Cryptography and MPC in Coinbase Wallet as a Service (WaaS)

Yehuda Lindell
Coinbase

June 15, 2023

Abstract

Coinbase WaaS (wallet as a service) utilizes advanced cryptographic techniques from the field of secure multiparty computation (MPC) in order to provide a unique combination of best-in-class usability and security, unlocking the ability for everyone to hold and use digital assets. Although MPC is a well-known technology today for addressing key management problems in blockchain and cryptocurrency applications, there are many subtleties in deploying an MPC wallet. In this technical white paper, we describe the cryptographic design of the key management mechanism used in Coinbase WaaS, and how it manages to overcome some of the biggest challenges facing today's wallet infrastructure.

1 The Self Custody Impediment to Broad Wallet Deployment

Coinbase WaaS enables organizations to bring cryptocurrencies, blockchain and Web3 technology to everyone. However, this requires everyone to have a wallet, and in particular it requires everyone to have digital assets. This introduces the challenge of how to protect those digital assets. There are two classic approaches to this. The first is to use a custodian or exchange, making it very easy to deploy (since the user doesn't actually have to hold anything except a method to authenticate), but this suffers from the risks associated with this being "not your keys; not your coins/assets". The second is for the users themselves to hold and protect their asset private keys, with them then assuming all the burden of preventing the keys from being lost or stolen. We discuss these two approaches, and then present a third approach based on MPC that provides the best of both approaches.

Self-custody. Most existing cryptocurrency and blockchain self-custody wallets hold the key in whole on the user's mobile device or browser extension, or in dedicated hardware. This provides the user with full control over their keys, but is very problematic both with respect to *usability* and with respect to *security*.

First, the *usability* of this solution is very problematic. Users have to export a mnemonic and store it safely, or risk losing their keys and all of their funds. In many cases, users are not even aware of this, and ask the wallet provider to "reset their password" if they lost their wallet, as if it were an exchange and everything can be recovered by merely authenticating to the organization. In a WaaS deployment scenario, where users should be able to install a wallet as they go without any friction, a traditional self-custodial wallet just won't work. If users are supposed to store their mnemonic somewhere safely before proceeding – or run the risk of losing their assets – then they

are going to very often just opt out. Stated differently, traditional self custody may be an excellent solution for sophisticated cryptocurrency and blockchain users, but it is not an approach that can work when bringing this technology to the masses.

The second problem is that of *security*: storing valuable keys on a single consumer device that is vulnerable to attack like a mobile phone can be risky. Furthermore, the backup of the mnemonic introduces additional security concerns. In fact, the better the backup of the mnemonic in terms of preventing loss, the higher the threat of theft. For example, backing it up in multiple locations provides multiple points of breach for theft. Storing it in a cloud storage account encrypted under a user password is risky since users still frequently use weak passwords or reuse their password from other services. Furthermore, users often forget their passwords (and the ability to remember any password that isn't vulnerable to an offline dictionary attack is questionable in any case[1]). On the flip side, storing it in plaintext in cloud storage significantly reduces the risk of loss, but greatly increases the risk of theft. Finally, the fact that the user sees a mnemonic and handles it makes the risk of social engineering attacks a real one. This problem imposes a major burden on the user for usability, as we have discussed above.

Exchanges and custody. The other option available to users is to use an exchange or cryptocurrency custodian. This has the advantage of moving the burden of managing the keys, preventing them from theft and loss, away from the user and to the exchange. Furthermore, a good exchange that really cares about security is far less likely to be breached than a regular (non-expert) user. Such an exchange can also deploy anti-fraud measures, policies limiting the transaction amounts in order to mitigate the damage in case of impersonation attacks, and more. However, this has its own problems. First, not all exchanges are equal, and we have unfortunately seen exchanges breached and bankrupted, with the result being users losing funds – completely beyond their control. Second, the decentralized vision of cryptocurrency – where parties hold their own funds by holding their own keys – is not aligned with the exchange custodial approach. Furthermore, personal assets like NFTs and Web3 accounts are problematic with respect to storage in a centralized exchange since exchanges typically pool assets together (something which just cannot be done in this case). Finally, the whole point of Web3 infrastructure is to guarantee users access to their assets without reliance on any third parties.

A new approach based on MPC. The current state of affairs described above is unsatisfactory and an obstacle to mass adoption. We need solutions that provide users with full control over their own keys but with usability and security that matches that of exchanges and custodians. Fortunately, MPC (secure multiparty computation) enables us to achieve exactly what is needed. Very briefly, MPC enables us to split the key between the user and an external entity that we will call the wallet provider (Coinbase in this case) and to carry out operations by running an advanced cryptographic protocol that generates a signature without ever bringing the key together. As a

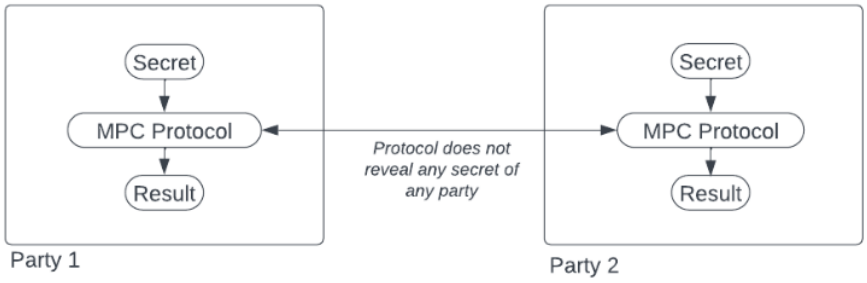
[1] In an offline dictionary attack, the attacker is able to guess passwords and verify the guess without interacting with the system. Consider the case of encrypting the user mnemonic using a hash of their password as the key. Given the ciphertext, the attacker can guess the user's password and try to decrypt the mnemonic. The attacker can then check if the mnemonic guess is correct (either if the file has a specific format, or by comparing the public keys or addresses derived from the mnemonic with the user's known public keys). Given that it's possible to check billions of hashes per second, this is an extremely effective attack. There are countermeasures like adding salt to the hash (to prevent rainbow table attacks) and iterating the hash (to slow down the attacker thousands-fold) but when the target is truly valuable these are merely cosmetic. The length of the user password needed to protect against such an attack is so great that it would need to be written down. But then one may as well just write down the mnemonic itself and we are back to square one.

result, the user's key remains protected even if the key share used to sign transactions is stolen from their device – since a single share of the key is meaningless without the other – and the wallet provider cannot generate a signature without the user since they also only hold one share. This means that the wallet provider does not own the user's keys or funds, with all of the implications that come with that. However, the user still relies on Coinbase's cooperation. In particular, since the user also only holds one share of the key, they cannot generate a signature without Coinbase's cooperation either. In order to give the user full control of their assets and keys, we provide the user with a special self-custody backup that enables them to securely recover their key and export it by themselves if needed. However, we expect that in the vast majority of cases this will never be used, and we provide users with another extremely simple restore from backup method that works in cooperation with Coinbase (with each party holding one share of the backup) that can be used if they lose their phone or the like.

In this white paper, we describe how all of this is achieved. We begin with an overview of what MPC is, and we then proceed to show how it is used to provide high usability together with strong security, all the while providing users full control of their assets and key.

2 Secure Multiparty Computation (MPC) and Crypto Wallets

Secure multiparty computation (MPC) considers a scenario where a number of distinct, yet connected, computing devices (or parties) wish to carry out a joint computation of some kind. The aim of an MPC protocol is to enable them to carry out the computation in a secure manner, even when some of the parties are corrupted and behave adversarially. The aim of this adversarial attack may be to learn private information or cause the result of the computation to be incorrect. Thus, two important requirements on any secure computation protocol are *privacy* and *correctness*. The privacy requirement states that nothing should be learned beyond what is absolutely necessary; more exactly, parties should learn their output and nothing else. The correctness requirement states that each party should receive its correct output. Therefore, the adversary must not be able to cause the result of the computation to deviate from the function that the parties had set out to compute. (We remark that adversarial behavior can be defined in multiple ways, but we consider here *malicious adversaries*. This means that in a two-party protocol, a party who fully controls one of the parties and can run arbitrary attack code, is unable to learn anything they shouldn't and cannot cause the output to be incorrect.) See [6] for a general technical introduction to MPC.



MPC Wallets: In the specific case of interest here, the function to be computed is a digital signature, and the private inputs that the parties hold are random shares of the private key. We consider the case of two MPC participants: Coinbase and the user's device. In order to sign on a transaction, the two parties interact in an MPC protocol – using their shares and the transaction

that they wish to sign upon – in order to generate a signature. In this context, *privacy* ensures that only the digital signature is learned and nothing else. In particular, this means that nothing is learned about the parties' private shares, and so it isn't possible for one of the parties who may be maliciously corrupted to sign in the future without the other. Furthermore, *correctness* means (among other things) that the signature generated is on the approved transaction only, and a malicious party cannot change the amount, recipient address or anything else. Importantly, since the MPC protocol requires participation from both sides, both need to approve and so it is possible to implement controls that are enforced on both sides.

The ability to compute on secrets without revealing them sounds impossible and so some people incorrectly think that it works by one party sending their secret to the other who quickly computes the result and then erases the secret to prevent it from being stolen. However, such a protocol does not provide *any security* in the case that the party receiving the secret from the other party is corrupted themselves. Thus, MPC protocols do not work in this way, and secrets are not revealed at any time. See Appendix A for some examples of how MPC protocols work. We stress that all MPC protocols used by Coinbase have been mathematically proven secure, and so achieve the aforementioned security properties.

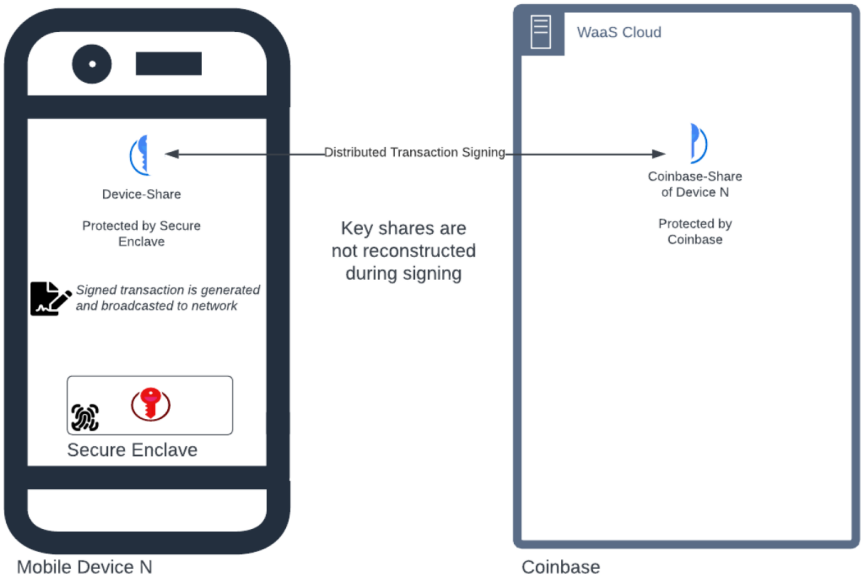
Before proceeding, it is important to understand what "corruptions" we are concerned about here. Needless to say, no one thinks that the user is corrupted and trying to steal from themselves. However, it is very possible that the user's device has been breached by malware of some kind. In a standard non-MPC wallet, such a breach suffices to steal the user's secret key.[2] Likewise, Coinbase has no interest in stealing from its users. However, this doesn't mean that there cannot be a breach at Coinbase either (albeit far less likely). Furthermore, and importantly in this context, the fact that Coinbase *cannot* generate a transaction without the user means that the paradigm of "your keys, your coins" is still fulfilled with an MPC wallet.

Two-party versus multiparty protocols: The current version of WaaS utilizes two-party protocols, with the key shared between Coinbase and the user. However, it is worth noting that everything here can be generalized to a multiparty setting, where the key is shared among additional machines and/or devices. Furthermore, it is possible to utilize threshold sharing of the key, meaning that the key can be shared between some n parties, and any m-of-n of those parties can participate (e.g., 2-of-3, 4-of-7, or whatever is desired). It is even possible to define different sets of parties, so that the only way to sign is if a threshold is reached in each set. For example, one could define a set of 3 parties and another set of 7 parties, and the key can be shared so that it's only possible to sign if 2-out-of-3 of the first set and 4-out-of-7 of the second set approve and participate. This can be utilized in some applications to provide strong and granular control over the key and signing operations.

MPC signing process: All signing in an MPC wallet is run via an MPC protocol involving a server in Coinbase and the user's device, which is a mobile phone (and in the future may also be a web browser on their laptop, for example). The application being built using WaaS may require additional user authentication before beginning the MPC process, using whatever method is suitable for the application (from a simple password to authenticator applications and even YubiKeys). The key share on the user's device is encrypted using the secure enclave, and requires biometric authentication to open (when supported by the device). In addition, policies can be

[2]Of course, this also requires the malware to be able to breach the memory space or storage of the wallet app. However, no operating system is perfect, and such vulnerabilities are impossible to fully prevent. Our aim here is to guarantee security even in the case of such a breach.

defined regarding what transactions are allowed. For example, such a policy can include things like daily amount limits, allow and disallow lists, time-of-day, and more, depending of course on the application. This makes it hard to steal even the single share on the user's device or to issue a fraudulent transaction in the name of the user.[3]

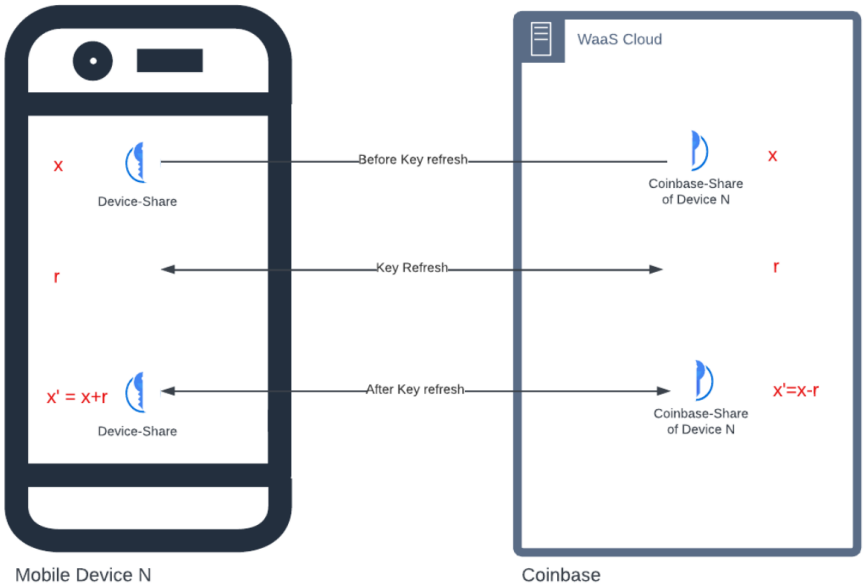


MPC key generation: As we have seen, using MPC to compute signatures given shares of the private key can prevent an attacker breaching one of the parties from doing anything malicious. However, how do we get the shares of the private keys in the first place? Clearly, we could generate the key in a regular manner and then split it, and this would already be much better than the current state of affairs with non-MPC wallets since the window of attack is significantly reduced to the few milliseconds where the key is generated. However, we can do even better than that and use another MPC protocol that generates the keys themselves without them *ever being exposed* on a single device. This ensures that the key is not exposed at any time throughout its entire life cycle.

Key-share refresh: MPC signing prevents an attacker who has broken into one of the parties' devices from stealing the key or cheating in any other way. However, a concern that arises is that over time (maybe even months), the attacker may be able to break into both parties, and at that point the key will be stolen. We mitigate against this threat by *refreshing* the key material *every time that we sign* so that the sharing (but the not key itself) continually changes. For example, consider the key sharing x_1, x_2 with $x_1 + x_2 = x$ for Schnorr/EdDSA signing above. Then, a key sharing refresh is achieved by the two parties running an MPC coin-tossing protocol to generate a random string r that both receive but neither can influence. Given this r, party P_1 can update their share to be $x'_1 = x_1 + r$, and party P_2 can update their share to be $x'_2 = x_2 - r$. Clearly, the sharing is of the same key since $x'_1 + x'_2 = x_1 + r + x_2 - r = x_1 + x_2 = x$. However, if an attacker broke into P_1's device and stole x_1, and later breaks into P_2's device and steals x'_2, then it learns *nothing* about the actual key x (since $x_1 + x'_2 = x + r$ which just looks completely random). Thus,

[3]It is important to note that an attacker does not need to steal a user's key if it can trick the parties into signing on a fraudulent transaction to transfer all of the user's assets to themselves. Thus, although the share on the user's device does not give any information to the attacker about the key itself, if it can be used to impersonate the legitimate user then this is a problem. This threat can be mitigated by strongly protecting the share on the user's device, adding user authentication, and including policies on what transactions are allowed.

the attacker actually has to breach both parties (in our context, both Coinbase and a user's device) at essentially the same time (or more exactly, before a refresh takes place).



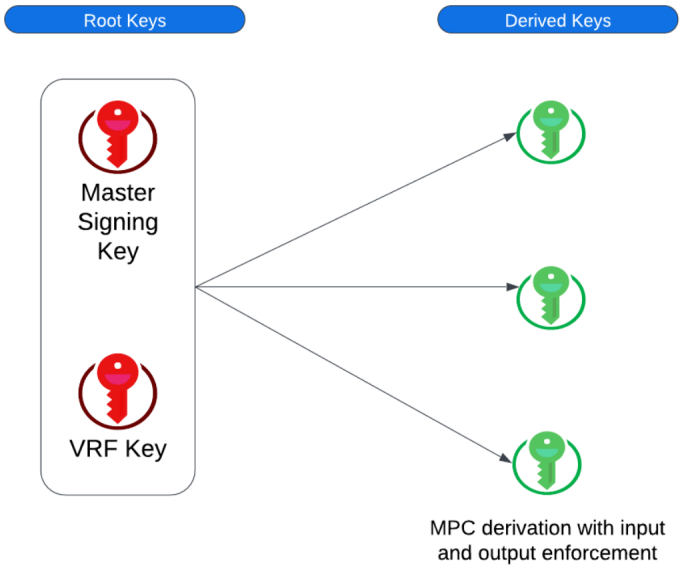
HD wallets in MPC: Most, if not all, crypto wallets are hierarchical deterministic (HD) wallets. This means that all keys are derived from an initial high-entropy seed, and thus it suffices to backup that seed and nothing else. Given the challenges of backup, this is crucial since otherwise every time you create a new key (for a new application, blockchain, or anything else), you would need to back it up. Indeed, generating even one manual backup that controls all of your assets is a huge issue, as we have discussed above regarding mnemonics. We will describe below how backup is taken care of; meanwhile, it suffices to say that by using an HD wallet, the backup needs only to be generated once upon wallet creation. From then on, all keys are derived from the seed, and thus can all be reconstructed by restoring only the seed from backup. This is advantageous operationally and so is the approach that we take. We actually have different HD seeds for different algorithms (e.g., ECDSA and EdDSA), and the description below holds for each separately.

The standard method for creating an HD wallet is described in the BIP-032 standard (with additional elements in BIP-039 and BIP-044). The way that BIP-032 works is to derive keys from the seed, using a specified public path string that is different per asset. The derivation function used is HMAC-SHA512, and it takes the seed as one of its inputs (the derivation is actually multi-step and so the seed is input in the first derivation and afterwards the intermediate derived keys are used, but that's not of importance here). Cryptographically, this method is sound based on the assumption that HMAC-SHA512 is a pseudorandom function, or PRF for short.[4] This means that outputs look random, and so can be used as cryptographic keys for signing. Although it is possible to compute HMAC-SHA512 in MPC, where the parties input shares of the seed (or an intermediate derived key) and receive shares of the derived key as output, it is quite expensive. A naive way of deriving keys would simply be for each party to hold a PRF key, and to locally derive shares of the key by locally computing their own PRF. Specifically, the "master seed" would be replaced

[4]This doesn't follow the standard modeling of HMAC since the seed, or intermediate derived key, is not given as the key to HMAC but rather as the input. Nevertheless, one can reasonably assume that HMAC behaves like a pseudorandom function even in this case, and one can certainly prove that this holds in the random-oracle model [1].

by party P_1 holding a random k_1 and party P_2 holding a random k_2. Then, in order to derive a key for a given path, P_1 would compute $x_1 = F_{k_1}(\mathsf{path})$ and P_2 would compute $x_2 = F_{k_2}(\mathsf{path})$, where F is a PRF. The x_1, x_2 values would then be used as *shares of the elliptic-curve signing key* in an MPC signing protocol for ECDSA, EdDSA or whatever is needed. Although appealing in its simplicity, this approach is extremely problematic. First, it does not support key-share refresh since the parties need to always hold the same k_1 or k_2 so that the derived key shares x_1 and x_2 can be recreated from backup when needed. Second, nothing prevents a cheating P_1 for example from using some other key value $\tilde{k}_1$ to compute $x_1 = F_{\tilde{k}_1}(\mathsf{path})$. This will look exactly the same externally (since the pseudorandom functions make all outputs like random and independent), but means that the backed-up k_1 will not be sufficient to obtain x_1 in the case of restore from backup. Another problem that arises is that even if we can force P_1 to use the correct k_1 to derive x_1, how can we force them to use the correctly-derived x_1 in the MPC signing protocol? This is not at all trivial since standard MPC protocols allow the parties to use any legitimate input. Once again, this means that a cheating P_1 could input a different $\tilde{x}_1$, and once again the backup of k_1 and k_2 will not be sufficient to recover the signing key if needed.

We therefore take a different path to solving this problem. Specifically, we use a *threshold verifiable random function* (VRF) for deriving the signing-key. By using a VRF, the parties are unable to change the pseudorandom value generated without being detected. We use the combination of a shared master key x_1, x_2 with $Q = (x_1 + x_2) \cdot G$ and a shared VRF key k_1, k_2 with $K = (k_1 + k_2) \cdot G$. The method is described in more detail in Appendix A.4 and ensures that the initial backed-up values suffice for reconstructing all keys at a later time, even if one of the parties was corrupted during key derivation. We remark that all of the above relates only to the *hardened derivations* used in BIP-032. In contrast, *normal derivation* works exactly as in BIP-032 with no change whatsoever (this is compatible for any hardened derivation method). As a result, the HD wallet is cryptographically indistinguishable from a standard BIP-032 HD wallet.[5]



[5]Note that the only difference is that the seed cannot be imported as is into a different wallet that follows the exact BIP-032 standard. This is because the key derivation method is different and so the hardened derivations from the seed will result in different keys. However, it is possible to export each hardened key directly, and all of the normally derived keys will still be valid.

In general, constructing HD wallets in MPC requires great care. Cryptographic methods need to be deployed to enforce that the parties use the correct input to the derivation and the correct output from the derivation. Otherwise, a malicious party can modify values in the key generation phase so that derived keys are no longer reconstructable from the backed-up secret. Our model of security in MPC is that a single corrupted party must not be able to carry out any attack. Given the fact that institutional wallets can contain large amounts of funds, it is critical that we have cryptographic certainty that the backup is valid and can be used to recover all asset keys, no matter what happens.

Backup and public verifiability: As we have discussed in the context of HD wallets, backup is a crucial element in any wallet. Every asset needs to be backed up to ensure that a lost or destroyed device doesn't result in any loss. Although this may not seem to be an especially hard problem, it is actually one of the more difficult features in building a wallet. This is because the potential loss can be huge and so a very high level of confidence in the backup must be reached. Furthermore, the standard approach to data backup of redundancy – storing the backup in many different places – cannot be naively deployed in this context due to the risk of theft.

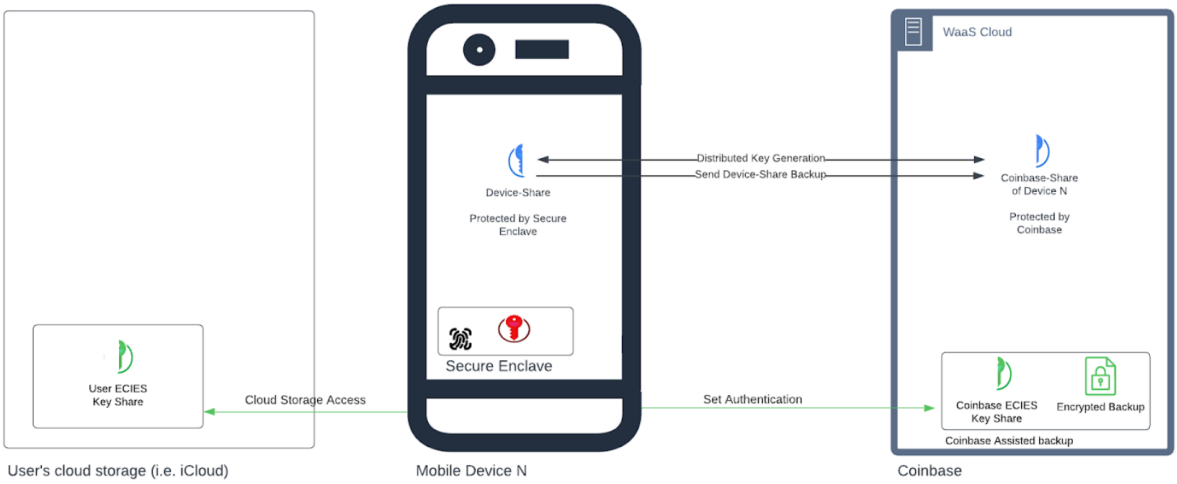
We provide two backup options, each with different properties. One is an easy-to-use backup that enables restore in collaboration with Coinbase (with Coinbase never having access to the user's key share), while the other is a "self-custody backup" that is held solely by the user, ensuring that the user fully controls their keys (i.e., Coinbase can never access the keys, while the user is able to singlehandedly restore from backup and obtain their keys, if they desire).

- *Coinbase-assisted backup:* This backup is used to easily restore the user's key shares to their mobile phone in the case that it is lost. In order to achieve this, when a wallet is generated, shares of the master key and VRF key (as described above) are generated in MPC. In addition, the parties generate an ECIES key in MPC, with each party holding a share.[6] Then, each party encrypts their shares under the shared ECIES key, achieving the property that neither Coinbase nor the user can decrypt the backup ciphertexts without the other's cooperation. All the ciphertexts are then stored by Coinbase in a secure and reliable way. Furthermore, Coinbase stores their share of the ECIES private key, and the user's share of the ECIES private key is either stored by the user in their own cloud storage (Google Drive or iCloud, etc.) or by the service provider deploying the application (as desired). We remark that the ECIES key for backup that is shared between the user and Coinbase can also be refreshed, as described above.

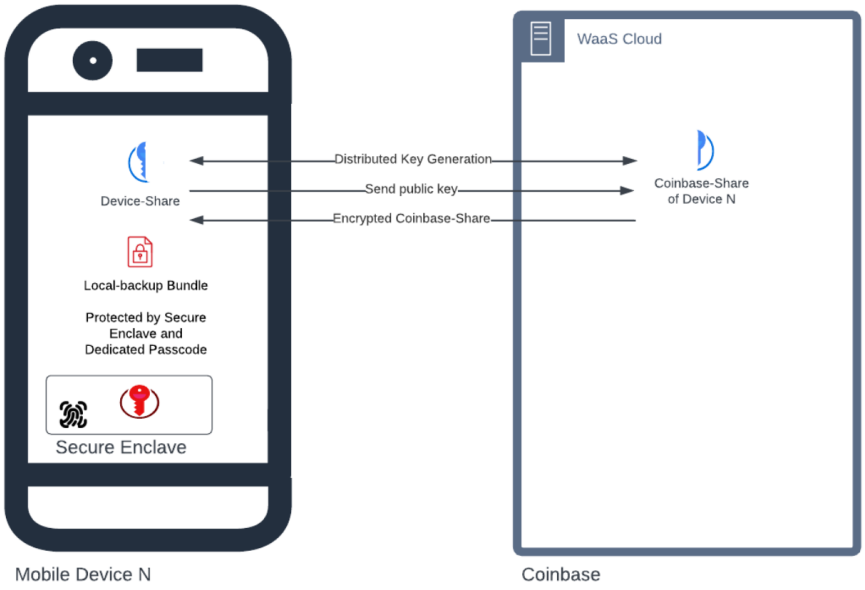
  In order to restore from backup, the user first authenticates (to the service provider, or potentially to the Coinbase server) in order to authorize the operation. After this, all that is required is for the user and Coinbase to run MPC decryption on the backup ciphertexts, with the result being that the original shares are restored at both Coinbase and the user.[7]

[6] Technically, the parties generate keys for a variant of ECIES, called TDH2, that is specifically tailored for MPC-decryption [3].

[7] Note that both shares (the user share and Coinbase's share) need to be restored, since the refresh mechanism makes the current Coinbase share incompatible with the original share encrypted by the user for backup.



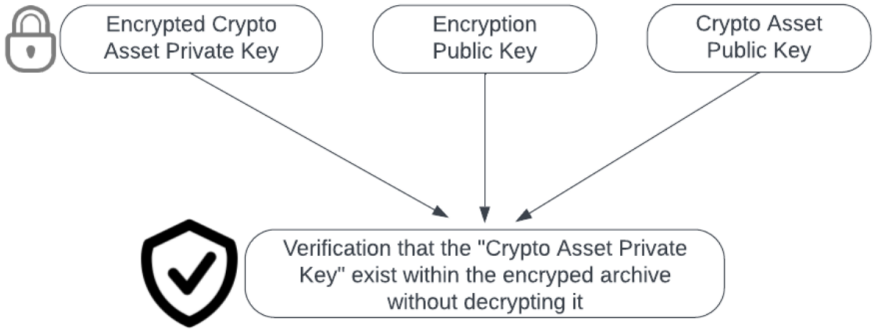
- *Self-custody backup:* In addition to the above backup, when the wallet is generated, both the shares are encrypted under a backup public keys that is owned *solely* by the user. Specifically, upon generating a wallet, the user provides a backup public key to the server, and both the Coinbase and user shares are encrypted with this public key and stored on the user's device. This public key can be in the mobile's enclave and protected with a biometric and/or passcode, and in the future it could be an external key (e.g., from a YubiKey). We stress that these ciphertexts (from Coinbase and from the user side) are stored for this backup *by the user*. This ensures that the user can obtain their keys independently of Coinbase (or even the service provider). Observe that although the user has full control over this backup, it is generated without the key ever being exposed. The security of this backup clearly relies on the security of the backup key; since this never needs to be used in ongoing operations, it can be strongly protected.



The Coinbase-assisted backup provides very high assurance against loss because Coinbase stores the ciphertexts (using robust DR techniques), and the user share is stored either in their cloud (with high resilience) or by the service provider. Regarding theft, the backup is strongly protected

since an attacker would need to breach both Coinbase's systems, obtaining the backup key and the backup ciphertexts, as well as the user in order to obtain anything. We believe that in the vast majority of cases, the Coinbase-assisted backup is what will be used. Indeed, the importance of the self-custody backup is to guarantee the user that they are not dependent on Coinbase to access their assets.

The above seems relatively simple, but there is an unaddressed threat in our above description. Specifically, the combination of the use of a cryptographically-enforced HD wallet and the above two backup mechanisms seems to provide all the guarantees needed. However, how do we know that at the time of wallet generation, the parties encrypt the correct values to backup? If one of the parties is corrupted (breached), then they may write garbage to backup. Since the backup is encrypted, this will go undetected. We will then use all of the cryptographic enforcement described regarding HD wallets, but they will all be relative to values that aren't actually backed up. One could of course decrypt the backups to check them, but this is extremely risky as it exposes sensitive key material. We solve this by using *publicly-verifiable encryption*. Informally speaking, a publicly-verifiable encryption of a secret share is an encryption together with a zero-knowledge proof that the encryption is valid. More concretely, in the key generation phase at wallet generation where we create the master key and VRF key, the parties generate shares (x_1, k_1) and (x_2, k_2), together with public keys $Q = (x_1 + x_2) \cdot G$ and $K = (k_1 + k_2) \cdot G$, and "public shares" $Q_i = x_i \cdot G$ and $K_i = k_i \cdot G$. Then, considering one specific case, when P_1 wants to generate a backup of x_1, it will encrypt x_1 under a given public key and provide a zero-knowledge proof that the resulting ciphertext encrypts a value that matches Q_1 (i.e., the value x_1 in the encryption is such that $x_1 \cdot G = Q_1$, as expected). This method is used for both the Coinbase-assisted backup and the self-custody backup; all of the encryptions generated are publicly verifiable. The beauty of this methodology is that all backups can be verified without ever accessing the private keys. This also opens the possibility of using an external backup service who can cryptographically verify that it received valid backups, without decrypting or exposing any private key material. This closes the potential threat, and ensures that the same keys backed up are the ones used to derive keys throughout, and the cryptographic enforcement methods implemented ensure that those derived keys are the ones used in all signing operations. We remark that we deploy publicly-verifiable encryption that works with *any* asymmetric encryption scheme. Thus, it can work with both ECIES and RSA, enabling the use of secure enclaves / TEEs in standard phones, standard YubiKeys, or any other device desired. The technical details on how our publicly-verifiable encryption works can be found in Appendix A.5.



Cloning: In some cases, users wish to (or even need to) have multiple copies of their wallet (on multiple mobiles, on their mobile and laptop, etc.). In order to achieve this, WaaS enables users

to generate clones of their wallet. This is carried out in the following way. The source device sends a copy of its share to the target device while the server creates a copy of its share. There are now two identical sharings of the same key: one sharing between the source device and the server, and another sharing between the target device and the server. The target device then runs a refresh procedure (as described above) that results in its sharing of the key being unique and independent of the original wallet. This completes the cloning procedure, and both wallets can work independently of each other. We note that the delivery of the share from the source device to target device is encrypted under a public-key generated on the target device, which is securely delivered (e.g., using an out-of-band method like the target device displaying a QR code of the public key, which the user scans into the source device).

How MPC overcomes the self custody impediment: The MPC solution described above provides an ongoing user experience that is comparable to that of a centralized exchange. In particular, operations are initiated via a user-friendly online interface, policy controls and anti-fraud verification can be implemented at both the user and Coinbase (reducing the possibility of key misuse), the key cannot be stolen from the user device, and the burden of backup is removed by having it both automatic and highly resilient. This means that the user can install and use a wallet easily and transparently, with all the "heavy lifting" typically associated with existing solutions being fully automated.

Furthermore, and importantly, due to the use of MPC, Coinbase has *no ability* to carry out any operation without the user's approval (even if all of the systems in Coinbase are corrupted). In addition, if the user wishes to obtain all of the keys to their assets without Coinbase's involvement, then they are able to do that. As a result, the wallet has the benefit of traditional self-custodial wallets, without the operational or security disadvantages of those solutions.

3 MPC Implementation at Coinbase

Conservative protocol choices: The strength of MPC rests on strong scientific foundations which have been researched extensively in academia going back as far as the 1980's. As a result, we have a very strong understanding of how MPC protocols work and how they can be proven secure. As with all cryptography, however, MPC protocols can be designed conservatively or aggressively. The conservative approach utilizes standard and well-established cryptographic primitives and values simplicity over speed, while the aggressive approach pushes the edge in order to achieve new and exciting scientific breakthroughs. There is no doubt that more aggressive design helps to spur innovation and is needed when other solutions are unavailable. Yet, in many cases, aggressive design performs only slightly better than more conservative approaches. Saving a few fractions of a second to sign on a transaction is not always of great importance, especially when doing so increases the possibility of mistakes. In many cases, the additional computation and time is not significant, especially in blockchain s which are not real-time.

At Coinbase, we place a strong emphasis on trust and security and often prefer a more conservative approach. This means that simplicity wins over complexity (wherever possible), and that standard well-established cryptographic hardness assumptions are preferred over newer ones. Fortunately, MPC has advanced to a stage where user and product needs can be met while taking the conservative approach. We are therefore able to provide state-of-the-art innovative products without sacrificing on the trust and security that has to be at the foundation of any key management solution.

MPC development process: MPC development requires great expertise in order to avoid catastrophic errors. At Coinbase we have a very strong team of cryptographers (researchers) with decades of expertise in MPC, and we have adopted a very rigorous process to MPC development, which we describe briefly here.

Stage 1 – theoretical protocol: The first stage of development is to choose each theoretical MPC protocol. This may be a published paper or a new protocol developed by the cryptography team. In both cases, the result is fully audited and verified by the cryptography team (a published paper is not trusted, and the proof is verified; a new protocol is verified by a different independent member of the team). A theoretical justification document is written, describing why this protocol is chosen and justifying any deviations from a published paper.

Stage 2 – specification: Cryptography papers often omit crucial details in protocol descriptions. After the protocol has been chosen, an exact specification is written, in order to ensure that all details are present, and there will be no errors in implementation. As with the theoretical protocol, the specification is independently audited by a different member of the team, ensuring that it matches the theoretical protocol.

Stage 3 – code review: After the protocol has been implemented by the cryptography engineering team, the cryptography team carries out code review to ensure that it matches the specification exactly.

Stage 4 – independent review: After all stages have been audited by the cryptography team, independent reviews are carried out by independent security teams inside and outside of Coinbase. This review includes a cryptographic review of the theoretical protocols and specifications, as well as an independent code review, penetration testing, and more.

The above process is very rigorous, and shows the care taken in this development process.

Additional security measures: Although the security of our solution is primarily based on the MPC protocols, we also employ a defense in depth approach. That means that secure enclaves are utilized on mobile devices to encrypt shares and biometrics are used to unlock them, YubiKeys are used for authentication and backup encryption, shares are protected in memory, and more. In general, we take a zero-trust approach trusting no single device, and a defense in depth approach that aims to make it hard to obtain anything even if a device is breached.[8]

4 Summary

Coinbase's Wallet as a Service based on MPC brings major innovation to the market. It is sometimes thought that in order to construct a secure MPC wallet, all one needs to do is to implement a secure signing protocol for ECDSA and EdDSA. However, as we have shown, there are many other elements required. In particular, the method of backup used is crucial for achieving the user-friendly experience without sacrificing self-custody, and this requires many cryptographic tools (HD derivation in MPC, input and output enforcement, publicly-verifiable backup, and more). The result is a solution that truly overcomes the usability difficulties associated with self-custody, making self-custody a realistic option for Web3 and other applications deploying WaaS, and unlocking the potential of truly bringing wallets to the masses.

[8] MPC guarantees that breaching a single device will not enable an attacker to do anything. However, that doesn't mean that we should rely only on it being hard to breach both Coinbase and the user. Rather, we make it hard at each location as well.

A Cryptographic Details

This appendix contains more technical details about the protocols and methods that we use, and is intended for the cryptographically inclined. We remark that all of our protocols have been proven secure in the presence of malicious adversaries under concurrent composition. We present a high-level explanation here in order to provide an idea as to how the protocols work, and refer to full papers for details where relevant.

A.1 Two-Party Elliptic-Curve Key Generation

In this section, we show how it is possible to generate a public-key Q for elliptic-curve signing (ECDSA or EdDSA) without either party knowing the private key. Rather, the private key is *shared* securely between the parties, without it being revealed. Let x be an elliptic-curve private key, and let Q be the associated public key (this means that $Q = x \cdot G$, where G is the generator of the elliptic-curve group). Our aim is to have party P_1 obtain x_1 and party P_2 obtain x_2, such that $x = x_1 + x_2$ (actually, the addition here is modulo the group order, but we will ignore that detail in this explanation). A first attempt at achieving this is simply for P_1 to choose a random x_1 and compute $Q_1 = x_1 \cdot G$, and for P_2 to choose a random x_2 and compute $Q_2 = x_2 \cdot G$. Then P_1 and P_2 send each other Q_1 and Q_2, respectively, and each finally defines $Q = Q_1 + Q_2$. By the basic properties of elliptic-curve groups, we have that $Q = Q_1 + Q_2 = x_1 \cdot G + x_2 \cdot G = (x_1 + x_2) \cdot G = x \cdot G$, as required. Thus the parties have succeeded in generating Q without either of them knowing x, but while they have *additive shares* of x in the form of x_1 and x_2.

The only problem with the above solution is that a corrupted P_2 could wait to receive Q_1 from P_1 before choosing its Q_2. This enables P_2 to *bias* the value of Q in some way, and even to choose Q_2 so that $Q_1 + Q_2$ equals a Q for which it knows the full private key (it can do this by choosing a random x and setting $Q = x \cdot G$, and then setting $Q_2 = Q - Q_1$). In order to prevent both of these attacks, we have P_1 send a *commitment* to Q_1, which is a cryptographic "envelope" that binds P_1 to the value of Q_1 without yet revealing it to P_2. Then, after P_2 sends Q_2, party P_1 can "open" the commitment by revealing Q_1. For example, P_1 can send a cryptographic hash of Q_1 to P_2 as the commitment. (By the collision resistance of the hash, it is not possible for P_1 to open the hash to any value other than Q_1. Furthermore, the hash hides the value of Q_1 to P_2, so P_2 is unable to choose Q_2 based on the value of Q_1.[9]) With this method, we achieve that Q_1 and Q_2 are chosen independently, and so as long as one party is honest and chooses it randomly, the result is random. Furthermore, neither party knows x, and the private key is only additively shared.

A.2 Two-Party ECDSA Signing

The ECDSA signing function is defined as follows. Let x be the private key and $Q = x \cdot G$ the public key. Then, a signature on m is defined by the following procedure:

1. Choose a random nonce k and compute $R = k \cdot G$; let r be the "x-coordinate" of the elliptic-curve point R
2. Compute $s = k^{-1} \cdot (H(m) + r \cdot x) \bmod q$
3. Output (r, s)

[9] Note that we actually include additional randomness in order to define the commitment.

We won't delve into why ECDSA is defined in this way or why it is secure. Rather, we will provide a high-level idea behind the protocol of [5] that we use for computing it without any single party knowing the private-key x. We remark that with ECDSA, if you are given the nonce k and the signature (r, s), then it is possible to extract the private-key x. Thus, our protocol needs to also ensure that neither party knows k.

Background – additively homomorphic encryption: Our protocol uses a tool called *additively homomorphic encryption*. This is a type of encryption with the property that if you are given two ciphertexts $c_1 = \mathsf{Enc}_{pk}(x)$ and $c_2 = \mathsf{Enc}_{pk}(y)$ then you can efficiently compute $c_3 = \mathsf{Enc}_{pk}(x+y)$. That is, you can add encrypted values without knowing anything about what they are (and in particular, without being able to decrypt). This type of encryption also supports *multiplication by a scalar* and so given c_1 as above and a plaintext value z, it is possible to efficiently compute $c_4 = \mathsf{Enc}_{pk}(z \cdot x)$, again without knowing anything about x.[10] The most common additively homomorphic encryption scheme is Paillier [8], and it is based on a similar hard problem to that of RSA.

Two-party computation of ECDSA: In two-party key generation for ECDSA, the parties choose random x_1 and x_2, respectively, and exchange $Q_1 = x_1 \cdot G$ and $Q_2 = x_2 \cdot G$ with each other (as described in Section A.1, this exchange takes place by first committing to values and then opening so that neither sees the other's value before presenting their own; this is needed to prevent one party biasing the key value). Next, one of the parties – let's call them P_1 – generates a Paillier key-pair (pk, sk) and sends $c_{\mathsf{key}} = \mathsf{Enc}_{pk}(x_1)$ to the other party P_2. In addition, P_1 provides a zero-knowledge proof that it generated the Paillier key correctly and that it encrypted the correct x_1 value (matching the Q_1 exchanged above). This completes the key generation phase.

In order to sign on a message m, the parties first need to generate k and $R = k \cdot G$ so that neither knows k. They do this by each party P_i choosing a random k_i (for $i = 1, 2$ respectively) and exchanging $R_1 = k_1 \cdot G$ and $R_2 = k_2 \cdot G$, and then setting $k = k_1 \cdot k_2$, implying that $R = k_1 \cdot R_2 = k_2 \cdot R_1$ (like in Diffie-Hellman key exchange). Once they both know R (but importantly neither know k), they can derive the value r. Next, P_2 can compute something that we call a "partial signature". In particular, using the homomorphic properties of Paillier encryption, P_2 takes $c_{\mathsf{key}} = \mathsf{Enc}_{pk}(x_1)$ and computes the following series (all inside the encryption which is possible since all operations are just addition and multiplication by a scalar):

1. $\mathsf{Enc}_{pk}(x_1) \to \mathsf{Enc}_{pk}(x)$ (by adding x_2)
2. $\mathsf{Enc}_{pk}(x) \to \mathsf{Enc}_{pk}(r \cdot x)$ (by multiplying r)
3. $\mathsf{Enc}_{pk}(r \cdot x) \to \mathsf{Enc}_{pk}(H(m) + r \cdot x)$ (by adding $H(m)$)
4. $\mathsf{Enc}_{pk}(H(m) + r \cdot x) \to \mathsf{Enc}_{pk}(k_2^{-1} \cdot (H(m) + r \cdot x))$ (by multiplying k_2^{-1})

P_2 then sends this ciphertext to P_1, who decrypts, multiplies it by k_1^{-1} to get s, and verifies that (r, s) is a valid signature. Observe that $k_2^{-1} \cdot (H(m) + r \cdot x)$ is "almost the s part of the signature". Specifically, the s part of the ECDSA signature is $k^{-1} \cdot (H(m) + r \cdot x)$ and so the only difference is the need to multiply in k_1^{-1} as well.

[10] This should not be confused with *fully homomorphic encryption* which enables you to also multiply ciphertexts; i.e., to compute $\mathsf{Enc}_{pk}(x \cdot y)$ given $c_1 = \mathsf{Enc}_{pk}(x)$ and $c_2 = \mathsf{Enc}_{pk}(y)$. Fully homomorphic encryption is much harder to construct than just additively homomorphic encryption.

Regarding security, the important thing to note is that if P_2 is honest then P_1 receives some $s' = k_2^{-1} \cdot (H(m) + r \cdot x)$ which reveals nothing more than the signature itself. Indeed, given a full signature (r, s), party P_1 can generate this s' by simply computing $k_1 \cdot s$.[11] As a result, this s' reveals nothing more to P_1 than the final signature itself (which is supposed to revealed).[12] Furthermore, if P_2 is corrupt, then the only thing it can do is provide an incorrect encryption to P_1. If this incorrect encryption is crafted carefully, it may leak some information about the private key. However, since P_1 first verifies the signature and only outputs it if it is correct, P_2 does not gain anything by sending an incorrect ciphertext.

For a complete description and full proof of security, see [5].

A.3 Two-Party Schnorr/EdDSA Signing

EdDSA is a variant of the Schnorr signing algorithm, defined over a specific elliptic curve (Ed25519) and including a specific (deterministic) way of deriving the random nonce used in the signing. Our two-party protocol for Schnorr works over every curve, and therefore also over Ed25519. We note, however, that we do not support the EdDSA way of deriving randomness. We stress that signatures generated in this way are *indistinguishable* from EdDSA signatures, with the exception that they are *not* deterministic. As a result, two signatures on the same message will be the same with standard EdDSA, but will be different when using our protocol. This is inconsequential for most blockchain applications; in particular, standard EdDSA verification works as usual and so all transaction signing is unchanged.[13]

The Schnorr signing algorithm is defined as follows:

1. Choose a random k and compute $R = k \cdot G$
2. Compute $e = H(m \| R)$
3. Compute $s = k + e \cdot x \bmod q$
4. Output (e, s)

We stress that there are different Schnorr variants and EdDSA is a very specific one. This influences the exact formatting of the hash computation, the exact definition of s (e.g., of either $k + e \cdot x$ or $k - e \cdot x$), and the exact output format. However, these make no difference to the protocol and our implementation is fully compatible with standard EdDSA in this respect.

As with ECDSA, if you are given the nonce k and the signature (e, s), then it is possible to extract the private-key x. Thus, our protocol needs to also ensure that neither party knows k. We

[11] Note that $k_1 \cdot s = k_1 \cdot k^{-1} \cdot (H(m) + r \cdot x) = k_1 \cdot k_1^{-1} \cdot k_2^{-1} \cdot (H(m) + r \cdot x) = k_2^{-1} \cdot (H(m) + r \cdot x) = s'$.

[12] We stress that this description is only partial and that P_2 also has to add random noise which is a multiple of the order of the curve group q. This is due to the fact that the Paillier operations are essentially over the integers and not in the same group as the ECDSA operations. We ignore these details here, but stress that they are crucial for security.

[13] We note one exception which is that some dApps that use non-standard methods derive a key from "standard wallets" by signing on a fixed message and using the hash of the result as a key. This method is technically sound since signatures must have high entropy, or they could be forged. However, the method is quite problematic since the security that user's receive is not necessarily as expected. For example, a user using a hardware wallet expects the key to never leave the hardware device, and a user using an MPC wallet expects the key to never be whole on any device. However in both of these cases, the signature is actually used as a key, and is thus exposed on the user's device. We stress that this method is only compatible with *deterministic* signatures, since the same key must be obtained every time.

use the protocol of [7] and provide a high-level description here. Before proceeding, note that unlike ECDSA, the Schnorr equation for s is *linear* (in particular, it does not contain multiplication by k^{-1}). This makes it much more amenable to MPC, making the protocol much simpler.

Two-party computation of Schnorr: Similarly to ECDSA, the parties run the two-party key generation described in Section A.1, resulting in the parties holding random x_1 and x_2, and the public key $Q = (x_1 + x_2) \cdot G$. This is all that is needed for the key generation in this case.

In order to sign on a message m, we begin in a similar way as for ECDSA. Specifically, the parties each choose random k_1, k_2 and exchange $R_1 = k_1 \cdot G$ and $R_2 = k_2 \cdot G$. Then, R is defined to be the sum $R_1 + R_2$, and each party can locally compute $e = H(m\|R)$. Now, all that is required is for P_1 to compute $s_1 = k_1 + e \cdot x_1$ and for P_2 to compute $s_2 = k_2 + e \cdot x_2$. Finally, by taking $s = s_1 + s_2$ we have the correct result (since $s_1 + s_2 = k_1 + e \cdot x_1 + k_2 + e \cdot x_2 = (k_1 + k_2) + e \cdot (x_1 + x_2) = k + e \cdot x$, as required).

We stress that there are subtleties in implementing this simple idea correctly, but this description covers the main ideas behind the protocol. A full description of the protocol and proof of security can be found in [7].

EdDSA key generation: One cryptographic challenge that arises with EdDSA is that the key itself is hashed (using SHA-512), and the result is separated into two parts: one part is used as the secret key x in Schnorr signing (over the Edwards curve Ed25519) and the other is used as the key for a PRF for deriving the randomness in signing (in order to achieve deterministic signing). Cryptographically, it is possible to generate the EdDSA key via SHA-512 in order to achieve full compatibility with EdDSA keys (using a protocol for securely computing via a Boolean circuit representation of SHA-512). However, for the sake of simplicity – as a security goal – we made the decision to currently generate x directly in MPC in the way described above. This means that EdDSA keys generated in the wallet can be used for signing, but for full compatibility to other wallets, one would need to rotate keys (or have the wallet accept x directly).

A.4 HD Wallets in MPC using VRFs

As described above, we use a verifiable random function (VRF) in order to cryptographically enforce correct key derivation. The VRF that we use is the one described in [9], as follows. Let $\mathbb{G}$ be an elliptic-curve group of order q with generator G in which the decisional Diffie-Hellman (DDH) assumption is assumed to be hard,[14] and let H be a hash function (random oracle) mapping strings into (random) elliptic curve points in the group. Then, the function $F_k(m) = k \cdot H(m)$ is a pseudorandom function (technically, this is a PRF on outputs over the elliptic-curve group). In order to see why, let $K = k \cdot G$ and observe that $(G, H(m), K, F_k(m)) = (G, H(m), k \cdot G, k \cdot H(m))$ is a Diffie-Hellman tuple. Therefore, under the DDH assumption, $F_k(m)$ looks like a completely random group element, even given $H(m)$ and K. Next observe that this can be made easily into a VRF by defining $K = k \cdot G$ to be the function "public key" and providing a zero-knowledge proof that $F_k(m)$ is correct. This zero-knowledge proof is extremely efficient since it is just a proof that $(G, H(m), K, F_k(m))$ is indeed a Diffie-Hellman tuple, for which there is a well-known and simple proof. Furthermore, an MPC version of this VRF is easily achieved by sharing the key k among the parties by them holding random k_1 and k_2 under the constraint that $k = k_1 + k_2$. Then, in order to

[14] The DDH assumption states that for a random H and r, the tuple $(G, H, r \cdot G, r \cdot H)$ is indistinguishable from a completely random tuple $(G, H, r \cdot G, s \cdot H)$. This is the assumption used to prove the security of Diffie-Hellman key exchange.

compute $F_k(m)$, party P_1 can compute $Y_1 = k_1 \cdot H(m)$ and party P_2 can compute $Y_2 = k_2 \cdot H(m)$, and the result is obtained by simply computing $F_k(m) = Y_1 + Y_2$. This is very similar to the RSA computation that we described above. Furthermore, by each party sending a zero-knowledge proof that their local computation is correct, the derivation can be verified and enforced, as required.

The natural way to use this for key derivation would be to define the derived key to be $F_k(\mathsf{path})$, where path is as defined in the BIP-032 standard. However, if $F_k(\mathsf{path})$ is the key then it cannot be revealed to either party. Furthermore, $F_k(\mathsf{path}) = k \cdot H(\mathsf{path})$ is an Elliptic-curve point and it needs to be hashed to a scalar value as a private key, which would break its clean algebraic structure. This makes it a challenge to build a highly-efficient MPC protocol that takes as input the shares $k_1 \cdot H(m)$ and $k_2 \cdot H(m)$ and outputs shares of a key for ECDSA or EdDSA/Schnorr (including all elements of input and output enforcement). We therefore take a different route and have the parties run standard MPC key generation to generate a "master key" x that is shared as x_1, x_2 with $x_1 + x_2 = x$, and with associated public-key $Q = x \cdot G$. In addition, they run MPC key generation to generation k_1, k_2 for the VRF with public-key $K = (k_1 + k_2) \cdot G = k \cdot G$. Then, the key in the HD tree defined by path is defined to be $x + F_k(\mathsf{path})$. In more detail, the parties use the distributed VRF computation in order for them both to obtain $\Delta = F_k(\mathsf{path})$; we stress that zero-knowledge proofs are used to prevent anyone cheating, and so we know that this is the correct value. Then, party P_1 defines its private key for this derivation to be $x'_1 = x_1 + \Delta$, and party P_2 leaves its private key for this derivation unchanged as $x'_2 = x_2$. Furthermore, both parties set the public key for this derivation to be $Q' = Q + \Delta \cdot G$. This is correct since $(x_1 + \Delta) + x_2 = (x_1 + x_2) + \Delta = x + \Delta$ and so $((x_1 + \Delta) + x_2) \cdot G = (x + \Delta) \cdot G = x \cdot G + \Delta \cdot G = Q + \Delta \cdot G$. The fact that both parties can compute this public key means that it cannot be changed (since both parties locally obtain the correct value). This therefore achieves our input and output enforcement requirements.

Regarding the security of this as a derivation method, observe that since the derivation function $F_k(\cdot)$ is a pseudorandom function, the derived private key $x' = x + \Delta$ and public key $Q' = x' \cdot G$ are indistinguishable from a random independently-generated key-pair (by the randomness of Δ). In particular, this means that the HD tree generated is indeed indistinguishable from a standard BIP-032 tree (which in turn is indistinguishable from a tree where each hardened-derived key is actually just chosen at random). It is important to note, however, that as far as the MPC participants themselves are concerned (in contrast to everyone else), the derived keys do not look independently random. This is because both parties know the "differences" between the keys. In particular, for a pair of keys $Q_i = Q + \Delta_i \cdot G$ and $Q_j = Q + \Delta_j \cdot G$, the MPC parties know that the difference between the private keys associated with Q_i and Q_j is exactly $\Delta_i + \Delta_j$, something which is not known for independent random keys. Nevertheless, this is exactly the property of normal derivation used in BIP-032, where the differences between all normally-derived keys (from a given hardened key) is known to all. Thus the security of signatures generated from keys derived in this way in the case that one of the parties is corrupted is exactly like that of normal derivation, which is formally justified in [2, 4] and in wide use.

We conclude by noting that with this method it isn't possible to delegate some keys externally (because if the Δ values are learned then it is possible to obtain other non-delegated keys). This is not an issue since delegation is not a feature offered in this type of wallet in any case. Furthermore, observe that normal derivation doesn't provide unlinkability, meaning that it is possible to know that different public keys belong to the same person. However, as described above, as far as any external observer is concerned, different derived keys look completely independent. The difference between different keys is known only to the MPC parties themselves and they can indeed know

that different public keys belong to the same person. However, the MPC parties themselves hold the wallet and so anyway know all of the public keys in the wallet. Thus this makes no difference whatsoever.

A.5 Publicly-Verifiable Backup

Publicly-verifiable encryption for elliptic curve private keys. The aim of encryption is to hide the plaintext. As such, it is infeasible to verify any property of an encrypted value without decrypting. In some cases, however, we want to verify some property of the plaintext without revealing anything else. Fortunately, zero-knowledge proofs provide the answer – these are cryptographic proofs that assert that the property holds without revealing anything beyond that about the plaintext.

A specific property that we are interested in for elliptic-curve applications is the following one. We are given a public key Q (with $Q = x \cdot G$) and we want to verify that a given ciphertext encrypts the associated private key x. There are generic zero-knowledge tools that enable this to be carried out, but they are very inefficient. It is also possible to build extremely efficient solutions for additively-homomorphic encryption schemes like Paillier, but this would severely limit applicability to where Paillier is available (and acceptable). In this section we describe how we achieve publicly-verifiable encryption of elliptic-curve secret keys using any asymmetric encryption scheme.

Constructing publicly-verifiable encryption. Let pk be the public-key for some asymmetric encryption scheme. Instead of encrypting x directly by computing $c = \mathsf{Enc}_{pk}(x)$, the encryptor chooses a random r and separately encrypts both r and $x + r$, obtaining $c_1 = \mathsf{Enc}_{pk}(r)$ and $c_2 = \mathsf{Enc}_{pk}(x+r)$. In addition, they provide the elliptic-curve point $R = r \cdot G$ together with the public-key Q and the two ciphertexts c_1 and c_2. Clearly, if we can be sure that c_1 encrypts r and c_2 encrypts $x + r$, then together we can decrypt and obtain the correct x (just decrypt both, and subtract r from $x + r$). This is therefore sufficient. But how can we verify that indeed c_1 and c_2 encrypt what they are supposed to?

In order to verify this, we obtain the values (Q, R, c_1, c_2) from the encryptor and then challenge them to open one of the ciphertexts. If they open the first, then we check that indeed c_1 encrypts a value r such that $r \cdot G = R$. If they open the second, then we check that indeed c_2 encrypts a value s such that $s \cdot G = Q + R$ (this is true if the encryptor is honest since $Q + R = x \cdot G + r \cdot G = (x+r) \cdot G = s \cdot G$). Now, if the encryptor prepared both ciphertexts correctly, then we are guaranteed to be able to decrypt and obtain the correct x as required. In contrast, if the encryptor cheats and provides an incorrect value in at least one of the ciphertexts, then we will catch them cheating if we ask them to open that incorrect value. We stress that the encryptor cannot provide ciphertexts that don't reveal x and still pass both challenges. This is because by definition, if c_1 encrypts a value r so that $r \cdot G = R$ and c_2 encrypts a value s so that $s \cdot G = Q + R$, then $s - r = x$ as required. (This can be seen from the fact that $s \cdot G - R = Q$ and so $(s - r) \cdot G = x \cdot G$.) Furthermore, and crucially for what we are trying to achieve, opening only one of the ciphertexts reveals nothing – all we see is a random r or a random $s = x + r$ (the first value clearly reveals nothing, but the same is true of the second value since the random r completely hides x in the sum s).

So far, the above seems to achieve what we want. However, clearly, the encryptor can provide a correct c_1 but an incorrect c_2 (or vice versa) and get away with it with probability $1/2$. This is because with probability $1/2$ we will ask them to open the correct one instead of the incorrect one.

Fortunately, we can reduce the chances of the encryptor cheating by repeating this process many times. Specifically, if we run this game 128 times, then the encryptor will succeed in cheating with probability only 2^{-128} which is so close to zero as to not matter (for context, the number of seconds since the big bang is less than 2^{59} and the number of microseconds less than 2^{79}).

The only remaining problem is that playing the above game requires interaction! The encryptor provides the ciphertexts and values, and we challenge them to open one ciphertext in each pair. However, this makes it not "publicly verifiable" in the sense that a third party cannot verify the results of the game at a later time. In particular, they will have no idea if the encryptor and "challenger" played the game correctly or colluded to cheat. This is overcome in the following way. The encryptor generates all of the ciphertexts and values and then applies a cryptographic hash function to them all and gets back 128 bits which say which ciphertext to open (e.g., if the i'th bit equals zero then open c_1 and otherwise open c_2). In this way, the encryptor can generate the entire proof by themselves, and later anyone can verify. This works (and doesn't enable the encryptor to cheat) since any change by the encryptor to any value results in a completely different and unpredictable hash result, and so it's not feasible for it to cheat in all pairs.

Publicly-verifiable backup. The above publicly-verifiable encryption is exactly what is needed to achieve publicly-verifiable backup. Specifically, given a public-key Q and an encryption generated as above, it is possible to verify that the encryption is indeed of the associated private key x without accessing the private decryption key. Furthermore, in the context of MPC, the result of the key generation is that the parties hold private key shares x_1, x_2 such that $x_1 + x_2 = x$, and also hold the public key $Q = x \cdot G$ as well as $Q_1 = x_1 \cdot G$ and $Q_2 = x_2 \cdot G$. As such, it is possible for party P_1 to generate a publicly-verifiable encryption of x_1 with public-key Q_1, and for party P_2 to generate a publicly-verifiable encryption of x_2 with public-key Q_2. Then, all that is needed to verify that the backup is valid is to verify that $Q_1 + Q_2 = Q$ is the correct public key, and that the encryptions are valid with respect to Q_1 and Q_2. We stress that this verification can be carried out by *anyone*, and no knowledge of the backup decryption key is needed. This means that the backup decryption key can be kept completely offline, or even belong to a different entity than the one verifying and storing the backup.

Extension to quorum backups. The above description shows how it is possible to backup a secret in a publicly verifiable manner. However, encrypting the entire key in this way yields to a potential risk in the owner of the backup decryption key. This can be solved in two ways. First, since signing keys are generated in MPC, each share is separately backed up (using a separate publicly-verifiable backup) and these can be encrypted under different backup keys. Second, it is possible to encrypt each share so that only a quorum of backup entities can decrypt. This is easily achieved by first secret sharing the share (e.g., with Shamir secret sharing) and then encrypting each Shamir subshare with the appropriate decryption key. More concretely, assume that we have five backup entities, and we wish to have three needed in order to restore the backup. Each backup entity has its own public key that should be used in order to encrypt their backup material. Then each party can secret share its share into five shares, so that any three suffice to reconstruct the original share, and then encrypt each share with one of the backup keys. Accordingly, any three backup entities who wish to restore can decrypt their share, and send it to the party who needs to restore the original share. Given the three Shamir shares the party is able to reconstruct the original share, concluding the restore process.

References

[1] Mihir Bellare, Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS 1993*, pages 62–73, 1993.

[2] Poulami Das, Andreas Erwig, Sebastian Faust, Julian Loss and Siavash Riahi. The Exact Security of BIP32 Wallets. In *ACM CCS 2021*, pages 1020–1042, 2021.

[3] V. Shoup and R. Gennaro. Securing Threshold Cryptosystems against Chosen Ciphertext Attack. In *Journal of Cryptology*, 15(2):75–96, 2002.

[4] Jens Groth and Victor Shoup. On the Security of ECDSA with Additive Key Derivation and Presignatures. In *EUROCRYPT 2022*, pages 365–396, 2022.

[5] Yehuda Lindell. Fast Secure Two-Party ECDSA Signing. In *CRYPTO 2017* and the *Journal of Cryptology*, 34(4):44, 2021.

[6] Yehuda Lindell. Secure Multiparty Computation (MPC). *Communications of the ACM* (CACM), 64(1):86–96, 2021.

[7] Yehuda Lindell. Simple Three-Round Multiparty Schnorr Signing with Full Simulatability. In *Cryptology ePrint Archive, Paper 2022/374*, 2022.

[8] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT 1999*, pages 223–238, 1999.

[9] Dimitrios Papadopoulos, Duane Wessels, Shumon Huque, Moni Naor, Jan Vcelak and Leonid Reyzin and Sharon Goldberg. Making NSEC5 Practical for DNSSEC. In *Cryptology ePrint Archive, Paper 2017/099*, 2017.

[10] Adi Shamir. How to Share a Secret. In *Communications of the ACM*, 22(11):612–613, 1979.