Structuring Computation for Privacy-Preserving Apps

Wei Dai
@_weidai
Bain Capital Crypto
April 21, 2022
Do you have prior knowledge of zero-knowledge proofs?
Zero-Knowledge Proofs (ZKPs)

**Properties**
- Succinct: \( \pi \) is short, verifier runtime is “small”
- Non-interactive: Only one message from P to V
- Transparent: No trusted setup
- Universal: No per-circuit trusted setup

**Security**
- Completeness: It works!
- Zero knowledge: Verifier learns nothing about \( w \)
- Knowledge soundness: Prover knows \( w \)

**History**
- Studied since the late 1980s
- Recent explosion, due to Z\{ero\}cash, Groth16, Sonic, Marlin, Plonk, ...

**Typical provers:** User wallets, proving services

**Typical verifiers:** Chains, EVM contracts

\( C \) - arithmetic circuit, “program execution”
\( x \) – public input, \( w \) – secret witness

\( \pi \) for \( x \sim \) “I know \( w \) such that \( C(x, w) = 1 \)”
Do ZKPs solve all privacy problems for blockchain apps?
(Think Uniswap, Aave, NFT auction)
No.
Agenda of this talk

1. **ZK** is in contention with **on-chain composability** and **shared states**.

2. **ZK** for private states, **transparent compute** for **shared states**.

3. **Threshold FHE** for **on-chain confidential compute on shared state**.

4. Framework to program **transparent, ZK, FHE** computation.
State Machines

Transition function computes \((s_{t+1}, o_{t}) = f(s_t, i_{t})\)

Blockchain State: \(s_t = (B_0, ..., B_i)\)

Blockchains

Smart Contracts

```solidity
contract LiquidityPool {
    uint public reserveX, reserveY;
    function swapXtoY( ... ) public {
        ...
    }
}
```
ZK State Machines Execution (Zexe / Aleo / Mina Snaps)

**Consensus updates $st_0$ to $st_1$ only if $\pi$ is valid**, i.e. $V_f(st_0, st_1, output_0, \pi) = 1$.

**Problem:** shared state give rise to race conditions.

**Alice:** $(st_0, x_A) \rightarrow st_A \text{ w/ } \pi_A$

**Bob:** $(st_0, x_B) \rightarrow st_B \text{ w/ } \pi_B$

Only one state update can be performed.

ZKP smart contracts do not support shared application state due to race conditions.

ZK prover for $f$

input

Chain

User

Update check w/ ZK verifier $V_f$

$\pi$ – zk proof certifying that “I have input so that $(st', output) = f(st, input)$”
On-chain vs off-chain apps

**On-chain apps**

- Chain
- App A
- App B
- App C

**Scalability**
**Privacy**
**Default composability**

**“Full-ZK” Apps**

- Chain
- App A
- App B
- App C

**Scalability**
**Privacy**
**Opt-in off-chain composability**
Structuring computation: Transparent vs ZK

**Contract** MyContract:

```plaintext
public st

DoStuff(cm, π):
  RangeCheck.verify(...)  
```

**Contract** ZCashOrchard:

```plaintext
public MT
  // Insert-only Merkle tree
public NS // nullifiers
Process(tx, π):
  Action.verify(MT.rt, tx, nf; π)
  Assert(nf ∉ NS)
  Ins(tx, MT); Ins(nf, NS)
```

**Contract** AleoApp:

```plaintext
public st // record

Update(st, st’, π):
  Update.verify(...)
```

- **On-chain**
- **Off-chain**

**ZKCirc** RangeCheck(cm; x, r):
  - Assert (cm = Commit(x; r)
  - Assert (x < k)

**ZKCirc** Action(rt, tx, nf; sk, ...):
  - “tx is valid spend against rt”
  - “tx declare correct value change”
  - “tx declare correct nf”

**ZKCirc** Update(st, st’; x):
  - Assert (st’ = f(st, x))

- ZKP touches no contract state
- New state does not invalid old proofs
- ZKP re-write contract state
Agenda of this talk

1. **ZK** is **in contention** with **on-chain composability** and **shared states**.

2. **ZK** for private states, **transparent compute** for **shared states**.

3. **Threshold FHE** for **on-chain confidential compute** on **shared state**.

4. Framework to program **transparent, ZK, FHE** computation.
Third type of computation?

Replicated on-chain
No privacy
Shared state

ZK off-chain
Supports private state and inputs
No shared state

Private input to confidential shared state?
Same trust assumption as consensus?

A: YES! w/ Multi-party computation (MPC) or Threshold Fully Homomorphic Encryption (FHE)
Fully Homomorphic Encryption

FHE: Computation over encrypted data

Problem: decryption key $dk$ is a master secret!

$(Kg, Enc, Dec, Eval)$

- **FHE [Gentry09]**: $C$ is **any** circuit
  - Active area of R&D in academia and industry. Efficiency improving.
  - Many variants: leveled [GSW, FV, BGV], per-gate bootstrapping [FHEW, TFHE]
  - “Current” state-of-the-art for binary FHE $2^{12}$ binary gates (xnor, mux) per second on GPU [cuFHE, nuFHE].
Threshold Cryptography

Liveness holds if k out of n servers cooperate

No security broken even if k – 1 servers collude

Threshold cryptography particularly applicable to blockchains / BFT protocols w/ k \sim 2n/3.

Threshold signatures

Dfinity: “Chain key cryptography”

Biconomy, Webb, Lit, ...

Threshold encryption / decryption

Anoma/Ferveo Penumbra

We know of protocols to maintain “Shamir threshold secret shares” among a dynamic set of nodes.

• Distributed key generation [DYXMK21, Groth21]
• Dynamic proactive secret-sharing [MZWLZJS19, GKMPS21, Groth21]
FHE with **Threshold Decryption**

\[(Kg, Enc, ThDec, Eval)\]

- Achievable with Shamir secret shares
- **Why?** *Consensus-based, programmable selective information disclosure*
  - AMM spot price
  - Trade validity
Decrypt part of the encrypted state `est` that is explicitly marked for decryption.

Can be replicated by any BFT-type consensus algorithm.
- Decryption available with a delay
- For privacy and safety, decryption => finalization

Rest of the talk: Assume a BFT-type blockchain system with fixed FHE public key `pk` that can replicate state machine with threshold decryption.

Q1: How to program this state machine?
Q2: Why is this useful?
<table>
<thead>
<tr>
<th>Types of Computation</th>
<th>Transparent On-chain</th>
<th>ZK Off-chain</th>
<th>Confidential On-Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EVM</td>
<td>Groth16</td>
<td>MPC</td>
</tr>
<tr>
<td></td>
<td>Solidity</td>
<td>Sonic</td>
<td>SEAL</td>
</tr>
<tr>
<td></td>
<td>Wasm</td>
<td>Marlin</td>
<td>FHE</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td>Palisade</td>
</tr>
<tr>
<td></td>
<td>Substrate</td>
<td></td>
<td>Concrete</td>
</tr>
<tr>
<td></td>
<td>ABCI</td>
<td></td>
<td>Supporting Shamir keys</td>
</tr>
</tbody>
</table>

- **Transparent On-chain**
  - EVM
  - Solidity
  - Wasm
  - Substrate
  - ABCI

- **ZK Off-chain**
  - Groth16
  - Sonic
  - Marlin
  - STARK / Plonk
  - Aztec
  - ZK-Garage
  - Halo2
  - Plonky{2}
  - Jellyfish
  - Risc0

- **Confidential On-Chain**
  - MPC
  - SEAL
  - FHE
  - Palisade
  - Concrete
  - Supporting Shamir keys
  - FHEW
  - GSW
  - Implementation?
Towards a Unified Framework: PESCA

Privacy-Enhancing Smart-Contract Architecture

All computation written in pseudocode!

PESCA

Transparent On-chain

ZK Off-chain

Confidential On-Chain
**Contract** ExampeContract:

**Public Func** ProcessA(input): // executed on-chain

`ValidateA.verify(input, π)`

`state' = ComputeOverA(enc_state, input)`

**Async** `d = ThDec():`

...

**User Func** GenerateA(): // executed off-chain

`input = ...`

`π = ValidateA.prove(input; ...)`

**ZK Circuit** ValidateA(): // proved off-chain, verified on-chain

**FHE Circuit** ComputeOverA(): // executed on-chain
Rest of the Talk: Privacy-preserving CFMM and Auctions

- ZCash-like ZK Circuits for token accounting
- Merkle tree and nullifier set
- Transparent Application logic
- Confidential inputs
- FHE circuits for application logic on confidential states
- Threshold decryption
- Information release
Token with composable private usage

Idea: modify existing ZCash orchard design: value commitment => value encryption.

**Contract** ShieldedToken:

public MT, NS // Merkle tree of notes and nullifier set

**ZK Circuit** Action(tx; ...):

\[ v = \ldots \]

\[ \text{Assert} \ (tx.ev == \text{FHE.Enc}_{pk}(v, r)) \]

**Private Func** Process (tx, \( \pi \)):

\[ \text{Action.verify}(\ tx; \ \pi \ ) \]

“Add spent notes nullifiers to NS”

“Add new notes commitment to MT”

**User Func** GenerateAction():

\[ tx = \ldots \]

\[ \pi = \text{ValidateA.prove}(tx; \ldots) \]
Constant Function Market Makers

Want to buy X

Want to buy Y

Liquidity providers:
Hold a position in both X and Y.

Privacy-preserving: trade origins and amounts are not revealed.

Information leakage:
- # of trade requests executed / dropped
- Spot price that is released **programmatically**
Privacy-preserving CFMM

**Contract** CFMM extends ShieldedToken:

```java
private est // FHE encrypted state encrypting reserves (x, y)
```

**FHE Circuit** `Trade((x, y), (dx, dy))`:

- If \((x + dx)(y + dy) \geq xy\) then Return \(((x + dx, y + dy), 1)\)
- Else Return \(((x, y), 0)\)

**Pub Func** `Trade(fund, refund, out)`:

- `Process(fund)`
- `Balance.verify(fund, refund)`
- \((est, eb) \leftarrow \text{FHE.Eval}(\text{Trade}, est, (tx_{\text{fund.ev}}, tx_{\text{out.ev}}))\)
- `Async b \leftarrow \text{ThDec}(eb)`:
  - If \(b = 1\) then `Process(out)`
  - Else `Process(refund)`
Preventing malicious decryptions

**Contract** FHEBase:

```python
InitFHEState(est, πs): // FHE states must be initialized via this method
for each (eb, π) in zip(est, πs):
    InitCheck.verify((this, eb), π)
```

**ZK Circuit** InitCheck(ContractID, eb; b, r):

```python
Assert (eb = FHE.Enc_{pk}(b; r))
```

---

**Contract** TargetContract

- public est

**Contract** AttackContract

- public est
- `Func DecryptAll: ...`

---

**Attack:** want to decrypt `est`, make new contract C and program C to release `est`.

**Mitigation:** FHE initial states and all FHE input needs **accompanying ZKPs** particular to each contract.
Privacy-preserving Sealed-bid Auctions

Sealed-bid: Bids not revealed to other bidders

Privacy-preserving: bids not revealed, to anyone, even after the auction is over.

Information leakage:
- Item seller learns settling price.
- Auction winner obtains item.
- All other bidders only learn that they did not win.
Privacy-preserving Sealed-bid Auctions

**Contract** FPSBA *extends ShieldedToken*:

private emax, ej // FHE encrypted state encrypting max_bid and winner index

**FHE Circuit** Bid[j]( (max_bid, index), bid ):
- If (bid > max_bid) then Return (bid, j)
- Else Return (max_bid, index)

**Pub Func** Setup( emax, ej ):
- j = 0; “state initiation checks”

**Pub Func** Bid( bid, refund, payout ):
- j += 1; “balance checks”; Process(bid)
- (emax, ej) = FHE.Eval(Bid[j], (emax, ej), bid.ev)

**Pub Func** Finalize():
- Async j = ThDec(ej):
  - Process(payout_j)
  - \( \forall i \neq j: \text{Process(refund}_i) \)
Closing Remarks

• Paper on PESCA to appear.

• We are **hiring**! If you are interested in benchmarking and implementation of ZK, FHE, or threshold cryptography, contact me!