How TEE is used in DeFi

DeFi MOOC ‘24 - Andrew Miller

The Initiative for CryptoCurrencies and Contracts

Privacy in smart contracts is an innovation bottleneck

The toolbox of **ZK** has done a great job of expanding what's possible, **MPC**, **FHE**, and **TEE** are coming along as well.

These all turn out to be *complementary*. You will eventually want TEE *plus* {ZK, MPC, FHE} in your dApp.

TEEs continue to be *underappreciated*, which I'm trying to fix

This talk: interventions to help blockchain industry overcome this bottleneck by using TEE as appropriate tech

The web3 TEE-in-blockchains Redemption Arc

secret

OASIS

Phala

obscuro

puffer

1RPC. Web3 Relay

Switchboard

Avalanche Bridge

Enclave Markets
Taiko
Marlin
shbots Builder
roll ZK+TEE


Shea Ketsdever @SheaKetsdever · May 3

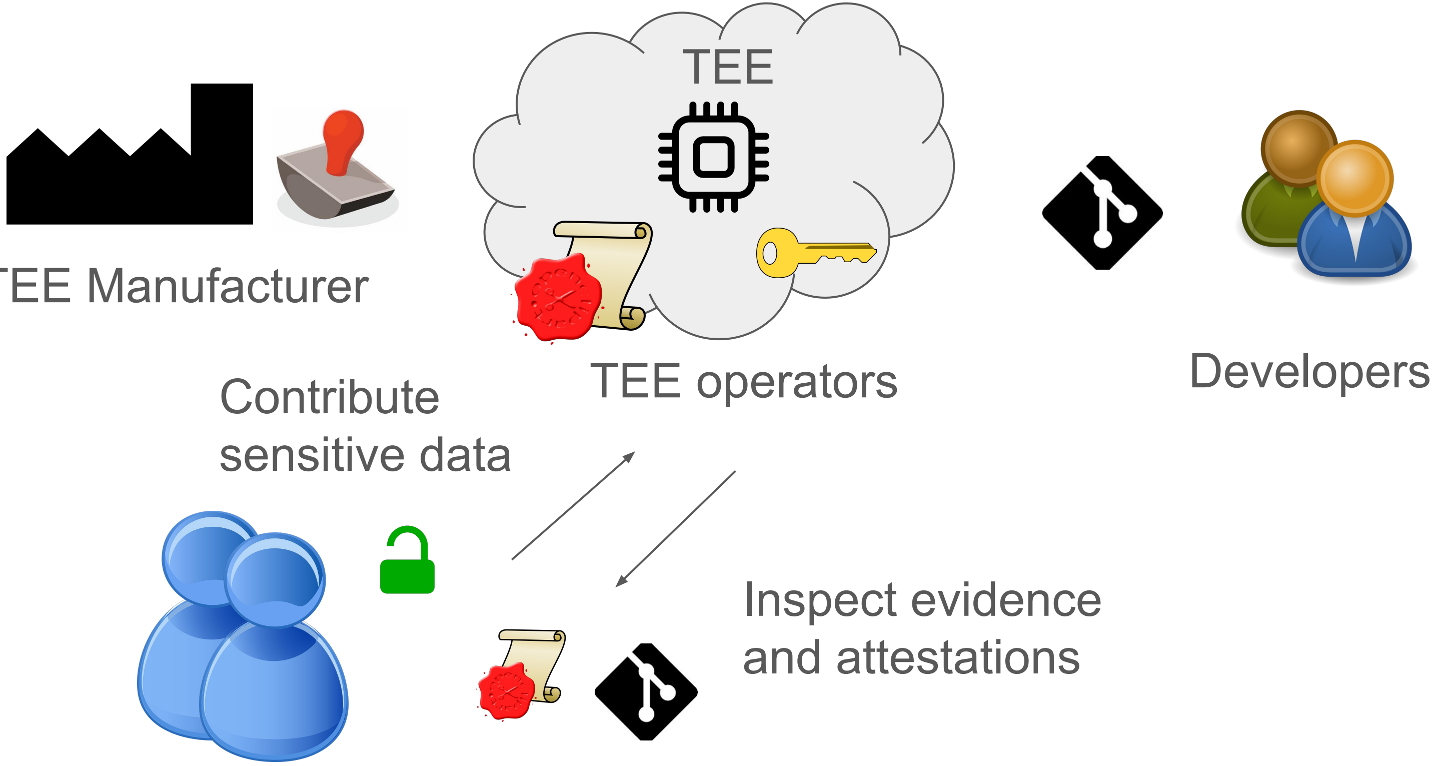
3.12.2023: First SGX block etherscan.io/block/16813125

4.30.2024: First TDX block etherscan.io/block/19767105

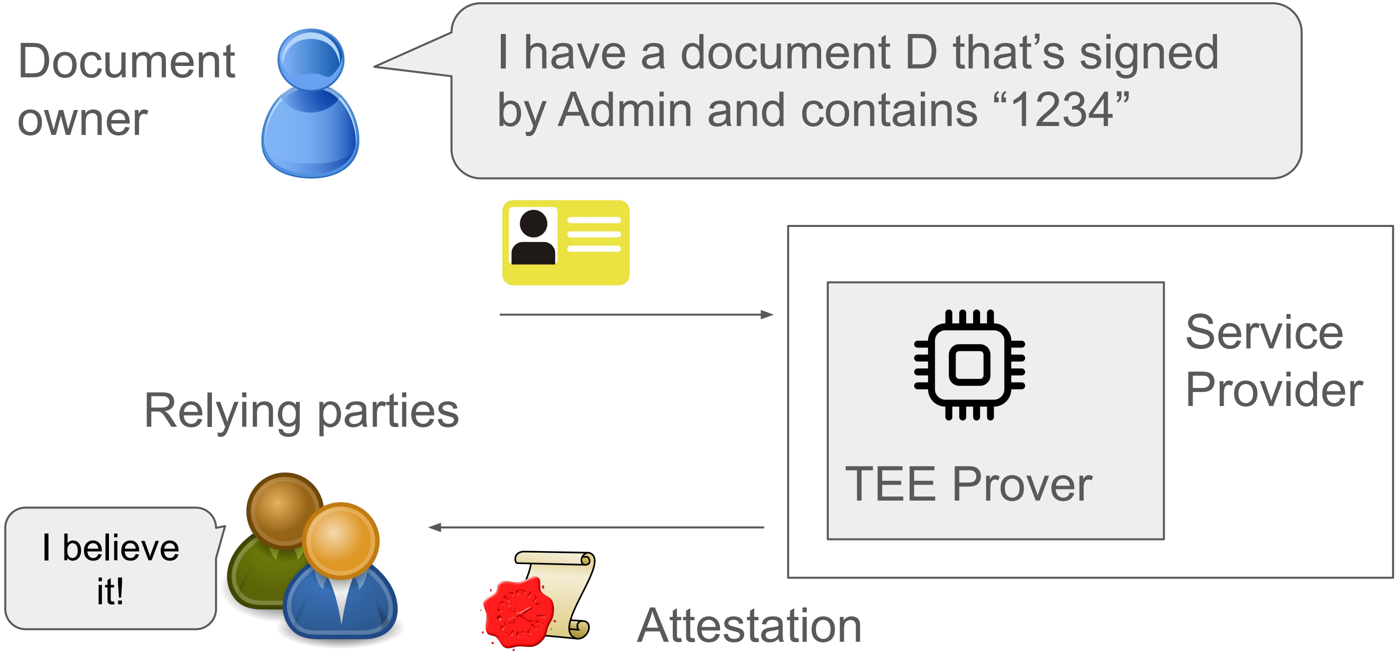
Soon: Majority of blocks built in TEEs

1 8 1.2K

How TEEs disintermediate app developers and clouds



Let's make a useful self-contained TEE application

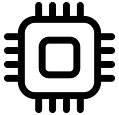


Self-contained example:
Trusted Setup using a TEE

Certificate chain

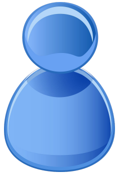
TEE Manufacturer

TEE



Samples p and q
Computes N = pq
Throw away p,q
Output N

Relying parties



N, att

Inspect the source code
CheckAtt(att, policy, N)

https://github.com/amiller/gramine-rsademo/blob/master/rsademo.py

```python
def sample_prime():
 p = random.randint(2**1023,2**1024-1)
 while not is_prime(p):p=random.randint(2**1023,2**1024-1)
 return p


if __name__ == '__main__':
 p = sample_prime()
 q = sample_prime()
 N = p*q
 del(p)
 del(q)
 print('RSA modulus:', N)
```

Rapid prototyping with Python in Gramine

Gramine is suitable for running python, so a “TEE-vm”

Check it out in CI-Examples/python

Where does it come from? Browse the manifest and see lib files

Comes with everything in the system python libs… but we could point it to a virtual env too.

Remote Attestation in Gramine

Gramine can produce remote attestations, that connect the root of trust (Intel's published certificate) to:

- An app-defined message (user report data)
- Summary of the app program (MRENCLAVE)
- and the configuration of the machine.

Accessed from /dev/attestation. Write to /dev/attestation/user_report_data.

We can parse and verify them with tools on a separate host

Remote Attestation verification in a Smart Contract

Often useful to post these to a public record. On-chain is good for this.

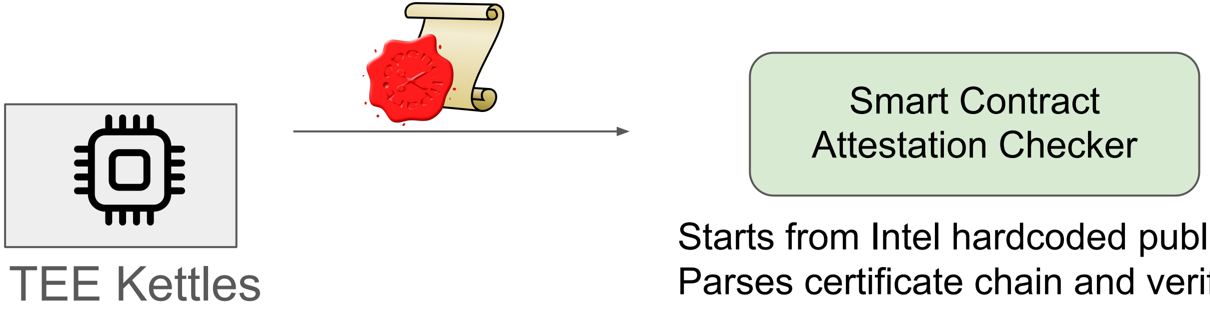
“Attestation Transparency” analogous to Certificate Transparency.

- Automata DCAP. Also implementations using ZK, from Phala and from Clique

    https://github.com/automata-network/automata-on-chain-pccs

SGX remote attestation on-chain contest

https://github.com/amiller/sgx-epid-contest/blob/master/README.md#good-riddance-to-epid-pre-deprecation-memorial-contest



Starts from Intel hardcoded public key
Parses certificate chain and verifies each signature.
Determines if the configuration is acceptable:

- in this case we allow all of them,
- but only 1 entry per configuration

https://optimistic.etherscan.io/address/0x490a428b0301d61db6ed45eddc55d615f2ea9f75#readContract

SGX remote attestation on-chain contest

Query

└ id *string*, timestamp *string*, version *string*, epidPseudonym *string*, advisoryURL *string*, advisoryIDs *string*, isvEnclave(
epidGroupID *string*

[**parse_epid_report(bytes)** method Response]

» **id** *string* : 22172948167720775901272948676082050685 9

» **advisoryIDs** *string* : ["INTEL-SA-00617","INTEL-SA-00657","INTEL-SA-00767"]

» **isvEnclaveQuoteStatus** *string* : GROUP_OUT_OF_DATE

» **platformInfoBlob** *string* :
150200650400010000080800000000000000000000000000000D00000C000000020000000000000C3D37B5D2657FF196BD6DBDB
D768DAA70E0054CE7B0CD0

» **isvEnclaveQuoteBody** *string* :
020001003d0c00000f000f000000000051caf5a48b450d624aefe3286d3148940000000000000000000000000508000000000(
00000000003000000000000008b1a4e11394366e979f8fa2716fdccee76ec79090f9c100543b931e3aa91993e00000000000000(
b47a90ce0f0dc550000000000000000000000000000000000000000000000000000000000000000000000000000000000000(
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
469756d2852292053696c766572204a35303430204350552040203322e303047487a

» **userReportData** *string* : 2a0e1753e8089dc964662310626587ec025b7b0074656c2852292050656e7469756d285229205369 6

» **epidGroupID** *string* : 3D0C0000

Tagging a release / Reproducible build

Here's a recipe for reliably producing the same MRENCLAVE:

Start from the Gramine dockerhub image

We can use the fixed version of python already present in the base image

The manifest will traverse library files in the base image

Anything tracked in this repo will be stable using git

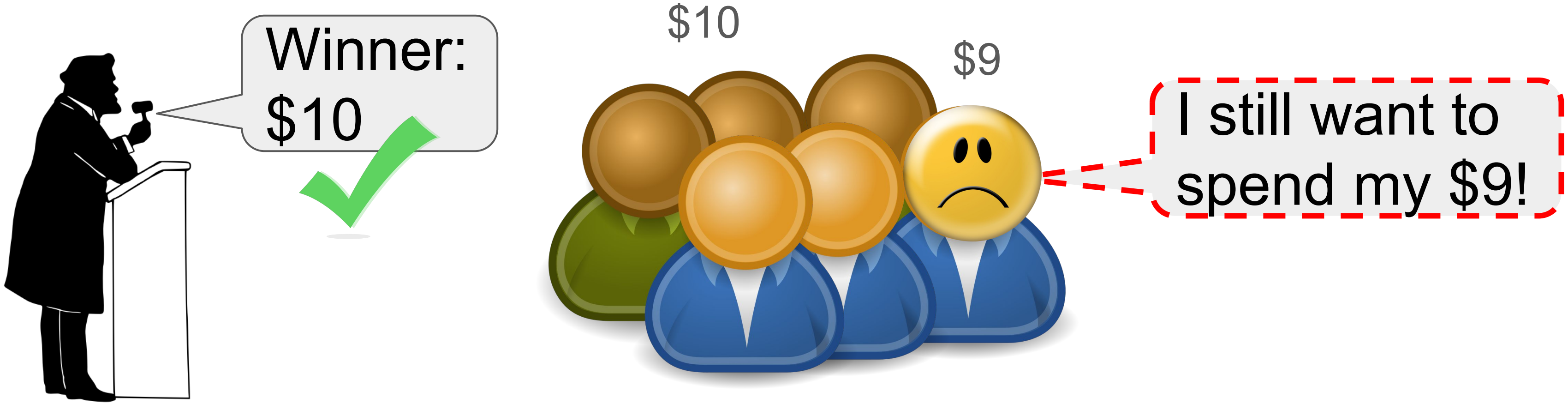
Further dependencies will need to be tracked (e.g., with nix)

Example: https://github.com/amiller/gramine-rsademo

Using TEE for DeFi

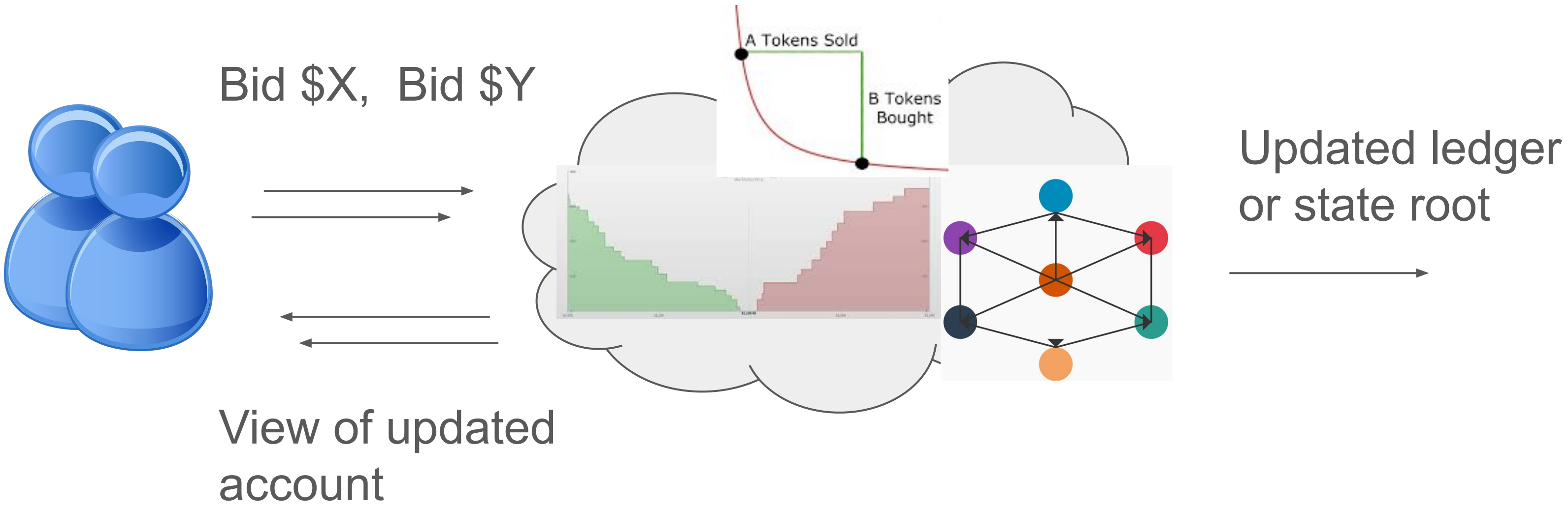
The Residual Bids problem

Example: an auction that conceals all the losing bids

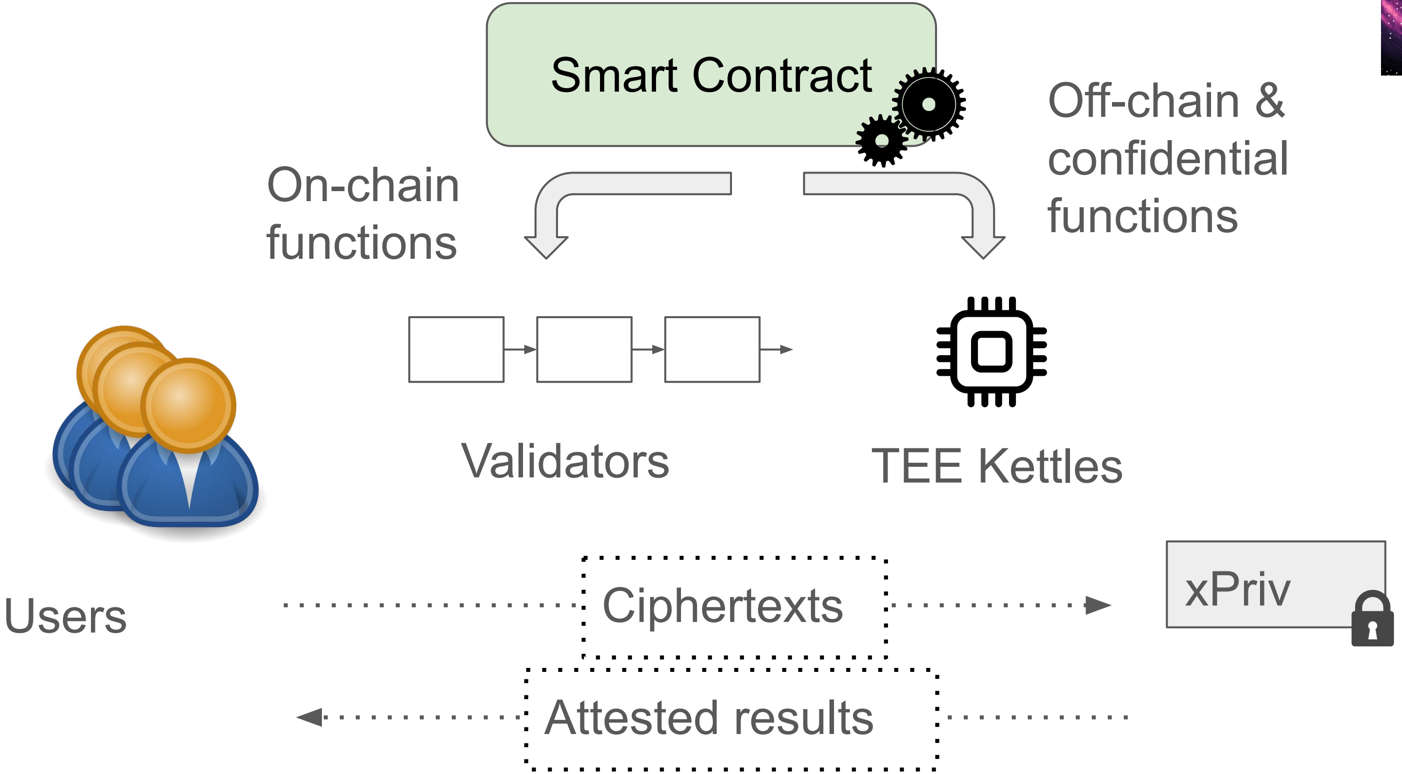


Today's unmet demand is tomorrow's bids!
This is strategic information to protect

Sufficient Motivating Application: Batch auctions



Sirrah: speedrunning a TEE Coprocessor

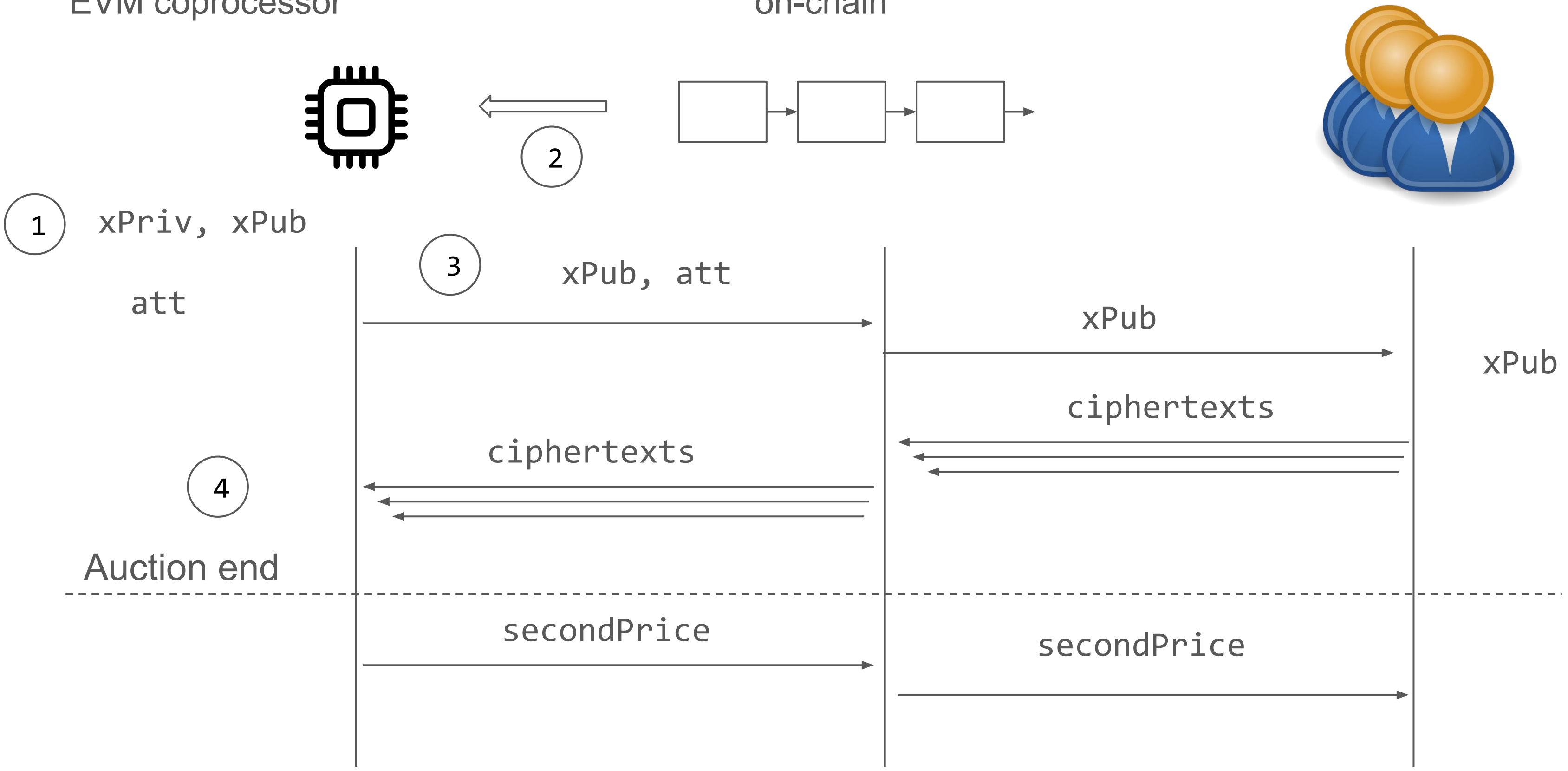


https://writings.flashbots.net/suave-tee-coprocessor

Sealed bid Auction using TEE Coprocessor

Confidential queries run on off-chain EVM coprocessor

Ordinary EVM functions run on-chain



Patching the auction using Sirrah (Before, plaintext)

```
contract LeakyAuction is AuctionBase {

  mapping (address => uint) public balance;
  uint public constant auctionEndTime = /* deadline */
  uint public secondPrice;
  mapping (address => uint) public bids;
  address[] public bidders;

  // Accept a bid in plaintext
  event BidPlaced(address sender, uint bid);
  function submitBid(uint bid) public virtual {
    require(block.number <= auctionEndTime);
    require(bids[msg.sender] == 0);
    bids[msg.sender] = bid;
    bidders.push(msg.sender);
    emit BidPlaced(msg.sender, bid);
  }
```

```
  // Wrap up the auction and compute the 2nd price
  event Concluded(uint secondPrice);
  function conclude() public {
    require(block.number > auctionEndTime);
    require(secondPrice == 0);
    // Compute the second price
    uint best   = 0;
    for (uint i = 0; i < bidders.length; i++) {
      uint bid = bids[bidders[i]];
      if (bid > best) {
        secondPrice = best; best = bid;
      } else if (bid > secondPrice) {
        secondPrice = bid;
      }
    }
    emit Concluded(secondPrice);
  }
}
```

Patching the auction using Sirrah (After, encrypted)

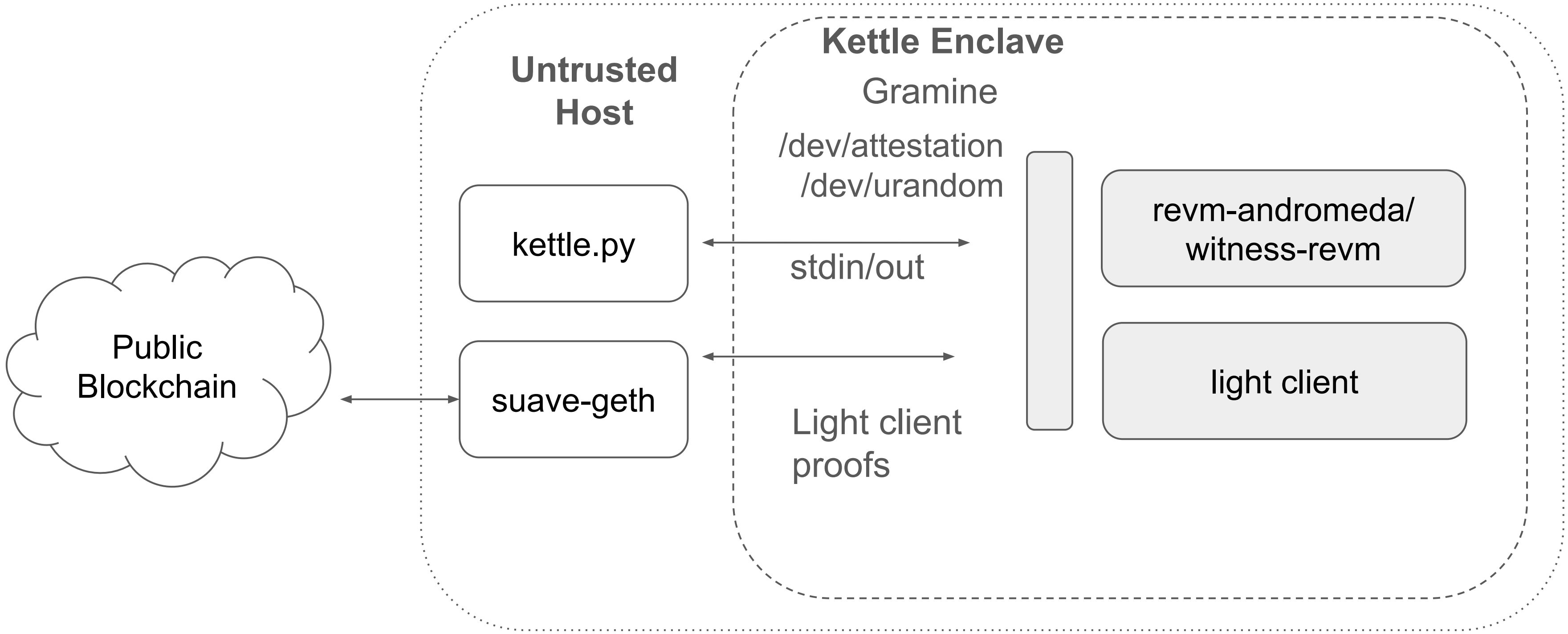
```
contract SealedBidAuction is AuctionBase, ... {
    ...
    mapping(address => bytes) encBids;
    function submitEncrypted(bytes memory ciphertext) public {
        require(block.number <= auctionEndTime);
        require(encBids[msg.sender].length == 0);
        encBids[msg.sender] = ciphertext;
        bidders.push(msg.sender);
    }
}
```

```
function finalize() public coprocessor {
  require(block.number > auctionEndTime);
  uint secondPrice_;
  for (uint i = 0; i < ciphertexts.length; i++) {
  uint bid = PKE.decrypt(xPriv(), ciphertexts[i]);
  ... // compute secondPrice_
  applyOnchain(secondPrice_) {
    secondPrice = secondPrice_;
    emit Concluded(secondPrice);
  }
}
```

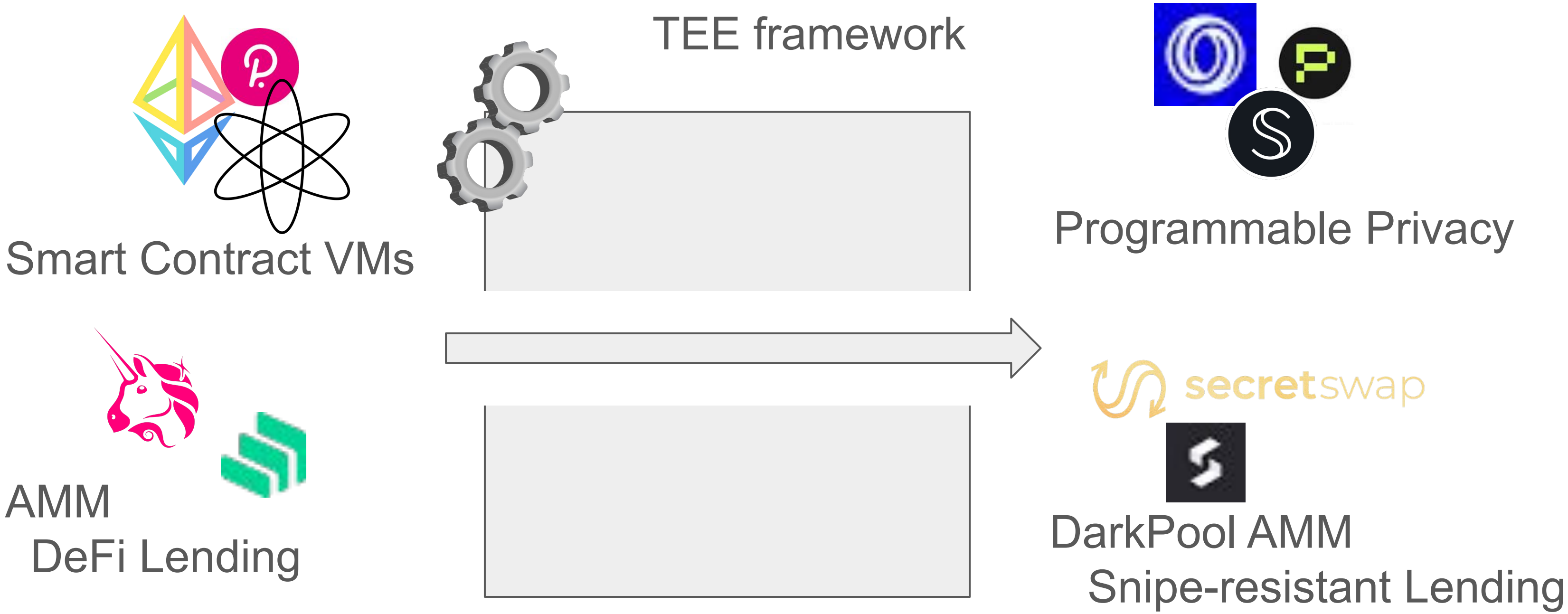
On-chain:
- Validates attestation
- Stores validated kettle addresses and public key

KeyManager.sol
Andromeda.sol

Off-chain:
- Generate private key
- Retrieve private key
- Generate attestation

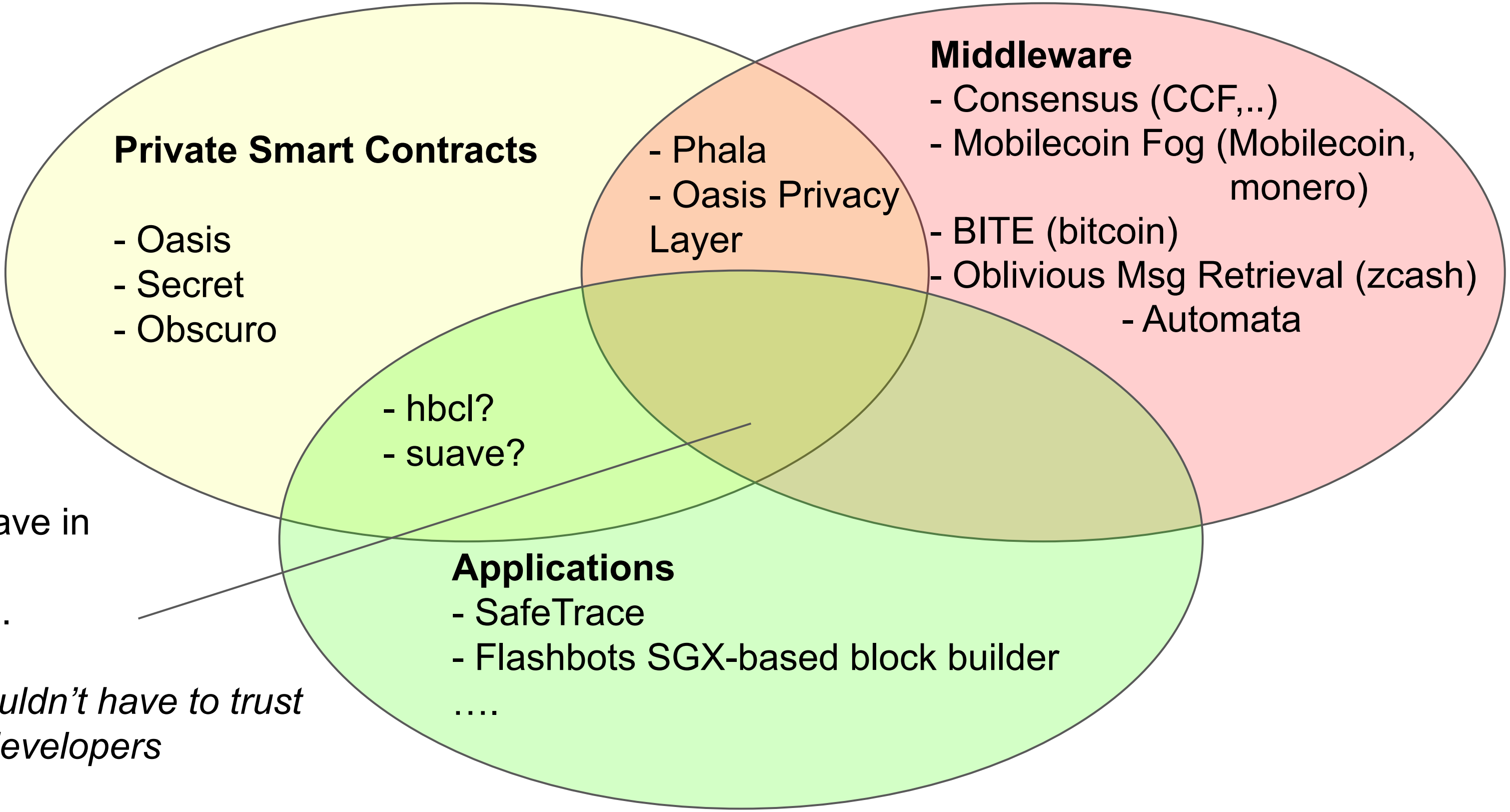


If you put a Smart Contract in a TEE, it gets upgraded with programmable privacy



TEE for web3 vs web2

Many sub-areas of Blockchain+TEE



What all of these have in common:
Disintermediation.

By design, you shouldn't have to trust the operators OR developers

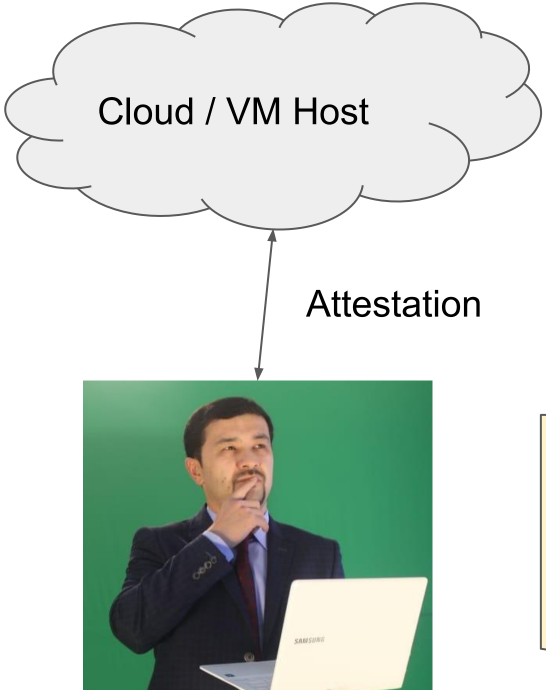
Cloud/Enterprise use case

- **Relying party:** the VM owner
- **Verifying an attestation** requires interacting with the enclave, e.g. over TLS
- **TCB Recovery** can be managed by the datacenter admin

Blockchain use case

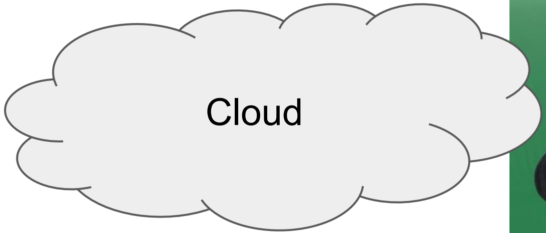
- **Relying party**: any user
- **Verifying an attestation** should be non-interactive, like verifying a certificate.
- **TCB Recovery** should be managed using through an trust-minimized process

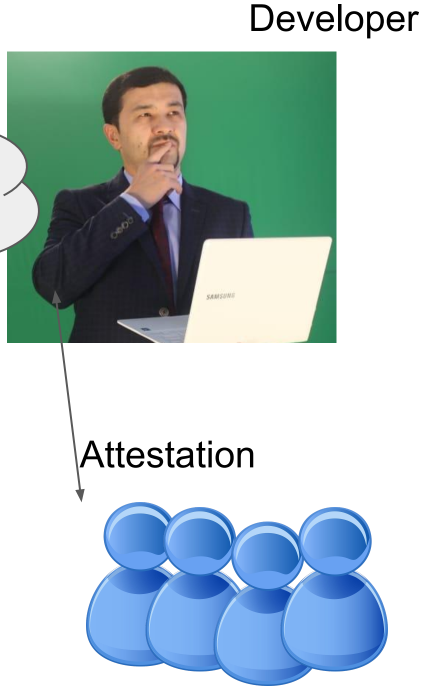
Cloud/Enterprise use case



Relying party:
Application Developer / VM Owner

Blockchain use case





Relying party:
Anyone/everyone in the public

AFAICT, requirements are being driven by Cloud/Enterprise side!
TDX, Nvidia, SEV

Security Time: Introducing “Controlled Channel Attacks”

https://github.com/amiller/gramine/commit/4763624

It’s not enough to “Run in the TEE”

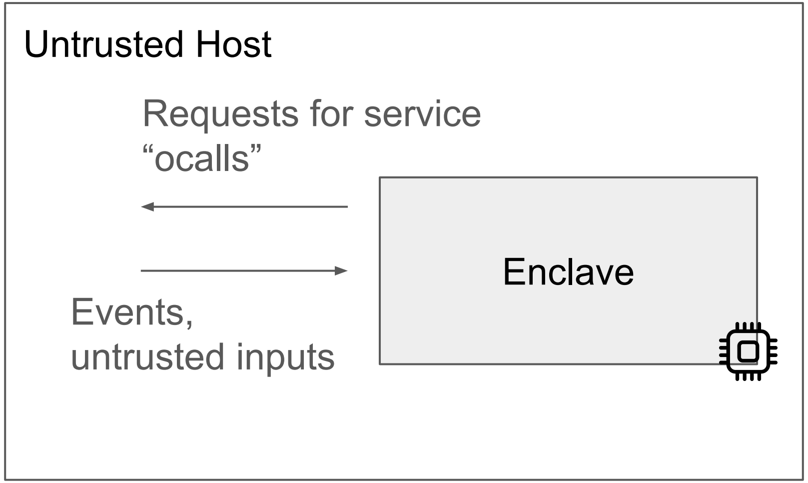
- Characterize and mitigate memory access pattern channels
- Prevent replay/grinding attacks and side channel amplification
- Avoid code-signing backdoors in the software upgrade process
- Rotate keys periodically for forward secrecy (prepare for vuln disclosures)
- Promptly reject vulnerable configurations after disclosures
- Make sure builds are reproducible
- Use “proof of cloud” to exclude TEEs in side channel labs

Proof of Cloud - a complement to hardware attestation

Thinking like a kernel/hypervisor attacker

Our threat model is a host that wants to learn more than they should about the enclave. There’s a gap between the default behavior (act like an ordinary OS) and what you can get away with (act like a “debugging tool”).

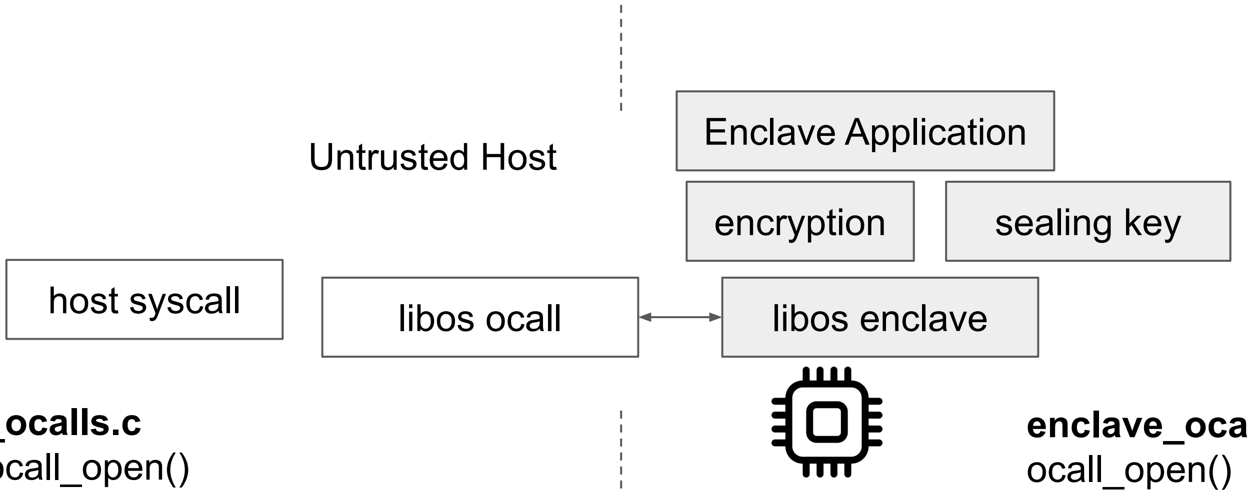
Between “running in SGX” and “is secure”



Channels controlled by untrusted OS:

- Interrupts
- **System calls**
- Page table entries
- …

How Gramine implements an encrypted filesystem



host_ocalls.c
sgx_ocall_open()
….
sgx_ocall_pwrite()

enclave_ocalls.c
ocall_open()
….
ocall_pwrite()

```
  { type = "encrypted", path = "/output/", uri = "file:output", key_name = "_sgx_mrenclave" },
]
```

“Spicy PrintF” demonstration

Sometimes, you can undermine an application just by monitoring an obvious “**controlled channel**” interface.

For example, with encrypted files we can modify the Gramine “ocall” to show the 4KB block being accessed.

Populating a user database

Making a data-dependent access

Controlled Channel Attacks - references

Shout out to this 2015 paper "Controlled Channel attacks" for explaining how page-fault oracles undermine legacy apps run in a TEE.

They can reconstruct a document in Word Processor from font renderer, or from spellcheck https://youtube.com/watch?v=fwUaN5ik8zE

https://ieeexplore.ieee.org/document/7163052
https://www.youtube.com/watch?v=fwUaN5ik8zE

These are still applicable today!

See also: https://github.com/jovanbulck/sgx-pte SGXonerated paper
https://www.comp.nus.edu.sg/~prateeks/papers/PigeonHole.pdf

Takeaways: Gramine and controlled channel attacks

Legacy applications that automatically “*run in Gramine/ SGX*”
are ***not*** automatically secure against controlled-channel attacks.

These aren’t even side-channels, they are documented you just have to choose to look at them.

Possible mitigations:

1. Design your application in a data-independent way
2. Automatically apply “ORAM” to make the queries data-independent
3. Abort if a page fault is detected during a transaction when it’s unexpected

Open Research challenges

- **End to end software chain for attestation.** Not yet fully implemented. Techdebt

- **Root of trust remains unsolved.** Decentralized open hardware?

- **Governance and upgrades.** Yet to define a best practice

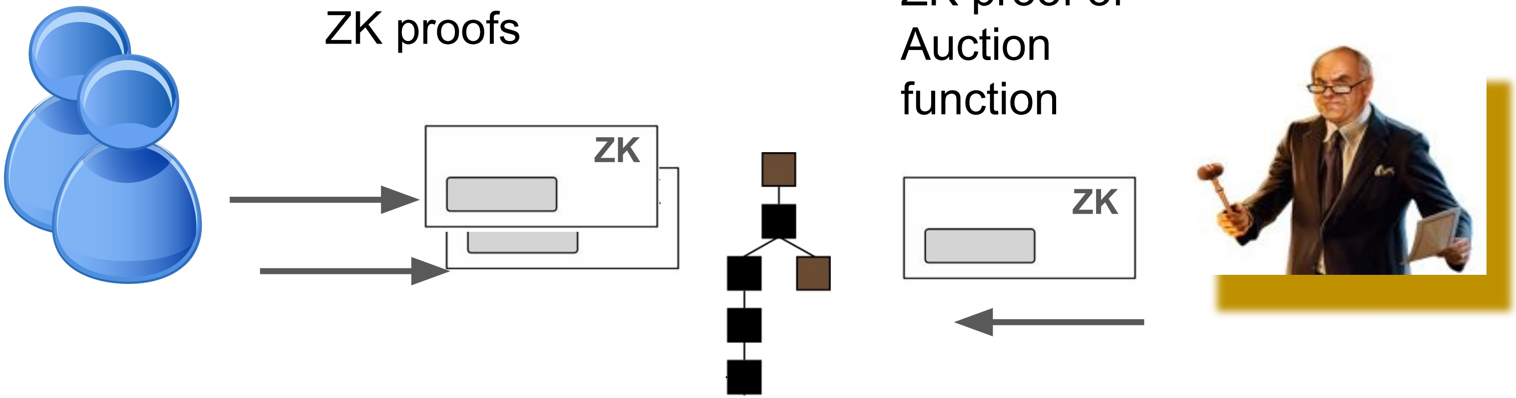
- **Integrating ORAM** and characterizing **side channels** remains open

Thank you!

Where ZKP falls short - sequencer has to see everything

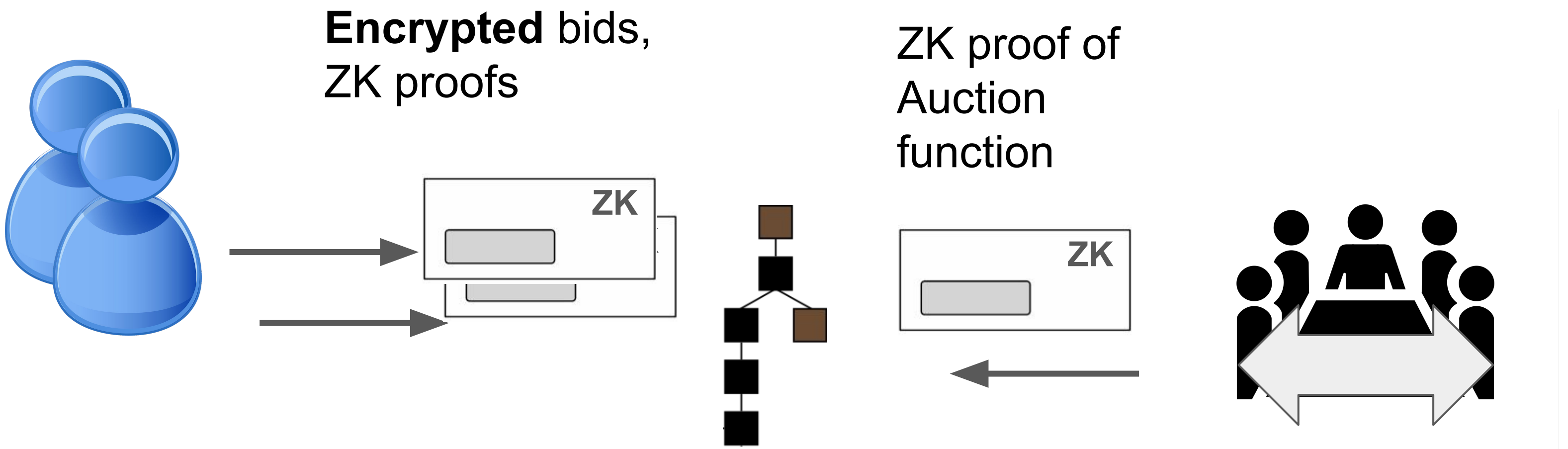
Encrypted bids,
ZK proofs

ZK proof of
Auction
function



But, the auctioneer must be able to **decrypt** the transactions in order to apply the auction computation.

MPC tolerates faults, but does nothing about collusion

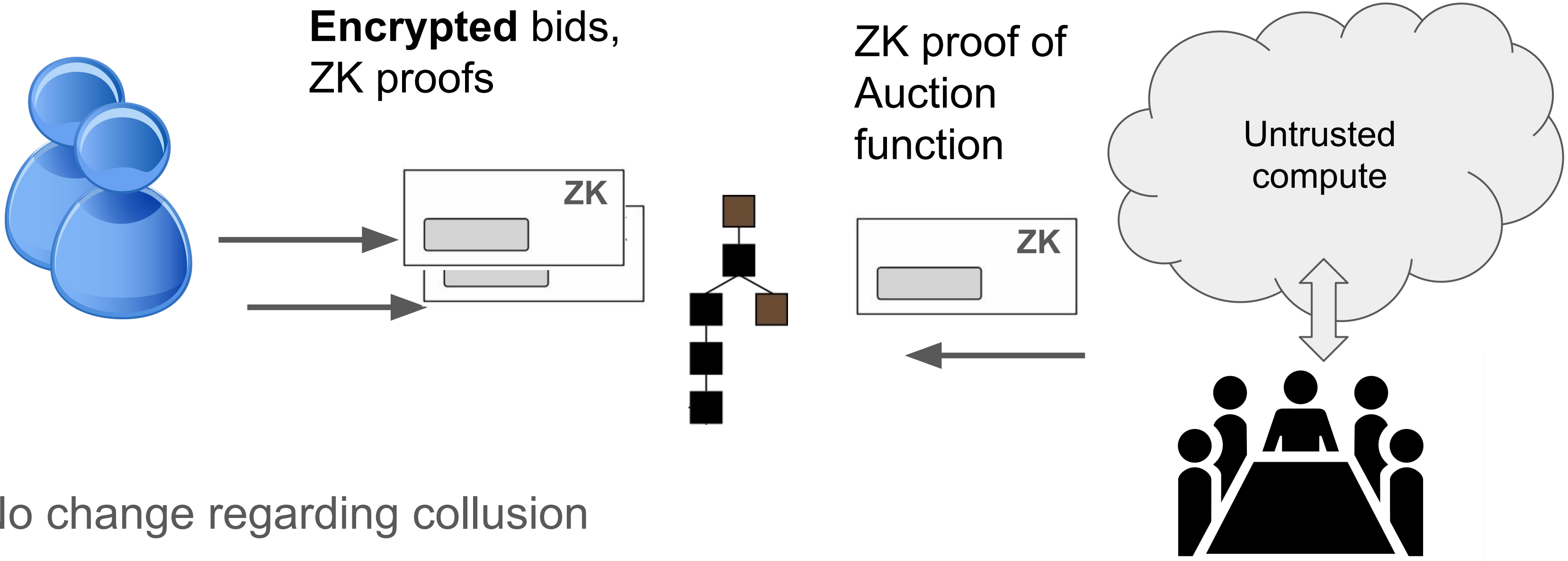


A “multisig” of key holder nodes

If a quorum of key holders collude, they could decrypt everything. Difficult to disincentivize, as it produces no evidence

FHE turns I/O bottleneck of MPC into compute tradeoff

Untrusted compute does the work



No change regarding collusion

Multisig only shows up to decrypt

ZK,MPC,TEE Design Space