Improving Security for Users of Decentralized Exchanges Through Multiparty Computation

Robert Annessi* and Ethan Fast†
Nash Exchange
Email: *robert@nash.io, †ethan@nash.io

Abstract—**Decentralized cryptocurrency exchanges offer compelling security benefits over centralized exchanges: users control their funds and avoid the risk of an exchange hack or malicious operator. However, because user assets are fully accessible by a secret key, decentralized exchanges pose significant internal security risks for trading firms and automated trading systems, where a compromised system can result in total loss of funds. Centralized exchanges mitigate this risk through API key based security policies that allow professional users to give individual traders or automated systems specific and customizable access rights such as trading or withdrawal limits. Such policies, however, are not compatible with decentralized exchanges, where all exchange operations require a signature generated by the owner's secret key. This paper introduces a protocol based upon multiparty computation that allows for the creation of API keys and security policies that can be applied to any existing decentralized exchange. Our protocol works with both ECDSA and EdDSA signature schemes and prioritizes efficient computation and communication. We have deployed this protocol on Nash exchange, as well as around several Ethereum-based automated market maker smart contracts, where it secures the trading accounts and wallets of thousands of users.**

I. Introduction

Centralized cryptocurrency exchanges hold custody of user funds and promise access to those funds when requested. As these exchanges often own large amounts of cryptocurrency funds in aggregate across their users, they are very attractive targets for criminals. Hundreds of millions of dollars of cryptocurrency have been lost over the years through hacks or malicious operators. These security problems are a major driver behind the success of decentralized[1] exchanges, where users hold their own secret keys.

Decentralized exchanges must overcome their own set of security challenges, however. In particular, automated trading algorithms deployed on decentralized exchanges must be trusted to hold and secure a secret key which fully controls user funds. Such trading algorithms are often hosted on cloud infrastructure and operate on shared accounts within trading firms, posing social and technical security risks. This is a significant limitation for professional users such as market makers. These users cannot provide restricted access to their account's capabilities such that their trading software may only trade; their software could as well withdraw funds, which significantly increases downside risk in the event their software or systems are compromised.

[1]In this paper, we use the established term *decentralized*, but we could equally use the terms *non-custodial* or *self-custodial*.

Centralized exchanges overcome such limitations by offering security policies, usually based on API keys, that include limiting the capabilities of trading algorithms to withdraw funds or allowing accounts to trade funds only within certain limits. This functionality is straightforward in centralized settings because the exchange can simply create random identifiers and passwords to enforce security policies via hosted software. While these policies are only as strong as the security of the centralized exchange itself, they do reduce the downstream damage if a user's automated trading infrastructure is compromised. Unfortunately, such security policies do not exist on most decentralized cryptocurrency exchanges today due to the fact that all operations are governed by digital signatures produced via a single secret key.

In this paper, we introduce a novel protocol that allows for the creation of API key based security policies for any decentralized cryptocurrency exchange. The protocol we describe is currently deployed on Nash [1], but can be applied to any smart contract based exchange protocol. In fact, the Nash mobile wallet also applies this protocol to Uniswap [2] and 1inch [3], allowing users to trade over these contracts without ever accessing their full secret key. The protocol we introduce has several important properties:

- Policies can be applied and extended to the signature schemes used by the most popular blockchains: in particular ECDSA and EdDSA and the most common elliptic curves on which they are used.
- There are no limitations on the kinds of security policies which can be enforced. Exchanges can create policies for withdrawal limits, trading limits, delayed withdrawals, or, for example, other more esoteric functionality such as geolocation or biometric information.
- Users fully control their assets and the secret keys that govern them. Secret keys are never accessible by any other system or party.
- Signatures produced via the protocol presented in this paper are efficient in computation, rounds of communication, and bandwidth; this is important to professional traders who require low latency for fast order execution and may trade many times per second.

While trading applications are the primary focus of this paper, the protocol we describe can easily be extended to enhance user wallet security as well.

II. Background

A. Digital Signature Schemes

A digital signature scheme consist of three algorithms [4]: a key generation algorithm, a signing algorithm, and a verification algorithm. The key generation algorithm generates a secret signing key and a public verification key. The signing algorithm is used to generate a signature for a particular message using the secret key. The verification algorithm is used with the public key in order to verify the signature for a particular message (and therefore confirm its authenticity).

B. ECDSA

An ECDSA signature consists of a tuple of integers (r, s) that is computed as follows [5]:

1) A cryptographically secure nonce k is generated with $0 < k < n$, with n the order of the elliptic curve. The implementer must ensure that the nonce k cannot be guessed and is not reused for any two messages. Otherwise, anyone who can guess the nonce used in one signature or who observes two signatures where the identical nonces were used twice can derive the secret key. Since there have been a quite a number of insecure implementations, a standard was created on how to generate cryptographically secure nonces deterministically, which is now used in most implementations [6].
2) A point on the elliptic curve is computed as $(x, y) = k \cdot G$, with G being the generator point of the curve. The x coordinate of that point is used as the first part of the signature, i.e., $r = x$.
3) The second part of the signature s is computed from the secret key d, the hash of the message z, the first part of the signature r, and the nonce k as follows: $s \equiv k^{-1} \cdot (z + r \cdot d) \mod n$.

C. EdDSA

Signature generation in EdDSA works similar to ECDSA. An EdDSA signature also consists of a tuple of integers (r, s), but computation differs slightly:

1) First, the secret key is hashed. The first half of the hash is used as signing key a and the second half is used in nonce generation.
2) A cryptographically secure nonce is also required in EdDSA, but generating it is not left to the implementer. Instead, the nonce k is computed by hashing the concatenation of the nonce-part generated in the first step with the message to be signed.
3) The first part of the signature (r) is computed in exactly the same way as in ECDSA (besides the fact that there is a different elliptic curve involved).
4) The second part of the signature s is computed as $s = (k + H(r|A|M) \cdot a) \mod L$, where A is the public key, M the message, and L the order of the Edwards curve. Note that in contrast to ECDSA, an EdDSA signature does not involve computing the inverse of the nonce k.

D. Threshold Signature Schemes

Threshold signature schemes are an application of secure multiparty computation. The basic idea in threshold signature schemes is that multiple parties are required to generate a signature. For this purpose, the secret key is split into multiple *key shares*, one for each party. The parties use their key shares to compute partial results, which are then combined to produce the signature. The resulting signature is indistinguishable from a signature computed by a single party with the secret key, such that only the signing and key generation algorithms are replaced in threshold signature schemes but the verification algorithm remains unchanged.

A threshold $t < n$ is defined such that any subset of t or more parties can jointly generate signatures, but any subset smaller than t can not. It is important to note that, with threshold signature schemes, the actual secret key never appears during signature generation.

Efficient threshold signature schemes have existed for some signature schemes (e.g., Schnorr [7]) for quite some time, but have only more recently been proposed for ECDSA. Designing an efficient ECDSA threshold signature scheme is challenging because signature generation involves the computation of the inverse of the nonce k, but the threshold setting dictates that no single party must know the nonce (as it could otherwise derive the secret key from the resulting signature). EdDSA, on the other hand, does not involve the computation of the inverse of the nonce so that designing a threshold EdDSA signature scheme is significantly less of a challenge.

III. Design Goals and Protocol Abstraction

Our API key protocol has been developed to account for three major design goals. First, it must be non-custodial, such that a third party never has control over user funds. Second, it must allow for the creation of flexible security policies (e.g., withdrawal limits based on time windows or addresses, market volume restrictions, etc.). Finally, it must minimize communication overhead, in terms of both bandwidth and rounds of communication for interacting parties.

The importance of the first design goal is self-evident, as the main benefit of interacting with a decentralized exchange is self-custody. A protocol for decentralized API keys should maintain that property. The second goal seeks to accommodate the widest possible variety of security policies; an ideal protocol would allow for the ability to construct any computable policy. The third goal is important as it allows the protocol to serve the needs of professional traders. Traders interacting with centralized exchanges place orders many times per second.

The protocol we have designed works broadly as follows (see technical details in Section V): Instead of generating signatures with a secret key residing just on a single machine, we employ a threshold signature scheme with two parties: the client and the exchange. Both parties need to cooperate in order to generate signatures. While the user is still in possession of the secret key, it is never actively used for trading. With the secret key, the user only creates API keys along with policies that describe the capabilities of these

API keys, i.e., under what circumstances the exchange should cooperate with the client on generating a signature. A policy can describe arbitrary properties. For example, a policy can restrict an API key to trade on certain markets only and to withdraw funds only to a specific address. So the secret key can be kept on a trusted, potentially even air gapped, device while API keys are used for day-to-day activities such as trading.

Theoretically, multi-signatures could also be used, but threshold signature schemes provide more flexibility than multi-signatures, because the underlying blockchains are not required to provide support specifically. Contrary to multi-signatures, the signature resulting from a threshold signature scheme is indistinguishable from a conventional ECDSA signing algorithm. For this reason, the verification algorithm can remain unchanged, and even the key generation algorithm can remain unchanged. Furthermore, it is impossible to secure an existing address by multi-signatures. A new address needs to be generated and funds transferred to that new address. An address can be secured with threshold signatures, however, without generating a new address and moving funds.

Unfortunately, threshold ECDSA schemes are challenging. We review existing threshold ECDSA schemes for their suitability to fit our protocol's design goals in Section IV. One major difference in our protocol is that the user knows their secret key. For this reason, API keys can be generated by the user offline on a trusted device, and the generated API key can then be transferred to some untrusted device without harming security. Furthermore, we aim to reduce the latency as perceived by clients as much as possible. We achieve this by shifting most of the computation and communication to earlier points in time, before actual trades are conducted. In this way, a client can conduct a trade within milliseconds (sub-millisecond even for EdDSA) with a single message sent to the exchange server.

IV. Evaluation of ECDSA Threshold Signature Schemes

We evaluate all recently proposed threshold ECDSA signature schemes (roughly in chronological order) with regard to their suitability to build the foundation of the protocol presented in this paper.

Green and Eisenbarth [12] improved upon original work from Ibrahim et al. [13] and Gennaro et al. [14]. The authors introduced fully distributed key generation and rekeying and modified the signing algorithm slightly to improve performance. However, the number of participants required is $\geq 2 \cdot t$, with t being the threshold required to generate signatures. Since neither the client nor the exchange must be able to generate signatures unilaterally, a threshold of 2 means that we would need to introduce additional parties, which is sub-optimal in the setting of decentralized cryptocurrency exchanges.

Gennaro, Goldfeder, and Narayanan [15] published the first threshold-optimal scheme (for DSA/ECDSA), with $n > t$. Signature generation requires 6 rounds of communication, however, which impairs fast execution.

Lindell proposed a 2-party (2-of-2) threshold signature scheme [8] [9][2]. While a 2-party threshold signature scheme seems like a significant limitation at the first glance, it is sufficient for the protocol that encompasses a client and an exchange. The scheme requires just 2 rounds of communication for generating a signature, is bandwidth-efficient, and computationally relatively fast. This efficiency is achieved by one party holding a homomorphic encryption of the other party's key share. This threshold ECDSA signature scheme is the first reasonable candidate to build the foundation of our protocol.

Boneh, Gennaro, and Goldfeder [16] improve upon Gennaro, Goldfeder, and Narayanan's work [15], reducing the number of communication rounds required for signature generation from 6 to 4, which is still more than the 2 rounds of [8], and the back-and-forth communication creates additional latency that impairs efficient trading.

The 2-of-2 and 2-of-n ECDSA threshold schemes proposed by Doerner et al. [10] do not rely on homomorphic encryption but instead use Oblivious Transfer (OT) multiplication. Signature generation is extremely performant and requires just 2 rounds of communication, but the bandwidth requirements are rather high. Nonetheless, it is a reasonable candidate.

Lindell and Nof [17] published a full threshold (t-of-n) ECDSA signature scheme. Instead of homomorphic encryption, the authors use ElGamal "in the exponent" to facilitate practical key generation in a multi-party setting and make signature generation more efficient. For signature generation, however, 8 rounds of communication are needed, which makes the scheme impractical in our protocol.

Gennaro and Goldfeder [18] improve upon Boneh, Gennaro, and Goldfeder's work [16], reducing communication overhead and the time to generate signatures. The number of communication rounds is increased to 9, however, which creates additional latency that impairs efficient trading.

Doerner et al. [19] [20] improve upon their previous work [10], extending it from a 2-of-n to a full t-of-n threshold signature scheme. The number of communication rounds required is $\log(t) + 6$, however, which makes the scheme impractical in our protocol.

Castagnos et al. [11] generalized Lindell's 2-of-2 threshold signature scheme [8] using hash proof systems. The scheme achieves good performance and requires low bandwidth, which makes it the third reasonable candidate to build the foundation of API keys for our protocol.

Castagnos et al. [21] build upon Gennaro and Goldfeder's t-of-n threshold signature scheme [18], improving computational and bandwidth efficiency. The number of communication rounds required for generating signatures is reduced from 9 to 8, which leaves the scheme still impractical for our use.

[2]Note that the full version of the paper [9] has received significant updates after the original publication.

TABLE I
SUMMARY OF ECDSA THRESHOLD SIGNATURE SCHEMES CANDIDATES

Scheme	Threshold-optimal	Rounds (signing)	Signing time	Signing bandwidth
Lindell [8] [9]	✓	2	$\sim$ 35 ms	$\sim$ 0.8 kB
Doerner et al. [10]	✓	2	$\sim$ 3 ms	$\sim$ 85.7 kB
Castagnos et al. [11]	✓	2	$\sim$ 150 ms	$\sim$ 5.6 kB

Summary of Evaluation

While most recent t-of-n ECDSA threshold signature schemes are threshold-optimal, our evaluation revealed that all of them require too many rounds of communication for signature generation. 2-of-2 or 2-of-n ECDSA threshold signature schemes, on the other hand, require only 2 rounds of communication, which significantly reduces latency and therefore facilitates efficient trading. It remains an open question whether the higher number of rounds of t-of-n ECDSA threshold signature schemes is a fundamental limitation or just requires further research. In any case, for our protocol, a 2-of-2 ECDSA threshold signature scheme is sufficient, and we continue our evaluation of the three candidate schemes.

Our evaluation resulted in three candidate schemes by Lindell [8], Doerner et al. [10], and Castagnos et al. [11]. All three candidates are threshold-optimal and require just 2 rounds of communication for signature generation. Apart from the cryptographic assumptions the schemes rely on, they differ in performance in terms of time required to generate signatures and the bandwidth required. Lindell's scheme [8] has good performance in terms of time required to generate a signature ($\sim$35 ms) and requires the least bandwidth ($\sim$0.8 kB). While Doerner et al.'s scheme [10] is the fastest, it also requires significantly more bandwidth. Traders conducting thousands of trades per second would require tens of Gbit of bandwidth solely for generating signatures, which represents a significant disadvantage for Doerner et al.'s scheme. Castagnos et al.'s scheme [21] is second of the three with regard to bandwidth required ($\sim$5.6 kB) but brings a significant disadvantage in terms of time to generate a signature ($\sim$150 ms), which makes it less favorable. So Lindell's scheme [8] remains as the best contender overall — in terms of bandwidth and in terms of time to generate a signature. Table I summarizes the results. And, as we will explain in Section V, we improve the perceived signing time such that it is almost en par with Doerner et al.'s scheme [10].

V. API KEYS FOR DECENTRALIZED EXCHANGES

A. ECDSA Threshold Signature Schemes

With a 2-of-2 threshold signature scheme, the two parties, i.e., the user and the exchange, are both required to cooperate during day-to-day operations, i.e., generate signatures on transactions. To this end, the full secret key x is split into two multiplicative secret shares x_1 and x_2 where $x \equiv x_1 \cdot x_2 \mod q$, with q being the order of the elliptic curve. One key share remains with the user's client, i.e., the machine holding the API key, and the other key share goes to the exchange. Signatures that would be valid under the public key cannot be generated with one key share alone and neither does knowledge of any one key share (i.e., x_1 or x_2) leak information about the full secret key x. Both parties can therefore control user's funds only when collaborating. In this way, the exchange can restrict an API key's capability (by refusing to collaborate) without requiring full access to the user's funds. Generally speaking, two parties interact in secret-key-operations (i.e., signing), and neither party can manipulate that operation to gain an advantage, even if that party is malicious. The most harmful thing a malicious party can do during signature generation is to prevent the operation from completing.

For the reasons outlined in Section IV, we build upon the signing algorithm from Lindell's scheme [8]. However, there exist (significant) differences:

1) API keys are generated solely by the client.
2) We significantly improve performance by using Diffie-Hellman key exchange instead of a coin tossing protocol.
3) We dramatically improve the perceived performance by employing a 2-phase approach for generating signatures.

B. API Key Generation

An API key is generated by the user's machine with access to the full secret key. Since that machine is considered a "trusted dealer", API key generation can happen entirely on the client. At first glance, this may sound counter-intuitive, but it is a result of the fact that we exploit knowledge of the full secret key on the client. With this trick, the client can create API keys efficiently.

An API key consists of two secret shares: the client secret share and the (encrypted) exchange secret share. The client sets the exchange's secret share x_1 initially to 1 and the client's secret share x_2 to the full secret key x. Then, a random number r (not to be confused with the r-part of an ECDSA signature) is generated by the client and the secret shares are updated as follows:

1) The update to the exchange's secret share is computed $x_1\prime \equiv x_1 \cdot r \mod q$.
2) The new exchange's secret share $x_1\prime$ is encrypted under the exchange's Paillier public key.
3) The update to the client's secret share is computed $x_2\prime \equiv x_2 \cdot r^{-1} \mod q$.

Note that $x \equiv x_1 \cdot x_2 \equiv x_1\prime \cdot x_2\prime \mod q$, and therefore the full secret key x, the public key, and the corresponding address remain unchanged such that – with the help of the API key – signatures can be generated that are indistinguishable from signatures generated by the full secret key and are therefore oblivious to blockchains.

The client using the API key must not be able to gain knowledge of the full secret key. For this reason, the exchange's secret share is encrypted under the exchange's Paillier public key. Note that the client must verify that the Paillier public key was generated correctly. A Paillier public key is defined as $N = p \cdot q$, with p, q being large primes. In order to certify that the exchange has generated the Paillier public key correctly, it needs to proof that $gcd(N, \phi(N)) = 1$ holds. To this end, we use the proof from Goldberg et al. [22] Section 3.2 with parameters $\alpha = 6370$ and $m_1 = m_2 = 11$ as suggested by Lindell and Nof [17] in Section 6.3.2. Since the exchange needs to create just a single Paillier key pair for all users, it needs to compute the proof of correctness just once. On the client side, verifying the correctness of the Paillier public key needs to be conducted only once as well, but is computationally rather efficient ($\sim$41 ms) anyway.

C. Signature Generation: Preparation Phase

For signature generation, we employ a 2-phase approach. The main idea with such a 2-phase approach is to shift some required communication and computation into a message-independent preparation phase such that the finalization phase can happen with a single message from the client to the exchange. By leveraging computational resources when a trader is not trading, performance is improved significantly.

With just one round of communication, client and server prepare a pool of (arbitrarily many) elliptic curve points, from which the first part of the signature is derived (as described in Section II-B). Recall that a point is computed as $R = k \cdot G$ where k is a cryptographically secure nonce. Knowledge of k allows deriving the secret key from a signature, so neither client nor exchange must know k. We employ the elliptic-curve Diffie-Hellman (ECDH) key agreement protocol in a bit of an unusual way. Commonly, ECDH is used to generate a shared secret over an insecure channel such that an observer of the communication cannot derive the shared secret. In our case, we employ ECDH such that the communicating parties arrive at a shared value (which will be made public afterwards) but do not know the other party's secret value. In the threshold ECDSA setting $k \equiv k_1 \cdot k_2 \mod q$, and each party knows R and either k_1 or k_2 but cannot derive k (unless that party could solve the elliptic curve discrete logarithm problem, which is supposedly intractable).

The main advantage of this 2-phase approach is that the signature finalization phase can be done computationally efficiently with just a single message from the client to the exchange. This efficiency is achieved by shifting most of the communication and computational work into the signature preparation phase. The computational work entails the (precomputed) randomness for Paillier encryption, which comprises modular exponentiation on big integers, and scalar multiplication. The storage overhead is negligibly small with 65 B per prepared point on the server (33 B for the compressed point representation and 32 B for the server's DH secret)[3]. Storage overhead is also small on the client with 321 B per prepared point on the client (33 B for the compressed point representation, 32 B for the client's DH secret, and 256 B for the Paillier randomness). Computational cost and bandwidth required for the preparation phase increases linearly with the number of points to prepare, but this preparational step can happen anytime before the user wants to trade (e.g., right after login) such that it does not affect the performance of creating the final signature (which is effectively what traders perceive as latency). Table II depicts the preparation phase of signature generation.

TABLE II
SIGNATURE GENERATION PROTOCOL: PREPARATION PHASE

Exchange		**Client**
		Choose random k_2 Compute $R_2 = k_2 \cdot G$
	$\xleftarrow{R_2}$	
Choose random k_1 Compute $R_1 = k_1 \cdot G$ Compute $R = k_1 \cdot R_2$ Store R, k_1		
	$\xrightarrow{R_1}$	
		Compute $R = k_2 \cdot R_1$ Compute random ρ Store R, k_2, ρ

ρ: (precomputed) randomness for the homomorphic Paillier encryption scheme.
G: generator of the curve.

D. Signature Generation: Finalization Phase

The signature finalization phase consists just of a single message from the client to the exchange, like with a centralized exchange where the client sends a single message indicating a trade. In this way, there is no effectively communication overhead in terms of communication rounds.

The client chooses any of the precomputed points and computes a *presignature* using the API key and that point. The client then sends the presignature along with the chosen point over to the exchange to complete the signature. It is essential for security that the client deletes the point used to prevent reuse (and therefore potential compromise of the secret key). Upon ensuring compliance with the signing policy (see Section V-F), the exchange completes the signature. Again, deletion of the point and DH secret used is paramount for security. Eventually, the correctness of the signature is verified. Table III depicts the finalization phase of signature generation.

E. Performance

With the 2-phase approach described, no communication and computation is spared as such, but most communication

[3]Note that the provided numbers represent the minimum and may be slightly higher depending on the encoding used.

TABLE III
SIGNATURE GENERATION PROTOCOL: FINALIZATION PHASE

Exchange		**Client**
		Compute r from R $c_3 = Enc(k_2^{-1} \cdot r \cdot d_2 \cdot d_{1enc} + k_2^{-1} \cdot m + \rho \cdot q)$ Delete R, k_2, and ρ
	$\xleftarrow{c_3, R}$	
Compute r from R Lookup k_1 with R Compute $s = k_1^{-1} \cdot Dec(c3) \mod q$ Delete R and k_1 Verify signature (r, s)		

d_2: the client's secret share.
d_{1enc}: the encrypted secret share of the exchange.
ρ: the (precomputed) randomness for the Paillier homomorphic encryption scheme.
m: the (message's) hash to be signed.
q: the order of the curve.

and computation is shifted to a point in time when it does not impair trading. In this way, trading can happen with a single message from the client to the exchange with relatively little computation required — the performance perceived by traders when conducting trades on a decentralized exchange is now comparable to trading on a centralized exchange.

TABLE IV
PERFORMANCE: PREPARATION PHASE

Exchange		**Client**
		Time: ∼13 ms
	$\xleftarrow{33\text{ B}}$	
Time: ∼0.16 ms		
	$\xrightarrow{33\text{ B}}$	

TABLE V
PERFORMANCE: FINALIZATION PHASE

Exchange		**Client**
		Time: ∼1.72 ms
	$\xleftarrow{545\text{ B}}$	
Time: ∼2.46 ms		

Tables IV and V show the performance for the preparation and finalization phases respectively. With the 2-phase approach, most of the communication and computation is moved into the preparation phase. The preparation phase takes few computational and bandwidth resources, and the overall time is dominated by communication delays. The finalization phase is very fast and takes just 4 ms in total and a single message from the client to the exchange[4]

[4]Note that the numbers depend on the elliptic curve used. For the measurements we used *secp256k1*.

F. Signing Policy

On a decentralized exchange, a full client without API keys signs a transaction and sends the signature to the exchange. For API keys, however, the exchange receives a presignature from the client and needs to complete it first. The exchange may base its agreement to complete a signature upon several factors, without taking custody of the user's funds. These factors are specified as *signing policy* by the user. Obviously, not the holder of the API key but only the user must be able to create and modify the signing policy. Such policies basically restrict the conditions under which the exchange participates in signature generation. Policies may involve for example:

- Daily / weekly / monthly withdrawal limits. The user can set an amount that is allowed to be moved from their exchange account each day. This can limit damage in the event of a compromised API key.
- IP address restrictions for trading and/or withdrawal.
- Location restrictions for trading and/or withdrawal.
- Withdrawal addresses restrictions.
- Daily / weekly / monthly trading limits. Limits are imposed upon how much users can trade.
- Time delayed withdrawals, where funds will be moved only after a 24- or 48-hour period. Such delay allow users in the event of a hacked account to reach out to the exchange and prevent fund movement within the delay period.
- Allow access only from specific devices.
- Allow access only to specific markets.
- Restrictions based on biometric information such as a fingerprint scan.

VI. DISCUSSION AND FUTURE WORK

A. Threshold EdDSA

We have extended the API key protocol described in Section V to also work with the EdDSA signature scheme. In contrast to threshold ECDSA (where multiplicative secret

TABLE VI
PAILLIER PERFORMANCE

Operation / Key size	512 bit	1024 bit	2048 bit	3072 bit	4096 bit	8192 bit
Precompute randomness	320 µs	2106 µs	13 595 µs	39 583 µs	74 151 µs	420 651 µs
Encrypt	<1 µs	1 µs	4 µs	7 µs	12 µs	35 µs
Decrypt	106 µs	453 µs	2081 µs	6032 µs	13 666 µs	78 387 µs

sharing is used), we use additive secret sharing for threshold EdDSA. For API key generation, the client generates a random scalar r which represents its secret share and computes the server's secret share as $a - r \mod L$, where a is the signing key. The client keeps its secret share and sends the (encrypted) exchange secret share to the exchange.

For signature generation we employ the same DH-like approach as we do for threshold ECDSA. The resulting point is computed a bit different though as R is the (point) addition of the public client point plus the public server point. While using that approach breaks EdDSA's deterministic approach, it does generates valid signatures nonetheless. Like in the threshold ECDSA scheme, we use the two-phased approach with a message-independent phase and a second phase in which the signature for a particular message is generated in order to facilitate signature generation with a single message from client to server and therefore improve the performance perceived by users.

Signature generation for threshold EdDSA is straightforward: the client can compute its part of the second part of the signature s as $s\_client = r\_client + h * client\_secret\_share$, with $h = H(r|A|M)$ (r: first part of the signature, A: public key, M: message). s_client is sent over to the server. The server computes its corresponding part and adds both numbers, which results in the second part of the signature s. The computations are very efficient, since the client does not need to do computations on the encrypted server secret share. For this reason, the computational performance of signature generation in threshold EdDSA is exceptional - just marginally slower than conventional, non-threshold EdDSA.

B. Generalization to Other Decentralized Services

In this paper, we explored the application of decentralized cryptocurrency exchanges, i.e., the Nash exchange. The protocol we presented, however, can be generalized to other decentralized services, and we have already applied the protocol to Uniswap and 1inch, allowing users to trade via the associated smart contracts without accessing their full secret key.

C. Paillier Key Size

The currently recommended key size for the Paillier cryptosystem is 2048 bit. Increasing the key size would be desirable but larger keys significantly increase the times for encrypting and decrypting. Table VI shows the results of our evaluation. Precomputing randomness takes longest, but is part of the signature preparation phase and does not affect signature finalization. Encryption increases as well with larger key sizes but the time it takes is negligible even with very large keys. Decryption, however, takes a significant amount of time and is part of the signature finalization phase. We leave the question of how to optimize the performance of Paillier decryption with large keys as future work.

VII. CONCLUSION

In this paper, we presented a protocol enabling API keys for decentralized cryptocurrency exchanges that is suitable for both security-sensitive end-users and professional traders alike. Given the fact that the full secret key is never required for any day-to-day operation, the risk of compromised secret keys is drastically reduced. The protocol presented in this paper allows users of an exchange to create many API keys associated with their account, each of which may be entitled to different (withdrawal and trading) limits. This allows trading companies to use decentralized exchanges without trusting their employees with credentials that have access to the full secret key material. To this end, client and exchange engage in an interactive signature generation protocol. The communication overhead is negligible, and the effective delay due to computations is roughly 4 ms, satisfying the low latency requirements of professional traders. High performance is achieved by employing a 2-phase approach, which shifts the majority of the computational time required as well as the additional communication delay (caused by the interactivity of the protocol) into a message-independent phase. In this way, a signature can be finalized with a single message from client to the exchange. We additionally provide an open source implementation of the protocol in Rust[5].

REFERENCES

[1] *Nash Exchange*. [Online; accessed 27-April-2021]. URL: https://app.nash.io.

[2] *Uniswap*. [Online; accessed 27-April-2021]. URL: https://app.uniswap.org/.

[3] *1inch*. [Online; accessed 18-June-2021]. URL: https://app.1inch.io.

[4] Jonathan Katz. *Digital Signatures*. Springer US, 2010. ISBN: 978-0-387-27711-0.

[5] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". In: *International Journal of Information Security* 1.1 (Aug. 2001), pp. 36–63.

[5] https://github.com/nash-io/nash-rust/tree/master/mpc-wallet/nash-mpc

[6] T. Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979 (Informational). RFC. Fremont, CA, USA: RFC Editor, Aug. 2013. DOI: 10.17487/RFC6979. URL: https://www.rfc-editor.org/rfc/rfc6979.txt.

[7] Claus P. Schnorr. "Efficient Identification and Signatures for Smart Cards". In: *Advances in Cryptology (CRYPTO 1990)*. Springer. 1990, pp. 239–252.

[8] Yehuda Lindell. "Fast secure two-party ECDSA signing". In: *Annual International Cryptology Conference (CRYPTO 2017)*. Springer. 2017, pp. 613–644.

[9] Yehuda Lindell. *Fast Secure Two-Party ECDSA Signing*. Cryptology ePrint Archive, Report 2017/552. https://eprint.iacr.org/2017/552. 2017.

[10] Jack Doerner et al. "Secure Two-party Threshold ECDSA from ECDSA Assumptions". In: *IEEE Symposium on Security and Privacy (SP18)*. 2018, pp. 980–997.

[11] Guilhem Castagnos et al. "Two-Party ECDSA from Hash Proof Systems and Efficient Instantiations". In: *Advances in Cryptology (CRYPTO 2019)*. Springer. 2019, pp. 191–221. ISBN: 978-3-030-26954-8. DOI: 10.1007/978-3-030-26954-8_7.

[12] Marc Green and Thomas Eisenbarth. *Strength in Numbers: Threshold ECDSA to Protect Keys in the Cloud*. Cryptology ePrint Archive, Report 2015/1169. https://eprint.iacr.org/2015/1169. 2015.

[13] M. H. Ibrahim et al. "A robust threshold elliptic curve digital signature providing a new verifiable secret sharing scheme". In: *IEEE Midwest Symposium on Circuits and Systems*. Vol. 1. 2003, pp. 276–280.

[14] Rosario Gennaro et al. "Robust Threshold DSS Signatures". In: *Advances in Cryptology (EUROCRYPT '96)*. Springer, 1996, pp. 354–371. ISBN: 978-3-540-68339-1.

[15] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. "Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security". In: *International Conference on Applied Cryptography and Network Security, ACNS 2016*. Springer, 2016. ISBN: 978-3-319-39555-5. DOI: 10.1007/978-3-319-39555-5_9.

[16] Dan Boneh, Rosario Gennaro, and Steven Goldfeder. "Using Level-1 Homomorphic Encryption to Improve Threshold DSA Signatures for Bitcoin Wallet Security". In: *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT 2017)*. Springer. 2017, pp. 352–377. ISBN: 978-3-030-25283-0. DOI: 10.1007/978-3-030-25283-0_19.

[17] Yehuda Lindell and Ariel Nof. "Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. 2018, pp. 1837–1854. ISBN: 9781450356930. DOI: 10.1145/3243734.3243788.

[18] Rosario Gennaro and Steven Goldfeder. "Fast Multiparty Threshold ECDSA with Fast Trustless Setup". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. 2018, pp. 1179–1194. ISBN: 9781450356930. DOI: 10.1145/3243734.3243859.

[19] Jack Doerner et al. *Threshold ECDSA from ECDSA Assumptions: The Multiparty Case*. Cryptology ePrint Archive, Report 2019/523. https://eprint.iacr.org/2019/523. 2019.

[20] Jack Doerner et al. "Threshold ECDSA from ECDSA Assumptions: The Multiparty Case". In: *IEEE Symposium on Security and Privacy (SP 2019)*. 2019, pp. 1051–1066. DOI: 10.1109/SP.2019.00024.

[21] Guilhem Castagnos et al. "Bandwidth-Efficient Threshold EC-DSA". In: *IACR International Conference on Public-Key Cryptography (PKC 2020)*. 2020, pp. 266–296. DOI: 10.1007/978-3-030-45388-6_10.

[22] Sharon Goldberg et al. "Efficient Noninteractive Certification of RSA Moduli and Beyond". In: *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2019)*. Springer, 2019, pp. 700–727. DOI: 10.1007/978-3-030-34618-8_24.