Flare Confidential Compute: Powering Interoperability for Flare through TEEs

Flare Network

Abstract

Trusted Execution Environments (TEEs) open up a new avenue in blockchain development: outsourcing computation to TEEs improves scalability without the traditional compromise of decentralisation and security. This paper introduces Flare Confidential Compute, smart contract infrastructure on Flare allowing users to access TEEs by issuing on-chain instructions. Several applications of this infrastructure are exhibited: firstly, the TEE Protocol Managed Wallet. These wallets allow Flare's users to issue transactions and manage accounts on external blockchains through Flare, all secured by a combination of the consensus of Flare's data providers and the security of TEEs. The design of the Protocol Managed Wallet is discussed in detail, including the methods by which users control the wallet. Secondly, Flare Confidential Compute's wider infrastructure allowing users to build custom TEE based applications, known as Flare Compute Extensions, on Flare. Finally, an upgrade to the Flare Data Connector, improving on various aspects of the FDC design by leveraging TEEs. These features work together to dramatically improve the interoperability between the Flare network and a wide range of sources, both Web 2 and Web 3.

1 Introduction

The Blockchain Trilemma stands as a folklore challenge within the blockchain community: how does one appropriately manage the necessary trade-off between scalability, decentralization, and security? Outsourcing computations is great for scalability, but undesirable for decentralization and can undermine security by opening up specific points of failure. Recent exploration [1] of the role of Trusted Execution Environments (TEEs) in blockchain systems represents a major new frontier in outsourcing, allowing for scalability and interoperability improvements without compromising the security properties of the network.

1.1 TEEs and Blockchains

To best understand the relationship between TEEs and blockchain, it is first important to understand their design. Put simply, a TEE is a hardware secure computing environment that is isolated from its parent operating system to prevent tampering or unauthorized access to its memory. More complex definitions of TEEs can be found in a variety of sources, for example [2]. In blockchain terminology, they are computing environments that support both confidentiality and cryptographically verifiable attestability. Confidentiality in the sense that memory and code running in the TEE cannot be accessed or tampered with, and attestability meaning that the TEE can prove that it is executing the correct code. These two properties enable the scalability offered to blockchains: computation can be outsourced to cloud-based TEEs [3], who can be relied upon to execute their role honestly due to attestability. Furthermore, the confidentiality allows for these computations to include security- or privacy-sensitive operations that are typically hard to move off-chain without introducing uncomfortable trust assumptions, and thus avoiding the usual pitfalls of the trilemma.

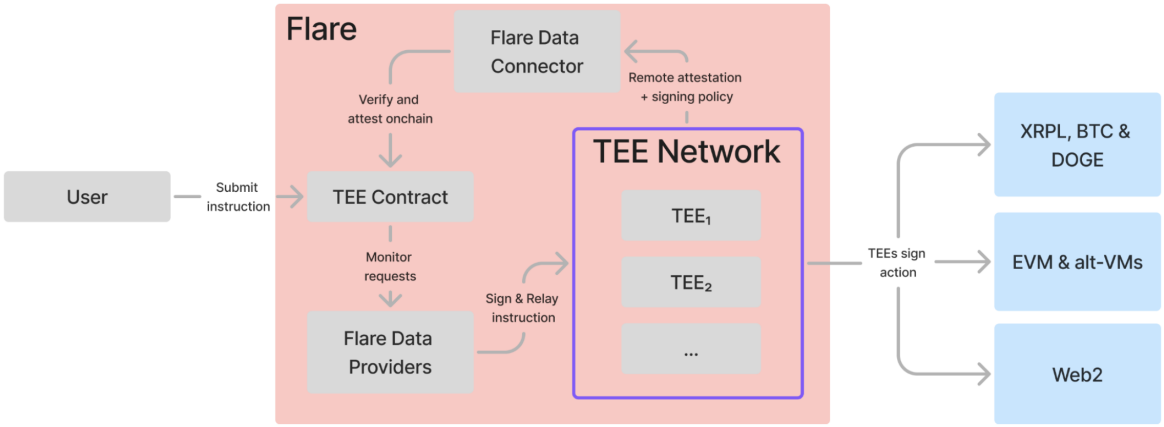

Figure 1: Extending Flare operations to other chains using TEEs. User requests are relayed to the TEE Network by Flare's data providers. The TEEs then sign the requested action, which can now be exported from Flare.

1.2 TEEs and Flare

Flare's vision for TEE deployment complements the network's position as an interoperability hub: leveraging TEEs to connect Flare to other blockchains, Web 2 sources, and the wider Decentralized Finance (DeFi) ecosystem. Protocols such as the Flare Data Connector [4] already connect Flare to other blockchains, bringing information on to Flare that is secured by Flare's data providers. TEEs allow Flare to dramatically widen the scope of its connectivity, while maintaining the fact that consensus among Flare's providers secures the entire system.

To this end, TEE based protocols on Flare will operate through its data providers. TEEs can be registered on a dedicated contract on Flare, with users able to command them through on-chain infrastructure. Users submit requests and instructions to the TEEs through smart contracts, which are picked up by Flare's data providers. The providers then package and relay the instructions to the appropriate TEEs. Once the majority of data providers have signed-off on an instruction, the TEE will execute the command and, due to their security properties, can be trusted to do so correctly.

Thus the combination of Flare's data providers and the TEE secures the instruction; the providers are responsible for consensus and relaying, and the TEE for execution. In this way, Flare's consensus is extended to other networks, and Flare's protocols are empowered to execute on other chains. These two features connect Flare's users to other blockchains, and even other data sources such as Web 2, all with Flare as a central hub for securing and managing their assets. This design is depicted in Figure 1.

1.3 Flare Deployment

This whitepaper describes the first phases of the deployment of Flare's TEE based architecture, known as Flare Confidential Compute (FCC). Two TEE based system protocols are defined: a TEE based protocol managed wallet (PMW) is introduced, a wallet that is simultaneously capable of executing transactions on other chains yet hosted and managed on Flare. Additionally, a TEE based variant of the FDC, known as the Flare Data Connector v2 (FDC2), is discussed. The system protocols are built as examples of Flare Compute Extensions (FCE), a structure for designing new user-defined TEE based protocols on Flare. The ability for users to design novel extensions on FCC is a primary feature of Flare's TEE deployment, and is explained in detail in this paper.

The novel protocols both function using the overarching TEE architecture highlighted

above: for the PMW, a TEE instructed on Flare holds the key for an account on another blockchain, and authorized users can instruct the TEE to execute transactions from this address via smart contracts on Flare. Flare's data providers enable this by picking up the user instructions from these contracts and relaying them to the TEEs. Similarly, for the FDC2 users submit attestation requests on-chain, which are collected and confirmed by data providers then relayed with confirmations to the TEE. In either case, once a TEE has received an instruction supported by the majority of Flare's data providers, it faithfully executes the next stage of the protocol: signing a transaction for the source chain for a PMW or preparing an attestation for the FDC2. Finally, data providers or other users relay the results to the appropriate destination.

Since TEEs can attest to their state, they can be trusted to manage their duties honestly, and since their memory is confidential, they can be trusted with private keys. Furthermore, the designs do not require TEEs to perform expensive monitoring of Flare or other networks themselves, or indeed any expensive computation. Instead, they act only in response to instructions from Flare's data providers, who are responsible for the heavy lifting; the TEEs are leveraged to provide an additional layer of security, signing off on the consensus of the data providers, but not overloaded with computations.

These simple yet flexible protocols connect Flare to other blockchains, opening up a variety of DeFi opportunities such as bridging and restaking. Of particular interest is the ability to allow movement of assets from chains that do not support smart contracts, such as XRP on the XRP Ledger, to Flare's smart contract ecosystem.

Roadmap. Before describing the novel system protocols, the paper begins with Section 2 which explains how TEEs are deployed and managed on Flare, and Section 3, which explains how FCEs on FCC are defined, built, and managed, opening up both the other designs in this paper and future use cases. Following this, the TEE PMW is described in detail in Section 4, exhibiting the design that enables the wallet to be secured by both Flare's data providers and the TEE infrastructure. Finally, Section 5 demonstrates how the FCC extension architecture is used to construct the TEE based upgrade of the FDC, the FDC2.

2 TEE Management On Flare

In the context of the Flare network, the purposes of TEEs are threefold. Firstly, their ability to attest to deployed code means that TEEs can be trusted to execute tasks honestly without requiring rationality assumptions or incentive structures typical to blockchains. Secondly, TEEs allow for the outsourcing of computations, such as moving gas intensive operations off-chain. Finally, TEEs enable connectivity between Flare and other networks, allowing actions on external chains in response to instructions on Flare. Combined, these properties support facilitating trustless off-chain outcomes from on-chain events, such as bridging operations, where the result of an on-chain transaction on Flare instructs the TEE network to release locked funds on an external chain.

The ability of TEEs to take actions based on instructions on Flare will be used in this paper to describe FCC operations such as the TEE PMW deployed. Before describing their implementation, the necessary infrastructure for deploying and operating TEEs as part of FCC is introduced here.

2.1 TEEs and the FSP

The Flare System Protocol (FSP) [5] is Flare's key underlying infrastructure, handling information necessary to support the network's core protocols. Relevant to FCC, its main

responsibilities revolve around managing signing on Flare and keeping track of the evolving weights of each of Flare's data providers over time. The features of the FSP used to support the TEE infrastructure on Flare are highlighted below.

Signing Policy and Weight. Before fulfilling an FCC instruction, TEEs will check that a sufficient proportion of Flare's data providers have signed off on it. Since Flare uses a stake based security model, this proportion is weighted; namely, an instruction is approved if the TEE sees signatures from more than a threshold value of accumulated weight of Flare's data providers. Usually, the required weight will be 50% or more, but certain instructions may set different thresholds. The weight used is known as the signing weight on Flare, and it is the same as is used in the FTSO, FDC, and FSP. The current set of data providers, signing weights, and their addresses is referred to as the signing policy on Flare.

To calculate the weight W_j of the jth data provider, first compute the proportion W_{j^*} of staked FLR tokens and (capped) delegated WFLR tokens assigned to each data provider address. Then, diversity weighting is applied across the weights of all data providers, so that the signing weight of the jth data provider is computed as

$$W_j = \frac{W_{j^*}^{3/4}}{\sum_i W_{i^*}^{3/4}}.$$

More details on the computation of signing weights can be found in Flare's other white papers such as that of the FTSO [6]. Each data provider has a public address on Flare corresponding to a public/private key pair, and will use this address to sign transactions submitted to the TEE contracts.

Reward Epochs. Flare's signing policy is updated at the end of each reward epoch, which have a duration of 3.5 days, or around 3360 blocks. At the end of each reward epoch, Flare's data providers sign the new signing policy for the next reward epoch, which tracks the signing weight of the data providers in the next epoch. Changes in signing weight are a result of changes of stake of Flare's data providers and changing delegations of Flare's users, in accordance with a delegated proof-of-stake model. Note that since the data providers in one epoch sign the policy for the next epoch, the first signing policy seen by a new entity such as a TEE must be trusted, after which all proceeding policies can be verified sequentially.

2.2 TEE Ownership on Flare

The ownership of TEEs operating on Flare is managed by permissioned entities. Since TEEs can attest to their state, there is only minimal trust required in the owner, who must be trusted not to power-off the machine or otherwise deny access to it. An initial set of TEEs will be managed by the Flare Foundation. Since the operation of TEEs will be rewarded on-chain, and different FCEs will have different requirements for their TEEs, operation is expected to be picked up by other providers in future. This section discusses the details of TEE management on Flare.

TEE Security. Ultimately, a TEE is a hardware-based security solution, separating a secure, tamper-proof segment of processing power away from the rest of a device's CPU. This enables the execution of secure code and storage of secure memory such as cryptographic keys. However, TEEs are an evolving technology, and the attack surface is still being explored [7]. Physical security is of particular importance, as physical access is a common attack vector for TEEs. On a cryptographic level, Flare's TEE based protocols will encourage redundancy

as a means of defense, requiring both data providers and several TEEs to sign off on a given instruction. For example, multi-sigs can be used to distribute trust across several TEEs that can have separate management and physical locations.

Nonetheless, for the purposes of this paper it is assumed that the underlying TEEs are sufficiently well-secured, capable of storing keys and executing code without attack. To this end, trusted providers running the TEE machines will be screened to ensure that they carefully control physical access to the devices. Furthermore, the make and models of TEEs used on Flare will be monitored, as will the state-of-the-art attacks on TEEs.

TEE Attestations. TEE machines can produce secure attestations to their state, including the code they are running and the integrity of their memory. Verifying these cryptographic proofs intermittently is an integral part of trusting the TEE environment. Data, in the form of public keys, required to confirm these attestations is stored on a TEE smart contract on Flare. Requests for attestations from specific TEEs are accompanied by fresh random nonces to ensure that the TEE is validly performing a live attestation rather than replaying an old one. These requests are handled by the FDC2, which publishes a verification of the TEE attestation together with an appropriate timestamp on-chain. Crucially, this means that confirming attestations is a responsibility of Flare's data providers, with the verification done according to consensus among a weighted majority of them. Similarly, a TEE registered on Flare can have its participation be paused in response to an FDC2 attestation checked by data providers stating that it is not currently available at the right URL and timestamp.

FCC Protocol Security. Regardless of the owner of the TEE, TEE operations on Flare are instructed according to a (weighted) threshold of signatures from Flare's data providers, in accordance with the signing policy of the reward epoch. Thus, the correctness of the execution of operations issued to the TEEs as part of FCC relies on the intersection of two sources:

- The security of the underlying TEE instance.
- The honest and correct behavior of Flare's data providers.

The security of TEE instances was already discussed, and comes down to trust in the underlying hardware and owner. The honesty of a majority of Flare's data providers is the core assumption of the network's security, and thus is a suitable building block for a TEE deployment managed on Flare. In the usual case where the threshold value is 50%, a weighted majority is required for both safety and liveness: without a weighted majority of signatures, neither a malicious instruction nor an honest instruction will be executed by a TEE.

Proxy Servers. The owner of each TEE controls access to the machine through a proxy, a dedicated server which relays information to and from the TEE. The TEE proxy corresponding to a TEE machine must also be registered on Flare with an appropriate public identity, for which it owns the corresponding private key. Essentially, these servers are what manage the flow of information in and out of the TEEs: they store and update signing policies, data providers communicate with them when instructing the TEE machines, and they store and distribute results of the machine's computations. Effectively, the proxy servers add a layer of security to the TEE by limiting direct access to the machine to the owner. The TEE owner is responsible for ensuring that information is relayed between the TEE and corresponding proxy server. TEEs will only execute instructions relayed to them if they are appropriately signed by Flare's data provider, and sign results of any FCC actions: thus, the owner is only trusted for availability of the TEE, and can neither force the TEE to take actions not permitted by FCC nor spoof a TEE operation.

In order to simplify exposition, proxy access will not be discussed in detail in this paper, and communication will sometimes be described as directly between TEEs and data providers: since these proxies do not meaningfully impact the user experience or design of the FCEs, this should not cause any misunderstandings.

3 Flare Confidential Compute

Flare Confidential Compute (FCC) is the name given to the entirety of the TEE architecture on Flare. It is designed to allow for a variety of applications to be deployed within the TEE network. In this section, the central technical mechanism of FCC is laid out: the ability of the TEE infrastructure to support general, user-defined instructions. These instructions are managed through a system of extensions, where an individual application is known as a Flare Compute Extension (FCE).

Definition 3.1 *A* Flare Compute Extension *is defined by the following*

- *A set of supported code versions, each stored as (a hash of) a virtual machine implementation so that they are reproducible accross TEE machines.*
- *A set of TEEs that are registered to the extension and running the appropriate code. Each such TEE is identified by a unique public identity* TEE_{id} *that is registered to the extension.*

The specified TEEs run the supported code, which implicitly specifies a set of TEE instructions *available on the extension. An instruction is a command issued by a user on Flare that is executed by one or more TEEs. These TEEs complete the instruction in response to signed and relayed messages from Flare's data providers specifying the instruction.*

Each FCE is identified by a unique ExtensionID, and all functions implemented by FCC are built as functions on an FCE. Though minimal, these requirements are sufficient to define meaningful TEE applications: the supported code defines the functionality of the TEE within the extension, and the set of TEEs defines which TEEs are eligible to participate.

In an FCE, TEEs will respond to instructions relayed to them by Flare's data providers. Their response to these instructions is determined by their code version and their state, which means that the deployed code must be able to perform the given instruction and they must hold any require keys. Similarly, TEE smart contracts on Flare must be able to parse given commands. Beyond this requirement, the instruction can be as general as the user desires, and in particular need not even be limited to blockchain or DeFi use cases. For example, an FCE can be put in place to provide Flare's users with access to AI services, or link actions on Flare to Web 2 sources. Adding or upgrading current instructions is straightforward: available instructions can be managed on-chain, and participating TEEs can be redeployed, updated to a new version supporting the instruction without changing the underlying identity.

System Extension. As part of the deployment of the FCC infrastructure, Flare will initialize the first extension with ExtensionID 0, known as the *system extension*. The system extension hosts two applications: the Protocol Managed Wallet infrastructure described in Section 4 and the FDC2 infrastructure described in Section 5.

Initializing an Extension. To set up a new FCE, a Flare user sends a register command to a specific TEE Extension Registry smart contract. This instruction defines the Flare address of the extension owner, also referred to as the governance address, from which the

extension can be managed. This address need not match the address of the user initializing the extension. In order to prevent a single user from controlling the extension, the governance account can support certain rules for its use; for example, it may be a multi-sig account.

Extension Management. Each extension registered to FCC is managed and owned by a governance address registered to that specific extension. This account is responsible for management duties for the extension, such as adding, updating, or deprecating supported code to be run by the registered TEEs. Each of these actions requires a signed instruction from the governance account to be implemented by the TEE. Additional common management instructions include adding or removing TEE machines from the extension and changing the ownership of the extension. More complicated instructions may be extension specific: extensions are able to support specialized TEE instructions that can only be issued from the management account, rather than requiring data provider signatures.

3.1 Registering TEEs to Extensions

In order to participate in a FCE, a participating TEE must first be registered on Flare. Registration is performed on a per-extension basis, with TEE instances signing up to participate in specific extensions rather than across FCC as a whole. A list of active TEE machines and their relevant extensions is stored on the TEE Registration smart contract on the Flare blockchain, with TEEs identified by:

- TEE_{id}, a unique identifier for the TEE. This is generated on boot, matching the address of a public/private key pair generated by the TEE during initialization.
- TEE_{URL}, the URL at which the TEE (proxy) can be found.
- The ExtensionID of the extension to which the TEE is registered.

The process for registering a TEE in reward epoch t utilizes the FDC2, and proceeds as follows:

1. The TEE is initialized alongside a TEE proxy, which provides the TEE with the signing policy for reward epoch t.
2. The TEE generates a public/private key pair and corresponding public address TEE_{id} and generates a request to register to the extension identified by ExtensionID.
3. An FDC2 request (with specialized attestation type) is made, asking data providers to confirm the TEE's ID and proxy address, that it is equipped with the correct signing policy, that is is running appropriate code for the extension, and that it provided a valid attestation of this information.
4. Assuming the attestation was accurate, the request is confirmed by the FDC2 and the TEE is registered.

A registered TEE can be unregistered by the machine owner at any time. At a later time it can also be re-registered, assuming that it is still running the same instance, for which it must produce a valid attestation with the correct time stamp.

Signing Policies. It is crucial that the TEE machine is initialized with the correct signing policy for reward epoch t when registering: at the end of each reward epoch, the TEE's copy of the signing policy is be updated to that of reward epoch $t + 1$ in accordance with the FSP, with the new signing policy relayed to the TEE machine by its proxy. However, as the new signing policy is signed by Flare's data providers in accordance with the old policy, the TEE needs the policy of epoch t to update securely. If the signing policy given to a TEE is incorrect, the TEE will not be able to determine whether or not instructions have been confirmed sufficiently by Flare's providers or correctly update to a new signing policy.

To ensure TEEs are equipped with the correct signing policy, their initial attestation submitted on registration includes the current signing policy, which is checked by data providers. Additionally, they are periodically required to attest to their current view of the signing policy to ensure that they are up to date.

Signing and Identity. Most instructions, including outputs from the PMW and FDC2 protocols, will result in the TEE publishing a signed response to their proxy server as part of their task. This could be direct, in the sense that the purpose of the task is to produce the signed response, or indirect, in the sense that the signature is the method by which a TEE confirms that it has taken some action such as generating a new key. In either case, the registration of the TEE under a public identity TEE_{id} is crucial: users are able to verify the signature against the corresponding public key, and able to trust that the results were generated correctly due to the fact that registered TEEs provide an attestation of their deployed code.

Stateful Functions. Although each TEE registered to a given extension will deploy the same code, this does not ensure that they will exhibit identical behavior. Certain functions required of the TEEs may be stateful, such as performing an operation with cryptographic keys stored in the TEE. For many applications this is not a problem as long as each participating TEE is correctly identified by its identity TEE_{id}. On the other hand, some applications may relay the same command to multiple TEEs and perform an aggregation over their outputs to ensure suitable redundancy.

3.2 TEE Instruction Work Flow

Once an extension is initialized, instructions can be defined on the extension. An instruction is a command issued on Flare to be completed by the TEEs. The implementation flow of an instruction proceeds as follows:

1. One or more TEEs registered to the extension identified by ExtensionID on FCC run code able to execute a certain instruction, or family of instructions, $I(\cdot)$ where the argument denotes a parameterization for the instruction. Each TEE registered to the extension is identified by its unique public identity TEE_{id}, for which it owns the corresponding private key.

2. A permissioned user U issues an instance of the instruction $\{I(p), T\}$, for input parameters p and T the list of TEEs to complete the instruction, to the TEE smart contract on Flare.

3. Flare's data providers pick up the user instruction $\{I(p), T\}$ on the contract, confirm that the instruction is a valid instruction on ExtensionID and that the user U has permission to issue it, and begin the process of relaying it to each TEE with $TEE_{id} \in T$. First,

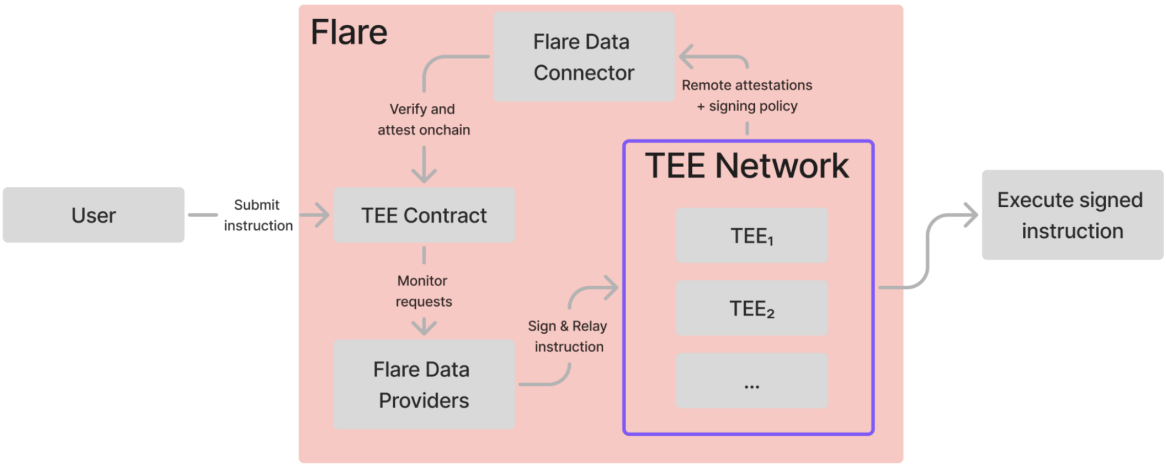

Figure 2: Issuing a Generalised Instruction from Flare

they assemble the details required by the TEEs necessary to complete $I(p)$, then they sign this package and relay it to the TEEs.

4. For each $\text{TEE}_{id} \in T$, once that TEE has received the instruction $I(p)$ and a combined weight $\sum_{j \in J} W_j > W_{I(p)}$ of the data provider signatures, they execute the given task. Here, where $W_{I(p)}$ is the signing threshold for instruction $I(p)$ and J the set of data providers whose signatures the TEE has received.

5. Once the instruction has been executed, the results can be fetched from the TEE machines on an external API hosted by their proxy. Any entity with access to the API can then relay the (signed) result to the appropriate location. For example, Flare's data providers relay signed FDC2 attestations back onto Flare.

Data Provider Augmentation. In this setting, the third point is where the data providers do the majority of the work: the user sends an instruction to the TEE contract on Flare asking for the data providers to instruct the TEEs. This instruction need not contain the necessary details for the TEEs to actually complete the instruction. Instead, it is up to the data providers to correctly assemble the package required by the TEE to successfully execute the instruction. For example, in the PMW case, this is where data providers assemble the transaction correctly; in the FDC2 setting, in this step the data providers are responsible for verifying the attestation. Thus, use cases need to be vetted and clearly defined in advance, so that Flare's data providers are able to bootstrap an instruction on Flare to an instruction that the TEEs can implement.

Since the data providers do their work off-chain, this allows substantial flexibility in the types of instruction supported by FCC: as long as the data providers can be convinced to support them, building use cases can be left to the imagination of Flare's wider community.

Instruction Management. Users are only able to issue instructions from a pre-defined list of available instructions on the extension. Adding and removing instructions from the extension is a responsibility of the extension management. TEEs demonstrate their capability to fulfill each instruction on the extension via an attestation, with the list of TEEs that support each instruction stored on the management contract. Included in the definition of an instruction is a list of users, or types of entity, who have permission to issue the instruction on the extension.

Signing Thresholds. The extension owner equips each possible instruction $I(\cdot)$ with a signing threshold $W_{I(\cdot)}$, so that the instruction will only be executed once the TEE machine receives signatures from data providers with a combined weight on the current signing policy exceeding this threshold $W_{I(p)}$. The default threshold is 50%, in accordance with Flare's usual signing threshold.

Cosigners. For particularly security conscious users, an option to augment the data provider signature requirement with an additional cosigner option is available. In this case, the user who is instructing TEE operations on Flare specifies up to n cosigner addresses and a parameter k for an (n, k) threshold signature scheme. When this option is enabled, the TEE requires both enough weight of data provider signatures and k or more cosigner signatures in order to execute an instruction.

The choice of cosigners is arbitrary and up to the instructing user. Note that since data provider signatures are still required to instruct TEE operations, misbehaving cosigners can only compromise liveness of the TEE operations. Thus, users choosing this option must be careful to nominate reliable cosigners who they expect to provide consistent availability.

Cosigner Enforcement. One additional security consideration is enforcement of cosigner signatures: data providers could, in theory, send the TEE machine a modified version of an instruction $I(p)$ for which the user has requested cosigners, redacted such that the cosigners field has been removed. Since it is not possible to prevent this on a per-instruction basis, each extension much have its own method for enforcing that cosigner signatures are checked. For example, in the PMW case (see Section 4), each key stored by a TEE for signing transactions has an optional metadata field storing parameters (n, k) and n cosigner addresses. When this field is enabled, all instructions using that key require at least k signatures from the designated cosigners, so that if the data providers redact cosigner information on an instruction it is automatically rejected.

Using Instruction Results. Typically, the instruction executed by the TEEs will result in them making signed data available on an API at their proxy, with results retrieved by data providers to be published on the Flare network. Results will be signed by the executing TEE, either with the key corresponding to TEE_{id} or by some other key owned by the TEE. Once they are published on the chain, users can verify the results and use them as inputs in smart contracts on the network. Some instructions, such as PMW transactions, will require that results be relayed to external chains: in this case, the extension must have its own method for ensuring that this step is completed.

3.3 Fees and Rewarding

In order to access FCC, users must pay fees, which are paid on Flare in its native FLR. Fees are charged to users for each function call made to the TEE contract, with each type of call having a different fee. The fees will be distributed to Flare's data providers to reward their role in securing the system and to the TEE operators to cover their infrastructure costs and reward their participation.

Rewards are distributed to the data providers in accordance with their weight and the quantity of requests that they handled in a timely manner. As part of their operation, TEEs will periodically compile and sign a list of which data providers gave their signatures in an allotted time frame for each relayed instruction. This signed rewarding data is then collated and published to the network by the data providers. Each data provider will receive a proportion of the rewards allocated to each instruction for which their signature was received

by the TEE in a timely manner, with the proportion received corresponding to their signing weight. Here, a timely manner means that either their signature arrived within the first 50% of the weight of signatures or within a short grace window after the required 50% majority was reached. Rewards that would be allocated to data providers who did not fulfill this criteria for a given instruction are burnt.

Rewards allocated to TEE machine owners will be assigned corresponding to the amount and type of instructions executed by the TEEs under their ownership, with each instruction having its own fee. In the case when multiple TEEs fulfill the same instruction, rewards will be split evenly among those TEEs.

4 TEE Protocol Managed Wallet

A TEE Protocol Managed Wallet (PMW) is a blockchain account managed on Flare but hosted on an external chain, such as the XRP Ledger or the Bitcoin Blockchain, that is dedicated to submitting payment transactions. It is a data structure contained within three places: on smart contracts on Flare, where its configurations and instructions are managed, on a set of TEE machines that store keys and sign transactions, and on the external network where the transactions are issued. The TEE PMW infrastructure is hosted as an application on the system extension of FCC. For clarity, a more precise definition follows.

Definition 4.1 *A TEE protocol managed wallet is an account W_C on a blockchain C whose private key or keys $sk_1, \ldots, sk_n$ are stored on one or more Trusted Execution Environments $TEE_1, \ldots$ TEE_m that are signed up to the system extension on FCC. These TEEs are deployed with code such that they only sign transactions from W_C in response to instructions that have been signed by a threshold of the signing weight of Flare's data providers. In turn, Flare's data providers only submit the instructions to the TEE in response to transaction instructions issued on the TEE smart contract by registered wallet owners on Flare. Once a transaction is signed by the appropriate TEEs, it is available to be relayed to the blockchain C so that it is correctly issued by W_C.*

In particular, the issuance of transactions from the wallet is triggered by the Flare user who own the wallet by issuing the corresponding instructions on Flare; these are then signed by Flare's data providers and relayed to the TEEs, who sign the final transaction. The TEEs run code controlling access to keys that manage the account W_C such that there is no other way for transactions to be issued

4.1 Initialization

Any user on Flare can setup and own a TEE PMW. A user who wishes to own a PMW must first register a Project, a data structure that stores multiple wallets with the same owner. Then, to initialize a wallet W_C on a chain C, the user issues an initialization instruction on the system extension containing the following parameters:

- **Project ID**: The ID of the project to which the wallet belongs.
- **Wallet ID**: A unique identifier for the wallet.
- **Admin Address**: The (Flare) address that instantiated and administrates the wallet. Note that this is not the address that issues payment instructions; rather, it is responsible for configuration and issuing configuration commands.
- **Wallet Address**: The (external) address W_C of the wallet on blockchain C.

- **Issuing Address**: The (Flare) address that is allowed to issue payment instructions.
- **Key Admin Information**: The (Flare) addresses of Key Admins, used as cosigners in the key recovery process of corrupted keys, and a signing threshold for their cosigning.
- **Key Definitions**: The information on which TEEs generate and hold the keys to sign transactions from the wallet address on blockchain C. Note that the external payment address may be a multi-sig address, thus more than one key may be needed. To approve each key, an FDC2 attestation request is made to verify the details of the holding TEE.

In response to an initialization call to the TEE smart contract, the specified TEEs generate keys according to the Key Definitions and initialize the account W_C on C.

Key Definitions and Storage. For each key related to the Wallet ID, the key definition parameter contains the required information about the structure of the key. As well as functional information, including the expiration time of the key and the signing method of the wallet, this also includes information around the TEE storage structure. A TEE with ID TEE_{id} that generates and stores the key is assigned for each key. Additionally, each key is backed up using a secret sharing scheme among Flare's data providers and additional key admins. More detailed information on key recovery can be found in Section 4.3.

Ongoing Management. Once a wallet has been initialized, its ongoing management is also handled through the TEE wallet contract. The complete list of possible calls is too long to enumerate here, but some particular functions of interest include:

- Deleting keys from the wallet.
- Triggering the restoration of a backup of a key on a specified TEE.
- Transferring ownership of the TEE wallet to a new user on Flare.
- Pausing, resuming, or deleting the wallet.

FCC Key Management. The management structure for keys described in this section, where data is organized in a system of projects, wallets, and keys with associated backup and management mechanics, is generic to the FCC. Since it is not specific to the PMW infrastructure, FCEs that require TEEs to handle private/public key pairs beyond those required for the TEE identities may re-use it if desired.

4.2 Submitting a Transaction

Once the TEE wallet has been set up, the user can now order transactions to be sent on blockchain C via Flare. To do so, they submit a payment instruction detailing the transaction to the smart contract on Flare. This instruction is picked up by the data providers, who compile the details required to complete the transaction on C, then sign and relay it to the key holding TEE(s). Once a TEE has received a weight of signatures exceeding the signing threshold (50% for PMW transactions) from the current signing policy, it signs the transaction with its stored keys. The transaction can then be retrieved from the TEE proxy and relayed to C. Assuming that all steps were completed correctly, the transaction will now be accepted on C.

Submitting a payment instruction in this way is an example of the generalized instruction flow depicted in Section 3.2. For clarity the method for sending a transaction is depicted in Figure 3 and laid out in detail below:

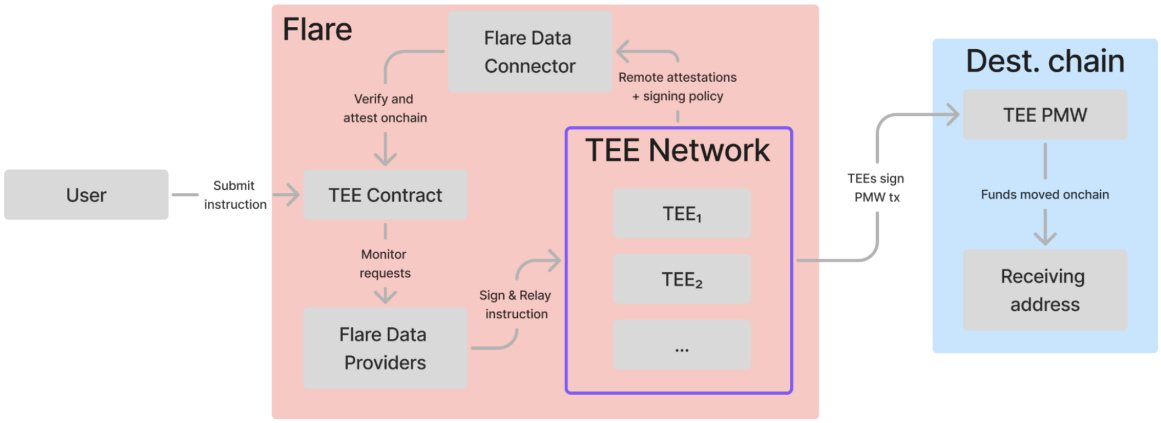

Figure 3: Issuing a PMW Transaction on Flare

1. The user submits an instruction to the TEE contract on Flare instructing a transaction T_C be issued on blockchain C from account W_C. The instruction specifies the Wallet ID of the TEE PMW. The transaction instruction includes the amount to be transferred, the receiving address, the offered fee (on the source chain), and a payment reference. The instruction and wallet address implicitly includes a list of the one or more TEE machines to which the instructions need to be relayed.

2. Upon picking up the payment instruction, Flare's data providers each independently assemble the transaction instruction T_{TEE}, a packet containing all the information necessary for the TEE(s) to sign the transaction from W_C. The contents of this package depend on the rules of the source chain C. The data providers then sign the packet, with the ith data provider producing a signature $\text{Sign}_i(T_{TEE})$, and send the signed instruction to the appropriate TEE(s). In cases of multiple TEEs, these instructions are issued independently by the data providers to each TEE.

3. Once a TEE has received signatures from a set of providers J whose combined weight in the current epoch exceeds the signing threshold, $\sum_{j \in J} W_j > 0.5$, the TEE signs the transaction T_C using the key corresponding to W_C and makes the transaction available. Again, in cases of multiple TEEs each completes this step independently e.g. waits until it has received enough data provider signatures and then submits its signature for T_C.

4. The transaction is retrieved from the TEE and relayed to C. Assuming all steps of the protocol were completed correctly, transaction T will be executed on blockchain C in accordance with the rules on that chain.

Executing a Transaction. In the third point of the above flow, the TEEs are responsible for signing the transaction once appropriately instructed by the data providers. However, in the fourth phase, these signatures and the transaction still need to be published on C. This is done permissionlessly: TEE proxies host APIs which can be queried to collect the signatures and transaction, so that any entity can collect and publish them on C. Since the process is permissionless and cannot be falsified, it is abstracted in this paper: once the TEEs sign the transaction, it is assumed that the transaction and signatures are published on C.

Cosigning. In the same manner as discussed in Section 3.2 for general instructions, security conscious users may combine the requirement for data provider signatures on PMW transactions with an additional (n, k) threshold signature scheme for n nominated cosigners, who

are chosen by the user. In this case, two phases of the protocol are changed. In phase two of the transaction, the cosigners each assemble the transaction instruction T_{TEE} and submit a signed copy of the instruction to the appropriate TEEs. Then, in phase three, the TEEs will not sign the transaction until they receive both a sufficient weight of data provider signatures and at least k cosigner signatures. Essentially, the cosigners function as a parallel set of signers alongside the data providers. Otherwise, the protocol remains unchanged.

If a user wishes to enable cosigners, this option is enabled on a per-key basis. That is, when a key is generated the owner may specify cosigning information. If so, then the TEE stores as metadata for that key the addresses and signing thresholds for its cosigning. Then, any transaction using the key must have been signed by at least k of the cosigners, rather than this option being enabled on a per-instruction basis. Users selecting this option should thus take care to choose cosigners carefully, as they are required for liveness of all submitted transactions.

Assembling a Transaction. Hidden in the second point of the transaction flow is a major duty of the data providers: correctly assembling a transaction. The TEE wallet infrastructure on Flare responds to transaction instructions by users, simple calls requesting a transaction from a TEE PMW on an external chain that specify a receiving address and an amount. The data providers are then responsible for correctly assembling this information into a transaction which can be handed over to the TEE to be signed, the complexity of which depends on the requirements of the underlying chain. Regardless, the required call to the TEE contract on Flare is kept simple for the user, with the work handled off-chain by Flare's data providers.

Multi-Sig Chains. The TEE PMW is capable of supporting multi-sig addresses on C. In this case, independent TEEs will hold keys corresponding to the multi-sig, and when a transaction is submitted on Flare the data providers can read this list from the wallet's configuration. The data providers then send signed transaction instructions to each TEE separately. Each of these TEEs independently sign the transaction once they have received a sufficient weight of signatures from Flare's data providers. Otherwise, the transaction flow listed above remains unchanged.

Batching. The TEE PMWs can support batching of transactions (sending multiple payments in a single transaction) as long as the underlying chain supports it. Users wishing to send a batch of transactions can enable a setting on their wallet specifying a batch size, the maximum number of transactions in a batch, and a batch duration, indicating the maximum amount of time transactions will be batched for. Transactions ordered from a wallet with this setting enabled will be batched together: each time a transaction is sent, if there is no open batch, a new batch is started. Otherwise, it is put into the current batch. A batch accepts new transactions until either the batch size or batch duration is reached, at which point no more transactions can be placed in the batch and the data providers begin to assemble and relay the completed batched transaction to the TEEs for signing.

Failed Transactions. From time to time transactions made from a TEE PMW may fail. For example, the offered fees may have been too low to be picked up on the source chain, or the user may have requested an impossible transaction. In this case, the TEE infrastructure on Flare hosts two mitigations. Firstly, users can instruct the reissuance of transactions using a specific reissue command, including identifying information for the transaction to be reissued and a new fee, typically larger than the old fee. This is particularly pertinent in cases where a transaction on a specific nonce (sequence number) has gotten stuck, and thus must be reissued

or nullified to push forward existing transactions on subsequent nonces that must be executed in sequence. Secondly, users can request that an FDC attestation request is made proving that a transaction was nullified or reverted, thus consuming the relevant nonce. In this case, the protocol determines whether or not a new payment instruction needs to be issued.

Latency. Unlike Flare's enshrined protocols, which proceed in a sequence of staggered rounds, instructions issued to the TEE PMWs do not have a fixed latency. Instead, Flare's data providers are incentivized to sign and relay transaction instructions to the TEE as quickly as possible. The TEE will then sign the transaction and make it available as soon as it has received the signatures of a weighted majority of data providers. Thus, the exact latency will vary with the response time of Flare's data providers, but transactions should be thought of as being issued in a matter of seconds.

Signing Threshold. In the third phase of the transaction flow, the TEEs submit signatures for a transaction upon receiving a sufficient weight of signatures from data providers. In this case, sufficient means 50% weight according to the signing policy of the current reward epoch on Flare, corresponding to an honest weighted majority of data providers. The 50% threshold was chosen for system extension applications (PMWs and the FDC2) to align the security of the protocols with that of the signing policy, and thus the network itself. In later designs this 50% threshold may be changed, or indeed made flexible at the discretion of the wallet owner. Increasing this threshold improves security, as more malicious data providers are needed to force through a false transaction, but damages availability, since a higher number of signatures is necessary to execute an honest transaction. However, without corresponding changes to Flare's signing policy the wallet protocol's security would deviate from that of the overall network's.

4.3 Key Management and Backup

TEEs registered to PMWs will need to handle the corresponding cryptographic keys. If a TEE machine is powered down, either accidentally or maliciously, all the contents of its memory are lost. In this case, this would include the keys for any wallet addresses (or other applications) that are stored in the TEE. Thus, precautions are required to manage the back up and restoration of keys owned by registered TEEs.

Keys are backed up by a two stage mechanism. First, the key owning TEE generates 2 shares $S_{\text{provider}}, S_{\text{admin}}$ for an additive Secret Sharing scheme, so that $S_{\text{provider}} + S_{\text{admin}} = S \mod N$, where S is the secret being shared.. These shares are then further split using two more secret sharing schemes, the shares of which are distributed to be stored by two sets of entities: one share is split among the data providers, and the other among a collection of specified key admins for the PMW.

Thus, recovering the original key consists of two stages: first, the data providers and key admins provide shares necessary to recover the two parts of the initial secret sharing, then these shares are recombined to recover the original key. Since the weight and identity of Flare's data providers evolves over time according to the signing policy, each key back up is rerolled on the introduction of a new policy. These processes are explained in more detail below, and summarized in Figure 4.

Triggering a Backup. Each key is backed up in two circumstances: on initial generation, a backup of the key is generated and distributed among data providers on the current signing policy and the key admins. Following this, each time the signing policy is updated at a TEE machine (which occurs roughly every 3.5 days) a fresh backup is generated for each key stored

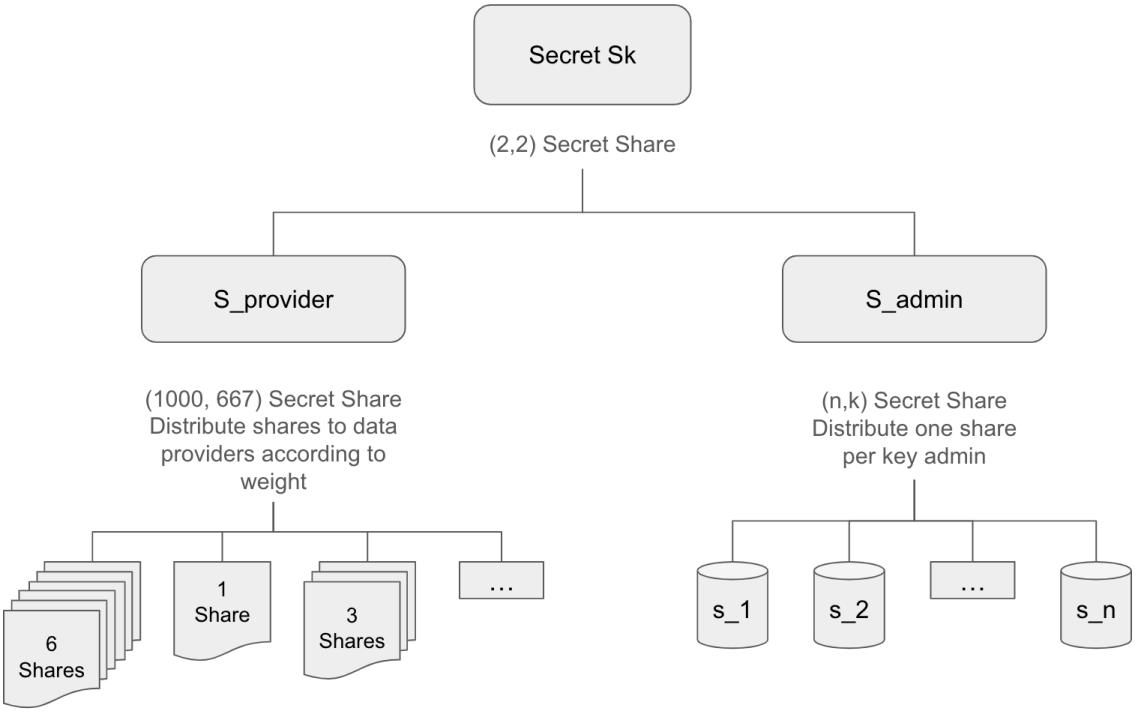

Figure 4: Backing Up a Key in Two Stages

by the TEE that is used in a PMW. The new backup is generated in accordance with the new signing policy, with data providers on that policy receiving the relevant amount of shares, so that backups evolve with the set of data providers.

Key Admins. The key backup information is distributed between two types of entities: Flare's data providers and (optionally) a set of key admins specified by the wallet owner. On initializing a key, the wallet owner specifies an arbitrary amount n_{admin} of key admins, each identified by a public identity corresponding to a public key $PK_1, \ldots, PK_{n_{\text{admin}}}$. Additionally, the user specifies a threshold value k_{admin} of admins required to recover the key admin share of a key backup for the key. These two parameters define a Shamir Secret sharing scheme that will be applied to S_{admin}. Each key admin receives a single share of the S_{admin}, and thus has an equal share of the responsibility during the backup process.

The identity and nature of the key admins is at the discretion of the user: thus, it is crucial that the user specifies key admins that it trusts, both to maintain liveness of the backup and to prevent the key admin share of the secret from leaking.

Distributing the Data Provider Share. The data provider share S_{provider} is split and distributed between the data providers in accordance with their weight, so that two thirds of the signing weight of data providers are required to recover the share. That is, a large integer n is chosen, and S_{provider} is split using an $(n_{\text{provider}}, \lceil \frac{2}{3} \cdot n_{\text{provider}} \rceil)$ Shamir Secret Sharing scheme, with each provider receiving a number of shares proportional to their signing weight[1]. Thus, the jth provider with normalized signing weight W_j receives $n_j := \lfloor W_j \cdot n_{\text{provider}} \rfloor$ shares $s_j^1, \ldots, s_j^{n_j}$ of S_{provider}, such that any combination of $\lceil \frac{2}{3} \cdot n_{\text{provider}} \rceil$ shares s_j^i distributed across any number of providers can be used to recover S_{provider}.

[1]The initial value for n_{provider} will be around 1000, but may be increased depending on the performance of the system.

Key Share Metadata. All key shares, both for data providers and key admins, are distributed with associated metadata, confirming the correctness and provenance of the share. This data is signed by the key holding TEE and allows for key shares to be uniquely identified and associated with the correct holder, key, and backup ID. The purpose of this metadata is twofold: during generation, it allows data providers and key admins to be sure that they have received valid key shares generated by the underlying TEE machine. During recovery, it ensures that data providers and key admins have provided the correct share and thus can not undermine the reconstruction process without detection, for example by intentionally presenting an invalid key share.

Specification. More formally, the key backup mechanism for a secret key sk proceeds as follows:

1. The key owning TEE with identity TEE_{id} triggers back up procedure of a key with identity Key_{id}, either as part of the generation of that key or on update of the signing policy.

2. The TEE begins the backup process for the secret part sk of Key_{id}. First, sk is split uniformly at random into two shares $S_{\text{provider}}, S_{\text{admin}}$ satisfying $S_{\text{provider}} + S_{\text{admin}} = sk \mod N$.[2]

3. Next, the TEE splits the data provider share S_{provider} into n_{provider} shares using an $(n_{\text{provider}}, \lceil \frac{2}{3} \cdot n_{\text{provider}} \rceil)$ Shamir Secret Sharing scheme. Then, it consults the relevant signing policy, either the current one or new one, and assigns these shares to the data providers so that the jth provider with signing weight W_j is allocated $n_j := \lfloor W_j \cdot n_{\text{provider}} \rfloor$ shares $s_j^1, \ldots, s_j^{n_j}$.

4. Following this, the TEE splits the key admin share S_{admin} into shares $s_1, \ldots, s_{n_{\text{admin}}}$ using an $(n_{\text{admin}}, k_{\text{admin}})$ Shamir Secret Sharing scheme, assigning the ith share to the ith listed key admin.

5. Finally, all assigned shares are encrypted under the public key of the entity that they are assigned to and assembled into a single backup package, posted to a backup URL. Shares in this package are indexed by the public key of the owner, so that providers can find their shares. Each data provider can decrypt its shares $s_j^1, \ldots, s_j^{n_j}$ of S_{provider} from the package as they are encrypted under the public key corresponding to its address. Similarly, each key admin can recover its share s_i of S_{admin} encrypted under its public key.

6. Each time the signing policy is updated, the TEE repeats this process for Key_{id} from Step 2, generating a fresh set of shares for each secret sharing scheme. This ensures that the set of data providers on the current signing policy can be used to restore the key, and that data providers on an old signing policy can not do so without colluding with the key admins.

Analogously, the recovery process works as follows:

1. The owner of the application using Key_{id} issues an instruction to recover the key on an associated TEE with identity $\text{TEE}_{id'}$.

[2]Here N is defined by the type of key that sk is e.g. an ECDSA key is an integer modulo some integer N that depends on the modulus of the signature scheme.

2. Each data provider encrypts a package containing its shares $s_j^1, \ldots, s_j^{n_j}$ of S_{provider} under the public identity key associated with $\text{TEE}_{id'}$ and sends this package as an instruction for the TEE.

3. Similarly, each key admin encrypts a package containing its share s_i of S_{admin} under the public key associated with $\text{TEE}_{id'}$ and sends this instruction.

4. Once at least $\lceil \frac{2}{3} \cdot n_{\text{provider}} \rceil$ encrypted data provider shares and k_{admin} encrypted key admin shares are available, the TEE decrypts the packages, recovering the individual provided data provider and key admin shares. It then recombines the data provider shares to recover S_{provider} and the key admin shares to recover S_{admin}.

5. Once $\text{TEE}_{id'}$ has recovered the data provider and key admin shares, it combines the 2 shares mod N to recover sk. Proof of the successful recovery is made available by the TEE, so that data providers can confirm that the key has been successfully restored.

In summary, the functions required for the key backup scheme are:

- **Key Backup**: For a TEE_{id} holding a key sk, given a list of key admin addresses $PK_1, \ldots, PK_{n_{\text{admin}}}$, a signing policy P, and a parameter k_{admin}, shares of the secret key sk are generated, encrypted, and batched into a backup package for the data providers and the key admins in accordance with the described process.

- **Key Restore**: On input a $\text{TEE}_{id'}$ permitted to restore a secret key sk corresponding to a key identified by Key_{id} and the signing policy on which the backup was generated, the data providers and key admins begin the recovery process, passing on the appropriate decrypted shares of the key to $\text{TEE}_{id'}$. From these shares, sk is regenerated on $\text{TEE}_{id'}$.

Security. Security of the backup process relies on the underlying secret sharing schemes: to compromise a backed up key, an attacker would require collusion between at least two thirds of the data providers and k_{admin} of the key admins. In particular, the user should only have selected trusted admins, and the data provider threshold is higher than the generic signing threshold on Flare.

A second more subtle requirement is that the same key can be backed up multiple times without compromising security. Fortunately, both components of the secret sharing mechanism have this property: the initial split is uniformly random, and Shamir Secret Sharing has the required dynamic property, where the same key can be shared multiple times without issue.

5 Flare Data Connector v2

As part of the system extension within FCC, a TEE based version of the FDC, known as the Flare Data Connector v2 (FDC2), is deployed. The FDC is Flare's enshrined protocol for providing attestations of external events in response to user requests, so that the truth of these events, validated by the data providers, can be used on Flare. The design of the FDC confirms requests periodically, collecting batches of requests every 90 seconds and confirming valid requests shortly afterwards. The FDC2 improves on the latency of the FDC by leveraging the TEE infrastructure to provide remote attestations in response to specific requests. This circumvents the 90 second collect phase that is required in the FDC to package many attestation proofs into a single Merkle root. As an additional benefit over the FDC, fees can be charged at a flat rate, rather than varying to incentivize inclusion in a given round. The FDC2 running on the system extension operates using TEE instructions, specifically as follows:

1. A contract on Flare receives FDC2 attestation requests from Flare's users. A collection of TEEs, identified by their public identities TEE_{id}, will be registered on FCC's system extension to execute the confirmation of attestation requests sent to this contract.

2. A user submits an attestation request $A(x)$ to the FDC2 contract asking for an attestation of data x.

3. The Attestation request $A(x)$ submitted to the FDC2 contract is picked up by Flare's data providers, who confirm the validity of the attestation request and the truth of the information x.

4. Assuming that the request is correct and accurate, the data providers each perform and sign the attestation, resulting in a confirmed attestation $A_j(x)$ of information x by provider j. They then relay the signature, confirmation data, and request to the TEEs participating in the FDC2.

5. Once a TEE has received signed attestations from a set of providers J satisfying $\sum_{j \in J} W_j > 0.5$ of the weight of data providers according to the current signing policy, they sign an attestation response $A_{id}(x)$ for $A(x)$. The attestation response $A_{id}(x)$ includes the request $A(x)$, confirmation data, and a package containing the signature of each data provider. The signature is performed using the private key corresponding to TEE_{id}.

6. Flare's data providers retrieve the signed attestation response from the TEE proxy and publish it on Flare. The attestation $A(x)$ is now confirmed on the network, and the truth of data x can be used.

This design exhibits and leverages the flexible nature of the FCE infrastructure. On an abstract level, all that is required to implement a general instruction is a contract on Flare where the data providers can receive requests and a collection of TEEs that are set up to execute appropriately signed commands. In this specific case, the instruction is simply to publish a confirmation that the user attestation request has been verified by Flare's data providers, and thus by the TEE. Since it is signed by both the TEE's public identity and the data providers, the user can trust the confirmation.

References

[1] Harry Roberts. *Trusted Execution Environments*. URL: `https://coinmarketcap.com/academy/glossary/trusted-execution-environments-tees`.

[2] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. 2015, pp. 57–64. DOI: `10.1109/Trustcom.2015.357`.

[3] Google. *Confidential Computing Overview*. URL: `https://cloud.google.com/confidential-computing/docs/confidential-computing-overview`.

[4] Flare Network. *The Flare Data Connector*. 2025. URL: `https://dev.flare.network/pdf/whitepapers/20240224-FlareDataConnector.pdf`.

[5] Flare Network. *Flare Systems Protocol*. URL: `https://dev.flare.network/network/fsp/`.

[6] Flare Network. *FTSOv2: More Data Feeds and Faster Updates to the FTSO*. 2024. URL: `https://dev.flare.network/pdf/whitepapers/20240223-FlareTimeSeriesOracleV2.pdf`.

[7] Antonio Muñoz et al. “A Survey on the (in)Security of Trusted Execution Environments”. In: *Computers and Security* 129 (2023), p. 103180. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2023.103180. URL: https://www.sciencedirect.com/science/article/pii/S0167404823000901.