped to an asymmetric Type-3 pairing [GPS06] in slightly different ways. Boneh et al.'s use case, however, is very different from ours: they envisage an optimistic fair exchange of signatures [ASW00], where a first signer encrypts her signature under an adjudicator's public key so that the countersigner can have it decrypted by the adjudicator if the first signer fails to reveal its real signature. A major difference is that in VES, the first signer knows her full signature, so there's no point in hiding it from the signer. Indeed, Calderon et al. pointed out that VES in fact do not require encryption at all and can be built from signatures alone [CMSW14]. (In fact, Section 10 of [ASW00] already made a very similar observation and gives a fair exchange protocol based only on signatures.) In a threshold setting, on the other hand, an individual signer does not know the full signature, and it is crucial that she doesn't learn the full signature from the encrypted signature.

Calypso [KAG$^+$20] is conceptually related in that it also enables private delivery of threshold-reconstructed decryption keys to users of a smart contract. It uses threshold ElGamal encryption instead of BLS, however, meaning that each ciphertext requires a separate threshold protocol to decrypt. By using BLS signatures as IBE keys, on the other hand, a single threshold evaluation can be used to decrypt many ciphertexts in one go. Calypso also doesn't have public verification of encrypted decryption shares and is presented without security proofs.

Benhamouda et al. [BGG$^+$20] and Goyal et al. [GKM$^+$20] describe how users can secret-share and conditionally reveal their secrets on a blockchain with a dynamically evolving committee of nodes, addressing some of the same use cases as our work. Secret-sharing (and especially, dynamically re-sharing) individual user secrets doesn't scale for billions of secrets, but works very well for a limited number of secret keys that are then used to encrypt other secrets. Indeed, their re-sharing protocols can be combined with our vetKD protocols to re-share the master secret key to a new set of nodes.

Another line of work [CGJ$^+$17, BMSV18, CZK$^+$19, KGM19, RAA$^+$19] protects user secrets in smart contracts using trusted execution environments (TEEs) such as Intel SGX or AMD SEV-SNP. Apart from adding a centralized trust assumption in the form of the TEE manufacturer, the unrelenting stream of attacks on TEEs [vSSY$^+$22, BJKS21] doesn't instill confidence in their security. It therefore seems fair to state that, in their current state, TEEs do not provide a very strong level of security for user data, and in particular, fail to provide any relevant protection against an adversary that has physical access to the node machines.

Time-lock encryption, that we mentioned as an example application of vetKeys, was originally proposed by Rivest, Shamir, and Wagner [RSW00] using two different approaches: time-lock puzzles that can be solved only with a predictable amount of computation, and trusted parties that only reveal information after a certain time has passed. Liu et al. [LJKW18] followed the former approach using witness encryption [GGSW13] and using the Bitcoin chain as a "computational reference clock". Boneh and Franklin [BF01] describe a solution that takes the latter approach by adding a timestamp to a user's identity in an identity-based encryption scheme and by distributing the authority in a threshold fashion. This idea was taken further to build a (non-threshold) time vault service [MHS02, BC04] that uses just the timestamp as identity. Sekar [Sek22] and Gailly, Melissaris, and Romailler [GMR23] then built a threshold version of this protocol in the context of blockchain networks and decentralized systems on top of the drand randomness beacon [dra].

The application of vetKeys to protect against MEV draws inspiration from fair transaction ordering in replicated systems, which has a much longer history. Reiter and Birman [RB94] describe a protocol that achieves "input causality", meaning that earlier requests cannot depend on the payload of later requests. Clients submit their requests encrypted under a threshold public key, upon which the servers agree on the order of encrypted requests and only then jointly decrypt and execute the requests. Cachin et al. [CKPS01] further refined and proved this protocol, referring to it as "secure causal atomic broadcast". Duan, Reiter, and Zhang [DRZ17] describe further variations, replacing threshold encryption with commitment-and-reveal and secret-sharing techniques. The TEX protocol [KGF19] uses the commit-and-reveal idea combined with automated opening through a verifiable delay function [BBBF18]. All of these solutions require a separate threshold evaluation for each transaction, or at the very least, for every transaction sender. By letting senders IBE-encrypt their transactions to a batch number, our solution can derive a single vetKey to (locally) decrypt all transactions in the batch, enabling a much higher transaction throughput.

Several solutions have been built to address the particular problem of MEV protection in blockchain networks. Helix [ACG$^+$18] encrypts transactions via a threshold public-key encryption scheme to limit censorship and front-running; once the order of transactions is final, their contents are decrypted. Kill Cord [Tou18] and Kimono [FH18] are Ethereum contracts that let a user generate a secret key, secret-share it among Ethereum nodes and incentivize them to decrypt a given ciphertext if a certain time passes. ClockWork [CDN20] lets users submit transactions encrypted with Rivest et al.'s time-lock puzzle [RSW00] and lets the exchange commit to an ordered batch of transactions before it can decrypt them. The Shutter network [Shu21] prevents MEV on the Ethereum network by having a dedicated set of nodes (so-called "Keypers") regularly generate new threshold encryption keys under which users encrypt their transactions. After the transactions appeared on chain, the Keypers reveal the secret key. ETHTID [SRMH21] implements threshold information disclosure on Ethereum, whereby a council generates a threshold encryption key so that the public key can be used to encrypt messages that are revealed when the council reconstructs the decryption key. Ferveo [BO22] prevents MEV on Tendermint-based proof-of-stake blockchains by letting the chain validators jointly generate keys for a threshold encryption scheme in Gap Diffie-Hellman groups [BZ03]. Users can encrypt information under this public key; to decrypt, the validators create decryption shares that a block proposer can aggregate and include in a block.

2 Usage in a Blockchain Scenario

We first present a syntax definition for vetKD schemes and illustrate their typical usage in a blockchain scenario. A vetKD scheme consists of the following algorithms:

- $\mathsf{DKG}(n, t, f) \rightarrow (mpk, mpk_1, \ldots, mpk_n, msk_1, \ldots, msk_n)$: A distributed key generation protocol performed by n servers $\mathcal{S}_1, \ldots, \mathcal{S}_n$, up to f of whom may be malicious, for a threshold t, so that each server $\mathcal{S}_i$ receives the master public key mpk, master public key shares $(mpk_1, \ldots, mpk_n)$, and a share of the master secret key msk_i, and so that a threshold of t servers is required to derive keys under mpk.

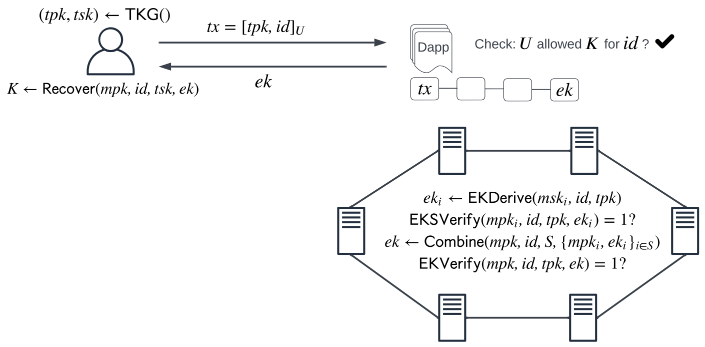

$(tpk, tsk) \leftarrow$ TKG()
$tx = [tpk, id]_U$
Dapp
Check: U allowed K for id ? ✓
$K \leftarrow$ Recover(mpk, id, tsk, ek)
ek
tx
ek
$ek_i \leftarrow$ EKDerive(msk_i, id, tpk)
EKSVerify$(mpk_i, id, tpk, ek_i) = 1$?
$ek \leftarrow$ Combine$(mpk, id, S, \{mpk_i, ek_i\}_{i \in S})$
EKVerify$(mpk, id, tpk, ek) = 1$?

Figure 1: Typical usage of vetKD in a blockchain scenario.

- $\mathsf{TKG}() \rightarrow (tpk, tsk)$: A transport key generation algorithm that a user can use to generate a transport public key tpk and corresponding secret key tsk.
- $\mathsf{EKDerive}(msk_i, id, tpk) \rightarrow ek_i$: An encrypted key derivation algorithm that a server $\mathcal{S}_i$ uses to compute an encrypted key share ek_i for identity id, encrypted under the transport public key tpk.
- $\mathsf{EKSVerify}(mpk_i, id, tpk, ek_i) \rightarrow 0/1$: An encrypted key share verification algorithm which verifies that ek_i indeed contains a valid share by server $\mathcal{S}_i$ with master public key share mpk_i of a derived key for identity id encrypted under transport key tpk.
- $\mathsf{Combine}(mpk, id, S, \{mpk_i, ek_i\}_{i \in S}) \rightarrow ek$: A combination algorithm that combines a set of at least t verified encrypted key shares ek_i for $i \in S$ for identity id into an encrypted key ek.
- $\mathsf{EKVerify}(mpk, id, tpk, ek) \rightarrow 0/1$: An encrypted key verification algorithm that lets anyone verify that ek contains a derived key for identity id under mpk, encrypted under the transport key tpk.
- $\mathsf{Recover}(mpk, id, tsk, ek) \rightarrow K$: A recovery algorithm that enables the user to recover the derived key K for identity id from the verified encrypted key ek using the transport secret key tsk.

We postpone a formal discussion of the desired security properties until later; for now it suffices to know that an adversary controlling at most $f < t$ corrupt servers cannot learn the derived key for any identity id for which it did not "legitimately" obtain at least $t - f$ encrypted key shares from honest servers, i.e., a key shares encrypted under a transport public key tpk created by the adversary.

A typical blockchain scenario is depicted in Figure 1. The vetKD servers will be the blockchain validator nodes, or a committee representing them. The dapp D can implement arbitrary policies to determine which users are allowed to derive the key for which identities. (The identity domains of different dapps are kept separate to avoid interference between dapps.)

When a user wants to derive the key for identity id from a dapp D, they generate a transport key pair (tpk, tsk) using TKG. The transport key is short-lived in principle; it only needs to be held for the duration of this vetKD evaluation, but can optionally be reused for multiple evaluations. The user submits a signed transaction $tx = [tpk, id]_{\mathcal{U}}$ to D that includes the transport public key and the derivation identity, where $[\cdot]_{\mathcal{U}}$ denotes the user's signature on the transaction.

When the transaction is included in a block, the dapp D is executed to check whether the user is allowed to obtain the key for id. If so, and if id is in the identity domain of D, then all nodes run

EKDerive and broadcast their encrypted key shares ek_i to all other nodes, who verify them using EKSVerify.

As soon as a node receives sufficiently many valid encrypted shares, it runs Combine to create an encrypted key ek and possibly broadcasts it to other nodes. The next block proposer then includes ek in the block as a response to the original transaction with id and tpk; other nodes can validate the correctness of ek by running EKVerify. When the user sees the response in the blockchain, they can run Recover to recover the derived key K for id.

In use cases where the dapp D itself needs a decryption key (e.g., in the front-running use case described above), D can trigger the same interaction for a "dummy" transport key generated by D and thus obtain the required key.

3 Overview

In this section, we provide a very brief overview of our protocols. As mentioned in Section 1.2, our protocols for verifiably encrypted threshold key derivation (vetKD) build on protocols for verifiably encrypted threshold BLS (vetBLS) signatures.

Our starting point is the $\mathcal{BLS}$ signature scheme [BLS04] adapted to the setting where we use an asymmetric bilinear map (a so-called Type-3 pairing)

$$\mathrm{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_\mathrm{T},$$

where $\mathbb{G}_1 = \langle g_1 \rangle$, $\mathbb{G}_2 = \langle g_2 \rangle$, and $\mathbb{G}_\mathrm{T}$ are multiplicative groups of prime order q. Let $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1$ be a hash function, modeled as a random oracle [BR93]. The secret key sk for $\mathcal{BLS}$ is a random element of $\mathbb{Z}_q$ and the public key is $pk = g_2^{sk}$. A signature on a message m is $\sigma = \mathsf{H}(m)^{sk}$ and can be verified by checking that $\mathrm{e}(\sigma, g_2) = \mathrm{e}(\mathsf{H}(m), pk)$. The security of $\mathcal{BLS}$ can be proved under the co-CDH assumption, which says that given $g_1^\alpha, g_1^\beta, g_2^\beta$ for random $\alpha, \beta \in \mathbb{Z}_q$, it is hard to compute $g_1^{\alpha\beta}$.

In all of our vetBLS protocols, we assume a secure protocol for distributed key generation. Such a protocol securely generates a $\mathcal{BLS}$ secret key sk and corresponding public key pk along with Shamir secret shares $sk_1, \ldots, sk_n$ and corresponding public key shares $pk_1, \ldots, pk_n$, and gives each server $\mathcal{S}_i$ the values $pk, (pk_1, \ldots, pk_n), sk_i$. Here, $sk_i = \omega(i)$ and $pk_i = g_2^{\omega(i)}$, where ω is a random polynomial over $\mathbb{Z}_q$ of degree less than t.

Our first vetBLS protocol, $\pi_{\text{vetbls-sim}}$, is a very simple protocol that relies on a secure signature scheme $\mathcal{SS}$ and a CPA-secure public-key encryption scheme $\mathcal{PKE}$. Each server $\mathcal{S}_i$ has its own signing key ssk_i for $\mathcal{SS}$, and we assume a secure PKI that allows every other server to reliably obtain $\mathcal{S}_i$'s corresponding public signature verification key spk_i. The protocol works as follows. A user $\mathcal{U}$ generates a transport public key tpk and transport secret key tsk by running the key generation algorithm for $\mathcal{PKE}$. Given a request for an encrypted signature share, consisting of tpk and a message m, a server $\mathcal{S}_i$ generates an encrypted signature share $es_i = (C_i, \sigma'_i)$, where C_i is an encryption of the BLS signature share $\sigma_i = \mathsf{H}(m)^{sk_i}$ under tpk, and σ'_i is a signature on (pk, m, tpk, C_i) under spk_i. Such an encrypted signature share es_i can be validated by validating the signature σ'_i. A valid encrypted signature es consists of a collection of $2t-1$ valid encrypted signature shares (from distinct servers). Given such a valid encrypted signature es, the user $\mathcal{U}$ can decrypt each ciphertext C_i using tsk to obtain a BLS signature share σ_i and can check that σ_i is valid by testing if $\mathrm{e}(\sigma_i, g_2) = \mathrm{e}(\mathsf{H}(m), pk_i)$. Since es consists of $2t-1$ valid encrypted signature shares, and there are at most $t-1$ corrupt servers, the user is sure to obtain a set of t valid BLS signature shares which can be combined via "interpolation in the exponent" to obtain the corresponding BLS signature.

One drawback of protocol $\pi_{\text{vetbls-sim}}$ is that encrypted signatures are of size proportional to t. Another drawback is that it requires that $n \geq 2t-1$. Our second vetBLS protocol, $\pi_{\text{vetbls-zkp}}$, which is based on non-interactive zero-knowledge proofs, eliminates the second drawback, and works as follows. A user $\mathcal{U}$ generates a transport public key tpk and transport secret key tsk based on ElGamal encryption — specifically, $\mathcal{U}$ generates $tsk \in \mathbb{Z}_q$ at random and sets $tpk = g_1^{tsk}$. Given a request (tpk, m) for an encrypted signature share, a server $\mathcal{S}_i$ generates an encrypted signature share $es_i = (C_i, \pi_i)$, where $C_i = (g_1^{t_i}, tpk^{t_i} \cdot \sigma_i)$ is an ElGamal encryption of the BLS signature share $\sigma_i = \mathsf{H}(m)^{sk_i}$ under tpk, where $t_i \in \mathbb{Z}_q$ is chosen at random, and π_i is a non-interactive zero-knowledge proof that C_i encrypts a valid BLS signature share σ_i. The proof π_i is a standard application of Fiat-Shamir [FS87] and Sigma protocols. Such an encrypted signature share es_i can be validated by validating the proof π_i. A valid encrypted signature es consists of a collection of t valid encrypted signature shares (from distinct servers). Given such a valid encrypted signature es, the user $\mathcal{U}$ can decrypt each ciphertext C_i using tsk to obtain a BLS signature share σ_i. The proofs ensure that these t BLS signature shares must be valid, and so they can be combined via "interpolation in the exponent" to obtain the corresponding BLS signature. In addition to the security properties (soundness and zero knowledge) of the proofs, the security of this protocol relies on the external Diffie-Hellman (XDH) assumption [Sco02, BBS04, BGdMM05], which states that the decisional Diffie-Hellman (DDH) problem is hard in $\mathbb{G}_1$.

Both of the above protocols suffer from the drawback that the encrypted signatures are of size proportional to t. To deal with this, we give two protocols, $\pi_{\text{vetbls-agg1}}$ and $\pi_{\text{vetbls-agg2}}$, in which encrypted signature shares are aggregated into a compact encrypted signature whose size is independent of n and t. Validation of encrypted signature shares and encrypted signatures is performed by checking simple pairing equations, and these protocols can be seen as threshold variants of the verifiably encrypted signature scheme of Boneh et al. [BGLS03], mapped to a Type-3 pairing setting. To simplify the discussion in this section, we focus here on protocol $\pi_{\text{vetbls-agg1}}$; protocol $\pi_{\text{vetbls-agg2}}$ and its security properties are very similar.

Protocol $\pi_{\text{vetbls-agg1}}$ works much like protocol $\pi_{\text{vetbls-zkp}}$, except that it uses pairing equations rather than non-interactive zero knowledge proofs. A user $\mathcal{U}$ chooses the transport secret key $tsk \in \mathbb{Z}_q$ at random and sets the transport public key to $tpk = (tpk_1, tpk_2) = (g_1^{tsk}, g_2^{tsk})$. Given a request (tpk, m) for an encrypted signature share, a server $\mathcal{S}_i$ first checks the validity of tpk by checking that $\mathrm{e}(tpk_1, g_2) = \mathrm{e}(g_1, tpk_2)$. If tpk is valid, $\mathcal{S}_i$ generates an encrypted signature share $es_i = (C_{i,1}, C_{i,2}) = (g_1^{t_i}, tpk_1^{t_i} \cdot \sigma_i)$, where $\sigma_i = \mathsf{H}(m)^{sk_i}$ is a BLS signature share and $t_i \in \mathbb{Z}_q$ is chosen at random; otherwise, $\mathcal{S}_i$ ignores the request. Such an encrypted signature share es_i can be verified by checking that $\mathrm{e}(C_{i,2}, g_2) = \mathrm{e}(C_{i,1}, tpk_2) \cdot \mathrm{e}(\mathsf{H}(m), pk_i)$. Given t valid encrypted signature shares, they can be combined into a single encrypted signature $es = (C_1, C_2)$ by "interpolation in the exponent". Such an encrypted signature can itself be verified by checking that $\mathrm{e}(C_2, g_2) = \mathrm{e}(C_1, tpk_2) \cdot \mathrm{e}(\mathsf{H}(m), pk)$. Given such a valid encrypted signature es, the user $\mathcal{U}$ can decrypt it to obtain $\sigma = C_2 \cdot C_1^{-tsk}$. The validity of es guarantees that σ is a valid BLS signature on m under pk. Note that instead of validating es as above (which requires three pairings), the user can instead simply decrypt es and check whether the resulting signature is a valid BLS signature (which requires just two pairings).

Protocol $\pi_{\text{vetbls-agg1}}$ in fact leaks some information about the BLS signatures corresponding to the signing requests. We can precisely characterize the extent of this leakage. Specifically, for a transport public key $tpk = (tpk_1, tpk_2)$ generated by an honest user $\mathcal{U}$, in addition to tpk itself, the adversary learns an ElGamal encryption $(g_1^t, tpk_1^t \cdot \sigma)$ of the BLS signature σ on m whenever an encrypted signature share request (tpk, m) is made to any server. We can prove that this leakage is harmless in several applications of vetBLS, including verifiably encrypted threshold IBE.

4 Verifiably Encrypted Threshold BLS

Our main vetKD construction will be based on the Boneh-Franklin IBE scheme [BF01]. As observed by Naor [BF01], the key derivation of any IBE scheme is also a signature scheme, which in the case of the Boneh-Franklin IBE scheme is the BLS signature scheme [BLS01].

At the core of our vetKD construction is therefore a verifiably encrypted threshold variant of the

BLS signature scheme. We define it here as a separate building block and give multiple instantiations that can be modularly plugged into our vetKD constructions. Before doing so, we recall some basic facts about BLS and threshold BLS signatures.

4.1 BLS and Threshold BLS Signatures

In the rest of this paper, we will assume that all algorithms, including adversarial ones, take a security parameter $\kappa \in \mathbb{N}$ as an implicit input.

We say that a function $f : \mathbb{N} \to \mathbb{R}$ is *polynomially bounded* if there exist $c, d \in \mathbb{R}$ such that for all $n \in \mathbb{N}$ we have $|f(n)| \leq n^c + d$, and we say that f is *negligible* if for all $c \in \mathbb{R}$ there exists $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ we have $|f(n)| < n^{-c}$.

We say that an algorithm is *efficient* if its running time is polynomially bounded in κ. Computational problems as well as security definitions are described as a game between two efficient algorithms, an experiment and an adversary, and the adversary is said to win the game if it produces an output that satisfies some condition defined by the experiment. The adversary's *advantage* is the probability that it wins the game as a function of the security parameter. We say that a computational problem is *hard* and that a scheme is *secure* according to a particular security definition if no efficient adversary exists that has non-negligible advantage in winning the associated game. Problems and schemes may involve mathematical structures such as groups; we see these as being generated by algorithms that take the security parameter as an input, so that one can define hardness and security in the same way.

A standard signature scheme $\mathcal{SS}$ consists of

- $\mathsf{KeyGen}() \to (pk, sk)$: a key generation algorithm that generates a public key pk and a matching secret key sk;
- $\mathsf{Sign}(sk, m) \to \sigma$: a signing algorithm that, on input the secret key and a message, produces a signature σ; and
- $\mathsf{Verify}(pk, m, \sigma) \to 0/1$: a verification algorithm that, given the public key, a message m, and a signature σ, outputs 0 or 1 indicating that the signature is rejected or accepted, respectively.

We consider the common security notion of existential unforgeability under chosen-message attack [GMR88] where the advantage of an adversary $\mathcal{A}$ is given by the probability that, on input an honestly generated public key pk and given access to a signing oracle $\mathsf{Sign}(sk, \cdot)$, $\mathcal{A}$ outputs a forgery (m^*, σ^*) so that $\mathsf{Verify}(pk, m^*, \sigma^*) = 1$ and it didn't query m^* from its signing oracle.

The $\mathcal{BLS}$ signature scheme [BLS04] was originally presented using a symmetric bilinear map (so-called *Type-1 pairing* [GPS06]); we use here the variant using an asymmetric bilinear map (so-called *Type-3 pairing*). Let $\mathrm{e} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_\mathrm{T}$ be a bilinear map where $\mathbb{G}_1 = (g_1)$, $\mathbb{G}_2 = (g_2)$, and $\mathbb{G}_\mathrm{T}$ are multiplicative groups of prime order q, and let $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1$ be a hash function, modeled as a random oracle [BR93]. The $\mathcal{BLS}$ signature scheme is given as follows:

- $\mathsf{KeyGen}()$: $sk \leftarrow_\$ \mathbb{Z}_q$, $pk \leftarrow g_2^{sk}$, return (pk, sk).
- $\mathsf{Sign}(sk, m)$: return $\sigma = \mathsf{H}(m)^{sk}$.
- $\mathsf{Verify}(pk, m, \sigma)$: return 1 if $\mathrm{e}(\sigma, g_2) = \mathrm{e}(\mathsf{H}(m), pk)$, otherwise return 0.

Definition 1 (Computational co-Diffie-Hellman Problem)**.** *The advantage of an algorithm $\mathcal{A}$ in solving the computational co-Diffie-Hellman (co-CDH) problem in $(\mathbb{G}_1, \mathbb{G}_2)$ is defined as*

$$\Pr\left[y = g_1^{\alpha\beta} \ : \ \alpha, \beta \leftarrow_\$ \mathbb{Z}_q \ , \ y \leftarrow_\$ \mathcal{A}(g_1^\alpha, g_1^\beta, g_2^\beta)\right] \ .$$

Theorem 1 ([BLS04, BS23])**.** *The $\mathcal{BLS}$ scheme is uf-cma secure in the random-oracle model if the co-CDH problem in $(\mathbb{G}_1, \mathbb{G}_2)$ is hard.*

A t-out-of-n threshold signature scheme $\mathcal{TS}$ consists of the following algorithms and protocols:

- $\mathsf{DKG}(n,t) \to (pk, pk_1, \ldots, pk_n, sk_1, \ldots, sk_n)$: a distributed key generation protocol, at the end of which each signer i obtains the public key pk, all signers' public key shares $pk_1, \ldots, pk_n$, and its own secret key share sk_i;
- $\mathsf{Sign}(sk_i, m) \to \sigma_i$: a signing algorithm that lets signer i sign a message m using its secret key share sk_i to produce a signature share σ_i;
- $\mathsf{SVerify}(pk_i, m, \sigma_i) \to 0/1$: a share verification algorithm that enables anyone to verify whether a signature share σ_i on message m was validly signed by signer i with public key share pk_i;
- $\mathsf{Combine}(pk, m, S, (pk_i, \sigma_i)_{i\in S}) \to \sigma$ a combination algorithm that takes as input the public key shares pk_i and signature shares σ_i of a set $S \subseteq [1,n]$ of signers with $|S| \geq t$, and produces a full signature σ;
- $\mathsf{Verify}(pk, m, \sigma) \to 0/1$: a verification algorithm that given the public key pk, message m, and full signature σ, checks whether the signature is valid.

The advantage of an adversary $\mathcal{A}$ in breaking the existential unforgeability against chosen-message attack (uf-cma) of $\mathcal{TS}$ is its probability in winning the following game. The adversary first outputs the corrupt signer indices $C \subset [1,n]$ with $|C| < t$. It then engages in the DKG protocol, where the adversary plays the role of the parties in C and the experiment plays the role of the honest parties $H = [1,n] \setminus C$. At the end of the DKG protocol, all honest parties output the public key pk, public key shares $pk_1, \ldots, pk_n$, and each honest party $i \in H$ ends up with its own secret key share sk_i.

The experiment then runs $\mathcal{A}$ while giving it access to a signing oracle that, on input a signer index $i \in H$ and a message m, returns $\sigma_i \leftarrow_{\$} \mathsf{Sign}(sk_i, m)$. Eventually, the adversary outputs its forgery (m^*, σ^*). It is said to win the game if $\mathsf{Verify}(pk, m^*, \sigma^*) = 1$ and it never queried its signing oracle for a signature on m^*.

The threshold BLS scheme $t\mathcal{BLS}$ by Boldyreva [Bol03] is the natural application of Shamir secret sharing [Sha79] to the secret keys of the $\mathcal{BLS}$ scheme. The basic principle is that the public key $pk = g_2^{sk}$ for a random $sk \in \mathbb{Z}_q$, while the secret key shares are determined by a random polynomial $\omega(X) = sk + \omega_1 \cdot X + \ldots + \omega_{t-1} \cdot X^{t-1} \in \mathbb{Z}_q[X]$ so that each signer i is given a secret key share $sk_i = \omega(i)$, with its public key share given by $pk_i = g_2^{sk_i}$.

Using Lagrange interpolation, one can rewrite $\omega(X)$ from the images of t points $S \subset \mathbb{Z}_q$ as $\omega(X) = \sum_{i\in S} \Lambda_{i,S}(X) \cdot \omega(i)$, where $\Lambda_{i,S}(X) = \prod_{j\in S\setminus\{i\}} \frac{X-j}{i-j}$ are the Lagrange basis polynomials. It therefore holds for all $S \subseteq [1,n]$ with $|S| \geq t$ that $sk = \sum_{i\in S} \Lambda_{i,S}(0) \cdot sk_i$.

The threshold BLS scheme $t\mathcal{BLS}$ is described as follows:

- $\mathsf{DKG}(n,t)$: For simplicity, we assume that a trusted dealer chooses a random polynomial $\omega(X) \leftarrow_{\$} \mathbb{Z}_q[X]$ of degree $t-1$, sets $pk = g_2^{\omega(0)}$ and $(pk_i, sk_i) = (g_2^{\omega(i)}, \omega(i))$ for $i \in [1,n]$, and hands $(pk, pk_1, \ldots, pk_n, sk_i)$ to each signer i.
- $\mathsf{Sign}(sk_i, m)$: Return $\sigma_i = \mathsf{H}(m)^{sk_i}$.
- $\mathsf{SVerify}(pk_i, i, m, \sigma_i)$: Return 1 if $\mathrm{e}(\sigma_i, g_2) = \mathrm{e}(\mathsf{H}(m), pk_i)$, otherwise return 0.
- $\mathsf{Combine}(pk, m, S, (pk_i, \sigma_i)_{i\in S})$: return $\sigma \leftarrow \prod_{i\in S} \sigma_i^{\Lambda_{i,S}(0)}$.
- $\mathsf{Verify}(pk, m, \sigma)$: return 1 if $\mathrm{e}(\sigma, g_2) = \mathrm{e}(\mathsf{H}(m), pk)$, otherwise return 0.

Theorem 2 ([Bol03, BS23])**.** *The $t\mathcal{BLS}$ scheme is uf-cma secure if $\mathcal{BLS}$ is uf-cma secure.*

4.2 The vetBLS Ideal Functionality

We analyze the security of our vetKD scheme in Canetti's universal composability (UC) framework [Can01]. Here, protocols are proved secure by showing that no efficient environment $\mathcal{E}$ can with non-negligible advantage distinguish whether it's running in a real-world execution with a real-world protocol π and a real-world adversary $\mathcal{A}$, or in an ideal-world execution with an ideal functionality $\mathcal{F}$ and a simulator $\mathcal{S}im$. In both worlds, the environment provides the inputs and receives the outputs of all honest parties. In the real world, the honest parties process their inputs as described by the protocol π, possibly communicating with other parties over a network that is completely controlled by $\mathcal{A}$. In the ideal world, the ideal functionality $\mathcal{F}$ acts as a central trusted third party that processes inputs and delivers outputs to honest and corrupt parties alike. Secure composition of protocols is modeled as a hybrid execution in which all the parties in the real-world protocol have access to an ideal sub-functionality $\mathcal{F}'$.

In Figure 2, we present a helper functionality for our vetKD construction, the verifiably encrypted threshold BLS functionality $\mathcal{F}_{\text{vetbls}}$. It models a set of servers $\mathcal{S}_i$ that jointly generate a BLS key pair and can jointly create encrypted BLS signatures if at least one honest server agrees to sign the same message m encrypted under tpk, where tpk are users' transport keys. (We must of course assume that the $t-1$ corrupt servers can always sign any messages they want, hence a single participating honest server enables the creation of an encrypted signature.) The actual BLS signature σ can only be recovered by the user who generated tpk.

The main guarantees that the $\mathcal{F}_{\text{vetbls}}$ functionality upholds are that

1. participation of at least one honest server is required to create a valid encrypted signature for an honest user's tpk,
2. an encrypted signature on m under an honest user's tpk does not reveal the BLS signature σ on m to the adversary,
3. the only way for the adversary to obtain a BLS signature on m is by having at least one honest server participate in an encrypted signature on m under a non-honest tpk,
4. and when the honest user who generated tpk decrypts an encrypted signature es that is valid for m and tpk, it recovers the BLS signature on m.

In designing the $\mathcal{F}_{\text{vetbls}}$ functionality in Figure 2, we made the following choices and assumptions:

- The functionality is parameterized with multiplicative groups $\mathbb{G}_1$ and $\mathbb{G}_2$ generated by g_1 and g_2, respectively. It keeps a key pair (pk, sk) in its internal state, as well as a map H initialized as $H[\cdot] = \perp$ to keep track of hash responses, a map TPK initialized as $TPK[\cdot] = \emptyset$ to keep track of honest users' public keys, and initially empty sets ES and V to keep track of created and verified encrypted signatures, respectively. We use the abbreviated notation $tpk \in TPK$ to denote $\exists\, \mathcal{U} : tpk \in TPK[\mathcal{U}]$.
- As prescribed by the UC framework, each input includes the session identifier sid of the protocol or functionality instance that it is addressed to. We assume that the set of all server identities $\mathcal{S}_i$ can somehow be derived from the session identifier sid as a function servers(sid). The server identities could for example be statically encoded in sid, or they could be obtained from the governance system of a blockchain. We assume the set of servers to be static, so the total number of servers is fixed as $n(sid) = |\text{servers}(sid)|$. We also assume that each server $\mathcal{S}_i$ is assigned a unique identifier $i \in [1, n(sid)]$ and that the threshold of servers that needs to be involved in a signing query is fixed as $t(sid)$. For brevity, we will usually refer to $n(sid)$ and $t(sid)$ simply as n and t.
- We assume static corruption of $t-1$ servers and an arbitrary number of other parties. At the beginning of the experiment, the adversary $\mathcal{A}$ outputs the identities of the parties that it

Functionality $\mathcal{F}_{\text{vetbls}}$

- On $(sid, \texttt{"init"})$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  If (pk, sk) aren't defined, choose $sk \leftarrow_\$ \mathbb{Z}_q$, compute $pk \leftarrow g_2^{sk}$, and store (pk, sk). Send $(\texttt{"init"}, \mathcal{S}_i, pk)$ to $\mathcal{Sim}$.
- On $(sid, \texttt{"output-pk"}, \mathcal{S}_i)$ from $\mathcal{Sim}$:
  If $\mathcal{S}_i \in \text{servers}(sid)$, output $(sid, \texttt{"output-pk"}, pk)$ to $\mathcal{S}_i$.
- On $(sid, \texttt{"hash"}, m)$ from $\mathcal{P}$:
  If $H[m] = \perp$, choose $H[m] \leftarrow_\$ \mathbb{G}_1$. Output $(sid, \texttt{"hash"}, m, H[m])$ to $\mathcal{P}$.
- On $(sid, \texttt{"transport-keygen"})$ from honest $\mathcal{U}$:
  Send a message $(\texttt{"transport-keygen"}, \mathcal{U})$ to $\mathcal{Sim}$ and wait for a response tpk from $\mathcal{Sim}$. Add tpk to $TPK[\mathcal{U}]$ and output $(sid, \texttt{"tpk"}, tpk)$ to $\mathcal{U}$.
- On $(sid, \texttt{"encsign"}, m, tpk)$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  If $tpk \in TPK$, add (m, tpk) to ES and send $(\texttt{"encsign"}, m, tpk, \mathcal{S}_i)$ to $\mathcal{Sim}$. Otherwise, add $(m, \perp)$ to ES, simulate an internal input $(sid, \texttt{"hash"}, m)$, compute $\sigma \leftarrow H[m]^{sk}$, and send $(\texttt{"encsign"}, m, tpk, \mathcal{S}_i, \sigma)$ to $\mathcal{Sim}$.
- On $(sid, \texttt{"output-encsig"}, m, tpk, \mathcal{S}_i, es)$ from $\mathcal{Sim}$:
  Simulate an internal input $(sid, \texttt{"verify"}, pk, m, tpk, es)$. If $(pk, m, tpk, es, \texttt{true}) \in V$ then output $(sid, \texttt{"encsign"}, m, tpk, es)$ to $\mathcal{S}_i$.
- On $(sid, \texttt{"verify"}, pk', m, tpk, es)$ from $\mathcal{P}$:
  Send $(\texttt{"verify"}, pk', m, tpk, es)$ to $\mathcal{Sim}$ and wait for a response β from $\mathcal{Sim}$. Add (pk', m, tpk, es, b) to V and output $(sid, \texttt{"verify"}, pk', m, tpk, es, b)$ to $\mathcal{P}$, where b is determined as follows:
  1. If $(pk', m, tpk, es, \gamma) \in V$, set $b \leftarrow \gamma$.
  2. Else, if $pk' \neq pk$, set $b \leftarrow \beta$.
  3. Else, if $tpk \notin TPK$ or $(m, tpk) \in ES$ or $(m, \perp) \in ES$, set $b \leftarrow \beta$.
  4. Else, set $b \leftarrow \texttt{false}$.
- On $(sid, \texttt{"decrypt"}, pk', m, tpk, es)$ from honest $\mathcal{U}$:
  Simulate inputs $(sid, \texttt{"verify"}, pk', m, tpk, es)$ and $(sid, \texttt{"hash"}, m)$. If $pk' \neq pk$ then send $(\texttt{"decrypt"}, pk', m, tpk, es)$ to $\mathcal{Sim}$, wait for σ from $\mathcal{Sim}$, and output $(sid, \texttt{"decrypt"}, pk', m, tpk, es, \sigma)$ to $\mathcal{U}$. Else, if $tpk \in TPK[\mathcal{U}]$ and $(m, tpk, es, \texttt{true}) \in V$, compute $\sigma \leftarrow H[m]^{sk}$ and output $(sid, \texttt{"decrypt"}, m, tpk, es, \sigma)$ to $\mathcal{U}$.

Figure 2: The verifiably encrypted threshold BLS functionality $\mathcal{F}_{\text{vetbls}}$.

wants to corrupt. From then on, $\mathcal{A}$ controls these parties, sees all their inputs and outputs, and can make them arbitrarily deviate from the protocol.

- We assume that the environment lets at least one honest server call the `"init"` interface before it makes any calls to the `"output-pk"`, `"encsign"`, `"verify"`, and `"decrypt"` interfaces.

- A crucial aspect of the $\mathcal{F}_{\text{vetbls}}$ functionality is that the secret key *sk* and any signatures delivered to honest parties remain hidden from the simulator. To issue the BLS signatures themselves, the $\mathcal{F}_{\text{vetbls}}$ functionality therefore does not follow the common approach of generic signature functionalities where the simulator hands the signatures [Can04] or the signing algorithm [Fis06] to the functionality. Instead, the $\mathcal{F}_{\text{vetbls}}$ functionality follows the approach of Groth-Shoup [GS22] by internally generating the secret key *sk* and computing all signatures $\mathsf{H}(m)^{sk}$, hidden from the simulator's view. It does, however, consult the simulator to produce encrypted signatures and users' transport keys, as these are meant to be visible to the adversary.

- The `"hash"` interface of the $\mathcal{F}_{\text{vetbls}}$ functionality behaves like an "internal random oracle", assigning random group elements to incoming messages. An external random-oracle functionality would not have worked here, as the simulator in the random-oracle-hybrid world would be able to program the random oracle so that it knows the corresponding BLS signatures, which we explicitly wanted to avoid. Modeling the hash function as a real-world function H would actually work for stand-alone applications of $\mathcal{F}_{\text{vetbls}}$, but would not have allowed the single-key composability of protocols in Section 8.

- Users can generate a transport public key *tpk* by invoking the `"transport-keygen"` interface; note that one user can generate multiple transport keys.

- Individual servers express their explicit agreement to create a BLS signature for m encrypted under *tpk* by providing an input $(sid, \texttt{"encsign"}, m, tpk)$ to the functionality. How *tpk* is transmitted from the user to the server is outside of the model of $\mathcal{F}_{\text{vetbls}}$; in a typical blockchain usage, it could be included in a user-signed transaction that appears on the blockchain.

- When an honest server agrees to create an encrypted signature for m under a *tpk* that is not controlled by an honest user, the $\mathcal{F}_{\text{vetbls}}$ functionality assumes that *tpk* is adversarially controlled and leaks the BLS signature on m to the simulator. When it agrees to do so under an honest user's *tpk*, however, nothing is leaked to the simulator.

- The verification and decryption interfaces do not assume that all participants have an authentic copy of the public key *pk*. Rather, they are called with an explicit public key pk'; any security guarantees only hold for $pk' = pk$, however.

- The verification interface enforces consistency, meaning that verifying the same encrypted signature for the same pk', the same message and the same *tpk* will always return the same result, as well as unforgeability, in the sense that the only way for the simulator to create a valid signature under *pk* for m and an honest user's *tpk* is if at least one honest server participated in an encrypted signing of m, either under *tpk*, or under a key that is not registered to an honest user. This is also in line with the game-based unforgeability definition of (non-threshold) verifiably encrypted signatures [BGLS03] that excludes trivial forgeries where m was previously queried to either a signing oracle or a decryption oracle.

- When an honest user uses the `"decrypt"` interface to decrypt a valid encrypted signature under *pk* for m under its own *tpk*, it always recovers the BLS signature on m. Note that the BLS signature is delivered in an output directly to the user, without its value being leaked to the simulator.

Functionality $\mathcal{F}_{\text{mca}}$

- On $(sid, \texttt{"register"}, v)$ from $\mathcal{P}$:
  Send $(sid, \texttt{"register"}, v, \mathcal{P})$ to $\mathcal{Sim}$ and wait for $\texttt{"ok"}$ from $\mathcal{Sim}$. If a record $(\mathcal{P}, v')$ exists then ignore, otherwise create a record $(\mathcal{P}, v)$.
- On $(sid, \texttt{"retrieve"}, \mathcal{P}')$ from $\mathcal{P}$:
  Send $(sid, \texttt{"retrieve"}, \mathcal{P}', \mathcal{P})$ to $\mathcal{Sim}$ and wait for $\texttt{"ok"}$ from $\mathcal{Sim}$. If a record $(\mathcal{P}, v)$ exists, output $(sid, \texttt{"retrieve"}, \mathcal{P}', v)$ to $\mathcal{P}$, otherwise output $(sid, \texttt{"retrieve"}, \mathcal{P}', \perp)$ to $\mathcal{P}$.

Figure 3: The multi-party certification authority ideal functionality $\mathcal{F}_{\text{mca}}$ based on Canetti's single-party version $\mathcal{F}_{\text{ca}}$ [Can04].

Functionality $\mathcal{F}_{\text{dkg}}$

- On $(sid, \texttt{"init"})$ from $\mathcal{S}_i$:
  If $\mathcal{S}_i \notin \text{servers}(sid)$ then ignore. If this is the first honest server in servers(sid) calling $\texttt{"init"}$, then choose a random polynomial $\omega(X) \leftarrow_{\$} \mathbb{Z}_q[X]$ of degree $t(sid)-1$, set $sk \leftarrow \omega(0)$ and $pk \leftarrow g_2^{sk}$, compute $pk_i \leftarrow g_2^{\omega(i)}$ for $i \in [1, n]$ where $n = |\text{servers}(sid)|$, and record $(\texttt{"init"}, pk, (pk_1, \ldots, pk_n), sk, \omega)$. Else, look up the record $(\texttt{"init"}, pk, (pk_1, \ldots, pk_n), sk, \omega)$. Send $(\texttt{"init"}, \mathcal{S}_i, pk, (pk_1, \ldots, pk_n))$ to $\mathcal{Sim}$.
- On $(sid, \texttt{"output-share"}, \mathcal{S}_i)$ from $\mathcal{Sim}$:
  If no record $(\texttt{"init"}, pk, (pk_1, \ldots, pk_n), sk, \omega)$ exists, then ignore. Else, compute $sk_i \leftarrow \omega(i) \bmod q$ and output $(sid, \texttt{"init"}, pk, (pk_1, \ldots, pk_n), \omega(i))$ to $\mathcal{S}_i$.

Figure 4: The distributed key generation functionality $\mathcal{F}_{\text{dkg}}$.

4.3 A Simple Construction

We first present a simple instantiation of $\mathcal{F}_{\text{vetbls}}$ called $\pi_{\text{vetbls-sim}}$ where a user's transport key pair is a standard public-key encryption (PKE) key pair, and where each server $\mathcal{S}_i$ simply encrypts its BLS signature share $\mathsf{H}(m)^{sk_i}$ under the transport public key and signs the resulting ciphertext. An encrypted signature consists of $2t-1$ such signed ciphertexts from different signers; given that at least t of these must be honest, the owner of the transport key can decrypt the individual signature shares, find a subset of t valid ones, and combine them into a full signature.

The details of protocol $\pi_{\text{vetbls-sim}}$ are given in Figure 5. Apart from a public-key encryption scheme $\mathcal{PKE}$ and a standard signature scheme $\mathcal{SS}$, it also assumes a multi-party certification authority ideal functionality $\mathcal{F}_{\text{mca}}$, described in Figure 3 as a variant of Canetti's single-party functionality $\mathcal{F}_{\text{ca}}$ [Can04], as well as a distributed key generation functionality $\mathcal{F}_{\text{dkg}}$ described in Figure 4. The latter distributes Shamir secret shares [Sha79] of the BLS secret key to the servers and certifies the resulting public key and public key shares. Distributed key generation is a research topic on its own; instantiations under various settings and assumptions exist in the literature [Fel87, Ped91, CKLS02, AF04, DYX+22, Gro21] and can be plugged into our schemes.

The `"encsign"` interface lets servers send their encrypted signature shares to a *combiner* who collects, verifies, and combines them, and sends the result back to the servers. The combiner does not need to be trusted; its role could be played by one or a subset of the servers, or by an additional party. In a typical blockchain scenario, the block maker will act as a combiner by including the encrypted signature in a block.

It is important to note that the $\pi_{\text{vetbls-sim}}$ protocol is only able to produce BLS signatures if

Protocol $\pi_{\text{vetbls-sim}}$

- On input $(sid, \texttt{"init"})$, $\mathcal{S}_i$ provides input $((sid, \texttt{"dkg"}), \texttt{"init"})$ to $\mathcal{F}_{\text{dkg}}$ and waits for an output $((sid, \texttt{"dkg"}), \texttt{"init"}, pk, (pk_1, \ldots, pk_n), sk_i)$. It also generates $(spk_i, ssk_i) \leftarrow_\$ \mathcal{SS}.\mathsf{KeyGen}()$ and registers spk_i with $\mathcal{F}_{\text{mca}}$. It stores pk, sk_i, and ssk_i in its local state and outputs $(sid, \texttt{"output-pk"}, (pk, (pk_1, \ldots, pk_n))$.
- On $(sid, \texttt{"hash"}, m)$, $\mathcal{P}$ returns $(sid, \texttt{"hash"}, m, \mathsf{H}(m))$.
- On $(sid, \texttt{"transport-keygen"})$, $\mathcal{U}$ generates a transport encryption key pair $(tpk, tsk) \leftarrow_\$ \mathcal{PKE}.\mathsf{KeyGen}()$, stores tsk in its local state, and outputs $(sid, \texttt{"tpk"}, tpk)$.
- On $(sid, \texttt{"encsign"}, m, tpk)$, $\mathcal{S}_i$ computes $\sigma_i \leftarrow \mathsf{H}(m)^{sk_i}$, $C_i \leftarrow_\$ \mathcal{PKE}.\mathsf{Enc}(tpk, \sigma_i)$, and $\sigma'_i \leftarrow \mathcal{SS}.\mathsf{Sign}(ssk_i, (pk, m, tpk, C_i))$, and sends $es_i \leftarrow (C_i, \sigma'_i)$ to a combiner.

  When a combiner receives this message, it retrieves spk_i for $\mathcal{S}_i$ from $\mathcal{F}_{\text{mca}}$ and verifies that $\mathcal{SS}.\mathsf{Verify}(spk_i, (pk, m, tpk, C_i), \sigma'_i) = 1$. As soon as it received validly signed encrypted signature shares from $2t - 1$ different servers $\mathcal{S}_i$, $i \in S$, it compiles $es \leftarrow (S, (C_i, \sigma'_i)_{i \in S})$ and sends (m, tpk, es) to all servers.

  When $\mathcal{S}_i$ receives this message, it verifies that $|S| = 2t - 1$ and that $\mathcal{SS}.\mathsf{Verify}(spk_i, (pk, m, tpk, C_i), \sigma'_i) = 1$ for all $i \in S$, where it retrieves spk_i for $\mathcal{S}_i$ from $\mathcal{F}_{\text{mca}}$. If all of these tests pass, it outputs $(sid, \texttt{"encsign"}, m, tpk, es)$.
- On $(sid, \texttt{"verify"}, pk', m, tpk, es)$, $\mathcal{P}$ parses es as $(S, (C_i, \sigma'_i)_{i \in S})$ and pk' as $(pk, \cdot)$. It verifies that $|S| = 2t - 1$ and checks that $\mathcal{SS}.\mathsf{Verify}(spk_i, (pk, m, tpk, C_i), \sigma'_i) = 1$ for all $i \in S$, where it retrieves spk_i for $\mathcal{S}_i$ from $\mathcal{F}_{\text{mca}}$. If all of these tests pass, it sets $result \leftarrow \texttt{true}$, otherwise it sets $result \leftarrow \texttt{false}$. It outputs $(sid, \texttt{"verify"}, pk', m, tpk, es, result)$.
- On $(sid, \texttt{"decrypt"}, pk', m, tpk, es)$, $\mathcal{U}$ parses es as $(S, (C_i, \sigma'_i)_{i \in S})$ and pk' as $(pk, (pk_1, \ldots, pk_n))$. It looks up tsk from its local state, decrypts $\sigma_i \leftarrow \mathcal{PKE}.\mathsf{Dec}(tsk, C_i)$ for all $i \in S$, and finds a subset $S' \subseteq S$ of size $|S'| = t$ such that $\mathrm{e}(\sigma_i, g_2) = \mathrm{e}(\mathsf{H}(m), pk_i)$ for all $i \in S'$, where it retrieves spk_i for $\mathcal{S}_i$ from $\mathcal{F}_{\text{mca}}$. If it finds such a set, $\mathcal{U}$ reconstructs $\sigma \leftarrow \prod_{i \in S'} \sigma_i^{\Lambda_{i,S'}(0)}$ and outputs $(sid, \texttt{"decrypt"}, m, tpk, es, \sigma)$.

Figure 5: The simple vetBLS protocol $\pi_{\text{vetbls-sim}}$ based on a public-key encryption scheme $\mathcal{PKE}$, a standard signature scheme $\mathcal{SS}$, a hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1$, the certification authority ideal functionality $\mathcal{F}_{\text{mca}}$ and a distributed key generation ideal functionality $\mathcal{F}_{\text{dkg}}$.

$n \geq 2t - 1$, i.e., $t \leq \lceil \frac{n}{2} \rceil$. One could consider an optimistic version of the protocol where, if after a certain timeout the combiner received less than $2t - 1$ but at least t encrypted shares, the combiner includes all shares that it received in the hope that t of them are valid. Such a protocol, however, would lose the guarantee that a valid encrypted signature can always be decrypted by the user, so would not securely realize the $\mathcal{F}_{\text{vetbls}}$ functionality.

The protocol relies on the standard definition of ind-cpa security [GM84] for a public-key encryption scheme $\mathcal{PKE}$ consisting of a key generation algorithm $(pk, sk) \leftarrow_\$ \mathsf{KeyGen}()$, an encryption algorithm $C \leftarrow_\$ \mathsf{Enc}(pk, m)$, and a decryption algorithm $m \leftarrow \mathsf{Dec}(sk, C)$. The adversary $\mathcal{A}$ is given a fresh public key pk as input. At some point, $\mathcal{A}$ outputs two challenge messages m_0, m_1, upon which the experiment chooses $b \leftarrow_\$ \{0, 1\}$ and sends the challenge ciphertext $C^* \leftarrow_\$ \mathsf{Enc}(pk, m_b)$ back to $\mathcal{A}$. The adversary outputs a bit b'. Its advantage in breaking the ind-cpa security is defined as the difference in its probability of outputting $b' = 1$ when $b = 1$ and outputting $b' = 0$ when $b = 0$.

It is worth pointing out that the theorem below doesn't involve any assumptions related to the security of BLS signatures themselves. This is of course because, as stated earlier, the $\mathcal{F}_{\text{vetbls}}$ functionality doesn't impose any unforgeability or other guarantees on the BLS signatures themselves.

Theorem 3. *If $\mathcal{SS}$ is uf-cma secure, $\mathcal{PKE}$ is ind-cpa secure, and H is modeled as a random oracle, then $\pi_{\textit{vetbls-sim}}$ securely realizes $\mathcal{F}_{\text{vetbls}}$ in the $(\mathcal{F}_{\text{dkg}}, \mathcal{F}_{\text{mca}})$-hybrid model.*

Proof. We prove the theorem by observing the following sequence of games:

Game 0: The real execution of $\mathcal{E}$ and $\mathcal{A}$ with $\pi_{\text{vetbls-sim}}$ in the $\mathcal{F}_{\text{mca}}$ and $\mathcal{F}_{\text{dkg}}$-hybrid world.

Game 1: The simulator $\mathcal{S}im$ takes over the execution of $\mathcal{A}$, $\mathcal{F}_{\text{mca}}$, $\mathcal{F}_{\text{dkg}}$, and all honest parties $\mathcal{P}$ by simply relaying messages to and from $\mathcal{E}$ as the real-world experiment would do. We refer to the simulated executions as "$\mathcal{A}$", "$\mathcal{F}_{\text{mca}}$", "$\mathcal{F}_{\text{dkg}}$", and "$\mathcal{P}$", respectively. The change is purely conceptual, so $\mathcal{E}$'s view is identical to that in $\mathcal{G}_0$.

Game 2: The simulator aborts whenever an honest party receives an encrypted signature es containing a ciphertext C_i and signature σ'_i attributed to an honest server $\mathcal{S}_i$ with a valid signature $\mathcal{SS}.\mathsf{Verify}(spk_i, (pk, m, tpk, C_i), \sigma'_i) = 1$, but "$\mathcal{S}_i$" never produced C_i on an input $(sid, \texttt{"encsign"}, m, tpk)$. Any environment causing $\mathcal{G}_3$ to abort easily gives rise to a uf-cma forger against $\mathcal{SS}$.

Note that it now also impossible for an honest party to successfully verify an encrypted signature es for m and tpk, even though no honest server ever participated in the creation of an encrypted signature on m under tpk.

Game 3: Rather than decrypting the individual ciphertexts in an encrypted signature es and verifying and interpolating the resulting signature shares, an honest user "$\mathcal{U}$" who generated tpk and receives a valid encrypted signature es for m and tpk immediately outputs the full BLS signature $\mathsf{H}(m)^{sk}$. Since there must be at least t honest servers in S, since those servers encrypted their correct BLS signature shares, and since all honest servers' ciphertexts in es were actually created by those honest servers, interpolation is guaranteed to yield a correct BLS signature. This change is therefore purely conceptual.

Game 4: When an honest server "$\mathcal{S}_i$" encrypts a signature share σ_i under a transport public key tpk generated by an honest user "$\mathcal{U}$", the simulator now encrypts g_1 instead. When "$\mathcal{U}$" has to decrypt that exact ciphertext, it uses σ_i as the outcome, without actually decrypting. A simple hybrid argument over the affected ciphertexts can be used to show that any environment distinguishing this game from the previous one can be used to break the ind-cpa security of $\mathcal{PKE}$. Essentially, the reduction would use its challenge public key as tpk for $\mathcal{U}$, use σ_i and g_1 as its challenge messages, and use its challenge ciphertext as encryption of σ_i. Note that

$\mathcal{U}$ never needs to decrypt incoming ciphertexts, as since the previous game, simulated honest users don't try to decrypt but, if *es* is valid, directly output the BLS signature.

Game 5: Instead of executing the real code of $\mathcal{F}_{\text{dkg}}$, the simulator lets "$\mathcal{F}_{\text{dkg}}$" generate $sk \leftarrow_\$ \mathbb{Z}_q$ and compute $pk \leftarrow g_2^{sk}$. It also chooses secret key shares $sk_i \leftarrow_\$ \mathbb{Z}_q$ for all $t-1$ corrupt servers $\mathcal{S}_i$, $i \in S$ and computes their public key shares as $pk_i \leftarrow g_2^{sk_i}$. For all honest $\mathcal{S}_i$, it computes $pk_i \leftarrow pk^{\Lambda_{0,S'}(i)} \cdot \prod_{j\in S} pk_j^{\Lambda_{j,S'}(i)}$, where $S' = S \cup \{0\}$.

Whenever an honest server $\mathcal{S}_i$ has to encrypt a signature share under a *tpk* that was not generated by an honest user, it first computes $\sigma \leftarrow \mathsf{H}(m)^{sk}$ and then computes the share as $\sigma_i \leftarrow \sigma^{\Lambda_{0,S'}(i)} \cdot \prod_{j\in S} \mathsf{H}(m)^{sk_j \cdot \Lambda_{j,S'}(i)}$.

It is clear that the t points $(0, sk)$ and (i, sk_i) for $i \in S$ implicitly define a unique polynomial $\omega(X) \in \mathbb{Z}_q[X]$ of degree $t-1$, and that by Lagrange interpolation $pk_i = g_2^{\omega(i)}$ and $\sigma_i = \mathsf{H}(m)^{\omega(i)}$ as in the real game. This change is therefore purely conceptual.

Note that the simulation of honest servers "$\mathcal{S}_i$" in Game 5 no longer depends on the secret key share sk_i. When encrypting to a *tpk* generated by an honest user, it encrypts a string of zeroes; when encrypting to any other *tpk*, it encrypts a signature share σ_i that is computed from $\sigma = \mathsf{H}(m)^{sk}$ and the secret key shares sk_i of corrupt servers. Also note that this is the only point where the simulator uses *sk* at all.

We can therefore easily turn Game 5 into a simulator $\mathcal{S}im$ that interacts with $\mathcal{E}$ and $\mathcal{F}_{\text{vetbls}}$ in the ideal world, as follows:

- On (`"init"`, $\mathcal{S}_i, pk$) from $\mathcal{F}_{\text{vetbls}}$, $\mathcal{S}im$ uses *pk* as the output of $\mathcal{F}_{\text{dkg}}$, chooses random secret key shares sk_i for all corrupt servers, and computes the public key shares as in Game 5.

- It "outsources" the random oracle $\mathsf{H}(\cdot)$ to the `"hash"` interface of $\mathcal{F}_{\text{vetbls}}$, which also implements a random oracle with range $\mathbb{G}_1$. This is the reason for the condition in Theorem 3 that H be modeled as a random oracle: the `"hash"` interface essentially implements a random oracle, so in order to substitute it for H, we have to model H as a random oracle.

- On (`"transport-keygen"`, $\mathcal{U}$) from $\mathcal{F}_{\text{vetbls}}$, it runs the honest simulated user "$\mathcal{U}$" on (sid, TKG) to obtain (sid, `"tpk"`, tpk), and sends (`"tpk"`, $\mathcal{U}, tpk$) to $\mathcal{F}_{\text{vetbls}}$.

- On (`"encsign"`, $m, tpk, \mathcal{S}_i$) from $\mathcal{F}_{\text{vetbls}}$, $\mathcal{S}im$ lets "$\mathcal{S}_i$" encrypt g_1 instead of the real signature share, as done in Game 4. Note that this ciphertext never needs to be decrypted by "$\mathcal{U}$", because `"decrypt"` inputs are handled locally by $\mathcal{F}_{\text{vetbls}}$, without intervention from $\mathcal{S}im$.

- On (`"encsign"`, $m, tpk, \mathcal{S}_i, \sigma$) from $\mathcal{F}_{\text{vetbls}}$, $\mathcal{S}im$ lets "$\mathcal{S}_i$" recompute σ_i from σ and the corrupt servers' secret key shares sk_i using Lagrange interpolation as in Game 5, encrypt it under *tpk*, and sign it.

- When an honest "$\mathcal{S}_i$" outputs (sid, `"encsign"`, m, tpk, es), $\mathcal{S}im$ sends (sid, `"output-encsig"`, $m, tpk, \mathcal{S}_i, es$) to $\mathcal{F}_{\text{vetbls}}$.

- On (`"verify"`, pk', m, tpk, es) from $\mathcal{F}_{\text{vetbls}}$, $\mathcal{S}im$ returns the result of calling the `"verify"` interface of $\pi_{\text{vetbls-sim}}$ with pk', m, tpk, es. Any forgeries, i.e., signatures deemed valid by $\pi_{\text{vetbls-sim}}$ but not by $\mathcal{F}_{\text{vetbls}}$, have been ruled out in Game 2.

- When "$\mathcal{U}$" receives a valid encrypted signature *es* to decrypt, it invokes the `"decrypt"` interface on $\mathcal{F}_{\text{vetbls}}$ to obtain the signature σ and outputs σ, exactly as in Game 3.

□

4.4 Construction with Zero-Knowledge Proofs

We now present a second scheme that only requires contributions from t servers to create an encrypted signature. An encrypted signature contains t BLS signature shares that are ElGamal-encrypted [ElG84] under the user's public key in $\mathbb{G}_1$, together with a zero-knowledge proof that they are indeed valid ElGamal-encrypted signature shares. Details of the $\pi_{\text{vetbls-zkp}}$ scheme are given in Figure 6.

Depending on the standard encryption and signature schemes used in the $\pi_{\text{vetbls-sim}}$ scheme, the gains in terms of signature length of $\pi_{\text{vetbls-zkp}}$ with respect to $\pi_{\text{vetbls-sim}}$ are probably offset by the longer size per share due to the ElGamal randomness and the zero-knowledge proof. However, waiting for the combiner to collect t signature shares instead of $2t-1$ can significantly reduce latency in many settings. Apart from that, the $\pi_{\text{vetbls-zkp}}$ scheme obviously has the advantage of supporting any threshold $t \leq n$, instead of being restricted to $t \leq \frac{n+1}{2}$.

Because we're using ElGamal encryption in $\mathbb{G}_1$, we have to rely on the external Diffie-Hellman (XDH) assumption [Sco02, BBS04, BGdMM05], which states that the decisional Diffie-Hellman (DDH) problem is hard in $\mathbb{G}_1$.

Definition 2 (External Diffie-Hellman Problem)**.** *The advantage of an algorithm $\mathcal{A}$ in solving the external Diffie-Hellman (XDH) problem in $\mathbb{G}_1$ is defined as*

$$\Big| \Pr\Big[b=1 \ : \ \alpha,\beta \leftarrow_\$ \mathbb{Z}_q \ , \ b \leftarrow_\$ \mathcal{A}(g_1^\alpha, g_1^\beta, g_1^{\alpha\beta})\Big] \\ - \Pr\Big[b=1 \ : \ \alpha,\beta,\gamma \leftarrow_\$ \mathbb{Z}_q \ , \ b \leftarrow_\$ \mathcal{A}(g_1^\alpha, g_1^\beta, g_1^{\gamma})\Big]\Big| \ .$$

Theorem 4. *If the XDH assumption holds in $\mathbb{G}_1$, the $\mathcal{BLS}$ signature scheme is uf-cma secure, and H, H' are modeled as random oracles, then the $\pi_{\text{vetbls-zkp}}$ protocol securely realizes $\mathcal{F}_{\text{vetbls}}$ in the $\mathcal{F}_{\text{dkg}}$-hybrid model.*

Proof. We prove the theorem by observing the following sequence of games:

Game 0: The real execution of $\mathcal{E}$ and $\mathcal{A}$ with $\pi_{\text{vetbls-zkp}}$ in the $\mathcal{F}_{\text{dkg}}$-hybrid world.

Game 1: The simulator $\mathcal{S}im$ takes over the execution of $\mathcal{A}$, $\mathcal{F}_{\text{dkg}}$, and all honest parties $\mathcal{P}$. This change is purely conceptual.

Game 2: *(Soundness of the zero-knowledge proof.)* When a simulated honest user "$\mathcal{U}$" has to decrypt an encrypted signature es, it no longer decrypts the individual shares and verifies the combined result as in the real protocol, but instead it verifies es as is done in the `"verify"` interface of $\pi_{\text{vetbls-zkp}}$, by checking that it contains encrypted shares from t different servers with valid zero-knowledge proofs according to Equation (2). If so, it outputs $(sid, \texttt{"decrypt"}, m, tpk, es, \sigma)$ straight away, where $\sigma \leftarrow \mathsf{H}(m)^{sk}$ if the verification was successful and $\sigma \leftarrow \perp$ if not. Note that this change implies that the decryption of a valid signature by the correct user always yields a correct BLS signature.

The only way for this game to be any different from the previous one is if es contains a valid zero-knowledge proof (c, s_1, s_2) for a ciphertext (C_1, C_2) that is not an encryption of $\mathsf{H}(m)^{sk_i}$ under tpk, i.e., the environment broke the soundness property of the zero-knowledge proof by creating a valid proof for a pair (C_1, C_2) that is not a member of the language described by Equation (1).

For $\mathcal{E}$ to do so, it must at some point make a random-oracle query

$$c = \mathsf{H}'\big(C_1, C_2, tpk, \mathsf{H}(m), pk_i, U_1, U_2, U_3\big) \ .$$

Protocol $\pi_{\textbf{vetbls-zkp}}$

- On input $(sid, \texttt{"init"})$, $\mathcal{S}_i$ provides input $((sid, \texttt{"dkg"}), \texttt{"init"})$ to $\mathcal{F}_{\text{dkg}}$ and waits for an output $((sid, \texttt{"dkg"}), \texttt{"init"}, pk, (pk_1, \ldots, pk_n), sk_i)$. It stores pk, $(pk_1, \ldots, pk_n)$, and sk_i in its local state and outputs $(sid, \texttt{"output-pk"}, (pk, (pk_1, \ldots, pk_n)))$.
- On $(sid, \texttt{"hash"}, m)$, $\mathcal{P}$ returns $(sid, \texttt{"hash"}, m, \mathsf{H}(m))$.
- On $(sid, \texttt{"transport-keygen"})$, $\mathcal{U}$ chooses $tsk \leftarrow_\$ \mathbb{Z}_q$, computes $tpk \leftarrow g_1^{tsk}$, stores tsk in its local state, and outputs $(sid, \texttt{"tpk"}, tpk)$.
- On $(sid, \texttt{"encsign"}, m, tpk)$, $\mathcal{S}_i$ computes $\sigma_i \leftarrow \mathsf{H}(m)^{sk_i}$. It ElGamal-encrypts σ_i as $(C_1, C_2) \leftarrow (g_1^t \ , \ tpk^t \cdot \sigma_i)$, and creates a generalized Schnorr zero-knowledge proof [CKY09]

$$(c, s_1, s_2) \leftarrow_\$ \mathsf{ZKP}\left\{t, sk_i \ : \ C_1 = g_1^t \ \wedge \ C_2 = tpk^t \cdot \mathsf{H}(m)^{sk_i} \ \wedge \ pk_i = g_2^{sk_i}\right\} \tag{1}$$

  by choosing $r_1, r_2 \leftarrow_\$ \mathbb{Z}_q$ and computing

$$c \leftarrow \mathsf{H}'\big(C_1, C_2, tpk, \mathsf{H}(m), pk_i, \ g_1^{r_1} \ , \ tpk^{r_1} \cdot \mathsf{H}(m)^{r_2} \ , \ g_2^{r_2}\big)$$
$$s_1 \leftarrow c \cdot t + r_1 \bmod q$$
$$s_2 \leftarrow c \cdot sk_i + r_2 \bmod q \ .$$

  It then sends $es_i \leftarrow (C_1, C_2, c, s_1, s_2)$ to a combiner.

  When a combiner receives this message, it verifies the zero-knowledge proof by checking that

$$c \ = \ \mathsf{H}'\big(C_1, C_2, tpk, \mathsf{H}(m), pk_i, \ g_1^{s_1} \cdot C_1^{-c} \ , \ tpk^{s_1} \cdot \mathsf{H}(m)^{s_2} \cdot C_2^{-c} \ , \ g_2^{s_2} \cdot pk_i^{-c}\big) \ . \tag{2}$$

  As soon as it received t valid encrypted signature shares es_i from different servers $\mathcal{S}_i$, $i \in S$, it compiles $es \leftarrow (S, (es_i)_{i \in S})$ and sends es to all servers.

  When $\mathcal{S}_i$ receives es, it verifies that $|S| = t$ and that each es_i satisfies Equation (2). If so, it outputs $(sid, \texttt{"encsign"}, m, tpk, es)$.
- On $(sid, \texttt{"verify"}, pk', m, tpk, es)$, $\mathcal{P}$ parses es as $(S, (es_i)_{i \in S})$ and pk' as $(pk, (pk_1, \ldots, pk_n))$. It verifies that $|S| = t$ and that all es_i satisfy Equation (2). If so, it sets $result \leftarrow \texttt{true}$, otherwise it sets $result \leftarrow \texttt{false}$. It outputs $(sid, \texttt{"verify"}, m, tpk, es, result)$.
- On $(sid, \texttt{"decrypt"}, pk', m, tpk, es)$, $\mathcal{U}$ parses es as $(S, (C_{i,1}, C_{i,2}, c_i, s_{i,1}, s_{i,2})_{i \in S})$ and pk' as $(pk, \cdot)$, and looks up tsk from its local state. It decrypts $\sigma_i \leftarrow C_{i,2} \cdot C_{i,1}^{-tsk}$ for all $i \in S$ and reconstructs $\sigma \leftarrow \prod_{i \in S'} \sigma_i^{\Lambda_{i,S'}(0)}$. If $\mathrm{e}(\sigma, g_2) \neq \mathrm{e}(\mathsf{H}(m), pk)$, it sets $\sigma \leftarrow \perp$. It outputs $(sid, \texttt{"decrypt"}, m, tpk, es, \sigma)$.

Figure 6: The zero-knowledge vetBLS protocol $\pi_{\text{vetbls-zkp}}$ based on hash functions $\mathsf{H} : \{0,1\}^* \rightarrow \mathbb{G}_1$ and $\mathsf{H}' : \{0,1\}^* \rightarrow \mathbb{Z}_{2^\ell}$, and the distributed key generation ideal functionality $\mathcal{F}_{\text{dkg}}$.

and come up with $s_1, s_2 \in \mathbb{Z}_q$ such that

$$\begin{aligned} U_1 &= g_1^{s_1} \cdot C_1^{-c} \\ U_2 &= tpk^{s_1} \cdot \mathsf{H}(m)^{s_2} \cdot C_2^{-c} \\ U_3 &= g_2^{s_2} \cdot pk_i^{-c} \ . \end{aligned}$$

If (C_1, C_2) is not an ElGamal encryption of $\mathsf{H}(m)^{sk_i}$ under tpk, then we can say that $(C_1, C_2) = (g_1^{t_1}, tpk^{t_2} \cdot \mathsf{H}(m)^{sk_i})$ for some $t_1, t_2 \in \mathbb{Z}_q$, $t_1 \neq t_2$. Let μ, u_1, u_2, and u_3 be the discrete logarithms of $\mathsf{H}(m)$, U_1, U_2, and U_3 with respect to bases g_1, g_1, g_1, and g_2, respectively. In order to satisfy the above equations, it needs to hold that

$$\begin{aligned} u_1 &= s_1 - c \cdot t_1 \bmod q \\ u_2 &= (s_1 - c \cdot t_2) \cdot tsk + (s_2 - c \cdot sk_i) \cdot \mu \bmod q \\ u_3 &= s_2 - c \cdot sk_i \bmod q \ . \end{aligned}$$

As $C_1, C_2, tpk, \mathsf{H}(m), pk_i$ are all included in the argument of the random-oracle call, the values of t_1, t_2, tsk, μ, sk_i are all fixed at the moment that $\mathcal{E}$ makes this query, as are the values of u_1, u_2, u_3. From the first and third equations, we get that

$$\begin{aligned} s_1 &= u_1 + c \cdot t_1 \bmod q \\ s_2 &= u_3 + c \cdot sk_i \bmod q \ . \end{aligned}$$

Filling these into the second equation, we find that if $t_1 \neq t_2 \bmod q$, there is only one value for $c \in \mathbb{Z}_q$, and hence at most one value in $\mathbb{Z}_{2^\ell}$, for which this system of equations has a solution, namely the one satisfying

$$u_2 = (u_1 + c \cdot (t_1 - t_2)) \cdot tsk + u_3 \cdot \mu \bmod q \ .$$

The probability that the random-oracle output hits this value is at most $1/2^\ell$. The probability that any of $q_{\mathsf{H}'}$ random-oracle queries hit this value is at most

$$1 - \left(1 - \frac{1}{2^\ell}\right)^{q_{\mathsf{H}'}} \leq \frac{q_{\mathsf{H}'}}{2^\ell} \ .$$

Game 3: *(Forgery of encrypted signatures.)* Whenever an honest party is asked to verify an encrypted signature es on a message m under a tpk that was generated by an honest user, and verification succeeds, even though no honest server "$\mathcal{S}_i$" ever participated in the creation of an encrypted signature for (m, tpk) or for (m, tpk') where tpk' was not generated by an honest user, the simulator aborts.

We show how any environment $\mathcal{E}$ and adversary $\mathcal{A}$ that can distinguish this game from the previous one can be used to build a uf-cma forger $\mathcal{B}$ for the $\mathcal{BLS}$ signature scheme. On input pk, $\mathcal{B}$ runs $\mathcal{E}$ and $\mathcal{A}$ as in Game 2, but using pk as the public key, for which it of course doesn't know sk, and relaying $\mathcal{A}$'s random-oracle queries $\mathsf{H}(\cdot)$ to its own random oracle.

The only time that $\mathcal{B}$ needs sk is to compute $\sigma = \mathsf{H}(m)^{sk}$ when an honest "$\mathcal{S}_i$" computes an encrypted signature share for a non-honest tpk. At this point, $\mathcal{B}$ queries its signing oracle on m to obtain σ.

When it obtains an encrypted signature es that causes Game 3 to abort, it decrypts es using tsk (which it can do because tpk was generated by an honest user "$\mathcal{U}$" and because Game 2 guarantees that valid encrypted signature shares decrypt correctly) to obtain σ, a valid message on m. This is a non-trivial forgery, because $\mathcal{E}$ never made an honest server create an encrypted signature on m under a non-honest key tpk', which would have made $\mathcal{B}$ query its signing oracle on m.

Game 4: *(Zero-knowledge.)* When a simulated honest server "$\mathcal{S}_i$" produces an encrypted signature share $es_i = (C_1, C_2, c, s_1, s_2)$, it creates a simulated proof by programming the random oracle for H'. In particular, it chooses $c \leftarrow_\$ \mathbb{Z}_{2^\ell}$ and $s_1, s_2 \leftarrow_\$ \mathbb{Z}_q$ and, if the entry for

$$\mathsf{H}'\big(C_1, C_2, tpk, \mathsf{H}(m), pk_i,\ g_1^{s_1} \cdot C_1^{-c}\ ,\ tpk^{s_1} \cdot \mathsf{H}(m)^{s_2} \cdot C_2^{-c}\ ,\ g_2^{s_2} \cdot pk_i^{-c}\big)\ .$$

is not yet defined, sets it to c. By the randomness of s_1, s_2, c, the probability that this entry is already defined when the table for H' has $q_{\mathsf{H}'}$ is $q_{\mathsf{H}'}/(q^2 \cdot 2^\ell)$. The probability that it is defined for any of q_{ES} encrypted signature queries is at most

$$1 - \left(1 - \frac{q_{\mathsf{H}'}}{q^2 \cdot 2^\ell}\right)^{q_{\mathrm{ES}}} \ \leq\ \frac{q_{\mathrm{ES}} \cdot q_{\mathsf{H}'}}{q^2 \cdot 2^\ell}\ .$$

Game 5: *(ElGamal encryption.)* When an honest server "$\mathcal{S}_i$" creates an encrypted signature share under an honest transport public key tpk, it uses two random group elements $C_1, C_2 \leftarrow_\$ \mathbb{G}_1$ instead of an ElGamal encryption of σ_i.

Note that the zero-knowledge proofs remain valid as Game 4 switched to simulated proofs. Also note that the decryption of (C_1, C_2) by "$\mathcal{U}$" will still yield σ_i, because of the changed decryption behavior in Game 2.

Indistinguishability from the previous game can be shown through a hybrid argument that gradually replaces all ciphertexts by honest servers under honest tpk. Any environment and adversary distinguishing the ith from the $(i-1)$st hybrid game gives rise to the following algorithm $\mathcal{B}$ solving the XDH problem.

On input $(A, B, C) = (g_1^\alpha, g_1^\beta, g_1^\gamma) \in \mathbb{G}_1^3$, $\mathcal{B}$ follows the code of Game 5, but generates the transport keys of all honest users "$\mathcal{U}$" as $tpk \leftarrow A \cdot g_1^{tsk}$ for $tsk \leftarrow_\$ \mathbb{Z}_q$. Note that this doesn't affect the handling of decryption inputs, as in Game 5 these no longer involve the transport secret key. Algorithm $\mathcal{B}$ uses random group elements $(C_1, C_2) \leftarrow_\$ \mathbb{G}_1^2$ in the first $i-1$ encrypted signature shares to honest users, uses real encryptions $(g_1^r, tpk^r \cdot \sigma_i)$ for the $(i+1)$st to last encrypted signature shares, and uses $(C_1, C_2) \leftarrow (B, C \cdot B^{tsk} \cdot \sigma_i)$ for the ith encrypted share, where tsk is the value it chose to compute $tpk = A \cdot g_1^{tsk}$. If $\gamma = \alpha\beta$, we have that $(C_1, C_2) = (g_1^\beta, g_1^{\alpha\beta + tsk\beta} \cdot \sigma_i) = (g_1^\beta, tpk^\beta \cdot \sigma_i)$ as in a real ciphertext, while if γ is random, (C_1, C_2) are random group elements. If $\mathcal{E}$ decides it's running in the $(i-1)$st hybrid game, $\mathcal{B}$ outputs 1, otherwise it outputs 0.

Game 6: *(Random key shares for corrupt servers.)* Instead of executing the real code of $\mathcal{F}_{\mathrm{dkg}}$, the simulator lets "$\mathcal{F}_{\mathrm{dkg}}$" choose random sk and $sk_i \leftarrow_\$ \mathbb{Z}_q$ for all corrupt servers $\mathcal{S}_i$, and derive public key shares for honest users as in Game 4 in the proof of Theorem 3. It also computes the signature share σ_i that an honest server "$\mathcal{S}_i$" encrypts to a non-honest tpk by interpolation from $\sigma \leftarrow \mathsf{H}(m)^{sk}$ and the corrupt servers' shares. Just as in the proof of Theorem 3, this change is purely conceptual.

Game 6 can be turned into the following simulator $\mathcal{Sim}$ that interacts with $\mathcal{E}$ and $\mathcal{F}_{\mathrm{vetbls}}$ in the ideal world:

- On (`"init"`, $\mathcal{S}_i, pk$) from $\mathcal{F}_{\mathrm{vetbls}}$, $\mathcal{Sim}$ uses pk as the output of $\mathcal{F}_{\mathrm{dkg}}$, chooses random secret key shares $sk_i \leftarrow_\$ \mathbb{Z}_q$ for all corrupt servers, and computes the public key shares as in Game 6.
- It lets the random oracle $\mathsf{H}(\cdot)$ be handled by the `"hash"` interface of $\mathcal{F}_{\mathrm{vetbls}}$.
- On (`"transport-keygen"`, $\mathcal{U}$) from $\mathcal{F}_{\mathrm{vetbls}}$, it runs the honest simulated user "$\mathcal{U}$" on (sid, TKG) to obtain (sid, `"tpk"`, tpk), and sends tpk to $\mathcal{F}_{\mathrm{vetbls}}$.

- On (`"encsign"`$, m, tpk, \mathcal{S}_i$) from $\mathcal{F}_{\text{vetbls}}$, $\mathcal{Sim}$ lets "$\mathcal{S}_i$" use random group elements $C_1, C_2 \leftarrow_\$ \mathbb{G}_1$ as in Game 5 and simulate the zero-knowledge proof (c, s_1, s_2) by programming the random oracle H' as in Game 4. Note that this ciphertext never needs to be decrypted by "$\mathcal{U}$", because $\mathcal{F}_{\text{vetbls}}$ handles decryptions independent of $\mathcal{Sim}$.

- On (`"encsign"`$, m, tpk, \mathcal{S}_i, \sigma$) from $\mathcal{F}_{\text{vetbls}}$, $\mathcal{Sim}$ lets "$\mathcal{S}_i$" recompute σ_i using Lagrange interpolation as in Game 6 and encrypts it under tpk as in the real protocol.

- When an honest "$\mathcal{S}_i$" outputs (sid, `"encsign"`$, m, tpk, es$), $\mathcal{Sim}$ sends (sid, `"output-encsig"`$, m, tpk, \mathcal{S}_i, es$) to $\mathcal{F}_{\text{vetbls}}$.

- On (`"verify"`$, pk', m, tpk, es$) from $\mathcal{F}_{\text{vetbls}}$, $\mathcal{Sim}$ runs $\pi_{\text{vetbls-zkp}}$ on input (sid, `"verify"`$, pk', m, tpk, es$) and returns the result to $\mathcal{F}_{\text{vetbls}}$.

□

5 Aggregated Constructions with Leakage

The simple and zero-knowledge schemes from the previous section have encrypted signatures that are linear in size in the number of combined shares, i.e., in the threshold t. In this section, we analyze the security of the $\pi_{\text{vetbls-agg1}}$ and $\pi_{\text{vetbls-agg2}}$ protocols, where the encrypted signature shares are aggregated into a compact verifiably encrypted signature, with size independent of n or t. Rather than relying on standard signatures or zero-knowledge proofs, verification of encrypted signatures is performed by checking simple pairing equations. The schemes can be seen as threshold variants of the verifiably encrypted signature scheme of Boneh et al. [BGLS03], mapped to a Type-3 pairing setting in two different ways.

Unfortunately, however, $\pi_{\text{vetbls-agg1}}$ and $\pi_{\text{vetbls-agg2}}$ do not securely realize the $\mathcal{F}_{\text{vetbls}}$ functionality as put forward in Figure 2. Whereas the simulators in $\pi_{\text{vetbls-sim}}$ and $\pi_{\text{vetbls-zkp}}$ could simulate encrypted signatures by honest servers to an honest tpk by encrypting bogus signature shares, the public verifiability by pairing equations of $\pi_{\text{vetbls-agg1}}$ and $\pi_{\text{vetbls-agg2}}$ prevents us from doing anything of the kind. This points to the fact that the encrypted signature shares in the compact constructions actually do leak some information about the signature share, namely, a verifiable ElGamal encryption of it.

5.1 Ideal Functionality with Leakage

We recover the compact constructions simply by accepting that leakage into the security model. Namely, we weaken the $\mathcal{F}_{\text{vetbls}}$ functionality by, whenever an honest server creates an encrypted signature for an honest tpk, giving the simulator access to leakage information derived from the actual BLS signature. This is modeled in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ ideal functionality in Figure 7, where the functionality is parameterized by a leakage algorithm $\mathcal{L}$ that, on input leakage parameters $lpars$ and a BLS signature σ, returns leakage information λ. The leakage parameters are generated as $lpars \leftarrow_\$ \mathcal{L}(\perp, \perp)$ at the initialization of the functionality.

The $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ functionality is weaker than the $\mathcal{F}_{\text{vetbls}}$ functionality in the sense that any protocol π that securely realizes $\mathcal{F}_{\text{vetbls}}$ also realizes $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ for any $\mathcal{L}$, but not vice versa. How severely leakage weakens the functionality depends on the type of leakage. On the one extreme, if $\mathcal{L}(lpars, \sigma)$ is independent of σ, $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ is equivalent to $\mathcal{F}_{\text{vetbls}}$. On the other extreme, if $\mathcal{L}(lpars, \sigma) = \sigma$, then it completely undermines the main security guarantee of the $\mathcal{F}_{\text{vetbls}}$ functionality, namely secure delivery of BLS signatures. Whether the leakage affects the security of a higher-level protocol depends on the exact type of leakage and on the higher-level protocol.

Functionality $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$

Identical to functionality $\mathcal{F}_{\text{vetbls}}$ in Figure 2, except for the addition of a new `"leakpars"` interface and a modified `"encsign"` interface as follows.

- On $(sid, \texttt{"leakpars"})$ from $\mathcal{S}im$:
  If *lpars* is not defined, generate and store $lpars \leftarrow_\$ \mathcal{L}(\bot, \bot)$. Output $(sid, \texttt{"leakpars"}, lpars)$ to $\mathcal{S}im$.
- On $(sid, \texttt{"encsign"}, m, tpk)$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  Simulate an input $(sid, \texttt{"hash"}, m)$ and compute $\sigma \leftarrow H[m]^{sk}$. If $tpk \in TPK$, add (m, tpk) to ES, compute $\lambda \leftarrow_\$ \mathcal{L}(lpars, \sigma)$ and send $(\texttt{"encsign"}, m, tpk, \mathcal{S}_i, \lambda)$ to $\mathcal{S}im$. Otherwise, add $(m, \bot)$ to ES and send $(\texttt{"encsign"}, m, tpk, \mathcal{S}_i, \sigma)$ to $\mathcal{S}im$.

Figure 7: The verifiably encrypted threshold BLS functionality with signature leakage $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, where $\mathcal{L}$ is a leakage algorithm.

5.2 BLS Signatures with Leakage

We first show that basic BLS signatures remain secure in the presence of a particular type of leakage that we will need to prove our aggregate vetBLS constructions.

We extend the definition of uf-cma security for standard signature schemes by giving the adversary access to an oracle that "leaks" partial information derived from signatures on messages of its choice. The adversary can query its leakage oracle "for free", without affecting the non-triviality condition. Meaning, having queried its leakage oracle on the leakage of a signature on message m doesn't preclude it from using a signature on m as its forgery.

We define the advantage of an adversary $\mathcal{A}$ in breaking the uf-cma$^{\mathcal{L}}$ security of $\mathcal{SS} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ as its probability in winning a game where the adversary is given a fresh public key pk as well as fresh leakage parameters $lpars \leftarrow_\$ \mathcal{L}(\bot, \bot)$ as input. It is then given access to a signing oracle $\mathsf{Sign}(sk, \cdot)$ as well as a leakage oracle $\mathcal{L}(lpars, \mathsf{Sign}(sk, \cdot))$. The adversary wins if it outputs a forgery m^*, σ^* so that $\mathsf{Verify}(pk, m^*, \sigma^*) = 1$ and it never queried m^* from its signing oracle.

We define two types of leakage based on ElGamal encryption of a signature:

1. *co-ElGamal-1 leakage* is given by an algorithm cEG_1 that, on input $(\bot, \bot)$, returns $lpars = (h_1, h_2) = (g_1^x, g_2^x)$ for $x \leftarrow_\$ \mathbb{Z}_q$, and that, on input $((h_1, h_2), \sigma)$, returns $\lambda = (g_1^r, h_1^r \cdot \sigma)$ for $r \leftarrow_\$ \mathbb{Z}_q$.
2. *co-ElGamal-2 leakage* is given by algorithm cEG_2 that, on input $(\bot, \bot)$, returns $lpars = h_1 = g_1^x$ for $x \leftarrow_\$ \mathbb{Z}_q$, and that, on input (h_1, σ), returns $\lambda = (g_1^r, g_2^r, h_1^r \cdot \sigma)$ for $r \leftarrow_\$ \mathbb{Z}_q$.

Theorem 5. *The $\mathcal{BLS}$ scheme is uf-cma$_1^{\mathsf{cEG}}$ and uf-cma$_2^{\mathsf{cEG}}$ secure in the random-oracle model if the co-CDH problem in $(\mathbb{G}_1, \mathbb{G}_2)$ is hard.*

Proof. We first prove the statement for the case of co-ElGamal-1 leakage. Given a uf-cma$_1^{\mathsf{cEG}}$ adversary $\mathcal{A}$, consider the following co-CDH adversary $\mathcal{B}$. On input $(A_1 = g_1^\alpha, B_1 = g_1^\beta, B_2 = g_2^\beta)$, $\mathcal{B}$ sets $pk \leftarrow B_2$, chooses $v \leftarrow_\$ \mathbb{Z}_q$ and sets $h_1 \leftarrow g_1^v / B_1$ and $h_2 \leftarrow g_2^v / B_2$. It runs $\mathcal{A}$ on input $(pk, (h_1, h_2))$ and simulates $\mathcal{A}$'s oracle queries as follows:

- Random oracle $\mathsf{H}(m)$: $\mathcal{B}$ guesses a random query index to be the hash of $\mathcal{A}$'s forgery m^*. For the query $\mathsf{H}(m^*)$, $\mathcal{B}$ returns A_1. For all other queries $\mathsf{H}(m)$, $\mathcal{B}$ chooses $r_m \leftarrow_\$ \mathbb{Z}_q$ and returns $g_1^{r_m}$.
- Signing oracle: When $\mathcal{A}$ makes a signing query for $m \neq m^*$, $\mathcal{B}$ returns $\sigma \leftarrow B_1^{r_m}$. If $\mathcal{A}$ queries for a signature on m^*, $\mathcal{B}$ gives up.

- Leakage oracle: If $\mathcal{A}$ makes a leakage query for m, $\mathcal{B}$ needs to return a tuple of the form $(g_1^r, h_1^r \cdot \mathsf{H}(m)^\beta)$. For $m \neq m^*$, it simply does so by computing a signature σ for m as above, choosing $r \leftarrow_\$ \mathbb{Z}_q$, and returning $(g_1^r, h_1^r \cdot \sigma)$. For $m = m^*$, it proceeds differently: it chooses $s \leftarrow_\$ \mathbb{Z}_q$ and outputs $\lambda = (\lambda_1, \lambda_2) = (A_1 \cdot g_1^s, A_1^v \cdot g_1^{vs}/B_1^s)$. This is correctly distributed because if $\lambda_1 = A_1 \cdot g_1^s = g_1^r$ for $r = \alpha + s$, then λ_2 is

$$\begin{aligned}
\lambda_2 &= h_1^r \cdot \mathsf{H}(m^*)^\beta \\
&= (g_1^v/B_1)^r \cdot A_1^\beta \\
&= g_1^{v(\alpha+s)-\beta(\alpha+s)} \cdot g_1^{\alpha\beta} \\
&= g_1^{v\alpha+vs-\beta s} \\
&= A_1^v \cdot g_1^{vs}/B_1^s \ .
\end{aligned}$$

When $\mathcal{A}$ outputs a forgery σ^* on m^*, it outputs $\sigma^* = \mathsf{H}(m^*)^\beta = g_1^{\alpha\beta}$, otherwise it gives up.

The proof for co-ElGamal-2 leakage is similar, except that $\mathcal{B}$ sets $h_1 \leftarrow g_1^v/A_1$ and simulates signature leakage for m^* as $(\lambda_1, \lambda_2, \lambda_3) \leftarrow (B_1 \cdot g_1^s, B_2 \cdot g_2^s, B_1^v \cdot g_1^{vs}/A_1^s)$, which one can check is of the form $(g_1^r, g_2^r, h_1^r \cdot \mathsf{H}(m^*)^\beta)$ for $r = \beta + s$. □

5.3 Two Aggregated vetBLS Constructions

We present two aggregated constructions $\pi_{\text{vetbls-agg1}}$ and $\pi_{\text{vetbls-agg2}}$ in Figures 8 and 9, respectively. Both constructions have constant-size encrypted signatures and can be seen as the natural threshold extension of the verifiably encrypted signature scheme of Boneh et al. [BGLS03], but mapped to a Type-3 pairing in two different ways.

In order to verify encrypted signatures by applying a pairing equation, we need more than $tpk \in \mathbb{G}_1$ and ElGamal ciphertext components $(C_1, C_2) \in \mathbb{G}_1^2$. The $\pi_{\text{vetbls-agg1}}$ scheme uses a double transport public key as $tpk = (g_1^{tsk}, g_2^{tsk})$, while the $\pi_{\text{vetbls-agg2}}$ scheme adds an element g_2^r to the ElGamal ciphertext. The effect is that $\pi_{\text{vetbls-agg1}}$ has to perform an extra pairing computation during the creation of encrypted signature shares, to check that the public key is correctly distributed, while $\pi_{\text{vetbls-agg2}}$ has to perform an extra pairing during the verification of encrypted signatures and signature shares, to ensure that the extended ElGamal ciphertext is correctly distributed.

We show that the $\pi_{\text{vetbls-agg1}}$ and $\pi_{\text{vetbls-agg2}}$ protocols securely realize the $\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_1}$ and $\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_2}$ functionalities, respectively. This is of course a weaker result than to realize the original $\mathcal{F}_{\text{vetbls}}$ functionality, but as we will see later, it actually turns out to suffice for usage in an IBE scheme, VRF, and signature scheme.

It is worth noting that, unlike Theorems 3 and 4, Theorem 6 does not rely on the hardness of the XDH problem, even though all four schemes use the same basic approach of ElGamal-encrypting the BLS signature in $\mathbb{G}_1$. However, by giving away an additional element in $\mathbb{G}_2$ that enables a pairing verification, the ElGamal encryption loses its semantic security based on the XDH assumption. Instead, the $\pi_{\text{vetbls-agg1}}$ and $\pi_{\text{vetbls-agg2}}$ protocols realize the weaker $\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_1}$ and $\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_2}$ functionalities.

Theorem 6. *If the co-CDH problem is hard and* H *is modeled as a random oracle, then the* $\pi_{\text{vetbls-agg1}}$ *and* $\pi_{\text{vetbls-agg2}}$ *protocols securely realize the* $\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_1}$ *and* $\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_2}$ *functionalities in the* $\mathcal{F}_{\text{dkg}}$*-hybrid model, respectively.*

Proof. We first prove the statement for the $\pi_{\text{vetbls-agg1}}$ protocol. Consider the following simulator $\mathcal{S}im$ that interacts with $\mathcal{E}$ and $\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_1}$ in the ideal world:

- On $(\texttt{"init"}, \mathcal{S}_i, pk)$ from $\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_1}$, $\mathcal{S}im$ uses pk as the output of "$\mathcal{F}_{\text{dkg}}$", chooses random secret key shares $sk_i \leftarrow_\$ \mathbb{Z}_q$ for all corrupt servers, and derives public key shares for honest users as in Game 4 in the proof of Theorem 3.

Protocol $\pi_{\text{vetbls-agg1}}$

The `"init"` and `"hash"` interfaces are identical to $\pi_{\text{vetbls-zkp}}$, except that the `"init"` interface outputs $(sid, \texttt{"output-pk"}, pk)$. The other interfaces are as follows:

- On $(sid, \texttt{"transport-keygen"})$, $\mathcal{U}$ chooses $tsk \leftarrow_{\$} \mathbb{Z}_q$, computes $tpk_1 \leftarrow g_1^{tsk}$, $tpk_2 \leftarrow g_2^{tsk}$, stores tsk in its local state, sets $tpk \leftarrow (tpk_1, tpk_2)$, and outputs $(sid, \texttt{"tpk"}, tpk)$.
- On $(sid, \texttt{"encsign"}, m, tpk)$, $\mathcal{S}_i$ parses $tpk = (tpk_1, tpk_2)$ and checks whether $\mathrm{e}(tpk_1, g_2) = \mathrm{e}(g_1, tpk_2)$; if not, it ignores this input. Otherwise, it computes $\sigma_i \leftarrow \mathsf{H}(m)^{sk_i}$, encrypts σ_i as $(C_1, C_2) \leftarrow (g_1^t\ ,\ tpk_1^t \cdot \sigma_i)$, and sends $es_i \leftarrow (C_1, C_2)$ to a combiner.

  When a combiner receives this message, it checks that $\mathrm{e}(C_2, g_2) = \mathrm{e}(C_1, tpk_2) \cdot \mathrm{e}(\mathsf{H}(m), pk_i)$. When it received t valid such $es_i = (C_{i,1}, C_{i,2})$ from different servers $\mathcal{S}_i$, $i \in S$, it computes

  $$es \;=\; (C_1, C_2) \;=\; \left(\prod_{i \in S} C_{i,1}^{\Lambda_{i,S}(0)}\ ,\ \prod_{i \in S} C_{i,2}^{\Lambda_{i,S}(0)} \right)$$

  and sends es to all servers.

  When $\mathcal{S}_i$ receives es, it verifies that $\mathrm{e}(C_2, g_2) = \mathrm{e}(C_1, tpk_2) \cdot \mathrm{e}(\mathsf{H}(m), pk)$. If so, it outputs $(sid, \texttt{"encsign"}, m, tpk, es)$.
- On $(sid, \texttt{"verify"}, pk', m, tpk, es)$, $\mathcal{P}$ parses es as (C_1, C_2) and verifies that $\mathrm{e}(C_2, g_2) = \mathrm{e}(C_1, tpk_2) \cdot \mathrm{e}(\mathsf{H}(m), pk')$. If so, it sets $result \leftarrow \texttt{true}$, otherwise it sets $result \leftarrow \texttt{false}$. It outputs $(sid, \texttt{"verify"}, pk', m, tpk, es, result)$.
- On $(sid, \texttt{"decrypt"}, pk', m, tpk, es)$, $\mathcal{U}$ parses es as (C_1, C_2) and looks up tsk from its local state. It decrypts $\sigma \leftarrow C_2 \cdot C_1^{-tsk}$ and checks whether $\mathrm{e}(\sigma, g_2) = \mathrm{e}(\mathsf{H}(m), pk')$. If so, it outputs $(sid, \texttt{"decrypt"}, pk', m, tpk, es, \sigma)$.

Figure 8: The first aggregate vetBLS protocol $\pi_{\text{vetbls-agg1}}$ based on hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1$ and the distributed key generation ideal functionality $\mathcal{F}_{\text{dkg}}$.

Protocol $\pi_{\text{vetbls-agg2}}$

The "init" and "hash" interfaces are identical to $\pi_{\text{vetbls-zkp}}$, except that the "init" interface outputs $(\mathit{sid}, \texttt{"output-pk"}, \mathit{pk})$. The other interfaces are as follows:

- On $(\mathit{sid}, \texttt{"encsign"}, m, \mathit{tpk})$, $\mathcal{S}_i$ computes $\sigma_i \leftarrow \mathsf{H}(m)^{sk_i}$, encrypts σ_i as $(C_1, C_2, C_3) \leftarrow (g_1^t\ ,\ g_2^t\ ,\ \mathit{tpk}_1^t \cdot \sigma_i)$, and sends $\mathit{es}_i \leftarrow (C_1, C_2, C_3)$ to a combiner.

  When a combiner receives this message, it checks that $\mathrm{e}(C_1, g_2) = \mathrm{e}(g_1, C_2)$ and that $\mathrm{e}(C_3, g_2) = \mathrm{e}(\mathit{tpk}, C_2) \cdot \mathrm{e}(\mathsf{H}(m), \mathit{pk}_i)$. When it received t valid such $\mathit{es}_i = (C_{i,1}, C_{i,2}, C_{i,2})$ from different servers $\mathcal{S}_i$, $i \in S$, it computes

  $$\mathit{es} \;=\; \big(C_1, C_2, C_3\big) \;=\; \left(\prod_{i\in S} C_{i,1}^{\Lambda_{i,S}(0)}\ ,\ \prod_{i\in S} C_{i,2}^{\Lambda_{i,S}(0)}\ ,\ \prod_{i\in S} C_{i,3}^{\Lambda_{i,S}(0)}\right)$$

  and sends es to all servers.

  When $\mathcal{S}_i$ receives es, it verifies that $\mathrm{e}(C_1, g_2) = \mathrm{e}(g_1, C_2)$ and that $\mathrm{e}(C_3, g_2) = \mathrm{e}(\mathit{tpk}, C_2) \cdot \mathrm{e}(\mathsf{H}(m), \mathit{pk})$. If so, it outputs $(\mathit{sid}, \texttt{"encsign"}, m, \mathit{tpk}, \mathit{es})$.

- On $(\mathit{sid}, \texttt{"verify"}, \mathit{pk}', m, \mathit{tpk}, \mathit{es})$, $\mathcal{P}$ parses es as (C_1, C_2, C_3) and verifies that $\mathrm{e}(C_1, g_2) = \mathrm{e}(g_1, C_2)$ and that $\mathrm{e}(C_3, g_2) = \mathrm{e}(\mathit{tpk}, C_2) \cdot \mathrm{e}(\mathsf{H}(m), \mathit{pk}')$. If so, it sets $\mathit{result} \leftarrow$ `true`, otherwise it sets $\mathit{result} \leftarrow$ `false`. It outputs $(\mathit{sid}, \texttt{"verify"}, \mathit{pk}', m, \mathit{tpk}, \mathit{es}, \mathit{result})$.

- On $(\mathit{sid}, \texttt{"decrypt"}, \mathit{pk}', m, \mathit{tpk}, \mathit{es})$, $\mathcal{U}$ parses es as $(C_1, C_2, C_3)$and looks up tsk from its local state. It decrypts $\sigma \leftarrow C_3 \cdot C_1^{-\mathit{tsk}}$ and checks whether $\mathrm{e}(\sigma, g_2) = \mathrm{e}(\mathsf{H}(m), \mathit{pk}')$; if not, it sets $\sigma \leftarrow \perp$. It outputs $(\mathit{sid}, \texttt{"decrypt"}, \mathit{pk}', m, \mathit{tpk}, \mathit{es}, \sigma)$.

Figure 9: The second aggregated vetBLS protocol $\pi_{\text{vetbls-agg2}}$ based on hash function $\mathsf{H} : \{0,1\}^* \to \mathbb{G}_1$ and the distributed key generation ideal functionality $\mathcal{F}_{\text{dkg}}$.

- It lets the random oracle $\mathsf{H}(\cdot)$ be handled by the `"hash"` interface of $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$.
- On (`"transport-keygen"`,$\mathcal{U}$) from $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$, it inputs ($sid$, `"leakpars"`) to $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$ to obtain output (sid, `"leakpars"`, $lpars$) where $lpars = (h_1, h_2)$. It then chooses $\delta \leftarrow_\$ \mathbb{Z}_q$, computes $tpk \leftarrow (h_1 \cdot g_1^\delta, h_2 \cdot g_2^\delta)$, adds (tpk, δ) to $TPK[\mathcal{U}]$, and sends tpk back to $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$.
- On (`"encsign"`, $m, tpk, \mathcal{S}_i, \lambda$) from $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$, $\mathcal{Sim}$ lets "$\mathcal{S}_i$" look up the user $\mathcal{U}$ such that $(tpk, \delta) \in TPK[\mathcal{U}]$, parse $\lambda = (\lambda_1, \lambda_2)$, compute $es_i \leftarrow (\lambda_1, \lambda_2 \cdot \lambda_1^\delta)$ and send es_i to the combiner.
- On (`"encsign"`, $m, tpk, \mathcal{S}_i, \sigma$) from $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$, $\mathcal{Sim}$ lets "$\mathcal{S}_i$" recompute σ_i via Lagrange interpolation from σ and the secret key shares sk_i of the corrupt servers. It then computes es_i as $(g_1^r, tpk_1^r \cdot \sigma_i)$ and sends es_i to the combiner.
- When an honest "$\mathcal{S}_i$" outputs (sid, `"encsign"`, m, tpk, es), $\mathcal{Sim}$ provides input (sid, `"output-encsig"`, $m, tpk, \mathcal{S}_i, es$) to $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$.
- On (`"verify"`, pk', m, tpk, es) from $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$, $\mathcal{Sim}$ runs $\pi_{\text{vetbls-agg1}}$ on input (sid, `"verify"`, pk', m, tpk, es) and returns the result to $\mathcal{F}_{\mathrm{vetbls}}$.

One can see that the view thus generated by $\mathcal{Sim}$ to $\mathcal{E}$ and $\mathcal{A}$ is distributed exactly as in the real world. In particular, encrypted signature shares by an honest "$\mathcal{S}_i$" to an honest tpk are correctly distributed as

$$\begin{aligned}(\lambda_1\ ,\ \lambda_3 \cdot \lambda_1^\delta) &= (g_1^r\ ,\ h_1^r \cdot \mathsf{H}(m)^{sk} \cdot g_1^{r\delta}) \\ &= (g_1^r\ ,\ tpk^r \cdot \mathsf{H}(m)^{sk})\ .\end{aligned}$$

To see why decryption of a valid encrypted signature for always results in a correct BLS signature, observe that the condition in the `"verify"` interface of $\pi_{\text{vetbls-agg1}}$ that $\mathrm{e}(C_2, g_2) = \mathrm{e}(C_1, tpk_2) \cdot \mathrm{e}(\mathsf{H}(m), pk)$ and the fact that $(tpk_1, tpk_2) = (g_1^{tsk}, g_2^{tsk})$ together imply that $C_2 = g_1^{t \cdot tsk + \mu \cdot sk}$, where $t = \mathrm{dlog}_{g_1}(C_1)$ and $\mu = \mathrm{dlog}_{g_1}(\mathsf{H}(m))$. The decryption as performed in the `"decrypt"` interface therefore always recovers $\sigma = C_2 \cdot C_1^{-tsk} = g_1^{\mu \cdot sk} = \mathsf{H}(m)^{sk}$.

The only difference between the simulated and the real experiment occurs when the `"verify"` interface of $\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$ rejects an encrypted signature that $\pi_{\text{vetbls-agg1}}$ deems valid. In particular, this happens when es verifies correctly for m and an honest user's tpk, even though no honest "$\mathcal{S}_i$" ever participated in a protocol for m and tpk or a non-honest tpk'.

Any environment $\mathcal{E}$ and adversary $\mathcal{A}$ that cause this event to happen can be used to build a uf-cmacEG$_1$ forger $\mathcal{B}$ against $\mathcal{BLS}$. Namely, on input $pk, (h_1, h_2)$, $\mathcal{B}$ runs $\mathcal{E}$ and $\mathcal{A}$ against a simulated "$\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$" and the simulator $\mathcal{Sim}$ described above, where $\mathcal{B}$ uses the same public key pk as in its own experiment, uses (h_1, h_2) as the leakage parameters for "$\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$", and relays `"hash"` queries to its own random oracle. It mostly generates honest users' public keys also as above, but for one random honest user, it generates the keys so that it knows the decryption key, i.e., as $tpk^* = g_1^{tsk^*}$ for $tsk^* \leftarrow_\$ \mathbb{Z}_q$.

On an `"encsign"` input for m and an honest $tpk \neq tpk^*$, $\mathcal{B}$ queries its leakage oracle on m to obtain $\lambda = \mathsf{cEG}_1(lpars, \mathsf{Sign}(sk, m))$ and computes the encrypted signature share as above. For a non-honest tpk' or for tpk^*, it queries its signing oracle to obtain σ and proceeds as $\mathcal{Sim}$ above.

If a discrepancy between the `"verify"` interfaces of "$\mathcal{F}_{\mathrm{vetbls}}^{\mathsf{cEG}_1}$" and $\pi_{\text{vetbls-agg1}}$ occurs for an encrypted signature $es = (C_1, C_2)$ on m^* under tpk^*, $\mathcal{B}$ decrypts es as $\sigma^* \leftarrow C_2 \cdot C_1^{-tsk^*}$ and outputs m^*, σ^* as its forgery. If such a discrepancy happens for some other honest key $tpk \neq tpk^*$, $\mathcal{B}$ gives up. The theorem statement for $\pi_{\text{vetbls-agg1}}$ then follows from Theorem 5.

The proof for $\pi_{\text{vetbls-agg2}}$ is extremely similar, the only differences being that $\mathcal{Sim}$ uses $tpk \leftarrow h_1 \cdot g_1^\delta$ as honest users' public keys, simulates encrypted signature shares to honest tpk as $es_i \leftarrow (\lambda_1, \lambda_2, \lambda_3 \cdot \lambda_1^\delta)$ and to non-honest tpk as $es_i \leftarrow (g_1^r, g_2^r, tpk^r \cdot \sigma_i)$. We leave the details as an easy exercise to the reader. □

Functionality $\mathcal{F}_{\text{vetibe}}$

- On $(sid, \texttt{"init"})$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  Send $(\texttt{"init"}, \mathcal{S}_i)$ to $\mathcal{S}im$. If mpk isn't defined, wait for a response mpk from $\mathcal{S}im$ and store mpk.
- On $(sid, \texttt{"output-mpk"}, \mathcal{S}_i)$ from $\mathcal{S}im$:
  If $\mathcal{S}_i \in \text{servers}(sid)$ and mpk is defined, output $(sid, \texttt{"output-mpk"}, mpk)$ to $\mathcal{S}_i$.
- On $(sid, \texttt{"init"}, mpk')$ from $\mathcal{U}$:
  If $ID[\mathcal{U}] = \bot$ set $ID[\mathcal{U}] \leftarrow \emptyset$ and set $MPK[\mathcal{U}] = mpk'$. Send $(\texttt{"init"}, \mathcal{U}, mpk')$ to $\mathcal{S}im$.
- On $(sid, \texttt{"ekderive"}, id, \mathcal{U})$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  If $ID[\mathcal{U}] = \bot$ then ignore. Add id to $EKD[\mathcal{U}]$ and send $(\texttt{"ekderive"}, id, \mathcal{U})$ to $\mathcal{S}im$.
- On $(sid, \texttt{"output-key"}, id, \mathcal{U})$ from $\mathcal{S}im$: If $ID[\mathcal{U}] = \bot$ then ignore. If $id \notin EKD[\mathcal{U}]$ and there does not exist a corrupt $\mathcal{U}'$ such that $id \in EKD[\mathcal{U}']$, then ignore. Add id to $ID[\mathcal{U}]$ and output $(\texttt{"ekderive"}, id)$ to $\mathcal{U}$.
- On $(sid, \texttt{"encrypt"}, mpk', id, m)$ from $\mathcal{P}$: If $mpk' = mpk$, send $(sid, \texttt{"encrypt"}, mpk, id, |m|)$ to $\mathcal{S}im$; otherwise, send $(sid, \texttt{"encrypt"}, mpk', id, m)$ to $\mathcal{S}im$. Wait for C from $\mathcal{S}im$ so that $\nexists\ (mpk', C, id, \cdot) \in CTXT$. Add (mpk', C, id, m) to $CTXT$ and output $(sid, \texttt{"encrypt"}, mpk', id, m, C)$ to $\mathcal{P}$.
- On $(sid, \texttt{"decrypt"}, id, C)$ from $\mathcal{U}$: If $\mathcal{U}$ is honest and $id \notin ID[\mathcal{U}]$ then ignore. If $\mathcal{U}$ is corrupt and there does not exist a corrupt $\mathcal{U}'$ such that $id \in EKD[\mathcal{U}]$ then ignore. If $\exists\ (MPK[\mathcal{U}], C, id, m) \in CTXT$ then output $(sid, \texttt{"decrypt"}, id, C, m)$ to $\mathcal{U}$. Otherwise, send $(sid, \texttt{"decrypt"}, MPK[\mathcal{U}], id, C)$ to $\mathcal{S}im$, wait for m from $\mathcal{S}im$, add $(MPK[\mathcal{U}], C, id, m)$ to $CTXT$, and output $(sid, \texttt{"decrypt"}, id, C, m)$ to $\mathcal{U}$.

Figure 10: The ideal functionality for identity-based encryption with verifiably encrypted threshold key derivation $\mathcal{F}_{\text{vetibe}}$.

6 Verifiably Encrypted Threshold IBE

We now return to our original goal of building an IBE scheme with verifiably encrypted threshold key derivation, or verifiably encrypted threshold IBE (vetIBE) scheme. As argued in the introduction, such a scheme has many applications in blockchain scenarios, from end-to-end encrypted on-chain messaging and social networks to preventing miner extracted value in DeFi applications.

We first define the concept of vetIBE by presenting an ideal functionality in the UC framework. We then show how the Boneh-Franklin FullIdent scheme [BF01] securely realizes that functionality in the $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$-hybrid world. Since any protocol that securely realizes $\mathcal{F}_{\text{vetbls}}$ also realizes $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$, that means that our scheme can be combined with any of the four vetBLS constructions of the previous sections.

6.1 The vetIBE Ideal Functionality

In Figure 10, we describe the vetIBE ideal functionality $\mathcal{F}_{\text{vetibe}}$. Rather than assigning a fixed identity to each user, which would severely restrict use cases, we follow Nishimaki et al. [NMO06] in allowing users to collect decryption keys for multiple identities. We further extend their model—as well as the set of use cases—by allowing multiple users to obtain the decryption key for the same identity, and also by guaranteeing secrecy for ciphertexts that are encrypted before any users are registered as recipients for that identity. These extensions are crucial for flexible use in many applications, including end-to-end encrypted messaging, social networks, or time-locked encryption.

Because of the latter extension, we run into the same "commitment problem" observed by Hofheinz et al. [HMM15] for IBE in the Constructive Cryptography framework [Mau11]. Namely, when messages are encrypted to an identity that only later gets assigned to a corrupt user, the simulator first has to "commit" to a ciphertext C without knowing the message m, and then later come up with a decryption key for id that makes C decrypt to m. A technically very similar problem occurs for public-key encryption in the face of adaptive corruptions, which was proved to be impossible to achieve in the standard model [Nie02a]. For IBE, the problem seems more inherent still, as it even shows up for static corruptions.

The $\mathcal{F}_{\text{vetibe}}$ functionality lets servers decide which users $\mathcal{U}$ get access to which identities id by participating in an encrypted key derivation `"ekderive"` for that $\mathcal{U}$ and id. The decryption key itself is kept internal to the functionality. As soon as it is delivered to $\mathcal{U}$ through the `"output-key"` interface, that user is able to decrypt ciphertexts encrypted to id through the `"decrypt"` interface, as long as the user was initialized with the correct $mpk' = mpk$.

The main security guarantee of the $\mathcal{F}_{\text{vetibe}}$ scheme is that messages encrypted under an id that no corrupt users have access to, meaning that no honest server ever participated in an encrypted key derivation for for a corrupt user, remain hidden from the adversary. Any ciphertext that encrypts a message m to identity id through the `"encrypt"` interface using the correct master public key mpk is guaranteed to decrypt to m for any user that was initialized with mpk and obtained a decryption key for id. Decryption of maliciously created ciphertexts or ciphertexts encrypted to a different identity id' or a different master public key mpk' can have arbitrary results; in particular, the $\mathcal{F}_{\text{vetibe}}$ functionality does not enforce any robustness property [ABN10].

The $\mathcal{F}_{\text{vetibe}}$ functionality in Figure 10 stores in its state the master public key mpk, a map ID initialized to $ID[\cdot] = \bot$ to keep track of which user received a decryption key for which identity, a map MPK initialized to $MPK[\cdot] = \bot$ to keep track of which master public key each user was initialized with, a map EKD initialized to $EKD[\cdot] = \emptyset$ to keep track of which user $\mathcal{U}$ had at least one honest server participating in an encrypted key derivation for which identity id, and a map $CTXT$ to keep track of ciphertexts and their decryptions.

An honest user with $id \in EKD[\mathcal{U}]$ will have to wait for the decryption key to be delivered through the `"output-key"` interface before id is added to $ID[\mathcal{U}]$ so that $\mathcal{U}$ can decrypt messages encrypted under id. A corrupt user, however, can decrypt under id if $id \in EKD[\mathcal{U}']$ for *any* corrupt user $\mathcal{U}'$, modeling that as soon as an honest server grants a corrupt users access to ciphertexts encrypted under id, one must assume that all corrupt users have this access.

6.2 A Construction based on the Boneh-Franklin IBE

Non-committing public-key encryption is known to be impossible to realize in the standard model [Nie02a], but relatively easy to instantiate in the random-oracle model [CLNS17]. It should therefore not come as a surprise that are able to instantiate the $\mathcal{F}_{\text{vetibe}}$ functionality in the random-oracle model; slight more surprising, perhaps, is the fact that the Boneh-Franklin FullIdent scheme [BF01] that was designed to be ind-cca secure also satisfies the required non-committing property.

The π_{vetibe} scheme in Figure 11 is presented in the combined $(\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model, where the former functionality is used for key derivation and the latter for registering users' transport keys. It is a rather straightforward adaptation of the Boneh-Franklin FullIdent scheme to a Type-3 pairing. The original scheme was presented for a Type-1 (i.e., symmetric) pairing; a chosen-plaintext secure variant for a Type-3 pairing is presented in [BS23]. We prove the π_{vetibe} protocol secure under the hardness of the bilinear co-Diffie-Hellman problem, which is an adaptation to Type-3 pairings of the bilinear Diffie-Hellman problem that [BF01] used to prove the FullIdent scheme secure, and is also a computational variant of the assumption under which the chosen-plaintext secure scheme in [BS23] was proved secure.

Definition 3 (Bilinear co-Diffie-Hellman Problem)**.** *The advantage of an algorithm $\mathcal{A}$ in solving the bilinear co-Diffie-Hellman (co-BDH) problem in $(\mathbb{G}_1, \mathbb{G}_2)$ is defined as*

$$\Pr\left[y = \mathrm{e}(g_1, g_2)^{\alpha\beta\gamma} \ : \ \alpha, \beta, \gamma \leftarrow_{\$} \mathbb{Z}_q \ , \ y \leftarrow_{\$} \mathcal{A}(g_1^{\alpha}, g_1^{\beta}, g_2^{\beta}, g_2^{\gamma}) \right] \ .$$

Protocol π_{vetibe}

- On $(sid, \texttt{"init"})$, server $\mathcal{S}_i$ provides input $(sid_{\text{vetbls}}, \texttt{"init"})$ to $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$. When it receives output $(sid_{\text{vetbls}}, \texttt{"init"}, mpk)$ from $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$, it outputs $(sid, \texttt{"output-mpk"}, mpk)$.
- On $(sid, \texttt{"init"}, mpk')$, user $\mathcal{U}$ provides input $(sid_{\text{vetbls}}, \texttt{"transport-keygen"})$ to $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$. When it receives output $(sid_{\text{vetbls}}, \texttt{"tpk"}, tpk)$, $\mathcal{U}$ stores mpk' and tpk, and provides input $(sid_{\text{mca}}, \texttt{"register"}, tpk)$ to $\mathcal{F}_{\text{mca}}$.
- On $(sid, \texttt{"ekderive"}, id, \mathcal{U})$, server $\mathcal{S}_i$ retrieves tpk for $\mathcal{U}$ from $\mathcal{F}_{\text{mca}}$. It then calls $(sid_{\text{vetbls}}, \texttt{"encsign"}, id, tpk)$ on $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$. When it receives output $(sid_{\text{vetbls}}, \texttt{"encsign"}, id, tpk, es)$, it sends (id, es) to $\mathcal{U}$.
- When $\mathcal{U}$ receives (id, es), it recovers mpk' from its state and checks whether it already has a tuple (id, K) in its state; if so it ignores. Otherwise, it first calls $(sid_{\text{vetbls}}, \texttt{"verify"}, mpk', id, tpk, es)$ on $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$ to obtain output $(sid_{\text{vetbls}}, \texttt{"verify"}, mpk', id, tpk, es, b)$. If $b = \texttt{true}$, it then calls $(sid_{\text{vetbls}}, \texttt{"decrypt"}, mpk', id, tpk, es)$ on $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$ to obtain $(sid_{\text{vetbls}}, \texttt{"verify"}, mpk', id, tpk, es, K)$, stores (id, K) in its state, and outputs $(\texttt{"ekderive"}, id)$ to $\mathcal{U}$.
- On $(sid, \texttt{"encrypt"}, mpk', id, m)$, $\mathcal{P}$ checks that $m \in \{0,1\}^{\ell}$. It calls $(sid_{\text{vetbls}}, \texttt{"hash"}, id)$ on $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$ to obtain response h. It then chooses $s \leftarrow_{\$} \{0,1\}^{\kappa}$, sets $t \leftarrow \mathsf{H}_3(s, m)$, and computes the ciphertext

$$C \;\leftarrow\; \Big(\; g_2^t \;,\; s \oplus \mathsf{H}_2\big(\mathrm{e}(h, mpk')^t\big) \;,\; m \oplus H_4(s) \;\Big)$$

  and outputs $(sid, \texttt{"encrypt"}, mpk', id, m, C)$.
- On $(sid, \texttt{"decrypt"}, id, C)$, user $\mathcal{U}$ checks whether it has a record (id, K) in its state. If not, it ignores this input. Otherwise, it parses $C = (C_1, C_2, C_3)$, computes $s \leftarrow C_2 \oplus \mathsf{H}_2\big(\mathrm{e}(K, C_1)\big)$, and recovers $m \leftarrow C_3 \oplus \mathsf{H}_4(s)$. It also computes $t \leftarrow \mathsf{H}_3(s, m)$ and checks whether $C_1 = g_2^t$. If so, it outputs $(sid, \texttt{"decrypt"}, id, C, m)$, otherwise it outputs $(sid, \texttt{"decrypt"}, id, C, \perp)$.

Figure 11: The Boneh-Franklin IBE with verifiably encrypted threshold key derivation π_{vetibe} in the $(\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model.

Theorem 7. *If the co-BDH problem is hard in* $(\mathbb{G}_1, \mathbb{G}_2)$ *and* $\mathsf{H}_2, \mathsf{H}_3, \mathsf{H}_4$ *are modeled as random oracles, then the* π_{vetibe} *protocol securely realizes* $\mathcal{F}_{\text{vetibe}}$ *in the* $(\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$*-hybrid model as well as in the* $(\mathcal{F}^{\mathsf{cEG}_2}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$*-hybrid model.*

We repeat that, because any secure realization of $\mathcal{F}_{\text{vetbls}}$ also securely realizes $\mathcal{F}^{\mathsf{cEG}}_{\text{vetbls}}$, the π_{vetibe} protocol can by the theorem above also be securely instantiated with an instantiation of the leakage-free functionality $\mathcal{F}_{\text{vetbls}}$.

Proof. We first prove the theorem for the $(\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model, and then sketch the differences for the case of the $(\mathcal{F}^{\mathsf{cEG}_2}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model.

Consider the following simulator $\mathcal{S}im$ that interacts with $\mathcal{E}$ and $\mathcal{F}_{\text{vetibe}}$ in the ideal world and that internally runs the real-world adversary $\mathcal{A}$ as well as the functionalities $\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$ and $\mathcal{F}_{\text{mca}}$ to provide $\mathcal{A}$ with a view of π_{vetibe} in the $(\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid world.

For brevity, we refer to the random oracle implemented by the `"hash"` interface of $\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$ as a function H_1. The random oracles for H_2, H_3, and H_4 are managed by $\mathcal{S}im$ by keeping maps H_2, H_3, and H_4, respectively, assigning new inputs to random elements from the appropriate ranges $\{0,1\}^\kappa$, $\mathbb{Z}_q$, and $\{0,1\}^\ell$. One limitation, however, is that $\mathcal{S}im$ aborts if a query $\mathsf{H}_2\big(\mathrm{e}(\mathsf{H}_1(id), mpk)^t\big)$, $\mathsf{H}_3(s, \cdot)$, or $\mathsf{H}_4(s)$ is made such that there exists $(C, id, t, s) \in CTXT$.

It interacts with $\mathcal{F}_{\text{vetibe}}$ and $\mathcal{A}$ as follows:

- On $(\texttt{"init"}, \mathcal{S}_i)$ from $\mathcal{F}_{\text{vetibe}}$, $\mathcal{S}im$ lets "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$" process an input $(sid_{\text{vetbls}}, \texttt{"init"})$ from "$\mathcal{S}_i$". The first time it makes such a call, "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$" will generate a secret key $sk \leftarrow_\$ \mathbb{Z}_q$ and public key $pk \leftarrow g_2^{sk}$. The simulator $\mathcal{S}im$ stores $(mpk, msk) \leftarrow (pk, sk)$ and returns mpk to $\mathcal{F}_{\text{vetibe}}$.

- When a simulated honest server "$\mathcal{S}_i$" outputs $(sid, \texttt{"output-mpk"}, mpk)$, $\mathcal{S}im$ sends $(sid, \texttt{"output-mpk"}, \mathcal{S}_i)$ to $\mathcal{F}_{\text{vetibe}}$.

- On $(\texttt{"init"}, \mathcal{U}, mpk')$ from $\mathcal{F}_{\text{vetibe}}$, $\mathcal{S}im$ lets "$\mathcal{U}$" follow the code of π_{vetibe} on input $(sid, \texttt{"init"}, mpk')$, but interacting with the simulated ideal functionalities "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$" and "$\mathcal{F}_{\text{mca}}$". Meaning, "$\mathcal{U}$" generates a transport public key by invoking the `"transport-keygen"` interface on "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$" and registering the resulting tpk with "$\mathcal{F}_{\text{mca}}$".

- On $(sid, \texttt{"encrypt"}, mpk', id, m)$ from $\mathcal{F}_{\text{vetibe}}$, $\mathcal{S}im$ runs the honest `"encrypt"` interface of π_{vetibe} to produce a ciphertext C and sends C back to $\mathcal{F}_{\text{vetibe}}$.

- On $(sid, \texttt{"encrypt"}, mpk, id, \ell)$ from $\mathcal{F}_{\text{vetibe}}$, $\mathcal{S}im$ sets $C \leftarrow (C_1, C_2, C_3)$ where $t \leftarrow_\$ \mathbb{Z}_q$, $C_1 \leftarrow g_1^t$, $C_2 \leftarrow_\$ \{0,1\}^\kappa$, and $C_3 \leftarrow_\$ \{0,1\}^\ell$. It chooses $s \leftarrow_\$ \{0,1\}^\kappa$, adds (C, id, t, s) to $CTXT$, and responds C to $\mathcal{F}_{\text{vetibe}}$.

- On $(\texttt{"ekderive"}, id, \mathcal{U}, \mathcal{S}_i)$ from $\mathcal{F}_{\text{vetibe}}$ for an honest user $\mathcal{U}$, $\mathcal{S}im$ lets "$\mathcal{S}_i$" follow the π_{vetibe} protocol by retrieving tpk for $\mathcal{U}$ from "$\mathcal{F}_{\text{mca}}$", calling the `"encsign"` interface on "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$", and, when it receives output es, sending (id, es) to "$\mathcal{U}$".

  When "$\mathcal{U}$" receives (id, es), it also follows π_{vetibe} by checking whether it already has a tuple (id, K) in its state. If not, it verifies es by calling the `"verify"` interface of "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$". If it is valid, it sends $(sid, \texttt{"output-key"}, id, \mathcal{U})$ to $\mathcal{F}_{\text{vetibe}}$. Note that it doesn't need to bother decrypting es, as $\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$ guarantees that a valid encrypted signature decrypts to the correct BLS signature.

- On $(\texttt{"ekderive"}, id, \mathcal{U}, \mathcal{S}_i)$ from $\mathcal{F}_{\text{vetibe}}$ for a corrupt user $\mathcal{U}$, it goes over all $(C, id, t, s) \in CTXT$ to

  - parse $C = (C_1, C_2, C_3)$,
  - send an input $(sid, \texttt{"decrypt"}, id, C)$ to $\mathcal{F}_{\text{vetibe}}$ to obtain a response $(sid, \texttt{"decrypt"}, id, C, m)$,

  - compute $T \leftarrow \mathrm{e}(\mathsf{H}_1(\mathit{id}), \mathit{mpk})^t$,
  - program the maps for random oracles H_2, H_3, and H_4 as

$$H_2[T] \leftarrow C_2 \oplus s \quad , \quad H_3[s,m] \leftarrow t \quad , \quad H_4[s] \leftarrow m \oplus C_3 \ ,$$

  - and to delete (C, id, t, s) from CTXT.

  Note that programming the random oracles this way always works unless $\mathcal{Sim}$ aborted earlier. Also note that, by programming the random oracles in this way, the `"decrypt"` interface of π_{vetibe} correctly decrypts C under id to m.

- On $(\mathit{sid}, \texttt{"decrypt"}, \mathit{id}, (C_1, C_2, C_3))$ from $\mathcal{F}_{\text{vetibe}}$, $\mathcal{Sim}$ proceeds as for the `"decrypt"` interface of π_{vetibe}, using $\mathsf{H}_1(\mathit{id})^{sk}$ as the secret key, to obtain a message m (that could be $\perp$). It sends m back to $\mathcal{F}_{\text{vetibe}}$.

One can see that $\mathcal{Sim}$ perfectly simulates the real-world environment of π_{vetibe} as long as it doesn't abort. For an environment that calls the `"encrypt"` interface of $\mathcal{F}_{\text{vetibe}}$ q_{E} times, we have that $|\mathit{CTXT}| \le q_{\text{E}}$. Since the random string s in each record $(C, \mathit{id}, r, s) \in \mathit{CTXT}$ does not affect $\mathcal{A}$'s view at all until $\mathcal{Sim}$ aborts, the probability that $\mathcal{A}$ makes $\mathcal{Sim}$ abort in one query to $\mathsf{H}_3(s, \cdot)$ or $\mathsf{H}_4(s)$ is at most $q_{\text{E}}/2^\kappa$. The probability that it does so in any of its q_{H} queries is at most

$$1 - \left(1 - \frac{q_{\text{E}}}{2^\kappa}\right)^{q_{\text{H}}} \quad \le \quad \frac{q_{\text{H}} \cdot q_{\text{E}}}{2^\kappa} \ .$$

Any environment $\mathcal{E}$ and adversary $\mathcal{A}$ causing $\mathcal{Sim}$ to abort by making a random-oracle query $\mathsf{H}_2\big(\mathrm{e}(\mathsf{H}_1(\mathit{id}), \mathit{mpk})^r\big)$ can be turned into an algorithm $\mathcal{B}$ that solves the co-BDH problem as follows. On input (U_1, V_1, V_2, W_2), $\mathcal{B}$ runs $\mathcal{E}$ and $\mathcal{A}$ in an environment like the one provided by $\mathcal{Sim}$ above, except that

- instead of letting "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$" run the real code of $\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$, it uses $pk \leftarrow V_2$,
- it simulates the `"hash"` interface of "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$", i.e., the random oracle $\mathsf{H}_1(\mathit{id})$, by choosing $r \leftarrow_\$ \mathbb{Z}_q$, storing $H_1[\mathit{id}] \leftarrow (g_1^r, r)$, and returning g_1^r, except for one randomly guessed query $\mathsf{H}_1(\mathit{id}^*)$ where it returns U_1,
- it simulates the `"leakpars"` interface of "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$" by responding with $(h_1, h_2) = (g_1^v/V_1, g_2^v/V_2)$ for $v \leftarrow_\$ \mathbb{Z}_q$,
- on input $(\mathit{sid}_{\text{vetbls}}, \texttt{"encsign"}, \mathit{id}, \mathit{tpk})$ for an honest user's tpk, "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$" computes the leakage λ sent to $\mathcal{A}$ as
  - if $\mathit{id} \neq \mathit{id}^*$: $\lambda \leftarrow (g_1^w, h_1^w \cdot \sigma)$, where $w \leftarrow_\$ \mathbb{Z}_q$, $H[\mathit{id}] = (g_1^r, r)$ and $\sigma \leftarrow U_1^r$, and
  - if $\mathit{id} = \mathit{id}^*$: $\lambda = (\lambda_1, \lambda_2) \leftarrow (U_1 \cdot g_1^w, U_1^v \cdot g_1^{vw}/V_1^w)$, which can be seen to be correctly distributed by considering that $\lambda_1 = g_1^{\beta+w}$ and

$$\begin{aligned} \lambda_2 &= U_1^v \cdot g_1^{vw}/V_1^w \\ &= g_1^{\alpha v + vw - \beta w} \\ &= g_1^{\alpha(v-\beta)+w(v-\beta)} \cdot g_1^{\alpha\beta} \\ &= h_1^{\alpha+w} \cdot \mathsf{H}_1(\mathit{id}^*)^\beta \ . \end{aligned}$$

- on input $(\mathit{sid}_{\text{vetbls}}, \texttt{"encsign"}, \mathit{id}, \mathit{tpk})$ for a non-honest tpk, "$\mathcal{F}^{\mathsf{cEG}_1}_{\text{vetbls}}$" computes the BLS signature σ sent to $\mathcal{A}$ as $\sigma \leftarrow V_1^r$ if $H[\mathit{id}] = (g_1^r, r)$, or gives up if $\mathit{id} = \mathit{id}^*$.

- on one randomly chosen input $(\mathit{sid}, \texttt{"encrypt"}, \mathit{mpk}, \mathit{id}, \ell)$ from $\mathcal{F}_{\text{vetibe}}$, it gives up if $\mathit{id} \neq \mathit{id}^*$; otherwise, it sets $C^* \leftarrow (W_2, C_2^*, C_3^*)$ for $C_2^* \leftarrow_\$ \{0,1\}^\kappa$ and $C_3^* \leftarrow_\$ \{0,1\}^\ell$, adds $(C^*, \mathit{id}^*, \bot, s)$ to CTXT for $s \leftarrow_\$ \{0,1\}^\kappa$, and responds C^* to $\mathcal{F}_{\text{vetibe}}$.

- On $(\mathit{sid}, \texttt{"decrypt"}, \mathit{id}, (C_1, C_2, C_3)$ from $\mathcal{F}_{\text{vetibe}}$, $\mathcal{B}$ doesn't decrypt using the secret key for id, but instead checks whether there exist s, m such that $C_1 = g_2^{H_3[s,m]}$. If so, then it additionally checks whether $C_2 = s \oplus \mathsf{H}_2(\mathrm{e}(\mathsf{H}_1(\mathit{id}), \mathit{pk})^t)$. If so, then it returns $m \leftarrow C_3 \oplus \mathsf{H}_4(s)$ to $\mathcal{F}_{\text{vetibe}}$; if not, or if no such s, m were found, it returns $m \leftarrow \bot$ to $\mathcal{F}_{\text{vetibe}}$.

  The only way that this decryption method produces a different outcome than the real decryption of π_{vetibe} is if during decryption, no entry in H_3 can be found such that $C_1 = g_2^{H_3[s,m]}$, but later a new random-oracle query results in the creation of exactly such an entry. For an environment that makes q_{D} decryption queries, the probability that some query $\mathsf{H}_2(s, m)$ is assigned a random value t such that $g_2^t = C_1$ for some previous decryption query is at most $q_{\text{H}} \cdot q_{\text{D}}/q$.

To cause the original simulator $\mathcal{S}\!\mathit{im}$ to abort, $\mathcal{A}$ must make a query $\mathsf{H}_1(T)$ so that there exists $(C, \mathit{id}, t, s) \in \mathit{CTXT}$ such that $T = \mathrm{e}(\mathsf{H}_1(\mathit{id}), \mathit{mpk})^t$. Algorithm $\mathcal{B}$'s strategy is that the offending query $\mathsf{H}_1(T)$ occurs for the tuple $(C^*, \mathit{id}^*, \bot, s) \in \mathit{CTXT}$, in which case we would have that

$$\begin{aligned} T &= \mathrm{e}(\mathsf{H}_1(\mathit{id}^*), \mathit{mpk})^{\mathrm{dlog}_{g_2} C_1^*} \\ &= \mathrm{e}(U_1, V_2)^{\mathrm{dlog}_{g_2} W_2} \\ &= \mathrm{e}(g_1, g_2)^{\alpha\beta\gamma} \ , \end{aligned}$$

the solution to $\mathcal{B}$'s co-BDH challenge.

Since $\mathcal{B}$ can't test which query $\mathsf{H}_1(T)$ is the offending one, it simply outputs the key of a randomly chosen entry in H_1, giving it a probability of at least $1/q_{\text{H}}$ to guess correctly. That query also has to trigger abortion for the tuple $(C^*, \mathit{id}^*, \bot, s) \in \mathit{CTXT}$, however, which happens with probability $1/q_{\text{E}}$ *if* $\mathcal{B}$ didn't give up prematurely.

There are two reasons that cause $\mathcal{B}$ to give up prematurely. The first is if an honest server "$\mathcal{S}_i$" calls "$\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_1}$" with $(\mathit{sid}_{\text{vetbls}}, \texttt{"encsign"}, \mathit{id}^*, \mathit{tpk})$ for a non-honest tpk. However, an honest server in π_{vetibe} only makes such a call if it received an input $(\mathit{sid}, \texttt{"ekderive"}, \mathit{id}^*, \mathcal{U})$ for a corrupt $\mathcal{U}$, upon which $\mathcal{S}\!\mathit{im}$ immediately deletes all tuples $(\cdot, \mathit{id}^*, \cdot, \cdot)$ from CTXT.

The other reason for $\mathcal{B}$ to give up early is if the randomly chosen "encrypt" input is for $\mathit{id} \neq \mathit{id}^*$. This can indeed happen, but since $\mathcal{A}$'s view is independent of $\mathcal{B}$'s choice of id^* as long as it doesn't give up, we have a probability $1/q_{\text{H}}$ that $\mathit{id} = \mathit{id}^*$.

Overall, $\mathcal{B}$ therefore outputs the solution to the co-BDH problem with probability at least $1/(q_{\text{E}} \cdot q_{\text{H}}^2)$.

The proof in the $(\mathcal{F}_{\text{vetbls}}^{\mathsf{cEG}_2}, \mathcal{F}_{\text{mca}})$-hybrid model is almost identical to the above, except that the algorithm $\mathcal{B}$ solves the co-BDH problem slightly differently. Namely, it still sets $\mathit{pk} \leftarrow V_2$ and $\mathsf{H}(\mathit{id}^*) \leftarrow U_1$, but it simulates the leakage parameters lpars as $h_1 \leftarrow g_1^v/U_1$ and simulates the leakage as $\lambda \leftarrow (g_1^w, g_2^w, h_1^w \cdot \sigma)$ if $\mathit{id} \neq \mathit{id}^*$, and as $\lambda \leftarrow (V_1 \cdot g_1^w, V_2 \cdot g_2^w, V_1^v \cdot g_1^{vw}/U_1^w)$. $\square$

7 Other vet Primitives

BLS signatures are so much more than just the decryption keys to the Boneh-Franklin IBE. They can of course be used as signatures, but also as a pseudo-random function (PRF) or verifiable random function (VRF). In this section, we describe how our vetBLS protocols can be used as a building block to create verifiably encrypted threshold variants of all of these primitives.

Functionality $\mathcal{F}_{\text{vetsig}}$

- On $(sid, \texttt{"init"})$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  Send $(\texttt{"init"}, \mathcal{S}_i)$ to $\mathcal{S}im$. If pk isn't defined, wait for a response pk from $\mathcal{S}im$ and store pk.
- On $(sid, \texttt{"output-pk"}, \mathcal{S}_i)$ from $\mathcal{S}im$:
  If $\mathcal{S}_i \in \text{servers}(sid)$ and pk is defined, output $(sid, \texttt{"output-pk"}, pk)$ to $\mathcal{S}_i$.
- On $(sid, \texttt{"init"}, pk')$ from $\mathcal{U}$:
  Set $PK[\mathcal{U}] \leftarrow pk'$ and send $(\texttt{"init"}, \mathcal{U})$ to $\mathcal{S}im$.
- On $(sid, \texttt{"encsig"}, m, \mathcal{U})$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  If $PK[\mathcal{U}] = \perp$ then ignore. Add m to $ES[\mathcal{U}]$ and send $(\texttt{"encsig"}, m, \mathcal{U})$ to $\mathcal{S}im$.
- On $(sid, \texttt{"output-sig"}, m, \sigma, \mathcal{U})$ from $\mathcal{S}im$: If $PK[\mathcal{U}] = \perp$ or $(PK[\mathcal{U}], m, \sigma, \texttt{false}) \in V$ then ignore. If $PK[\mathcal{U}] = pk$ and $m \notin ES[\mathcal{U}]$ and there does not exist a corrupt $\mathcal{U}'$ such that $m \in ES[\mathcal{U}']$, then ignore. Add $(PK[\mathcal{U}], m, \sigma, \texttt{true})$ to V, add m to DS, and output $(\texttt{"output-sig"}, m, \sigma)$ to $\mathcal{U}$.
- On $(sid, \texttt{"verify"}, pk', m, \sigma)$ from $\mathcal{P}$:
  Send $(\texttt{"verify"}, pk', m, \sigma)$ to $\mathcal{S}im$ and wait for a response β from $\mathcal{S}im$. Add (pk', m, σ, b) to V and output $(sid, \texttt{"verify"}, pk', m, \sigma, b)$ to $\mathcal{P}$, where b is determined as follows:
  1. If $(pk', m, \sigma, \gamma) \in V$, set $b \leftarrow \gamma$.
  2. Else, if $pk' \neq pk$, set $b \leftarrow \beta$.
  3. Else, if $m \in DS$ or there exists a corrupt $\mathcal{U}$ such that $m \in ES[\mathcal{U}]$, set $b \leftarrow \beta$.
  4. Else, set $b \leftarrow \texttt{false}$.

Figure 12: The ideal functionality for verifiably encrypted threshold signatures $\mathcal{F}_{\text{vetsig}}$.

7.1 Verifiably Encrypted Threshold Signatures

We define verifiably encrypted threshold signatures (vetSIG) through the $\mathcal{F}_{\text{vetsig}}$ functionality in Figure 12. It enables a group of servers to securely transmit signatures to authenticated users if at least t servers agree to do so.

Just as in threshold signatures [ADN06], a valid signature on m can only be registered after at least one honest server agrees to participate in creating a (in our case, encrypted) signature on m. Unlike threshold signatures, however, the $\mathcal{F}_{\text{vetsig}}$ functionality will only allow such a signature to be registered *after* the signature was delivered to its honest recipient; if the recipient is corrupt, valid signatures can be registered immediately after one honest server participates.

The functionality stores a public key pk, a map $PK[\cdot]$, initially $\perp$, to keep track of the public key each user was initialized with, a map $ES[\cdot]$, initially $\emptyset$, to keep track of which messages were encrypted-signed to which users users, and an initially empty set DS to keep track of messages for which signatures were delivered to their recipients, and an initially empty set V to keep track of verified signatures.

We describe a simple protocol π_{vetsig} in the $(\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model in Figure 13 where the resulting signatures are BLS signatures. Users first call $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ to generate a transport key and register it with $\mathcal{F}_{\text{mca}}$. A server encrypted-signs a message to user $\mathcal{U}$ by encrypted-signing m using $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ under the transport key that it looks up in $\mathcal{F}_{\text{mca}}$. Verification is performed as in the $\mathcal{BLS}$ signature scheme.

Theorem 8. *Protocol π_{vetsig} securely realizes $\mathcal{F}_{\text{vetsig}}$ in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model if the $\mathcal{BLS}$ signature scheme is uf-cma$^{\mathcal{L}}$ secure when H is modeled as a random oracle.*

Proof. The simulator simply runs internal instance of $\mathcal{F}_{\text{vetbls}}$ and $\mathcal{F}_{\text{mca}}$ and lets simulated honest

Protocol π_{vetsig}

The `"init"` interfaces for servers $\mathcal{S}_i$ and users $\mathcal{U}$ are identical to π_{vetibe} in Figure 11. The other interfaces are implemented as:

- On $(sid, \texttt{"encsig"}, m, \mathcal{U})$, server $\mathcal{S}_i$ retrieves tpk for $\mathcal{U}$ from $\mathcal{F}_{\text{mca}}$ and calls $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ with $(sid_{\text{vetbls}}, \texttt{"encsign"}, m, tpk)$. When it receives outputs $(sid_{\text{vetbls}}, \texttt{"encsign"}, m, tpk, es)$, it sends (m, es) to $\mathcal{U}$.
- When $\mathcal{U}$ receives (m, es), it recovers pk' from its state and verifies es by calling $(sid_{\text{vetbls}}, \texttt{"verify"}, pk', m, tpk, es)$ on $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$. If es is valid, $\mathcal{U}$ decrypts the BLS signature σ by calling $(sid_{\text{vetbls}}, \texttt{"decrypt"}, pk', m, tpk, es)$ and outputs $(sid, \texttt{"output-sig"}, m, \sigma)$.
- On $(sid, \texttt{"verify"}, pk', m, \sigma)$, a party $\mathcal{P}$ first calls $(sid_{\text{vetbls}}, \texttt{"hash"}, m)$ on $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ to obtain the hash value h. It outputs $(sid, \texttt{"verify"}, pk', m, \sigma, b)$ with $b = \texttt{true}$ if $\mathrm{e}(\sigma, g_2) = \mathrm{e}(h, pk')$, otherwise with $b = \texttt{false}$.

Figure 13: The verifiably encrypted threshold BLS signatures protocol π_{vetsig} in the $(\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model.

parties follow π_{vetsig} based on the inputs from $\mathcal{F}_{\text{vetsig}}$. In particular,

- on $(\texttt{"init"}, \mathcal{S}_i)$, $\mathcal{S}im$ lets "$\mathcal{S}_i$" proceed as π_{vetsig} does on input $(sid, \texttt{"init"})$. If this is the first honest server calling `"init"`, it will trigger "$\mathcal{F}_{\text{vetbls}}$" to generate a fresh key pair (pk, sk), allowing $\mathcal{S}im$ to respond pk to $\mathcal{F}_{\text{vetsig}}$.
- when "$\mathcal{S}_i$" outputs $(sid, \texttt{"output-pk"}, pk)$, $\mathcal{S}im$ sends $(sid, \texttt{"output-pk"}, \mathcal{S}_i)$ to $\mathcal{F}_{\text{vetsig}}$.
- on $(\texttt{"init"}, \mathcal{U}, pk')$ from $\mathcal{S}im$ for an honest user $\mathcal{U}$, $\mathcal{S}im$ lets "$\mathcal{U}$" proceed as π_{vetsig} does on input $(sid, \texttt{"init"}, pk')$.
- when $\mathcal{A}$ registers a key with "$\mathcal{F}_{\text{mca}}$" in the name of a corrupt user $\mathcal{U}$, $\mathcal{S}im$ sends $(sid, \texttt{"init"}, pk)$ to $\mathcal{F}_{\text{vetsig}}$ as coming from $\mathcal{U}$.
- on $(\texttt{"encsig"}, m, \mathcal{U})$ from $\mathcal{F}_{\text{vetsig}}$, $\mathcal{S}im$ lets "$\mathcal{S}_i$" proceed as π_{vetsig} on input $(sid, \texttt{"encsig"}, m, \mathcal{U})$.
- when a simulated honest user "$\mathcal{U}$" receives a message (m, es), it also proceeds as prescribed by π_{vetsig}. If "$\mathcal{U}$" outputs $(sid, \texttt{"output-sig"}, m, \sigma)$, $\mathcal{S}im$ provides $(sid, \texttt{"output-sig"}, m, \sigma, \mathcal{U})$ to $\mathcal{F}_{\text{vetsig}}$.
- on $(\texttt{"verify"}, pk', m, \sigma)$ from $\mathcal{F}_{\text{vetsig}}$, $\mathcal{S}im$ proceeds as π_{vetsig} on an input $(sid, \texttt{"verify"}, pk', m, \sigma)$, i.e., queries the `"hash"` interface of "$\mathcal{F}_{\text{vetbls}}$" to obtain the hash value h for m and checks whether $\mathrm{e}(\sigma, g_2) = \mathrm{e}(h, pk')$. If so, $\mathcal{S}im$ responds $\beta = \texttt{true}$ to $\mathcal{F}_{\text{vetsig}}$, otherwise it responds $\beta = \texttt{false}$.

One can see that the only tangible difference in the view thus provided by $\mathcal{S}im$ to $\mathcal{A}$ is in the verification interface. Namely, whereas the simulation by $\mathcal{S}im$ always follows the verification equation of π_{vetsig}, i.e., $\mathrm{e}(\sigma, g_2) = \mathrm{e}(h, pk')$, the ideal functionality $\mathcal{F}_{\text{vetsig}}$ follows its own rules that could lead to a discrepancy between the ideal execution with $\mathcal{F}_{\text{vetsig}}$ and the hybrid execution with π_{vetsig}. In particular, such a discrepancy can occur if either

- a signature σ output by an honest user for a message m doesn't satisfy the verification equation $\mathrm{e}(\sigma, g_2) = \mathrm{e}(h, pk)$, or

- a signature σ does satisfy the verification equation for pk and m, but a valid signature for m was never delivered to an honest user, or encrypted-signed by an honest server to a corrupt user.

The first case is easily excluded as $\mathcal{F}_{\text{vetbls}}$ guarantees that honest users only output valid signatures. We show that any environment $\mathcal{E}$ and adversary $\mathcal{A}$ causing a discrepancy of the second type yields a successful uf-cma$^{\mathcal{L}}$ adversary $\mathcal{B}$ against $\mathcal{BLS}$ signatures.

On input public key pk and leakage parameters $lpars$, $\mathcal{B}$ runs $\mathcal{E}$ and $\mathcal{A}$ in an experiment much like that with the simulator $\mathcal{S}im$ above, but instead of letting "$\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$" execute the real code of $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, $\mathcal{B}$ lets "$\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$" use pk as public key and uses its own random oracle for $\mathsf{H}(\cdot)$ to respond to `"hash"` queries by $\mathcal{A}$. The only interfaces that involve the secret key sk, which $\mathcal{B}$ of course doesn't know, are the `"encsign"` and `"decrypt"` interfaces, that it simulates as follows:

- On $(sid_{\text{vetbls}}, \texttt{"encsign"}, m, tpk)$ from a simulated honest server "$\mathcal{S}_i$" and a tpk that is registered in "$\mathcal{F}_{\text{mca}}$" to an honest user, $\mathcal{B}$ queries its leakage oracle on m to obtain signature leakage λ that it includes in a message $(\texttt{"encsign"}, m, tpk, \mathcal{S}_i, \lambda)$ to $\mathcal{A}$. On such an input for a tpk that is not registered to an honest user, $\mathcal{B}$ queries its signing oracle on m to obtain signature σ to include in $(\texttt{"encsign"}, m, tpk, \mathcal{S}_i, \sigma)$ to $\mathcal{A}$.

- On $(sid_{\text{vetbls}}, \texttt{"decrypt"}, m, tpk, es)$ from a simulated honest user "$\mathcal{U}$" for a valid encrypted signature es, $\mathcal{B}$ queries its signing oracle on m and includes the resulting signature σ in the output $(sid, \texttt{"decrypt"}, m, tpk, es, \sigma)$ to "$\mathcal{U}$".

If at some point $\mathcal{E}$ calls $(sid, \texttt{"verify"}, pk, m, \sigma)$ that causes a discrepancy of the second type above, $\mathcal{B}$ outputs m, σ as its forgery. Note that this is a non-trivial forgery, because the fact that no valid signature for m was ever delivered to an honest user and m was never encrypted-signed by an honest server means that $\mathcal{B}$ never had to query its signing oracle on m. $\square$

7.2 Verifiably Encrypted Threshold PRF

The uniqueness of BLS signatures makes them easily amenable to a pseudo-random function (PRF) [GGM86] and a verifiable random function (VRF) [MRV99] in the random-oracle model. Whereas BLS signatures themselves are unpredictable, but not necessarily pseudo-random, a pseudo-random output is easily obtained by applying a hash function modeled as a random oracle.

The relation between unique signatures and VRFs has mainly been studied in the standard model [MRV99, Lys02]. The relation in the random-oracle model seems folklore; concrete schemes based on RSA and CDH have been proved secure [PWH+17].

Distributed PRFs [NPR99, Nie02b] and VRFs [Dod03] secret-share the secret key over multiple servers, so that a quorum of them is needed to evaluate the function. The random beacon of the Internet Computer [HMW18, CDH+22] uses the hash of a threshold BLS signature to obtain common randomness and was analyzed as a decentralized VRF in [GLOW21].

We describe how to use vetBLS to create verifiably encrypted threshold PRFs (vetPRFs) and VRFs (vetVRFs) that enable users to securely evaluate and obtain proofs from a threshold of servers so that the PRF/VRF output value remains hidden from the adversary. (It does not try to hide the input value, though.)

The $\mathcal{F}_{\text{vetprf}}$ ideal functionality in Figure 14 describes the ideal behavior of a vetPRF. The public key pk that is output by servers is used as a "public handle" to the PRF, allowing users to verify that they indeed obtain the correct output value for the chosen PRF instance. Users are initialized with a public key pk'; the correct output guarantee only holds if the user was initialized with $pk' = pk$.

When an honest server evaluates the PRF on a new input x for an honest user $\mathcal{U}$, a random output y is chosen and privately delivered to $\mathcal{U}$. The only way for the simulator to learn the output value is when an honest server evaluates x for a corrupt user.

Functionality $\mathcal{F}_{\text{vetprf}}$

- On $(sid, \texttt{"init"})$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  Send $(\texttt{"init"}, \mathcal{S}_i)$ to *Sim*. If pk isn't defined, wait for a response pk from *Sim* and store pk.
- On $(sid, \texttt{"output-pk"}, \mathcal{S}_i)$ from *Sim*:
  If $\mathcal{S}_i \in \text{servers}(sid)$ and pk is defined, output $(sid, \texttt{"output-pk"}, pk)$ to $\mathcal{S}_i$.
- On $(sid, \texttt{"init"}, pk')$ from $\mathcal{U}$:
  Set $PK[\mathcal{U}] \leftarrow pk'$ and send $(\texttt{"init"}, pk', \mathcal{U})$ to *Sim*.
- On $(sid, \texttt{"enceval"}, x, \mathcal{U})$ from honest $\mathcal{S}_i \in \text{servers}(sid)$:
  If $PK[\mathcal{U}] = \bot$ or pk is not defined then ignore. Add x to $EE[\mathcal{U}]$ and send $(\texttt{"enceval"}, x, \mathcal{U}, \mathcal{S}_i)$ to *Sim*.
- On $(sid, \texttt{"deliver-eval"}, x, \mathcal{U})$ from *Sim*:
  If $PK[\mathcal{U}] = \bot$ then ignore. If $PK[\mathcal{U}] = pk$ and $x \notin EE[\mathcal{U}]$ and there does not exist a corrupt $\mathcal{U}'$ such that $x \in EE[\mathcal{U}']$, then also ignore.

  If $PK[\mathcal{U}] = pk$ and $Y[pk, x] = \bot$ then choose $Y[pk, x] \leftarrow_{\$} \{0,1\}^{\ell}$. If $PK[\mathcal{U}] \neq pk$ and $Y[PK[\mathcal{U}], x] = \bot$ then send $(sid, \texttt{"deliver-eval"}, PK[\mathcal{U}], x)$ to *Sim*, wait for a response y from *Sim*, and set $Y[PK[\mathcal{U}], x] \leftarrow y$.

  Add x to $DE[\mathcal{U}]$ and output $(sid, \texttt{"deliver-eval"}, x)$ to $\mathcal{U}$.
- On $(sid, \texttt{"eval"}, x)$ from $\mathcal{U}$:
  If $x \notin DE[\mathcal{U}]$ then ignore, else output $(sid, \texttt{"eval"}, x, Y[PK[\mathcal{U}], x])$ to $\mathcal{U}$.

Figure 14: The ideal functionality for a verifiably encrypted threshold pseudo-random function $\mathcal{F}_{\text{vetprf}}$.

The functionality maintains the public key pk, a map $PK[\cdot]$ initialized to $\bot$ keeping track of the public keys with which users are initialized, a map $EE[\cdot]$ initialized to $\emptyset$ keeping track of which inputs were encrypted-evaluated for each user, a map $DE[\cdot]$ initialized to $\emptyset$ to keep track of the delivered evaluations for each user, and a map $Y[\cdot]$ initialized to $\bot$ to maintain consistency of the randomly chosen outputs.

The π_{vetprf} protocol in the $(\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model described in Figure 15 uses the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ functionality to have BLS signatures securely delivered to users, who have their transport public keys registered in $\mathcal{F}_{\text{mca}}$. The output of the PRF on an input x is given by $\mathsf{H}'(pk, x, \sigma)$, where H' is a hash function and σ is the BLS signature on $m = x$.

Theorem 9. *Protocol π_{vetprf} securely realizes $\mathcal{F}_{\text{vetprf}}$ in the $(\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model if H' is modeled as a random oracle and the $\mathcal{BLS}$ signature scheme is uf-cma$^{\mathcal{L}}$ secure when H is modeled as a random oracle.*

Proof. Consider a simulator *Sim* that runs simulated instances of $\mathcal{F}_{\text{mca}}$ and $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, written as "$\mathcal{F}_{\text{mca}}$" and "$\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$", as well as simulated honest parties "$\mathcal{S}_i$" and "$\mathcal{U}$". It mostly runs these running the real code of $\mathcal{F}_{\text{mca}}$, $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, and π_{vetprf}, but in the simulation of the random oracle H', it only programs the output values generated by $\mathcal{F}_{\text{vetprf}}$ as they become known to *Sim*. We show that any environment $\mathcal{E}$ and adversary $\mathcal{A}$ that make that programming fail give rise to a uf-cma$^{\mathcal{L}}$ forger for $\mathcal{BLS}$ with signature leakage $\mathcal{L}$.

The simulator *Sim* proceeds as follows:

- On $(\texttt{"init"}, \mathcal{S}_i)$ from $\mathcal{F}_{\text{vetprf}}$, *Sim* lets "$\mathcal{S}_i$" follow the instructions of π_{vetprf} on input $(sid, \texttt{"init"})$. When "$\mathcal{S}_i$" outputs $(sid, \texttt{"output-pk"}, pk)$, then it sends pk to $\mathcal{F}_{\text{vetprf}}$ if this is the first honest server to output a public key, and outputs $(sid, \texttt{"output-pk"}, \mathcal{S}_i)$ to $\mathcal{F}_{\text{vetprf}}$.
- On $(\texttt{"init"}, pk', \mathcal{U})$ from $\mathcal{F}_{\text{vetprf}}$, *Sim* lets "$\mathcal{U}$" follow π_{vetprf} on input $(sid, \texttt{"init"}, pk')$.

Protocol π_{vetprf}

- On $(sid, \texttt{"init"})$, server $\mathcal{S}_i \in \text{servers}(sid)$ calls $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ with $(sid_{\text{vetbls}}, \texttt{"init"})$. When it receives an output $(sid_{\text{vetbls}}, \texttt{"output-pk"}, pk)$ from $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, it stores pk and outputs $(sid, \texttt{"output-pk"}, pk)$.
- On $(sid, \texttt{"init"}, pk')$, user $\mathcal{U}$ calls $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ with $(sid_{\text{vetbls}}, \texttt{"transport-keygen"})$ and waits for output $(sid_{\text{vetbls}}, \texttt{"tpk"}, tpk)$ from $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$. It then registers tpk with $\mathcal{F}_{\text{mca}}$ and stores tpk and pk'.
- On $(sid, \texttt{"enceval"}, x, \mathcal{U})$, server $\mathcal{S}_i$ recovers the stored public key pk and retrieves the transport public key tpk registered for $\mathcal{U}$ from $\mathcal{F}_{\text{mca}}$; any of these is not found, it ignores this input. It then calls $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ with $(sid_{\text{vetbls}}, \texttt{"encsign"}, x, tpk)$. When it receives an output $(sid_{\text{vetbls}}, \texttt{"encsign"}, x, tpk, es)$ from $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, it sends (pk, x, tpk, es) to $\mathcal{U}$.
- When $\mathcal{U}$ receives a message (pk, x, tpk', es), it recovers tpk and pk' from its storage. If $pk \neq pk'$ or $tpk \neq tpk'$, it ignores this message. Otherwise, it calls $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ with $(sid_{\text{vetbls}}, \texttt{"decrypt"}, pk', x, tpk, es)$. If $\mathcal{U}$ then receives an output $(sid_{\text{vetbls}}, \texttt{"decrypt"}, pk', x, tpk, es, \sigma)$ from $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, it computes $y \leftarrow \mathsf{H}'(pk', x, \sigma)$, stores (x, y, σ) in its state, and outputs $(sid, \texttt{"deliver-eval"}, x)$.
- On $(sid, \texttt{"eval"}, x)$ from $\mathcal{U}$:
  If no tuple (x, y, σ) is stored in the state, ignore. Else, output $(sid, \texttt{"eval"}, x, y)$.

Figure 15: The verifiably encrypted threshold pseudo-random function protocol π_{vetprf} in the $(\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model with hash function $\mathsf{H}' : \{0,1\}^* \rightarrow \{0,1\}^{\ell}$.

- When $\mathcal{A}$ registers a key tpk with "$\mathcal{F}_{\text{mca}}$" in name of a corrupt user $\mathcal{U}$, $\mathcal{Sim}$ follows the code of $\mathcal{F}_{\text{mca}}$ but additionally inputs $(sid, \texttt{"init"}, pk)$ in name of $\mathcal{U}$ to $\mathcal{F}_{\text{vetprf}}$.
- On $(\texttt{"enceval"}, x, \mathcal{U}, \mathcal{S}_i)$ from $\mathcal{F}_{\text{vetprf}}$, $\mathcal{Sim}$ acts differently depending whether $\mathcal{U}$ is honest or corrupt. If $\mathcal{U}$ is honest, it lets "$\mathcal{S}_i$" follow π_{vetprf} on input $(sid, \texttt{"enceval"}, x, \mathcal{U})$, causing it to send a message (pk, x, tpk, es) to "$\mathcal{U}$".

  If $\mathcal{U}$ is corrupt, however, $\mathcal{Sim}$ first provides an input $(sid, \texttt{"deliver-eval"}, x, \mathcal{U})$ to $\mathcal{F}_{\text{vetprf}}$ and a subsequent input $(sid, \texttt{"eval"}, x)$ as coming from $\mathcal{U}$ so that $\mathcal{Sim}$, who plays the role of the ideal-world $\mathcal{U}$, obtains output $(sid, \texttt{"eval"}, x, y)$. It then programs the random oracle for H' by setting $H'[pk, x, \sigma] \leftarrow y$, where $\sigma \leftarrow h^{sk}$ and h is obtained by calling $(sid_{\text{vetbls}}, \texttt{"hash"}, x)$ on "$\mathcal{F}_{\text{vetbls}}$"; if the entry $H'[pk, x, \sigma]$ is already defined, $\mathcal{Sim}$ aborts.
- When "$\mathcal{U}$" receives a message (pk'', x, tpk', es), it initially follows the honest code of π_{vetprf}, but if $pk'' = pk$ and "$\mathcal{U}$" receives a $\texttt{"decrypt"}$ output from "$\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$", it doesn't make an internal random-oracle call $\mathsf{H}'(pk, x, \sigma)$ but simply sets $y \leftarrow \perp$.

  When "$\mathcal{U}$" produces an output $(sid, \texttt{"deliver-eval"}, x, y)$, $\mathcal{Sim}$ provides an input $(sid, \texttt{"deliver-eval"}, x, \mathcal{U})$ to $\mathcal{F}_{\text{vetprf}}$. If it then receives $(sid, \texttt{"deliver-eval"}, pk'', x)$ from $\mathcal{F}_{\text{vetprf}}$, it responds with y to $\mathcal{Sim}$.
- If $\mathcal{A}$ makes a random-oracle query $\mathsf{H}'(pk', x, \sigma)$, $\mathcal{Sim}$ proceeds as follows. If $pk = pk'$, $\sigma = H[x]^{sk}$, and $H'[pk, x, \sigma] = \perp$, $\mathcal{Sim}$ aborts. Otherwise, it returns $H'[pk', x, \sigma]$, assigning it a random value from $\{0,1\}^{\ell}$ if it is not yet defined.

The simulation provided by $\mathcal{Sim}$ is perfect as long as $\mathcal{Sim}$ does not abort. Any environment $\mathcal{E}$ and $\mathcal{A}$ that cause $\mathcal{Sim}$ to abort can be turned into a uf-cma$^{\mathcal{L}}$ adversary $\mathcal{B}$ against $\mathcal{BLS}$ as follows.

Algorithm $\mathcal{B}$ runs $\mathcal{E}$ and $\mathcal{A}$ in the same environment as with $\mathcal{F}_{\text{vetprf}}$ and $\mathcal{Sim}$ above, except that "$\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$" no longer runs the real code of $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, but uses information from its own uf-cma$^{\mathcal{L}}$ experiment to simulate it. In particular, on input $(pk, lpars)$, $\mathcal{B}$ uses pk as the public key of $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$; forwards queries to its $\texttt{"hash"}$ interface to its own random oracle for H; queries its signing oracle

Functionality $\mathcal{F}_{\text{vetvrf}}$

The functionality contains all the interfaces of $\mathcal{F}_{\text{vetprf}}$ in Figure 14 and adds the following ones:

- On $(sid, \texttt{"proof"}, x)$ from $\mathcal{U}$:
  If $x \notin DE[\mathcal{U}]$ then ignore. Else, send $(\texttt{"proof"}, PK[\mathcal{U}], x, Y[PK[\mathcal{U}], x])$ to $\mathcal{S}im$ and wait for a response ϕ from $\mathcal{S}im$ so that $(PK[\mathcal{U}], x, \phi, \texttt{false}) \notin V$. Add $(PK[\mathcal{U}], x, \phi, \texttt{true})$ to V and output $(sid, \texttt{"proof"}, x, \phi)$ to $\mathcal{U}$. Output $(sid, \texttt{"proof"}, x, y, \phi)$ to $\mathcal{U}$.
- On $(sid, \texttt{"verify-proof"}, pk', x, y, \phi)$ from $\mathcal{P}$:
  Send $(\texttt{"verify-proof"}, pk', x, y, \phi)$ to $\mathcal{S}im$ and wait for a response β from $\mathcal{S}im$. Add (pk', x, y, ϕ, b) to V and output $(sid, \texttt{"verify-proof"}, pk', x, y, \phi, b)$ to $\mathcal{P}$, where b is determined as follows:
  1. If $(pk', x, y, \phi, \gamma) \in V$, set $b \leftarrow \gamma$.
  2. Else, if $pk' \neq pk$, set $b \leftarrow \beta$.
  3. Else, if $y \neq Y[pk', x]$, set $b \leftarrow \texttt{false}$.
  4. Else, if $\exists\ (pk, x, y, \cdot, \texttt{true}) \in V$, set $b \leftarrow \beta$.
  5. Else, if there exists a corrupt $\mathcal{U}$ such that $x \in EE[\mathcal{U}]$, set $b \leftarrow \beta$.
  6. Else, set $b \leftarrow \texttt{false}$.

Figure 16: The ideal functionality for a verifiably encrypted threshold verifiable random function $\mathcal{F}_{\text{vetvrf}}$.

on m when it needs to compute $\sigma = H[m]^{sk}$ to respond to an `"encsign"` input for a corrupt user's tpk; queries its leakage oracle on m when it needs to compute $\lambda \leftarrow_{\$} \mathcal{L}(lpars, \sigma)$ to respond to an `"encsign"` input for an honest user's tpk; and replaces the check whether $\sigma = H[x]^{sk}$ in the simulation of H' with the equivalent check whether $\mathrm{e}(\sigma, g_2) = \mathrm{e}(\mathsf{H}(x), pk)$.

One can see that $\mathcal{S}im$ aborts only when $\mathcal{A}$ makes a random-oracle query $\mathsf{H}'(pk, x, \sigma)$ with $\sigma = \mathsf{H}(x)^{sk}$ before x was encrypted-evaluated to a corrupt user. If this happens, $\mathcal{B}$ outputs x, σ as its forgery, which is non-trivial because $\mathcal{B}$ would only query its signing oracle on x if it gets encrypted-evaluated to a corrupt user. □

7.3 Verifiably Encrypted Threshold VRF

The attentive reader will have noticed that the BLS signature σ in the PRF construction above could be used as a verifiable proof the the PRF output is correct, turning the PRF into a VRF. In Figures 16 and 17, we describe an ideal functionality $\mathcal{F}_{\text{vetvrf}}$ and a protocol π_{vetvrf} for a verifiably encrypted threshold VRF that add proving and verification interfaces to the PRF functionality and protocol above. The resulting protocol can be seen as a verifiably encrypted variant of the DFINITY decentralized VRF that is used as the random beacon for the Internet Computer [HMW18, CDH$^+$22, GLOW21].

Theorem 10. *Protocol π_{vetvrf} securely realizes $\mathcal{F}_{\text{vetvrf}}$ in the $(\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model if H' is modeled as a random oracle and the $\mathcal{BLS}$ signature scheme is uf-cma$^{\mathcal{L}}$ secure when H is modeled as a random oracle.*

Proof. The simulator and proof are very similar to the proof for π_{vetprf} in Theorem 9, so we only sketch the differences here. Consider the simulator $\mathcal{S}im$ that behaves identically to that of Theorem 9, but adds the following actions:

- On $(\texttt{"proof"}, pk', x, y)$ from $\mathcal{F}_{\text{vetvrf}}$, $\mathcal{S}im$ finds a simulated user "$\mathcal{U}$" with a state that contains pk' as well as a tuple $(x, \cdot, \sigma)$, which must exist if $\mathcal{F}_{\text{vetvrf}}$ sent this message. If $pk' = pk$ and

Protocol π_{vetvrf}

The protocol contains all of the interfaces of π_{vetprf} in Figure 15 and adds the following ones:

- On $(sid, \texttt{"proof"}, x)$, $\mathcal{U}$ checks whether its state contains a tuple (x, y, σ). If so, it outputs $(sid, \texttt{"proof"}, x, y, \phi)$.
- On $(sid, \texttt{"verify-proof"}, pk, x, y, \phi)$, $\mathcal{P}$ calls $(sid_{\text{vetbls}}, \texttt{"hash"}, x)$ on $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ to obtain a response h. It then checks whether $\mathrm{e}(\phi, g_2) = \mathrm{e}(pk, h)$ and whether $\mathsf{H}'(pk, x, \sigma) = y$. It outputs $(sid, \texttt{"verify-proof"}, pk, x, y, \phi, b)$ where $b \leftarrow \texttt{true}$ if both checks pass and $b \leftarrow \texttt{false}$ otherwise.

Figure 17: The verifiably encrypted threshold verifiable random function protocol π_{vetvrf} in the $(\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}, \mathcal{F}_{\text{mca}})$-hybrid model with hash function $\mathsf{H}' : \{0,1\}^* \rightarrow \{0,1\}^\ell$.

$H[pk, x, \sigma] = \perp$, it programs the random-oracle entry $\mathsf{H}'(pk, x, \sigma)$ by setting $H'[pk, x, \sigma] \leftarrow y$. It then responds σ to $\mathcal{F}_{\text{vetvrf}}$.

- On input $(\texttt{"verify-proof"}, pk', x, y, \phi)$ from $\mathcal{F}_{\text{vetvrf}}$, $\mathcal{Sim}$ executes the honest code of π_{vetvrf} on input $(sid, \texttt{"verify-proof"}, pk', x, y, \phi)$ to obtain output $(\texttt{"verify-proof"}, pk', x, y, \phi, \beta)$ and responds β to $\mathcal{F}_{\text{vetvrf}}$.

In particular, $\mathcal{Sim}$ simulates the random oracle for H' in the same way as in Theorem 9, aborting whenever a query $\mathsf{H}'(pk, x, \sigma)$ is made with $\sigma = H[x]^{sk}$ and $H'[pk, x, \sigma] = \perp$.

Entries in H' are programmed at two occasions now: when an honest server encrypted-evaluates x for a corrupt user, and when an honest user calls the "proof" interface. Note that the latter event immediately leads to a valid proof ϕ for x being added to V in $\mathcal{F}_{\text{vetvrf}}$, so that the two occasions where the random oracle gets programmed correspond exactly with cases 4 and 5 of the "verify-proof" interface of $\mathcal{F}_{\text{vetvrf}}$.

Programming these random-oracle entries never fails, because then the simulator must have aborted on an earlier random-oracle query already. One can prove that any environment and adversary that cause $\mathcal{Sim}$ to abort can be turned into a uf-cma$^{\mathcal{L}}$ forger $\mathcal{B}$ against $\mathcal{BLS}$ in a similar way as in Theorem 9, except that now $\mathcal{B}$ also consults its signing oracle on x to respond to incoming message $(\texttt{"proof"}, pk, x, y)$ from $\mathcal{F}_{\text{vetvrf}}$, excluding x from being used in $\mathcal{B}$'s forgery. This is fine, because $\mathcal{B}$ programs $H'[pk, x, \sigma] \leftarrow y$ immediately after that, so that random-oracle queries on x can no longer cause $\mathcal{Sim}$ to abort.

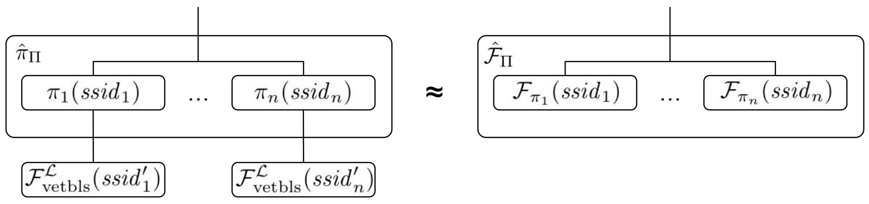
We leave further details as an exercise to the reader. $\square$

8 Secure Single-Key Composition

Multi-session functionalities and protocols. Consider the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ functionality depicted in Figure 7. Let Π be a set of protocols in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model so that $\pi | \mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ securely realizes some functionality $\mathcal{F}_\pi$ for every $\pi \in \Pi$.

Analogously to the multi-session extension of a single functionality [CR03], we define the multi-session extension $\hat{\mathcal{F}}_\Pi$ of a set of functionalities $\{\mathcal{F}_\pi : \pi \in \Pi\}$ as follows. Every input to $\hat{\mathcal{F}}_\Pi$ specifies, apart from the session identifier sid, also a sub-session identifier $ssid$. Let $\{SSID_\pi : \pi \in \Pi\}$ be a set of disjoint sets of sub-session identifiers, each associated with a different protocol $\pi \in \Pi$. When the multi-session extension $\hat{\mathcal{F}}_\Pi$ receives a message for a new sub-session $ssid \in SSID_\pi$, it internally creates a new instance of $\mathcal{F}_\pi$ to which all future inputs for $ssid$ get routed.

Let $\hat{\pi}_\Pi$ be the corresponding multi-instance protocol that for every $ssid \in SSID_\pi$ internally runs an instance of π. Standard UC composition [Can01] guarantees that $\hat{\pi}_\Pi$ securely realizes $\hat{\mathcal{F}}_\Pi$ in the


$\hat{\pi}_\Pi$
$\pi_1(ssid_1)$
...
$\pi_n(ssid_n)$
$\approx$
$\hat{\mathcal{F}}_\Pi$
$\mathcal{F}_{\pi_1}(ssid_1)$
...
$\mathcal{F}_{\pi_n}(ssid_n)$
$\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}(ssid'_1)$
$\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}(ssid'_n)$

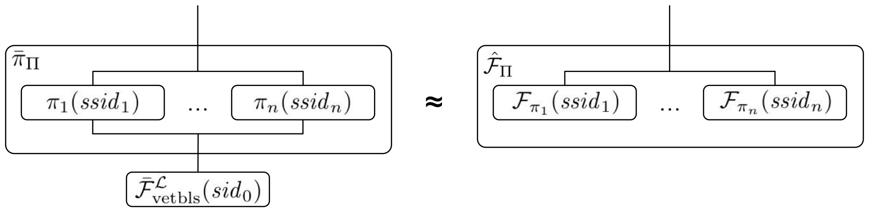

$\bar{\pi}_\Pi$
$\pi_1(ssid_1)$
...
$\pi_n(ssid_n)$
$\approx$
$\hat{\mathcal{F}}_\Pi$
$\mathcal{F}_{\pi_1}(ssid_1)$
...
$\mathcal{F}_{\pi_n}(ssid_n)$
$\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}(sid_0)$

Figure 18: Top: The multi-session composition $\hat{\pi}_\Pi$ of a set of protocols Π securely realizes the multi-session functionality $\hat{\mathcal{F}}_\Pi$ in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model, as implied by standard UC composition. Bottom: The single-key composition $\bar{\pi}_\Pi$ in the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model that we want to achieve.

$\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model, as depicted at the top of Figure 18. For this guarantee to hold, however, each instance of any $\pi \in \Pi$ must use its own separate instance of $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, each with its own key pair (pk, sk).

Single-key composition. We are rather interested in the security of a *single-key composition* $\bar{\pi}_\Pi$ as depicted at the bottom of Figure 18, where all protocol instances use a single instance of a multi-session but single-key functionality $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$. (Note that joint-state universal composability [CR03] doesn't guarantee such composition, as it uses the multi-session extension $\hat{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ of $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ as the common underlying functionality, which is a multi-key functionality.)

Let $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ be the functionality that internally runs a single instance of $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, generating a single key pair (pk, sk), but that separates the message spaces of the different sub-sessions by prepending the sub-session identity *ssid* to all messages. More particularly, on an input $(sid, ssid, v)$ from a party $\mathcal{P}$, $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ passes an input (sid, v) to the internal instance of $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ as coming from $\mathcal{P}$, but replacing any message m occurring in v with $m' = (ssid, m)$. Likewise, when such a call to the internal instance of $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ produces an output (sid, v) for a party $\mathcal{P}'$, $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ outputs $(sid, ssid, v)$ to $\mathcal{P}'$, but replacing any message $m' = (ssid, m)$ in v with m.

We define the single-key composition $\bar{\pi}_\Pi$ of Π as the following protocol in the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model:

- when called with a sub-session $ssid \in SSID_\pi$ for $\pi \in \Pi$, $\bar{\pi}_\Pi$ routes the input to an internal instance of π associated with *ssid*;
- an instance *sid* of $\bar{\pi}_\Pi$ invokes only a single instance sid_0 of $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$;
- whenever an internal instance of π associated with *ssid* instructs a party $\mathcal{P}_i$ to send an input (sid', v) to $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, $\bar{\pi}_\Pi$ sends an input $(sid_0, ssid, v)$ to $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$;
- whenever a party $\mathcal{P}_i$ running $\bar{\pi}_\Pi$ receives an input $(sid_0, ssid, v)$ from $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ for $ssid \in SSID_\pi$, it follows the instructions of π on receiving the message (sid', v) from $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$.

It is tempting to expect that if $\pi|\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ securely realizes $\mathcal{F}_\pi$ for all $\pi \in \Pi$, then $\bar{\pi}_\Pi|\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ securely realizes $\hat{\mathcal{F}}_\Pi$. Unfortunately, that turns out to be false: consider, for example, a functionality $\mathcal{F}$ that lets all parties agree on a common random group element, together with a protocol π that

Protocol $\tilde{\pi}$

Identical to protocol π, but adding the interface:

- On $(sid, \texttt{"extended-pk"})$, $\tilde{\mathcal{P}}$ provides input $(sid_0, \texttt{"extended-pk"})$ to $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ to receive output (sid_0, pk_1, pk_2), and outputs $(sid, \texttt{"extended-pk"}, pk_1, pk_2)$.

Functionality $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$

Identical to functionality $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, but adding the interface:

- On $(sid, \texttt{"extended-pk"})$ from $\tilde{\mathcal{P}}$:
  If (pk, sk) is defined, output $(sid, \texttt{"extended-pk"}, g_1^{sk}, g_2^{sk})$ to $\tilde{\mathcal{P}}$, else output $(sid\texttt{"extended-pk"}, \perp)$ to $\tilde{\mathcal{P}}$.

Functionality $\tilde{\mathcal{F}}_{\pi}$

Identical to functionality $\mathcal{F}_{\pi}$, but adding the interface:

- On $(sid, \texttt{"extended-pk"})$ from $\mathcal{P} \in \{\tilde{\mathcal{P}}, \mathcal{S}im\}$:
  If (pk_1, pk_2) isn't defined yet, generate $x \leftarrow_\$ \mathbb{Z}_p$, store $(pk_1, pk_2) \leftarrow (g_1^x, g_2^x)$. Output $(sid, \texttt{"extended-pk"}, pk_1, pk_2)$ to $\mathcal{P}$.

Figure 19: The imposed-key extensions $\tilde{\pi}$, $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$, and $\tilde{\mathcal{F}}_{\pi}$ for a protocol π that securely realizes $\mathcal{F}_{\pi}$ in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model. The party $\tilde{\mathcal{P}}$ is a new dedicated party that cannot be corrupted.

realizes it in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model by using the public key as the common group element. The single-key composition $\bar{\pi}_{\{\pi\}}$ of multiple instances of π in the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model obviously doesn't securely realize $\hat{\mathcal{F}}_{\{\pi\}}$, because all sub-sessions will end up with the same group element, instead of independently random ones.

Imposed-key simulatability. We want to go even further than a protocol π that securely single-key composes with itself; we want it to securely single-key compose with different protocols π', too. It would of course be very inconvenient to have to prove that secure composition separately for each combination of protocols.

Rather, we define a requirement for a single protocol π in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model that, if all protocols in Π satisfy it, guarantees that they can all be securely single-key composed as $\bar{\pi}_{\Pi}$ in the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model. Namely, we require that the protocol remains simulatable if, in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid world, the environment is additionally given the extended public key (g_1^{sk}, g_2^{sk}), while in the ideal world, both the environment and the simulator are given a pair of elements (g_1^x, g_2^x) for a random $x \leftarrow_\$ \mathbb{Z}_p$. This imposes an additional restriction on the simulator, as it must now be able to simulate the real-world protocol for a given public key for the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$ functionality, without knowing the corresponding secret key.

We define this formally by considering the standard UC experiment for the *imposed-key extensions* $\tilde{\pi}$, $\tilde{\mathcal{F}}_{\pi}$, and $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ of π, $\mathcal{F}_{\pi}$, and $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, respectively, as described in Figure 19. Since the environment in the UC framework cannot access protocols and functionalities directly, we introduce a dedicated incorruptible party $\tilde{\mathcal{P}}$ through which the environment can obtain the extended public key. We say that π *imposed-key realizes* $\mathcal{F}_{\pi}$ in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model if $\tilde{\pi}$ securely realizes $\tilde{\mathcal{F}}_{\pi}$ in the $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model, as graphically depicted at the bottom of Figure 20.

Theorem 11. *Let* Π *be a set of protocols and* $\{\mathcal{F}_{\pi} : \pi \in \Pi\}$ *be a set of corresponding functionalities.*

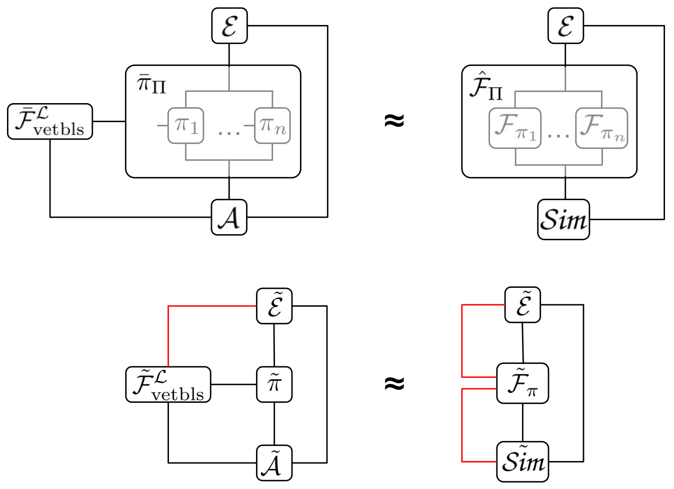


Figure 20: Top: The statement to be proved in Theorem 11, with the real-world experiment on the left where an environment $\mathcal{E}$ faces the single-key multi-session protocol $\bar{\pi}_\Pi$ in the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model, and the the ideal world on the right where $\mathcal{E}$ interacts with the multi-session functionality $\hat{\mathcal{F}}_\Pi$. Bottom: The precondition of Theorem 11 that $\tilde{\pi}$ securely realizes $\tilde{\mathcal{F}}_\pi$ in the $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model. The red wires depict the additional access to the `"extended-pk"` interfaces.

If for all $\pi \in \Pi$, π imposed-key realizes $\mathcal{F}_\pi$ in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model, then the single-key composition $\bar{\pi}_\Pi$ securely realizes the multi-session functionality $\hat{\mathcal{F}}_\Pi$ in the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model.

Proof. The statement that we're looking to prove is graphically depicted in Figure 20: we want to show that there exists a simulator $\mathcal{S}im$ such that no efficient environment $\mathcal{E}$ and adversary $\mathcal{A}$ can distinguish the real experiment with the single-key protocol $\bar{\pi}_\Pi$ and functionality $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ from the ideal experiment with the multi-session functionality $\hat{\mathcal{F}}_\Pi$.

We prove the theorem through a hybrid argument. Consider a sequence of games $\mathcal{G}_0, \ldots, \mathcal{G}_n$, where n is an upper bound on the number of sub-sessions created by the environment. The sequence gradually changes the experiment from the real world on the left of Figure 20 to the ideal world on the right.

Because $\tilde{\pi}$ securely realizes $\tilde{\mathcal{F}}_\pi$ in the $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model for all $\pi \in \Pi$, there exists a simulator $\tilde{\mathcal{S}im}_\pi$ so that no efficient environment $\tilde{\mathcal{E}}$ and adversary $\tilde{\mathcal{A}}$ can distinguish $\tilde{\pi}$ in the $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid world from $\tilde{\mathcal{F}}_\pi$ in combination with $\tilde{\mathcal{S}im}_\pi$ in the ideal world.

In game $\mathcal{G}_i$, graphically depicted in Figure 21, the environment interacts with a hybrid protocol $\bar{\pi}_i$ and a simulator $\mathcal{S}im_i$. The hybrid protocol $\bar{\pi}_i$ replaces the first i protocol sub-sessions $\pi_1, \ldots, \pi_i$ in $\bar{\pi}_\Pi$ with their ideal functionalities $\mathcal{F}_{\pi_1}, \ldots, \mathcal{F}_{\pi_i}$. The simulator $\mathcal{S}im_i$ then presents the adversary $\mathcal{A}$ with a simulation of the first i protocol sub-sessions provided by the imposed-key simulators $\tilde{\mathcal{S}im}_{\pi_1}, \ldots, \tilde{\mathcal{S}im}_{\pi_i}$, and an unmodified view of $\pi_{i+1}, \ldots, \pi_n$ for the remaining sub-sessions.

In more detail, the simulator $\mathcal{S}im_i$ works as follows:

- It generates a secret key $sk \leftarrow_\$ \mathbb{Z}_p$ and computes the corresponding public key $pk \leftarrow g_2^{sk}$ that will be used for the simulation of $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$.
- It internally runs a separate instance of $\tilde{\mathcal{S}im}_{\pi_j}$ associated with $ssid_j$ for $j = 1, \ldots, i$, relaying inputs and outputs between $\tilde{\mathcal{S}im}_{\pi_j}$ and the internal instance of $\mathcal{F}_{\pi_j}$ associated with the same $ssid_j$ in $\bar{\pi}_i$. However, when $\tilde{\mathcal{S}im}_{\pi_j}$ makes a call to the `"extended-pk"` interface of $\tilde{\mathcal{F}}_{\pi_j}$, $\mathcal{S}im_i$

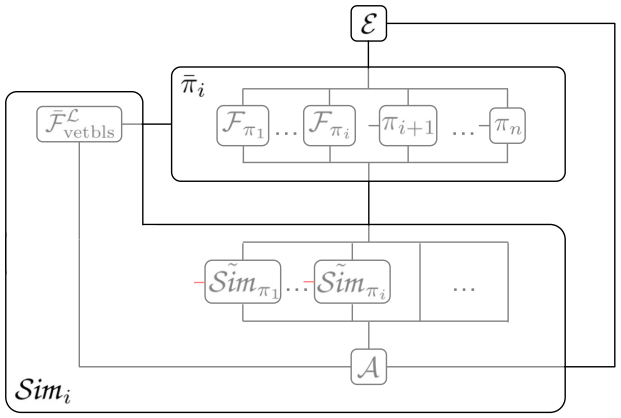


Figure 21: Game $\mathcal{G}_i$ in the hybrid argument in the proof of Theorem 11. The red wires extending to the left of $\tilde{\mathcal{Sim}}_{\pi_j}$ indicate the access to the "extended-pk" interface of $\tilde{\mathcal{F}}_{\pi_j}$ that must be simulated by $\mathcal{Sim}_i$.

responds with $(\texttt{"extended-pk"}, g_1^{sk}, g_2^{sk})$, thereby imposing the public key pk to the simulation provided by $\tilde{\mathcal{Sim}}_{\pi_j}$.

- It simulates the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ functionality honestly (i.e., following the code of $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$) for all queries involving sub-sessions $ssid_{i+1}, \ldots, ssid_n$, but relays responses from $\tilde{\mathcal{Sim}}_1, \ldots, \tilde{\mathcal{Sim}}_i$ for all calls to sub-sessions $ssid_1, \ldots, ssid_i$ of $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ in the following way.

  When $\bar{\pi}_i$ or $\mathcal{A}$ provides an input $(sid_0, ssid, v)$ to $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ for some $ssid \in \{ssid_1, \ldots, ssid_i\}$, it provides an input (sid', v) to the simulated $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ functionality of the internal instance of $\tilde{\mathcal{Sim}}_{\pi_j}$ associated with $ssid_j$, and relays responses back to $\bar{\pi}_i$ and $\mathcal{A}$. This means in particular that an input $(sid_0, ssid_j, \texttt{"hash"}, m)$ will result in the same value h being responded as $\tilde{\mathcal{Sim}}_{\pi_j}$ uses internally for a query $(sid', \texttt{"hash"}, m)$. Moreover, because $\mathcal{Sim}_i$ imposed the same public key pk onto the internal simulation of $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ by $\tilde{\mathcal{Sim}}_{\pi_j}$, a BLS signature on m will also be the same in the simulation by $\mathcal{Sim}_i$ and the internal simulation by $\tilde{\mathcal{Sim}}_{\pi_j}$, namely h^{sk}.

- It provides a simulated protocol view to the adversary $\mathcal{A}$ where the first sub-sessions $j = 1, \ldots, i$ are simulated by their respective imposed-key simulator $\tilde{\mathcal{Sim}}_{\pi_j}$ interacting with the internal instance of $\mathcal{F}_{\pi_j}$ within $\bar{\pi}_i$, while the other sub-sessions $j = i+1, \ldots, n$ are linked directly to the internal instances of π_j within $\bar{\pi}_i$.

Game $\mathcal{G}_0$ is clearly equivalent to the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid world on the left of Figure 20, where the simulator $\mathcal{Sim}_0$ includes the honest execution of $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ and all sub-sessions are simulated by real protocol instances within $\bar{\pi}_0 = \bar{\pi}_\Pi$. Game $\mathcal{G}_n$ fits the right-hand side of Figure 20 because $\bar{\pi}_n$ exclusively runs internal instances of $\mathcal{F}_{\pi_j}$, just like $\hat{\mathcal{F}}_\Pi$, and using $\mathcal{Sim} = \mathcal{Sim}_n$ as a simulator. Note that $\mathcal{Sim}$ no longer needs to run an internal instance of $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$, but simply generates a key pair that it imposes on all internal instances of $\tilde{\mathcal{Sim}}_{\pi_j}$.

We can therefore prove that $\bar{\pi}_\Pi$ securely realizes $\hat{\mathcal{F}}_\Pi$ in the $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model by showing that two subsequent games $\mathcal{G}_{i-1}$ and $\mathcal{G}_i$ are indistinguishable. We do so by contradiction: given an efficient environment $\mathcal{E}$ and adversary $\mathcal{A}$ that can distinguish $\mathcal{G}_{i-1}$ from $\mathcal{G}_i$, we construct an environment $\tilde{\mathcal{E}}$ and adversary $\tilde{\mathcal{A}}$ that can distinguish an execution of $\tilde{\pi}_i$ in the $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid world from an ideal execution of $\tilde{\mathcal{F}}_{\pi_i}$, contradicting the fact that π_i imposed-key realizes $\mathcal{F}_{\pi_i}$.

Namely, consider the environment $\tilde{\mathcal{E}}$ that first inputs $(\texttt{"extended-pk"})$ to obtain output $(\texttt{"extended-pk"}, pk_1, pk_2)$. It then runs $\mathcal{E}$ and $\mathcal{A}$ in an experiment similar to game $\mathcal{G}_i$, but where

$\tilde{\mathcal{E}}$ relays all of $\mathcal{E}$'s inputs and outputs involving $ssid_i$ to its own experiment, adding and removing sub-session identifiers as needed. Also, it simulates $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ in a different way than $\mathcal{S}im_i$, namely:

- for "`hash`" queries involving $ssid_j$, $j \in 1, \ldots, i-1$, it re-routes the responses provided by the internal instance of $\tilde{\mathcal{S}im}_j$, as done by $\mathcal{S}im_i$;
- for "`hash`" queries involving $ssid_i$, $\tilde{\mathcal{E}}$ re-routes responses obtained by $\tilde{\mathcal{A}}$ in its own experiment;
- and for "`hash`" and "`encsign`" queries involving $ssid_j$, $j \in i+1, \ldots, n$, $\tilde{\mathcal{E}}$ simulates "`hash`" responses so that it knows their discrete logarithm, i.e., by choosing $r \leftarrow_{\$} \mathbb{Z}_q$, recording $H[(ssid_j, m)] \leftarrow (r, h = g_1^r)$, and returning $(\texttt{"hash"}, ssid_j, m, h)$, so that it can simulate "`encsign`" responses by returning $\sigma = pk_1^r$;

So essentially, the only differences between $\mathcal{G}_i$ and the view produced by $\tilde{\mathcal{E}}$ is that all inputs and outputs related to $ssid_i$ are relayed to and from $\tilde{\mathcal{E}}$'s experiment, and that "`encsign`" queries for $ssid_{i+1}, \ldots, ssid_n$ are simulated as pk_1^r instead of h^{sk}, but these are of course the exact same value $g_1^{r \cdot sk}$.

One can therefore see that if $\tilde{\mathcal{E}}$ and $\tilde{\mathcal{A}}$ are running in the $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$-hybrid world with $\tilde{\pi}_i$, then the view produced by $\tilde{\mathcal{E}}$ to $\mathcal{E}$ and $\mathcal{A}$ is exactly that of $\mathcal{G}_{i-1}$ where $ssid_i$ is run with the real protocol π_i, while if they are running in the ideal world with $\tilde{\mathcal{F}}_{\pi_i}$ and $\tilde{\mathcal{S}im}_{\pi_i}$, then that view is exactly that of $\mathcal{G}_i$ where $ssid_i$ is run by $\mathcal{F}_{\pi_i}$ and $\tilde{\mathcal{S}im}_{\pi_i}$. Therefore, by repeating $\mathcal{E}$'s output, $\tilde{\mathcal{E}}$ obtains the same probability in winning its own game as $\mathcal{E}$ has in distinguishing $\mathcal{G}_{i-1}$ from $\mathcal{G}_i$. □

To show that arbitrarily many instances of our vetIBE, vetSIG, vetPRF, and vetVRF protocols can run in parallel while sharing a common vetBLS instance, we have left to show that they all satisfy the precondition of Theorem 11, namely that they all imposed-key realize their respective functionalities in the $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$-hybrid model.

Theorem 12. *For prim* $\in \{\text{vetibe}, \text{vetsig}, \text{vetprf}, \text{vetvrf}\}$, π_{prim} *imposed-key realizes* $\mathcal{F}_{prim}$ *for* $\mathcal{L} \in \{\mathsf{cEG}_1, \mathsf{cEG}_2\}$.

Corollary 1. *The single-key composition* $\bar{\pi}_\Pi$ *securely realizes the multi-session functionality* $\hat{\mathcal{F}}_\Pi$ *in the* $\bar{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$*-hybrid model for* $\Pi = \{\pi_{\text{vetibe}}, \pi_{\text{vetsig}}, \pi_{\text{vetprf}}, \pi_{\text{vetvrf}}\}$ *and* $\mathcal{L} \in \{\mathsf{cEG}_1, \mathsf{cEG}_2\}$.

Proof (Theorem 12). We sketch the proof for *prim* = vetibe here, the proofs for the other primitives are analogous.

We have to show that there exists a simulator $\tilde{\mathcal{S}im}$ such that no environment $\tilde{\mathcal{E}}$ can tell whether it's interacting with $\tilde{\pi}_{\text{vetibe}}$ and $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ in the real world or with $\tilde{\mathcal{S}im}$ and $\tilde{\mathcal{F}}_{\text{vetibe}}$ in the ideal world, as depicted in Figure 20.

The only way that $\tilde{\pi}_{\text{vetibe}}$, $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$, and $\tilde{\mathcal{F}}_{\text{vetibe}}$ differ from π_{vetibe}, $\mathcal{F}^{\mathcal{L}}_{\text{vetbls}}$, and $\mathcal{F}_{\text{vetibe}}$ is by the presence of the "`extended-pk`" interface that, in the real world, outputs (g_1^{sk}, g_2^{sk}) for the sk used by $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$, and in the ideal world, outputs (g_1^x, g_2^x) for a random $x \leftarrow_{\$} \mathbb{Z}_q$. Meaning, the simulator $\tilde{\mathcal{S}im}$, on input $(mpk_1, mpk_2) = (g_1^x, g_2^x)$, must be able to provide a simulation of $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ and $\tilde{\pi}_{\text{vetibe}}$.

Consider the simulator $\mathcal{S}im$ from the proof of Theorem 7. The simulator $\tilde{\mathcal{S}im}$ behaves exactly like $\mathcal{S}im$, but runs a simulated instance "$\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$" that uses $mpk = mpk_2$ as its master public key, and that responds to calls to its "`hash`" interface so that it knows the discrete logarithm of the returned hash values, i.e., by choosing $r[m] \leftarrow_{\$} \mathbb{Z}_q$ and setting $H_1[m] \leftarrow g_1^{r[m]}$. Whenever the real $\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$ would use $H_1[m]^{sk}$, the simulated "$\tilde{\mathcal{F}}^{\mathcal{L}}_{\text{vetbls}}$" uses mpk_1^r instead. Also, whenever $\mathcal{S}im$ uses $\mathsf{H}_1(id)^{sk}$ as part of the "`decrypt`" interface, $\tilde{\mathcal{S}im}$ uses $mpk_1^{r[id]}$ instead.

A similar approach works for the vetsig, vetprf, and vetvrf cases. Crucially, for all of them, the simulator gets to decide the (master) public key that is used by the protocol, enabling $\tilde{\mathcal{S}im}$ to substitute mpk_2 for it.

Earlier in this section, we mentioned a functionality $\mathcal{F}$ that generates a common random group element as a counterexample that is not single-key composable. Indeed, such a functionality would internally generate the common group element, preventing the simulator from setting it to the imposed master public key, so the same proof technique would not go through. $\square$

9 Integration and Evaluation

DFINITY is currently integrating vetKeys as a system service into the Internet Computer (IC) [DFI22], the blockchain created by the DFINITY Foundation.

The IC is unique in that it can serve web content straight from the blockchain, so that users can interact with it using an ordinary web browser. Because all web assets are certified by the blockchain through a threshold signature created by the nodes, the IC realizes an interesting security model where any scripting code that is downloaded and executed in the browser (e.g., Javascript or WASM code) is threshold-signed by the IC.

In a traditional client-server setting, cryptographic client-side scripts cannot protect the user from a malicious server, because a bad server can always tamper with the script code to make it leak all the user's secrets to the server. On the IC, however, that code is certified by the entire blockchain, rather than by a single node.[2] The code can therefore be trusted to be correct, and can usefully include cryptographic routines that lets the user hide secrets from the blockchain nodes, including for example the transport secret key, any derived vetKeys, and any content encrypted under these vetKeys.

Integration into the IC. The IC is subdivided into so-called *subnets*, each running its own blockchain to implement a replicated state machine for the dapps running on that subnet. Dapps on different subnets can asynchronously communicate with each other through cross-net messages that are authenticated by threshold signatures. One can therefore secret-share a vetBLS secret key over the nodes of a single subnet and allow dapps on the local subnet as well on other subnets to derive vetKeys from the secret-shared vetBLS secret key. When the composition of a subnet changes, i.e., when nodes join or leave the subnet, the vetBLS secret key is re-shared to the new members in the same way as the IC already does for the threshold signature keys [Gro21]. The vetBLS requests from different dapps are isolated from each other as described in Section 8, by pre-fixing a domain separator that includes the unique identifier of the dapp.

Dapps can trigger a vetBLS request by calling a special system interface. This call contains the derivation path (that, together with the dapp identifier, is used as a domain separator) and the message, as well as the transport public key *tpk* under which the BLS signature will be encrypted; typically, *tpk* will have been included in a user-supplied ingress message that appeared on the blockchain earlier.

The effect of the system call is that each node will create and broadcast an encrypted BLS signature share; verify incoming shares until sufficiently many encrypted shares are received; combine these shares into a full encrypted signature and include it in a block proposal as the response to the system call; and verify the validity of the encrypted signature included in a block as part of block validation.

Efficiency estimates. Figure 22 depicts a breakdown of the estimated CPU time spent by each node per vetBLS request, as well as their encrypted signature sizes. The numbers are based on benchmarks of basic cryptographic operations on the hardware of a typical Internet Computer node

[2]One remaining caveat is that the threshold signature on the downloaded web assets needs to be properly verified in the browser, with the correct public key. The code to do so is currently included in a service worker served by the TLS endpoint, a so-called *boundary node*. The boundary node thereby becomes a single point of failure in the security model; a more secure implementation would include the threshold signature verification in a browser plugin, or in a dedicated IC client application.

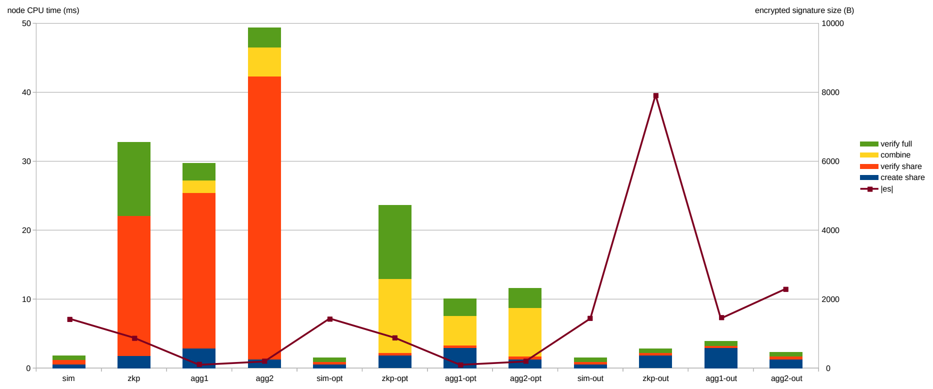


Figure 22: CPU time spent per vetBLS request by each node in a subnet of $n = 13$ nodes with threshold $t = 5$, as well as the size of an encrypted signature in bytes.

(a server machine with an AMD EPYC 7232P 3.10GHz processor); we hope to update this section soon with actual experimental results.

The four leftmost columns in Figure 22 display a breakdown of the CPU time of the basic vetBLS schemes as described in this paper. One can immediately see that for all schemes other than $\pi_{\text{vetbls-sim}}$, by far the most CPU time is spent on the verification of encrypted signature shares (depicted in orange in Figure 22).

The verification of encrypted shares is important to discard bad shares provided by corrupt nodes. Nodes in a typical blockchain network, however, are usually incentivized to behave honestly, so that one can expect that nodes will provide correct shares most of the time. One can therefore improve best-case efficiency by considering an *optimistic* variant of each scheme that

- adds to each encrypted signature share a standard (e.g., Ed25519) signature by the node,
- limits encrypted share verification to verifying this standard signature,
- after having received encrypted shares signed by t different nodes, optimistically combines the encrypted shares and verifies the resulting full encrypted signature,
- and only if that fails, resorts to fully verifying the individual encrypted shares and combining t valid ones.

The middle four columns in Figure 22 depict the best-case efficiency estimates for the optimistic variants of all four schemes; worst-case efficiency in the presence of $t-1$ corrupt nodes is of course identical to the basic schemes. One can see that encrypted share verification now takes a minimal amount of CPU time, while keeping the size of the full encrypted signature the same.

Another optimization stems from the observation that the network nodes will be handling many requests from many different dapps and users simultaneously, but each individual user will only be deriving one or a few vetKeys at the same time. It can therefore make sense to consider an *outsourced* variant of each scheme, where encrypted shares include a standard signature by the nodes as in the optimistic variant, but the full encrypted signature is simply the concatenation of $2t-1$ encrypted shares that are validly signed by different nodes. Full verification of individual shares and combination into a valid encrypted signature is then left to the user's device, that probably has more idle CPU time to spend on it. Since at most $t-1$ nodes are corrupt, at least t of the $2t-1$

		π_{vetbls}				optimistic				outsourced			
		sim	zkp	agg1	agg2	sim	zkp	agg1	agg2	sim	zkp	agg1	agg2
$n = 13$	rps	545	30	33	20	647	42	99	86	647	264	254	427
$t = 5$	$\|es\|$	1440	880	96	192	1440	880	96	192	1440	7920	1440	2304
$n = 13$	rps	370	19	23	13	414	24	74	59	414	77	208	311
$t = 9$	$\|es\|$	2720	1584	96	192	2720	1584	96	192	2720	26928	2720	4352
$n = 40$	rps	225	10	12	7	285	16	63	46	285	31	170	232
$t = 14$	$\|es\|$	4320	2464	96	192	4320	2464	96	192	4320	66528	4320	6912
$n = 100$	rps	97	4	5	2	127	6	34	22	127	5	97	115
$t = 34$	$\|es\|$	10720	5984	96	192	10720	5984	96	192	10720	400928	10720	17152

Table 1: Estimated performance overview in different settings of the vetBLS schemes in this paper. The columns contain estimates for the basic, optimistic, and outsourced variants of the $\pi_{\text{vetbls-sim}}$, $\pi_{\text{vetbls-zkp}}$, $\pi_{\text{vetbls-agg1}}$, and $\pi_{\text{vetbls-agg2}}$ schemes. Each row contains the number of vetBLS requests per second (rps) that a network of n single-core servers can theoretically handle with a threshold t, as well as the size of an encrypted signature es in bytes.

encrypted shares are bound to be valid shares, so that the user is guaranteed to obtain the requested vetKey.

The outsourced variant of the four vetBLS schemes is depicted in the rightmost four columns of Figure 22. One can see that CPU time spent per request by each node is now drastically reduced, at the cost of increased encrypted signature sizes. Because the full encrypted signature appears in the blockchain, it very much depends on the concrete parameters (number of nodes, threshold, and price of bandwidth on the blockchain) whether this optimization is worth it.

Table 1 contains estimates for the throughput of the network, measured in vetBLS requests per second on single-core node machines, as well as the corresponding encrypted signature sizes. Note that the cryptographic operations to handle simultaneous vetKey requests are highly parallelizable, so that on multi-core nodes one would expect to achieve a multiple of these throughput numbers.

On the 13-node application subnets and 40-node system subnets that are currently deployed on the Internet Computer, one would therefore expect a vetKey throughput of hundreds of requests per second. Of course, these estimates neglect the cost of running the dapps and the blockchain itself; we hope to update the paper with actual experimental results soon.

References

[ABN10] Michel Abdalla, Mihir Bellare, and Gregory Neven. Robust encryption. In Daniele Micciancio, editor, *TCC 2010: 7th Theory of Cryptography Conference*, volume 5978 of *Lecture Notes in Computer Science*, pages 480–497, Zurich, Switzerland, February 9–11, 2010. Springer, Heidelberg, Germany.

[ACG$^+$18] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. Helix: A scalable and fair consensus algorithm resistant to ordering manipulation. Cryptology ePrint Archive, Report 2018/863, 2018. `https://eprint.iacr.org/2018/863`.

[ADN06] Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold RSA with adaptive and proactive security. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 593–611, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

[AF04] Masayuki Abe and Serge Fehr. Adaptively secure feldman VSS and applications to universally-composable threshold cryptography. In Matthew Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 317–334, Santa Barbara, CA, USA, August 15–19, 2004. Springer, Heidelberg, Germany.

[ASW00] N. Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. *IEEE J. Sel. Areas Commun.*, 18(4):593–610, 2000.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 757–788, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matthew Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 41–55, Santa Barbara, CA, USA, August 15–19, 2004. Springer, Heidelberg, Germany.

[BC04] Ian F. Blake and Aldar C-F. Chan. Scalable, server-passive, user-anonymous timed release public key encryption from bilinear pairing. Cryptology ePrint Archive, Report 2004/211, 2004. `https://eprint.iacr.org/2004/211`.

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[BGdMM05] Lucas Ballard, Matthew Green, Breno de Medeiros, and Fabian Monrose. Correlation-resistant storage via keyword-searchable encryption. Cryptology ePrint Archive, Report 2005/417, 2005. `https://eprint.iacr.org/2005/417`.

[BGG$^+$20] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 260–290, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany.

[BGI$^+$01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[BGLS03] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 416–432, Warsaw, Poland, May 4–8, 2003. Springer, Heidelberg, Germany.

[BJKS21] Robert Buhren, Hans Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. One glitch to rule them all: Fault injection attacks against AMD's secure encrypted virtualization. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2875–2889, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of

Lecture Notes in Computer Science, pages 514–532, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany.

[BLS04] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[BMSV18] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. Private data objects: an overview. *CoRR*, abs/1807.05686, 2018.

[BO22] Joseph Bebel and Dev Ojha. Ferveo: Threshold decryption for mempool privacy in bft networks. Cryptology ePrint Archive, Paper 2022/898, 2022. `https://eprint.iacr.org/2022/898`.

[Bol03] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Yvo Desmedt, editor, *PKC 2003: 6th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 31–46, Miami, FL, USA, January 6–8, 2003. Springer, Heidelberg, Germany.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[BS23] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography, Version 0.6.* 2023. `https://cryptobook.us`.

[BZ03] Joonsang Baek and Yuliang Zheng. Simple and efficient threshold cryptosystem from the gap diffie-hellman group. In *GLOBECOM '03. IEEE Global Telecommunications Conference (IEEE Cat. No.03CH37489)*, volume 3, pages 1491–1495 vol.3, 2003.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[Can04] Ran Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, page 219. IEEE Computer Society, 2004.

[CDH$^+$22] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet computer consensus. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 81–91. ACM, 2022.

[CDN20] Dan Cline, Tadge Dryja, and Neha Narula. Clockwork: An exchange protocol for proofs of non front-running. https://dci.mit.edu/clockwork, 2020.

[CGJ$^+$17] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 719–728, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[CKLS02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In Vijayalakshmi Atluri, editor, *ACM CCS 2002: 9th Conference on Computer and Communications Security*, pages 88–97, Washington, DC, USA, November 18–22, 2002. ACM Press.

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[CKY09] Jan Camenisch, Aggelos Kiayias, and Moti Yung. On the portability of generalized Schnorr proofs. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 425–442, Cologne, Germany, April 26–30, 2009. Springer, Heidelberg, Germany.

[CLNS17] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. UC-secure non-interactive public-key encryption. In Boris Köpf and Steve Chong, editors, *CSF 2017: IEEE 30th Computer Security Foundations Symposium*, pages 217–233, Santa Barbara, CA, USA, August 21–25, 2017. IEEE Computer Society Press.

[CMSW14] Theresa Calderon, Sarah Meiklejohn, Hovav Shacham, and Brent Waters. Rethinking verifiably encrypted signatures: A gap in functionality and potential solutions. In Josh Benaloh, editor, *Topics in Cryptology – CT-RSA 2014*, volume 8366 of *Lecture Notes in Computer Science*, pages 349–366, San Francisco, CA, USA, February 25–28, 2014. Springer, Heidelberg, Germany.

[CR03] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[CZK$^+$19] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *IEEE European Symposium on Security and Privacy, EuroS&P 2019*, pages 185–200. IEEE, 2019.

[DFI22] The DFINITY Team. The internet computer for geeks. Cryptology ePrint Archive, Paper 2022/087, 2022. `https://eprint.iacr.org/2022/087`.

[Dod03] Yevgeniy Dodis. Efficient construction of (distributed) verifiable random functions. In Yvo Desmedt, editor, *PKC 2003: 6th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 1–17, Miami, FL, USA, January 6–8, 2003. Springer, Heidelberg, Germany.

[dra] drand. Distributed randomness beacon. `https://drand.love`.

[DRZ17] Sisi Duan, Michael K. Reiter, and Haibin Zhang. Secure causal atomic broadcast, revisited. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017*, pages 61–72. IEEE Computer Society, 2017.

[DYX$^+$22] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew K. Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2518–2534. IEEE, 2022.

[ElG84] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18, Santa Barbara, CA, USA, August 19–23, 1984. Springer, Heidelberg, Germany.

[Fel87] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science*, pages 427–437, Los Angeles, CA, USA, October 12–14, 1987. IEEE Computer Society Press.

[FH18] Paul Fletcher-Hill. Kimono — trustless secret sharing using time-locks on ethereum. `https://medium.com/@pfh/kimono-trustless-secret-sharing-using-time-locks-on-ethereum-8e7e696494d`, 2018. Accessed 2022-08-12.

[Fis06] Marc Fischlin. Round-optimal composable blind signatures in the common reference string model. In Cynthia Dwork, editor, *Advances in Cryptology – CRYPTO 2006*, volume 4117 of *Lecture Notes in Computer Science*, pages 60–77, Santa Barbara, CA, USA, August 20–24, 2006. Springer, Heidelberg, Germany.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany.

[Gen22] Ekin Genç. What is MEV, aka maximal extractable value? `https://www.coindesk.com/learn/what-is-mev-aka-maximal-extractable-value/`, 2022.

[GG17] Rishab Goyal and Vipul Goyal. Overcoming cryptographic impossibility results using blockchains. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part I*, volume 10677 of *Lecture Notes in Computer Science*, pages 529–561, Baltimore, MD, USA, November 12–15, 2017. Springer, Heidelberg, Germany.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 467–476, Palo Alto, CA, USA, June 1–4, 2013. ACM Press.

[GKM$^+$20] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. Cryptology ePrint Archive, Report 2020/504, 2020. `https://eprint.iacr.org/2020/504`.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 39–56, Santa Barbara, CA, USA, August 17–21, 2008. Springer, Heidelberg, Germany.

[GLOW21] David Galindo, Jia Liu, Mihai Ordean, and Jin-Mann Wong. Fully distributed verifiable random functions and their application to decentralised random beacons. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 88–102. IEEE, 2021.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[GMR23] Nicolas Gailly, Kelsey Melissaris, and Yolan Romailler. tlock: Practical timelock encryption from threshold bls. Cryptology ePrint Archive, Paper 2023/189, 2023. `https://eprint.iacr.org/2023/189`.

[GPS06] S.D. Galbraith, K.G. Paterson, and N.P. Smart. Pairings for cryptographers. Cryptology ePrint Archive, Report 2006/165, 2006. `https://eprint.iacr.org/2006/165`.

[Gro21] Jens Groth. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339, 2021. `https://eprint.iacr.org/2021/339`.

[GS22] Jens Groth and Victor Shoup. On the security of ECDSA with additive key derivation and presignatures. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 365–396, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

[HMM15] Dennis Hofheinz, Christian Matt, and Ueli Maurer. Idealizing identity-based encryption. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015, Part I*, volume 9452 of *Lecture Notes in Computer Science*, pages 495–520, Auckland, New Zealand, November 30 – December 3, 2015. Springer, Heidelberg, Germany.

[HMW18] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[JLS21] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 60–73. ACM, 2021.

[KAG$^+$20] Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. CALYPSO: private data management for decentralized ledgers. *Proc. VLDB Endow.*, 14(4):586–599, 2020.

[KGF19] Rami Khalil, Arthur Gervais, and Guillaume Felley. TEX - A securely scalable trustless exchange. Cryptology ePrint Archive, Report 2019/265, 2019. `https://eprint.iacr.org/2019/265`.

[KGM19] Gabriel Kaptchuk, Matthew Green, and Ian Miers. Giving state to the stateless: Augmenting trustworthy computation with ledgers. In *ISOC Network and Distributed System Security Symposium – NDSS 2019*, San Diego, CA, USA, February 24–27, 2019. The Internet Society.

[LJKW18] Jia Liu, Tibor Jager, Saqib A. Kakvi, and Bogdan Warinschi. How to build time-lock encryption. *Des. Codes Cryptogr.*, 86(11):2549–2586, 2018.

[Lys02] Anna Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 597–612, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[Mau11] Ueli Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011.

[MHS02] Marco Casassa Mont, Keith Harrison, and Martin Sadler. The HP time vault service: Innovating the way confidential information is disclosed, at the right time. Technical Report HPL-2002-243, 2002.

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science*, pages 120–130, New York, NY, USA, October 17–19, 1999. IEEE Computer Society Press.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[Nie02a] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 111–126, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[Nie02b] Jesper Buus Nielsen. A threshold pseudorandom function construction and its applications. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 401–416, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[NMO06] Ryo Nishimaki, Yoshifumi Manabe, and Tatsuaki Okamoto. Universally composable identity-based encryption. In Phong Q. Nguyen, editor, *Progress in Cryptology - VIETCRYPT 06: 1st International Conference on Cryptology in Vietnam*, volume 4341 of *Lecture Notes in Computer Science*, pages 337–353, Hanoi, Vietnam, September 25–28, 2006. Springer, Heidelberg, Germany.

[NPR99] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 327–346, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany.

[Ped91] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract) (rump session). In Donald W. Davies, editor, *Advances in Cryptology – EUROCRYPT'91*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526, Brighton, UK, April 8–11, 1991. Springer, Heidelberg, Germany.

[PWH$^+$17] Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Včelák, Leonid Reyzin, and Sharon Goldberg. Making NSEC5 practical for DNSSEC. Cryptology ePrint Archive, Report 2017/099, 2017. `https://eprint.iacr.org/2017/099`.

[RAA$^+$19] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, Julien Maffre, Thomas Moscibroda, Kartik Nayak, Olya Ohrimenko, Felix Schuster, Roy Schwartz, Alex Shamis, Olga Vrousgou, and Christoph M. Wintersteiger. Ccf: A framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft, 2019.

[RB94] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, 1994.

[RSW96] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, 1996.

[RSW00] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, 2000.

[Sch22] John Schmidt. Why does bitcoin use so much energy? `https://www.forbes.com/advisor/investing/cryptocurrency/bitcoins-energy-usage-explained/`, 2022.

[Sco02] Mike Scott. Authenticated ID-based key exchange and remote log-in with simple token and PIN number. Cryptology ePrint Archive, Report 2002/164, 2002. `https://eprint.iacr.org/2002/164`.

[Sek22] Venkkatesh Sekar. Preventing front-running attacks using timelock encryption. Master's thesis, University College London, 2022.

[SG98] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 1–16, Espoo, Finland, May 31 – June 4, 1998. Springer, Heidelberg, Germany.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[Shu21] Shutter Network. Introducing Shutter network – combating front running and malicious MEV using threshold cryptography. `https://blog.shutter.network/introducing-shutter-network-combating-frontrunning-and-malicious-mev-using-threshold-cry` 2021. Accessed 2022-08-12.

[SRMH21] Oliver Stengele, Markus Raiber, Jörn Müller-Quade, and Hannes Hartenstein. ETHTID: deployable threshold information disclosure on ethereum. *CoRR*, abs/2107.01600, 2021.

[Tou18] Nathan Toups. killcord. `https://github.com/nomasters/killcord`, 2018. Accessed 2022-08-12.

[vSSY$^+$22] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. 2022.