Hello Ethereum research, I've been working on a simple system for interoperable smart contracts that I could use some feedback on. It's still in the early stages, but I think there's enough to give a general idea of how it works.

The problem

I'm trying to build a system for general-purpose smart contracts where its possible to support virtually any crypto-asset. I think such a system would be useful because it would open up asset exchange between all assets and make it much easier to create diverse financial instruments, utilising any kind of asset.

I believe existing attempts to solve this problem are inadequate because they rely on high level abstractions like TX formats, specific consensus algorithms, or OP code availability, that differs vastly between asset types. The most suitable solution currently uses multi-party computation for programmable secret-sharing, but this approach introduces network latency, and makes it too costly to run more complex systems.

The solution

My solution is based on programmable, tamper-proof release of private keys. Here's a diagram for those visual thinkers among you https://imgur.com/a/QxeRRFZ .

The client portion follows these steps:

1. Clients use standard trusted execution environments (TEE) like Samsung Knox, Intel Enclaves, AMD Trustzone, and so on (UUICs and eSIM would also work!) Within this environment a program generates N ECDSA key pairs.

2. The pub keys from (2) are exposed to a crypto wallet. The TEE produces a proof for the wallet to validate before proceeding (proofs might look a little like this: https://pastebin.com/PmHa8vRd

That's what a report looks like from an Intel Enclave attesting to a certain set of parameters.)

1. Clients crypto wallet sends funds to the corresponding pub keys using a relevant address format. The size of the deposit amount is based on a standard rate in USD: for example - say $50 USD worth of X coin, at current market value.

Now you may be thinking: trusted computing. Surely you have seen the recent attack mounted on Intel Enclaves (plundervault.) But my answer is the same the last time there was a problem with enclaves: avoid over-centralisation of resources, and use a layered approach to security. A single malicious client should not be able to take down the whole system.

The next part to introduce is the server. The server acts as a risk management engine for clients. Server code runs inside a TEE, and produces proofs for the clients that they can validate against a signed build system (signed binary + open source code that made it is needed.) The server manages risk but its other function is to act as a switch board for value allocation between contracts entered into or created by other clients. Private keys are not intelligibly divisible, so the engine will need to be able to deal with "chunks" of coins and allocate them as efficiently as possible.

The work flow for the server might look something like this:

1. Client Request:

I'm looking for X worth of Y, to be bound by contract Z. Let me know if you know about anyone who's interested.

1. Server Response:

I have a set of […] from clients […]. I'm not giving you too many from the same person though.

1. Client:

Validate received balances and TEE signatures. Return accept to server. Run contract inside TEE. Release any private keys to counterparties based on contract logic.

1. Client:

If a private key is received, claim the funds and move the money to a new 'address.' Indicate success to server. If a double-spend occurs here (e.g. can't claim funds.) Indicate failure to the server.

The client who issued the request now validates any returned set of deposits using their wallet. This means that there's zero blockchain-specific or ledger-specific code to maintain, and all wallets can easily support the protocol with functionality they already have. All that is needed is a common interface.

So to recap: our client now has entered into a contract with enough parties to match their interests. The contract will be carried out / enforced by their TEEs. Funds will change hands by swapping private keys if they need to be (depends on contract rules – remember could be anything – its general purpose.) Clients may move funds after receiving private keys (depending on scaling vs risk preferences.)

How security is maintained

I would not expect anyone to fully trust a TEE with their money. There are a number of techniques that can be used to improve the security of the system in addition to using a TEE:

1. Basic risk management (don't fund a contract solely with the same inputs from the same TEE.)
2. Reputation management + Sybil bond (if using Samsung Knox for the TEE, a deposit bond is already in place = the cost of phones is high! Computation in SIM cards and signatures from mobile networks is another potential sybil factor.)
3. Additional factors added to key exchange (like threshold ECDSA, key escrow, time-locks, etc.)
4. Penalties???
5. Tit-for-tat transfer of private keys

Handling failure or double-spends

If a client indicates they are unable to 'claim' funds it may indicate a double-spend. The server needs to be able to adjust reputation, contract allocation, and other factors to control for possible loss from theft of funds.

The platform doesn't have to be 100% secure all of the time for it to be valid. It just needs to work well enough that it can guarantee there will always be enough funds to cover legitimate users. That may be possible with a small sign-up fee, usage costs, or insurance.

If you've made it this far thanks so for reading!

That should hopefully be enough for an overview. Any feedback would be appreciated. The rest of this is optional to read but is further commentary on security aspects.

Note: I'll add some smart contract pseudo-code when I next have time to show what a DSL might look like for this.