CASA
CYBER SECURITY IN THE AGE OF LARGE-SCALE ADVERSARIES

Heinz Nixdorf Stiftung

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

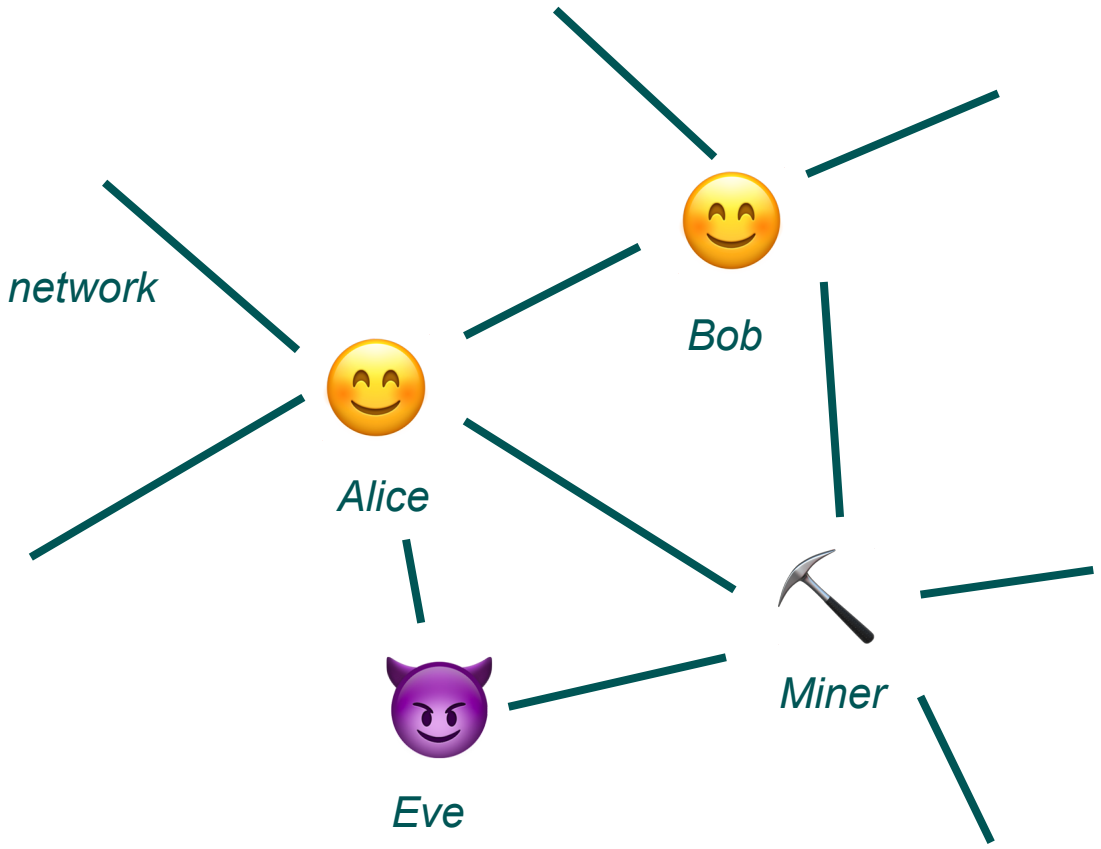
Combining Program and Protocol Analysis for Frontrunning Resistance in Smart Contracts

FCS 2026

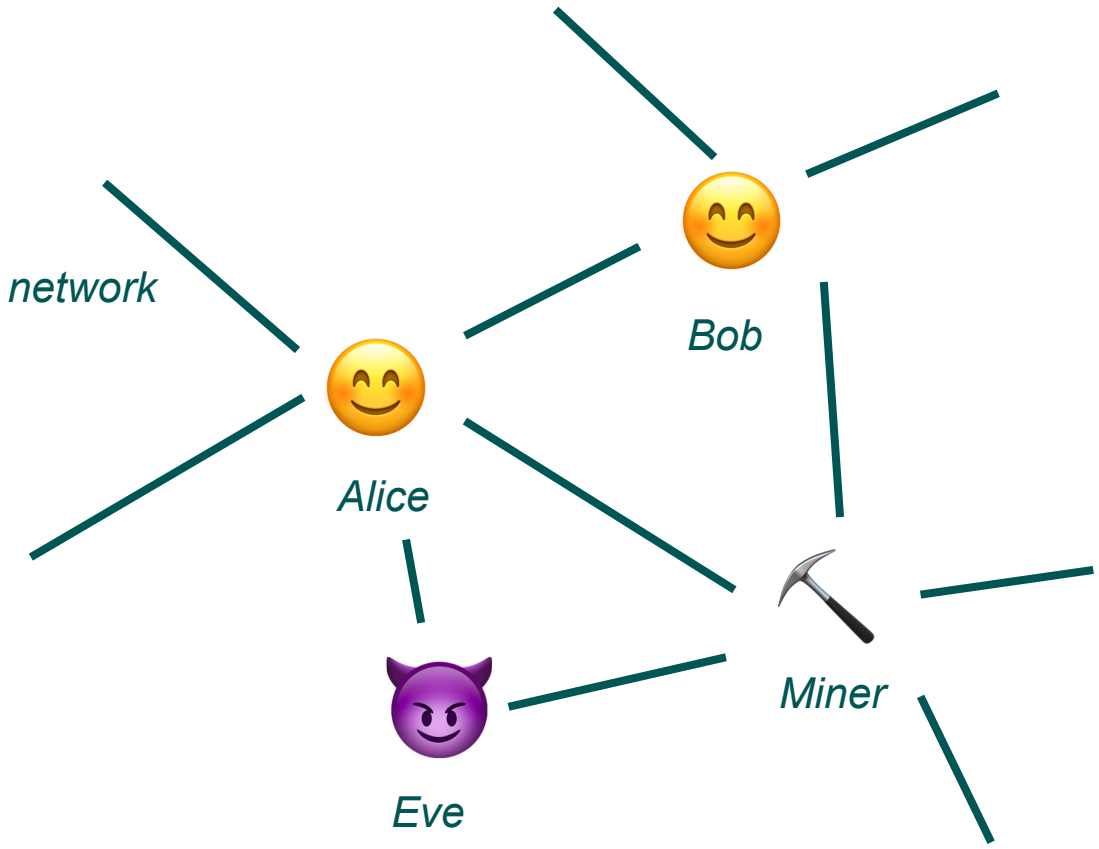
Sebastian Holler, **Yan Farba**, Clara Schneidewind

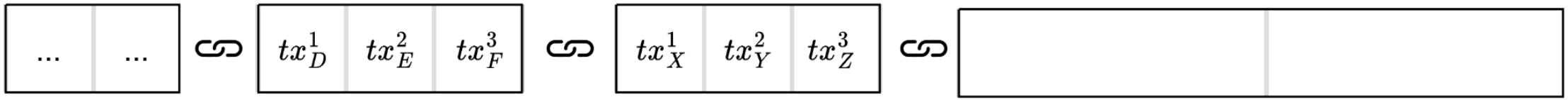
Introduction

Blockchain 101



Blockchain 101

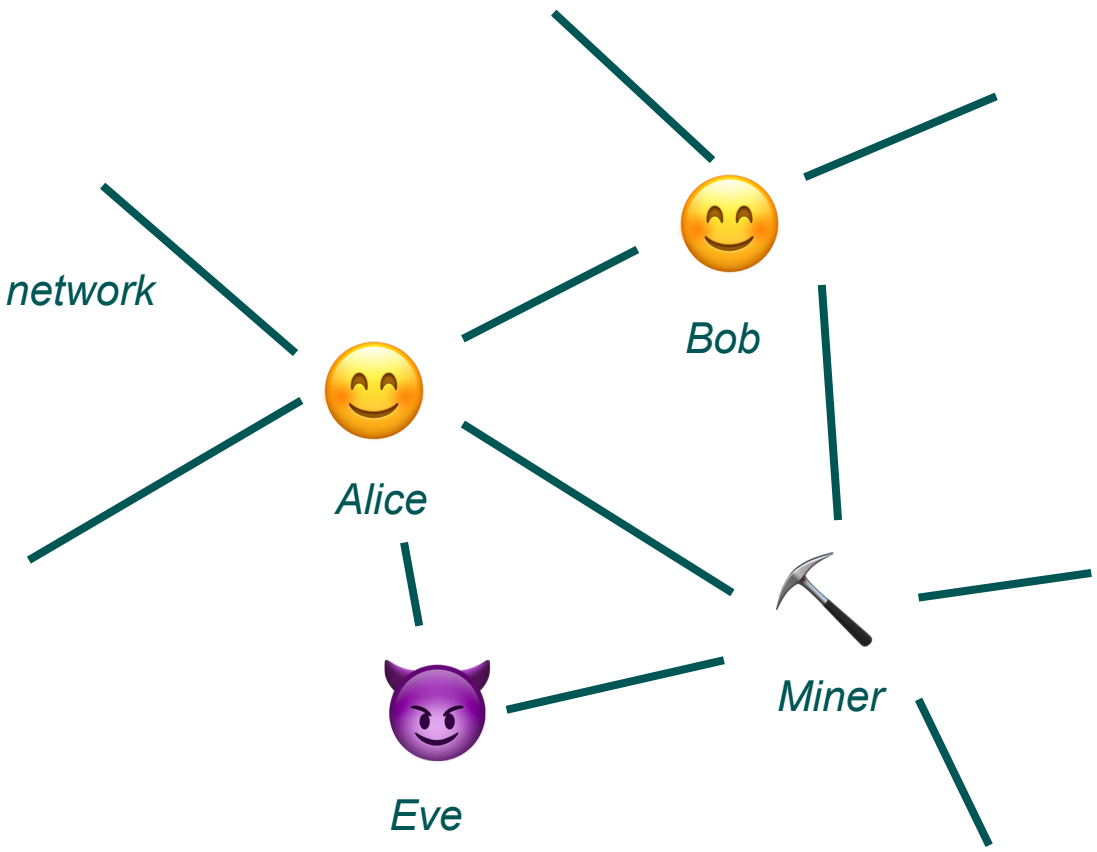


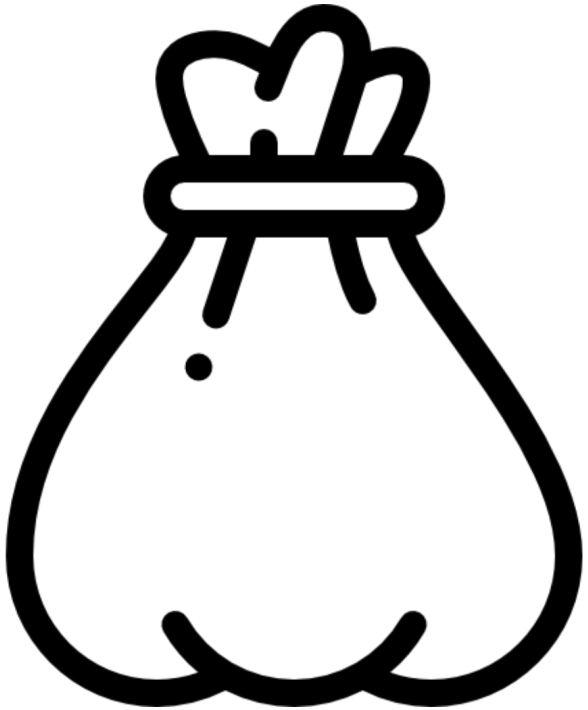


Blockchain

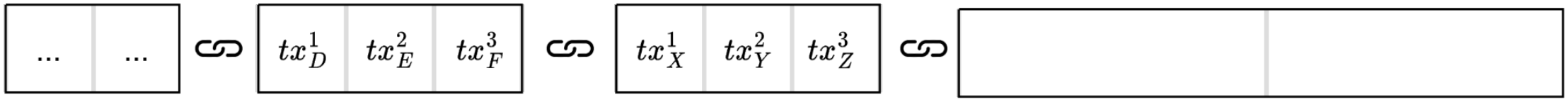
upcoming block

Blockchain 101





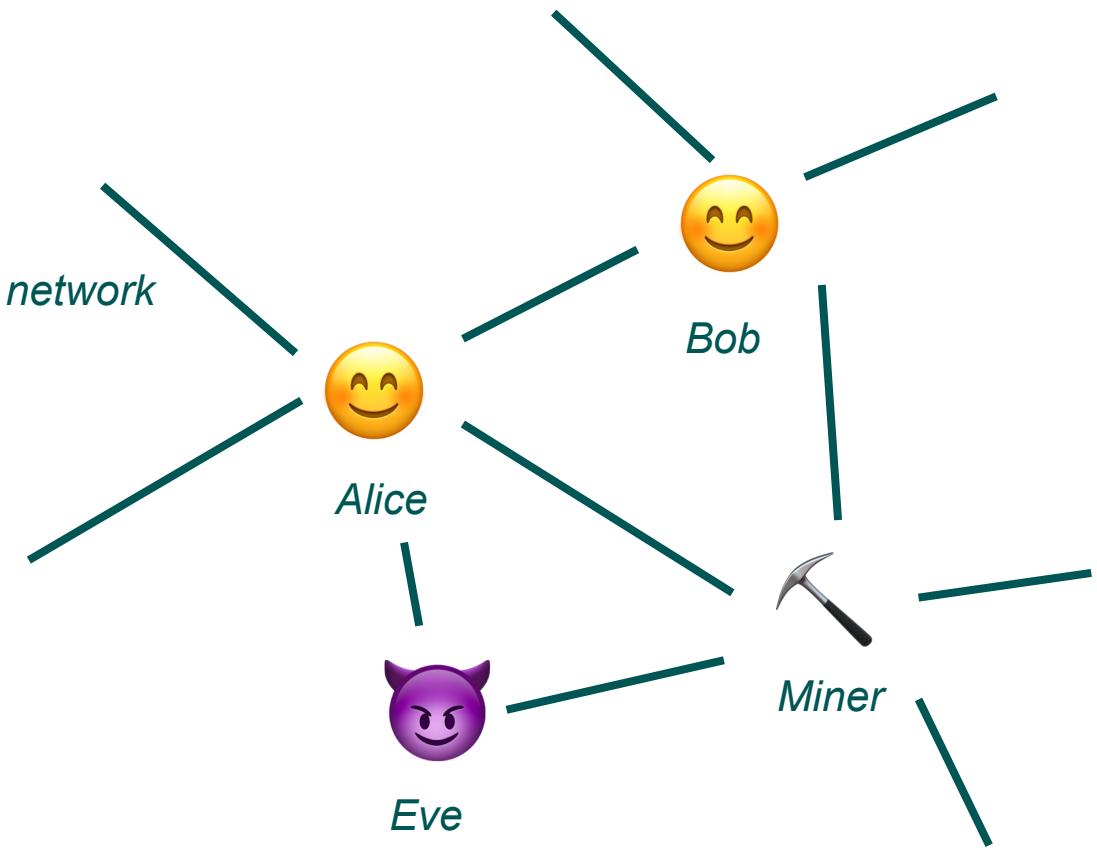
Mempool

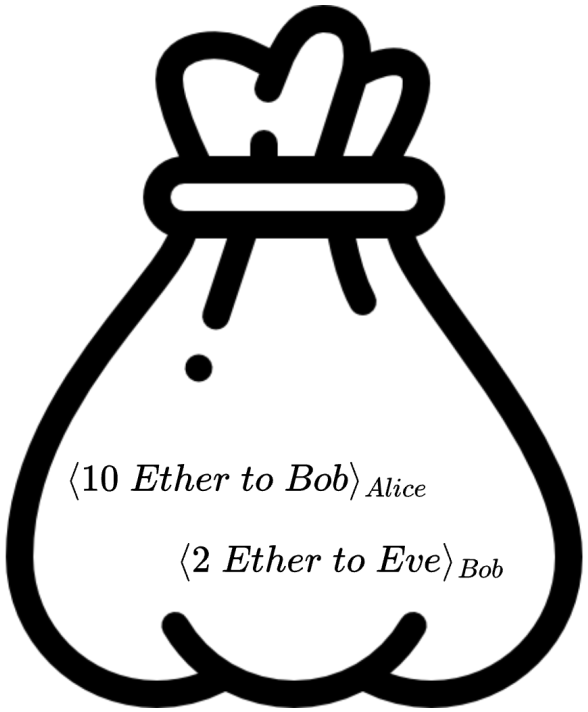


Blockchain

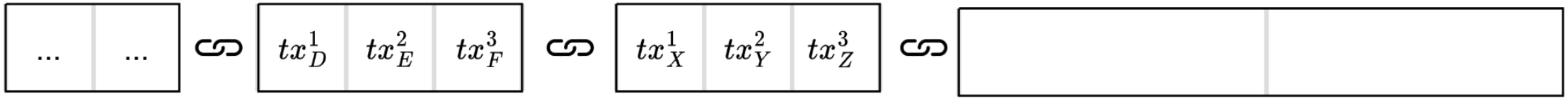
upcoming block

Blockchain 101





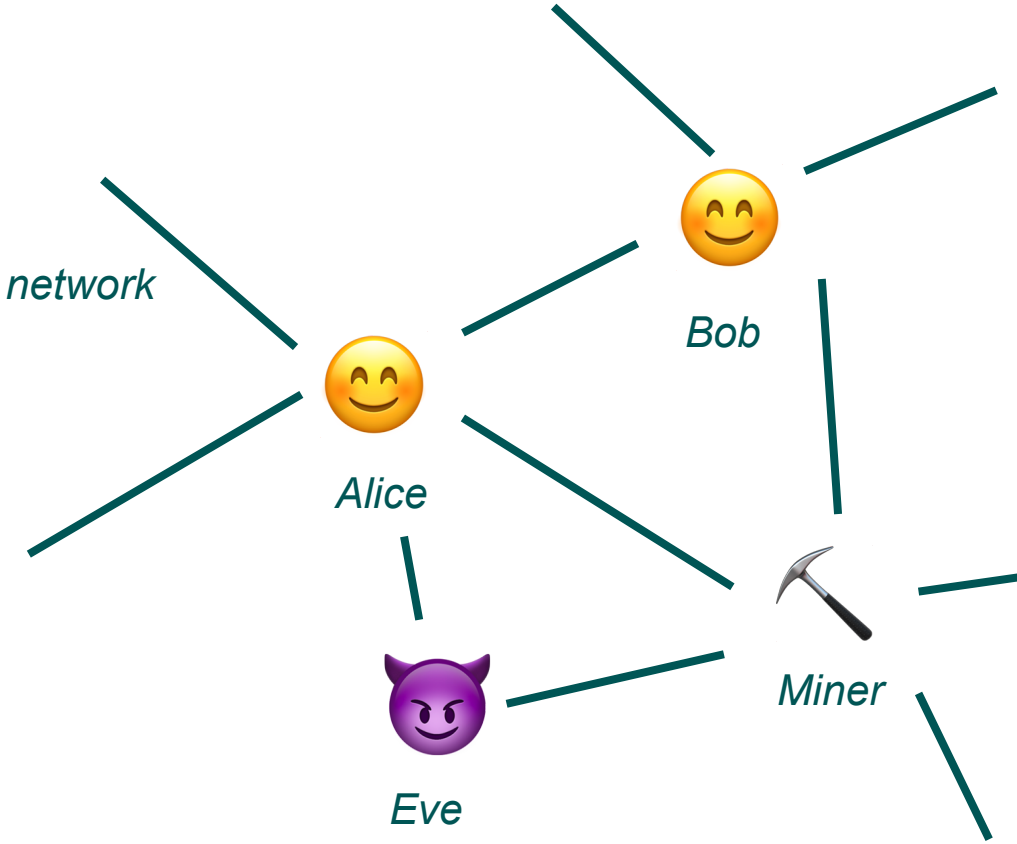
Mempool

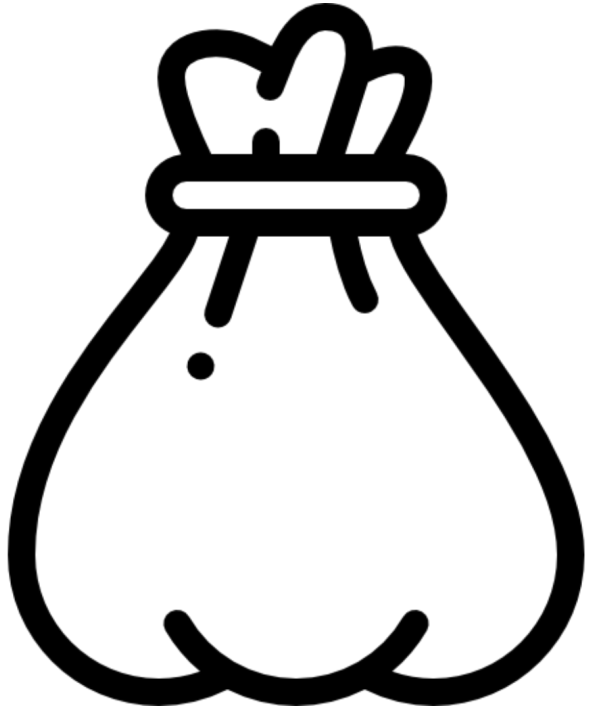


Blockchain

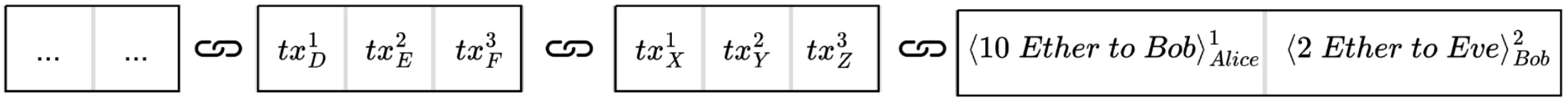
upcoming block

Blockchain 101





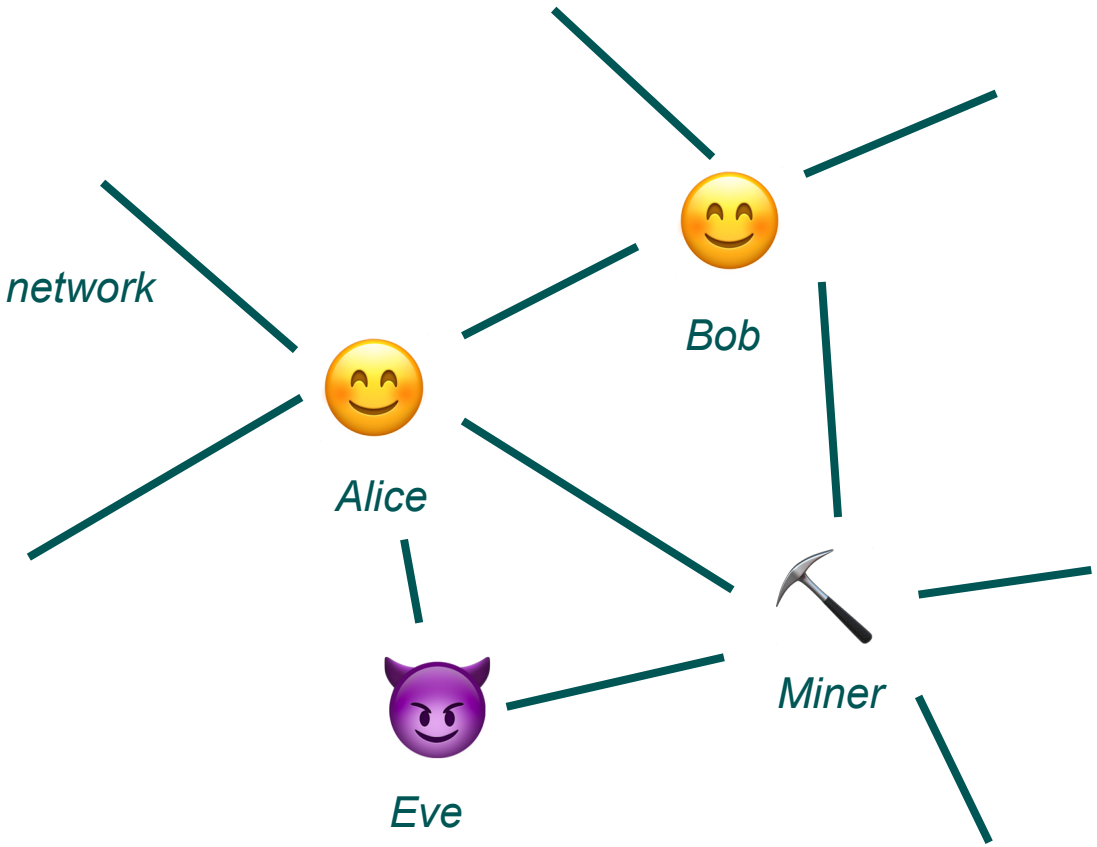
Mempool

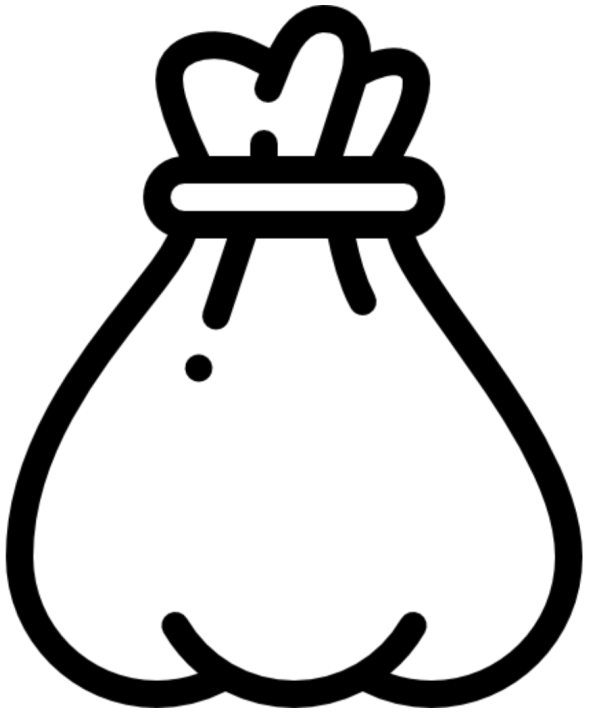


Blockchain

new *block*

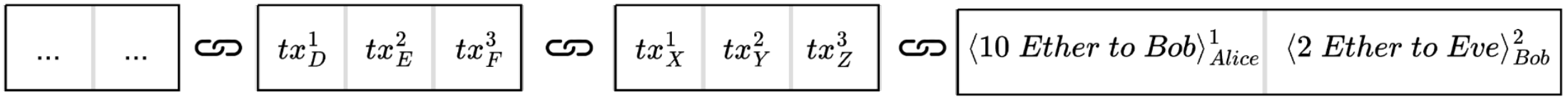
Blockchain 101





Mempool

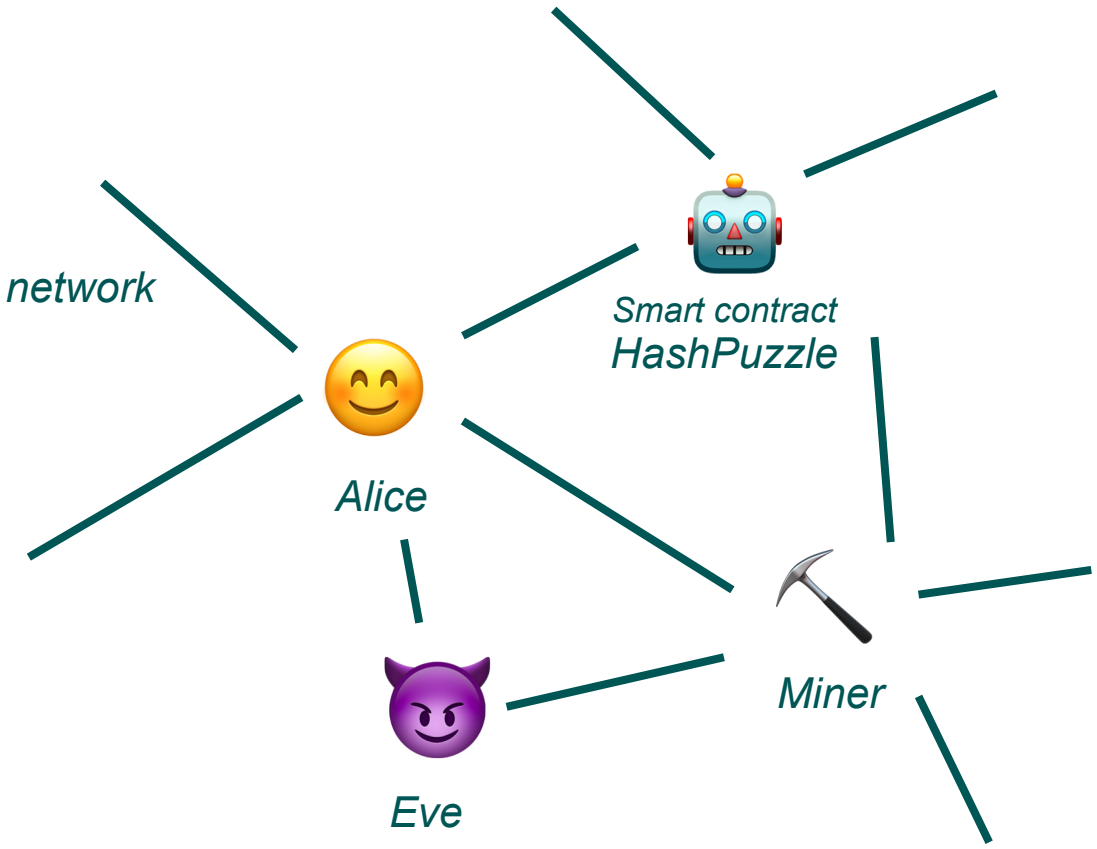
We focus on **Ethereum** blockchain, which supports **Smart Contracts**.

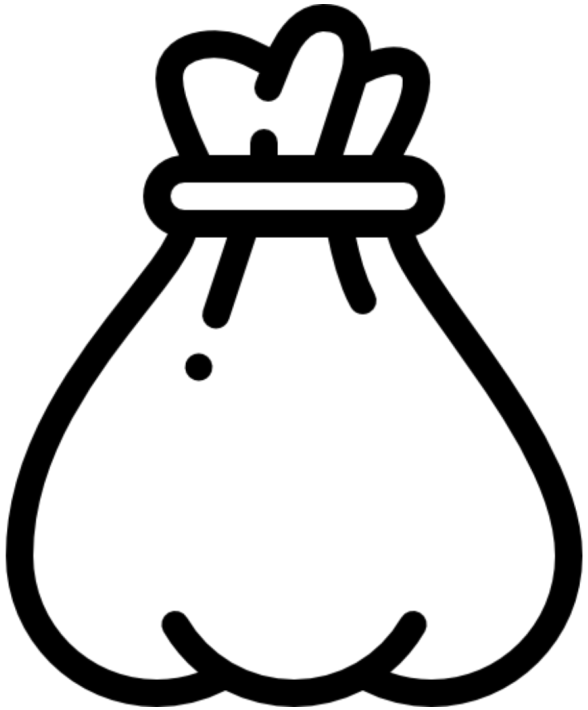


Blockchain

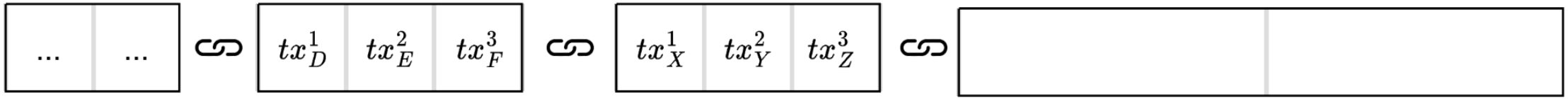
***new** block*

Smart Contracts 101





Mempool



Blockchain

upcoming block

Smart Contracts 101

```
contract HashPuzzle {
    bytes32 public challenge;
    address public winner;
    function solve(bytes32 solution) {
        ...
    }
}
```

Written in Solidity.

Smart Contracts 101

```
contract HashPuzzle {
    bytes32 public challenge;
    address public winner;
    function solve(bytes32 solution) {
        require(
                sha256(solution) == challenge);
        ...
    }
}
```

Written in Solidity.

Smart Contracts 101

```
contract HashPuzzle {
    bytes32 public challenge;
    address public winner;
    function solve(bytes32 solution) {
        require(
                        sha256(solution) == challenge);
                payable(msg.sender)
                        .transfer(this.balance);
        this.winner = msg.sender;
    }
}
```

Written in Solidity.

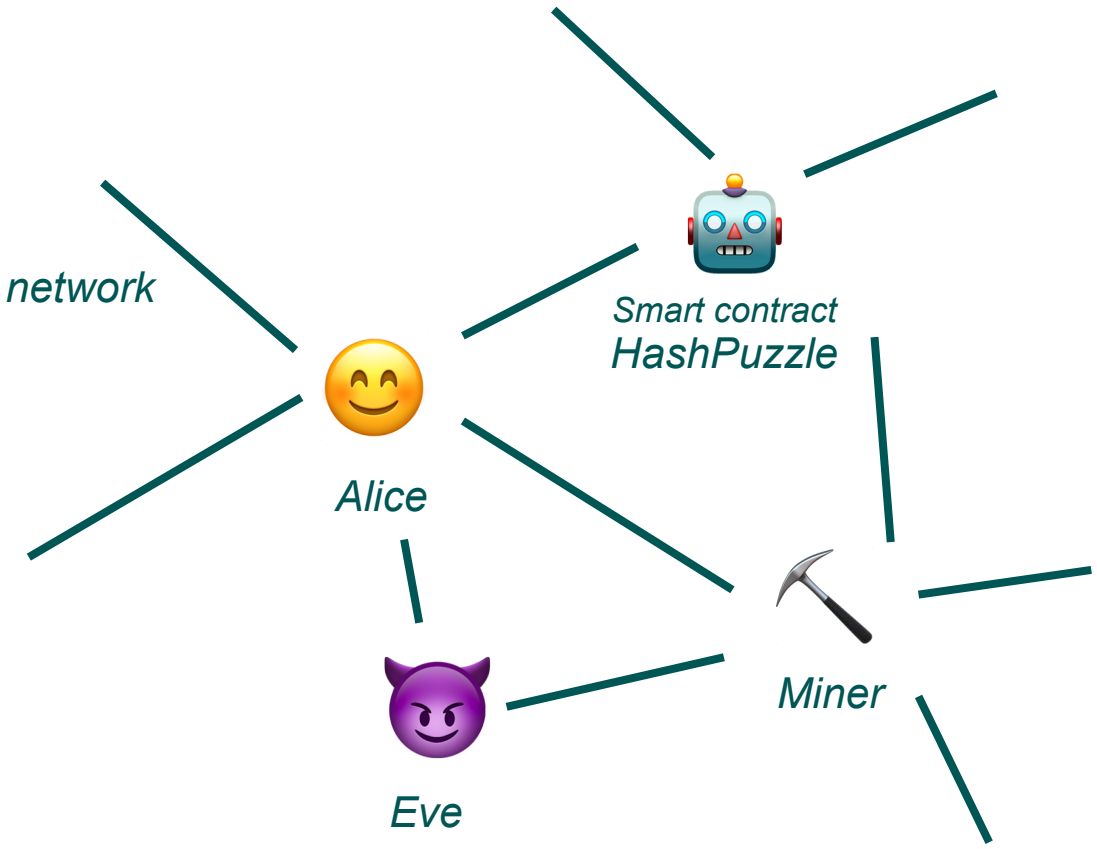
Smart Contracts 101

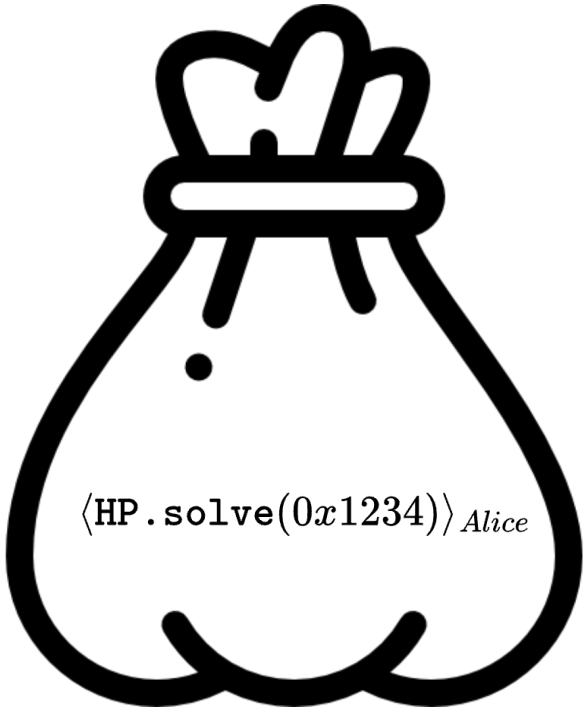
```
contract HashPuzzle {
    bytes32 public challenge;
    address public winner;
    function solve(bytes32 solution) {
        require(
                        sha256(solution) == challenge);
                payable(msg.sender)
                        .transfer(this.balance);
        this.winner = msg.sender;
    }
}
```

Written in Solidity.

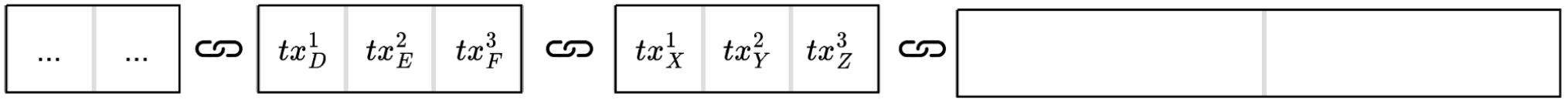
The smart contracts' **state** (state variables, code and balance) is public**.**

Smart Contracts 101





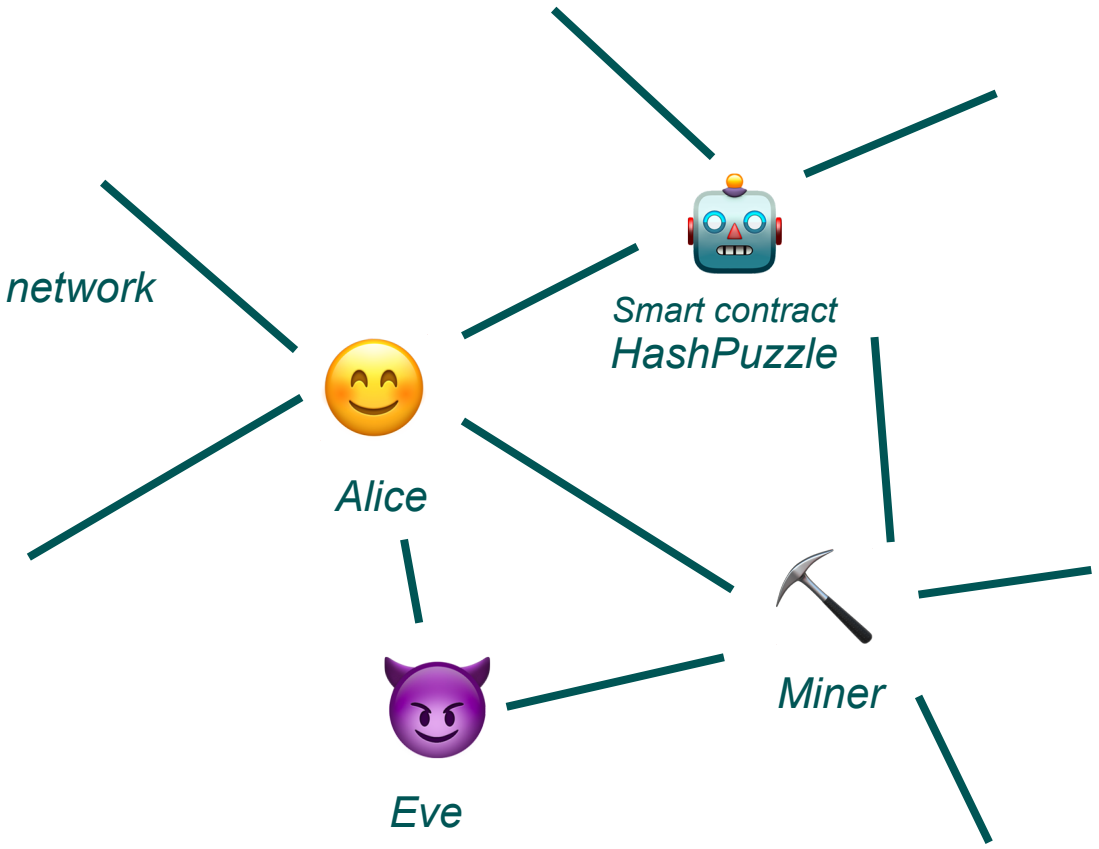
Mempool

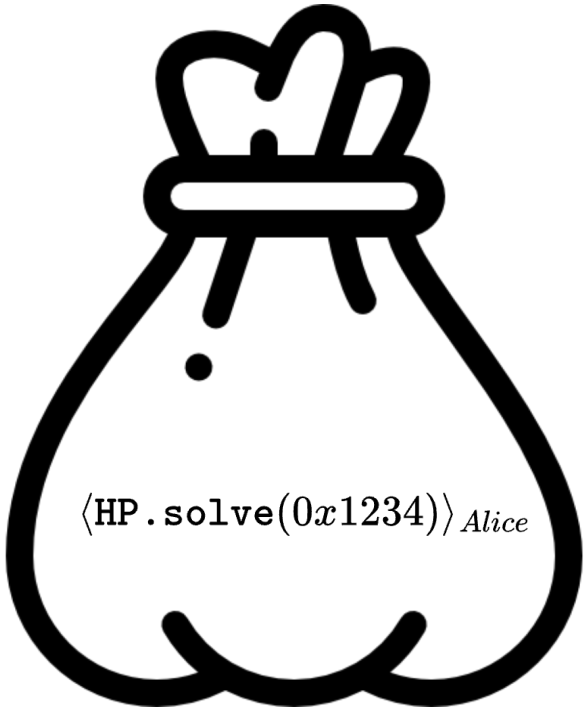


Blockchain

upcoming block

Blockchain as a Transition System





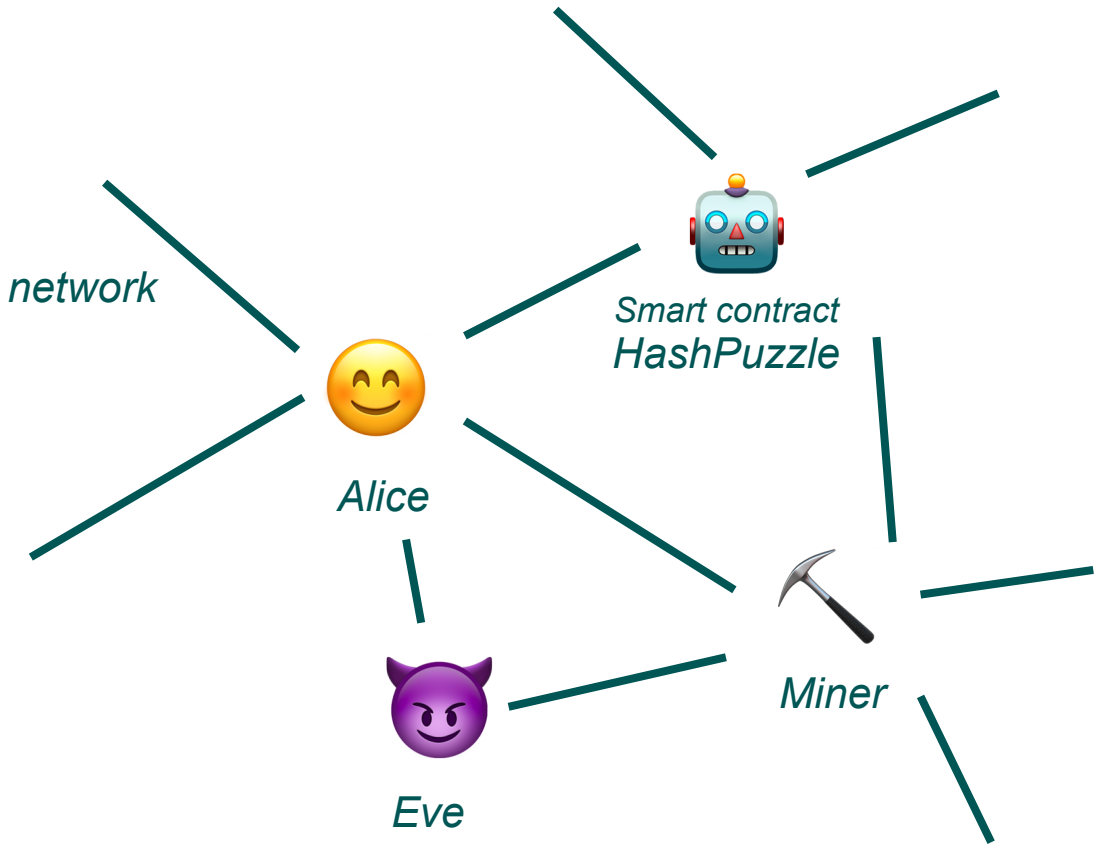
Mempool

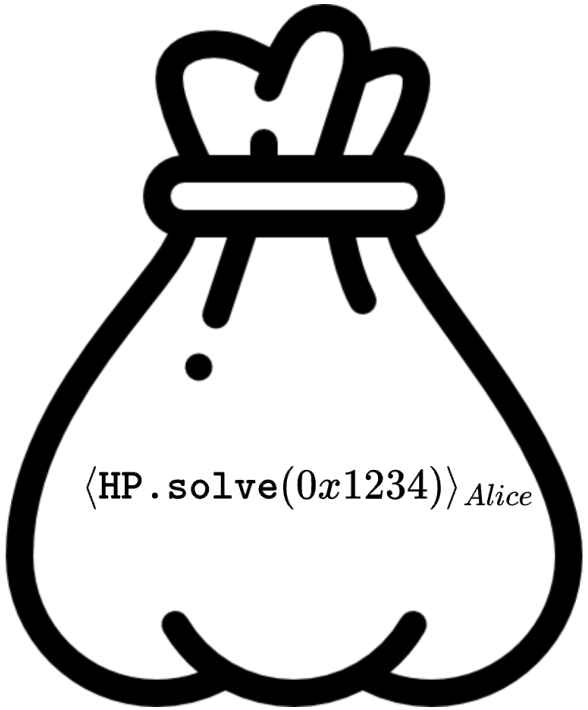
With blockchain semantics $\Gamma \xrightarrow{\langle tx \rangle} \Gamma'$, we reason about blockchain traces:

$$\Gamma_{Init} \xrightarrow{\langle tx_1 \rangle} \Gamma_1 \xrightarrow{\langle tx_2 \rangle} \ldots \xrightarrow{\langle tx_n \rangle} \Gamma_n$$

Motivation

Frontrunning *HashPuzzle*

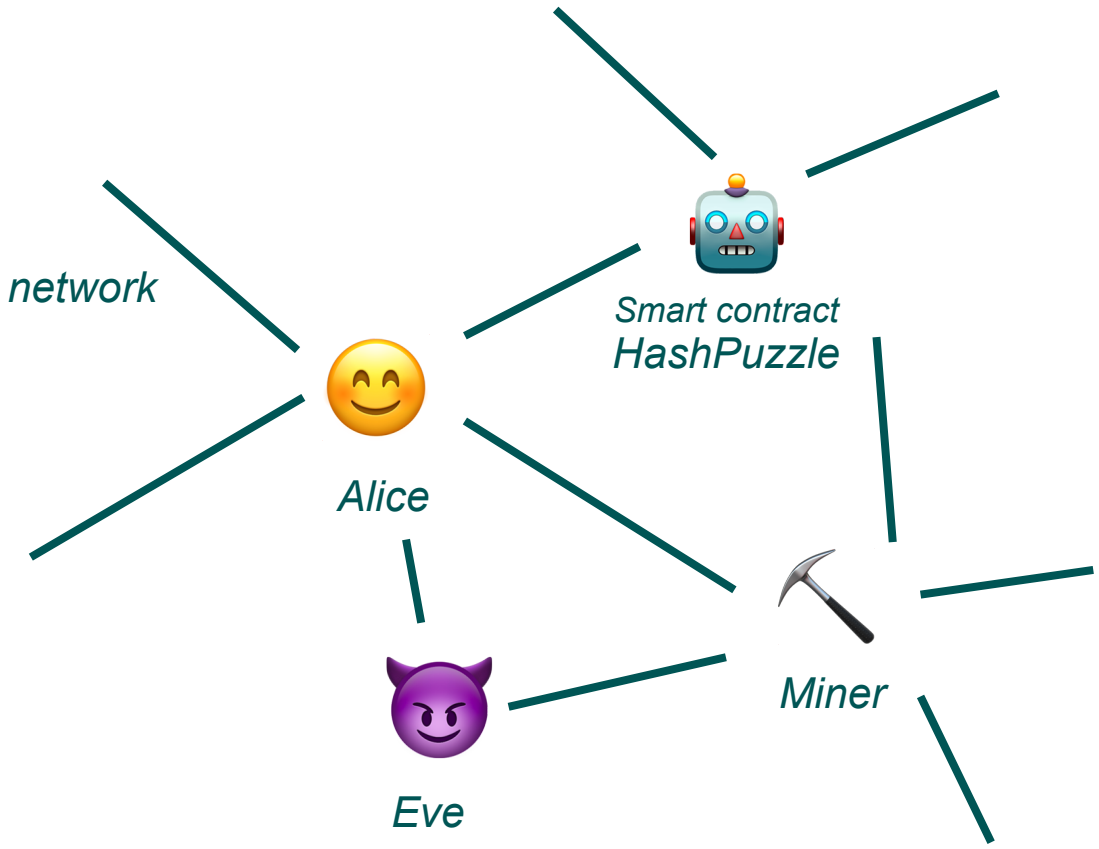


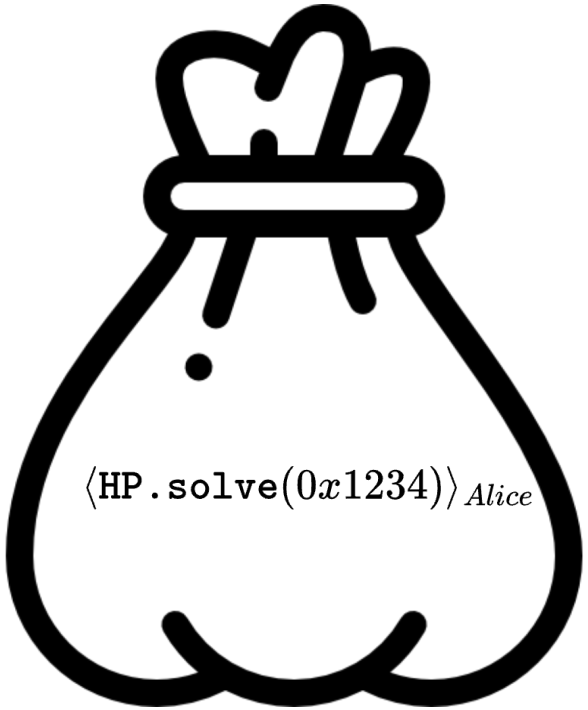


Mempool

Alice thinks, that she *calls* HashPuzzle.

Frontrunning *HashPuzzle*

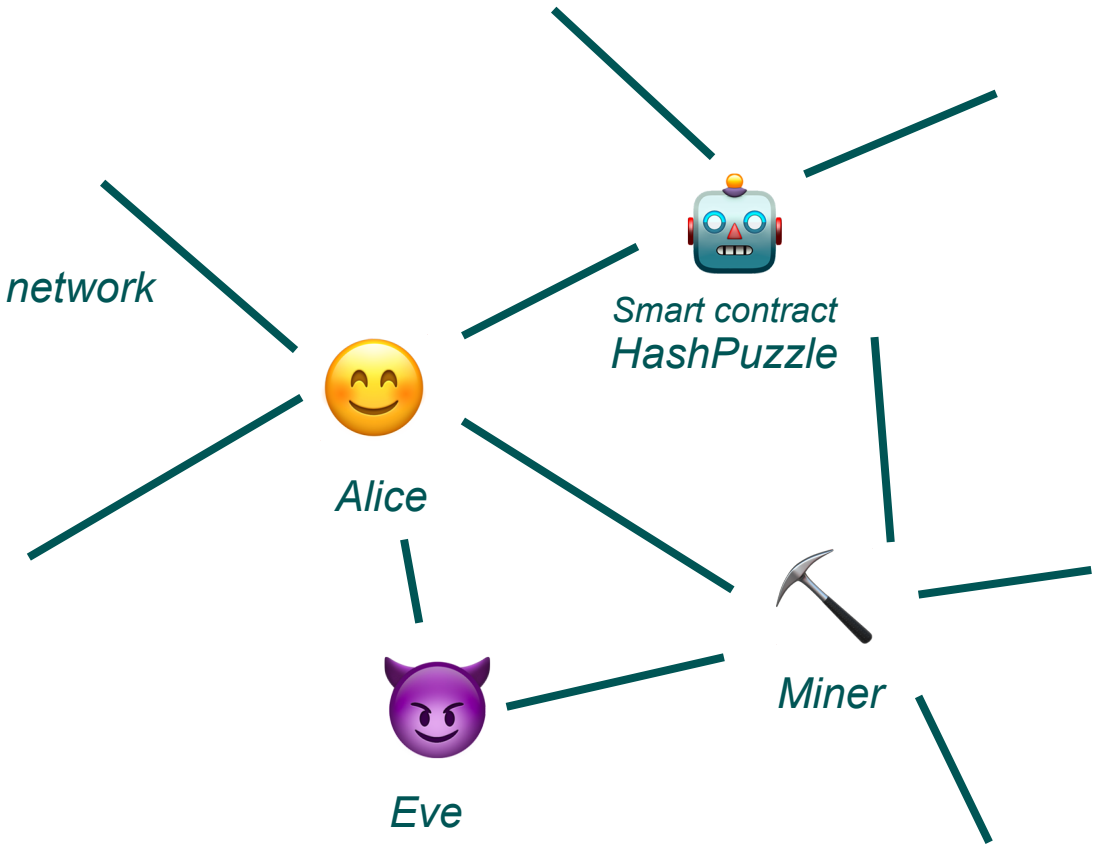


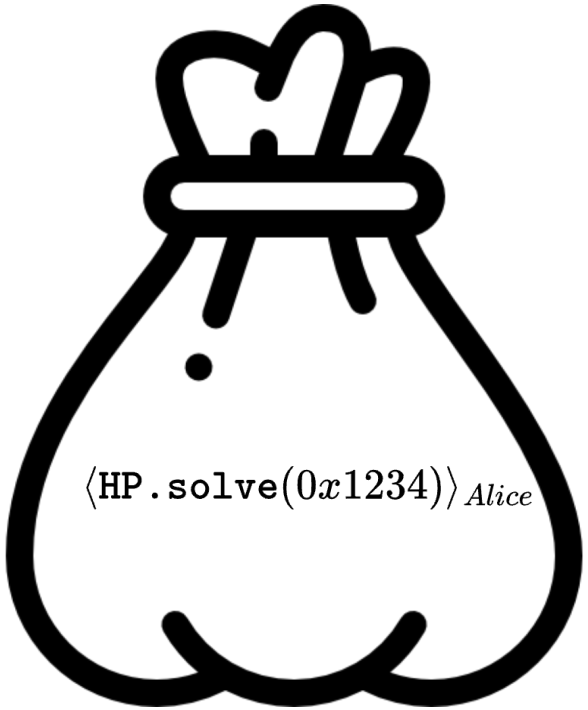


Mempool

Alice thinks, that she *calls* HashPuzzle. But before the block with a transaction is appended, the call has no effect.

Frontrunning *HashPuzzle*

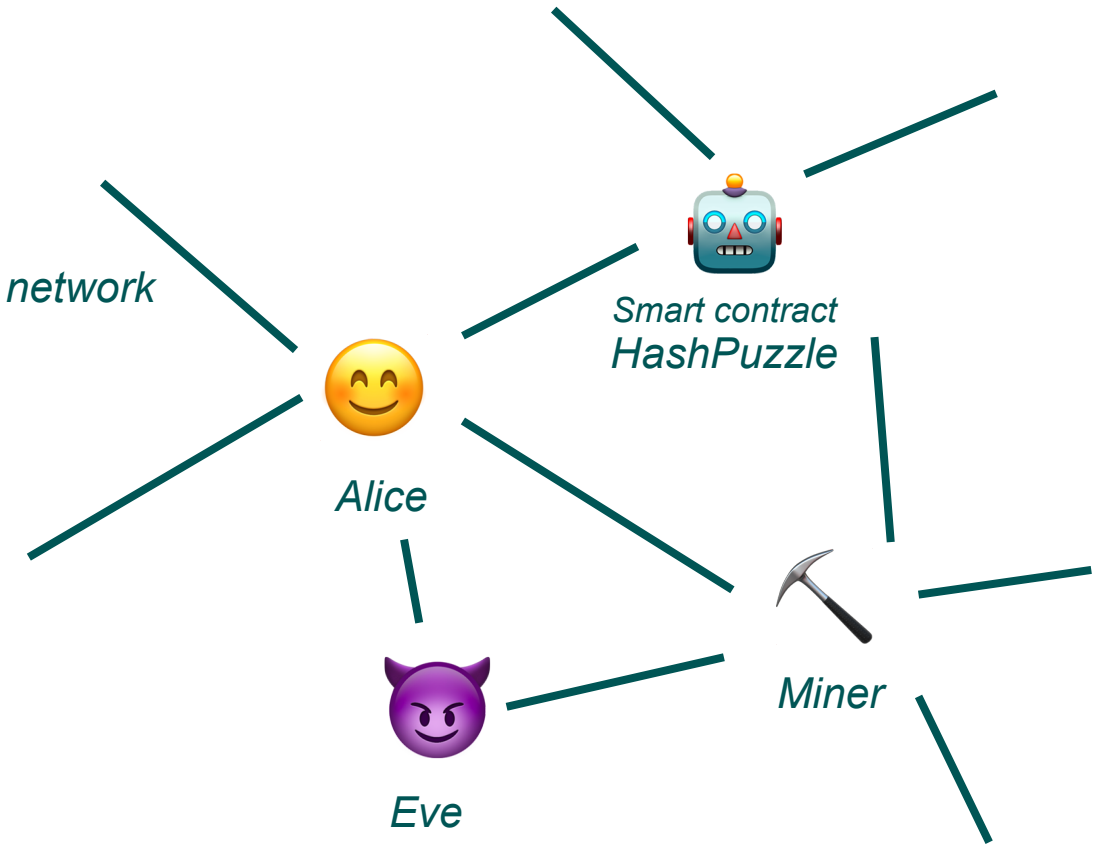


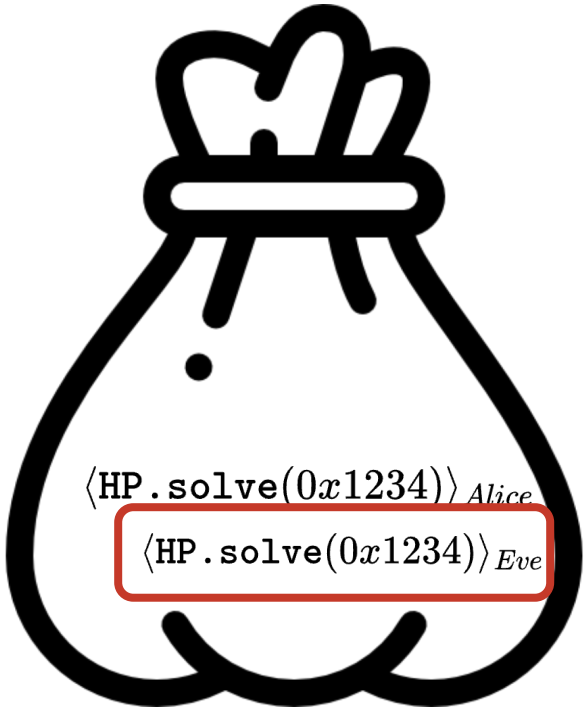


Mempool

Eve sees Alice's transaction **in the mempool** and uses it.

Frontrunning *HashPuzzle*



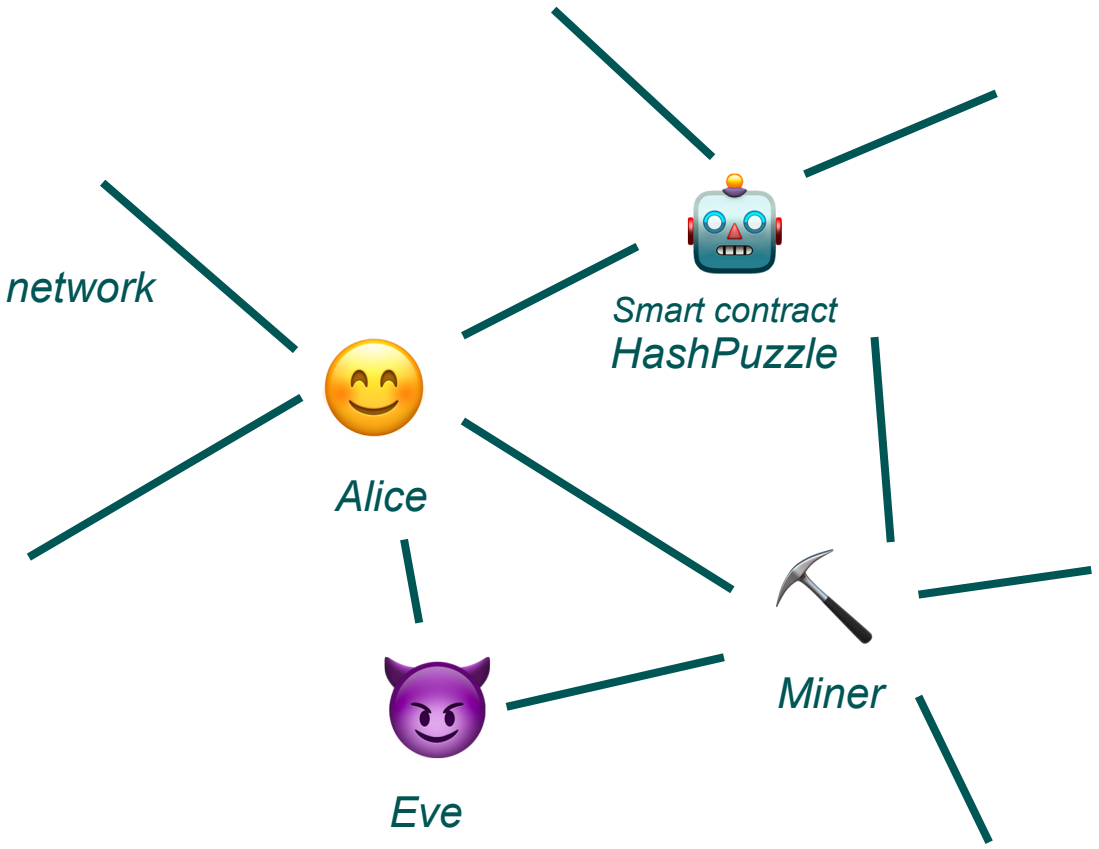


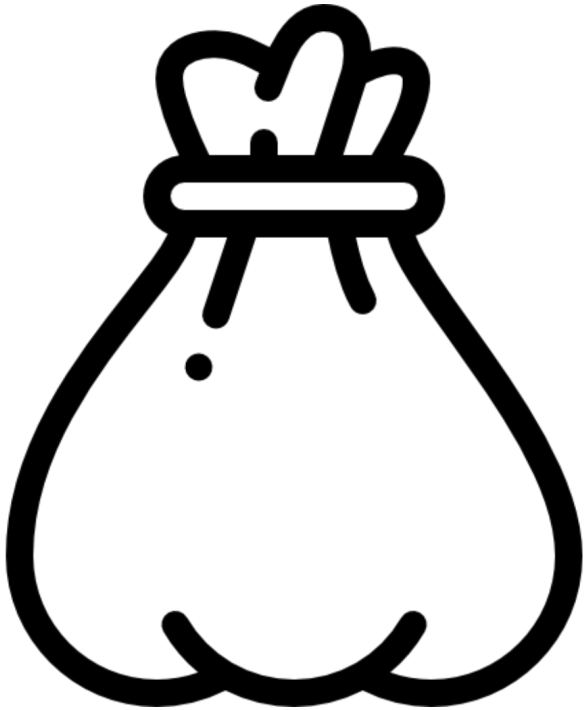
Mempool

😈 sees Alice's transaction **in the mempool** and uses it.

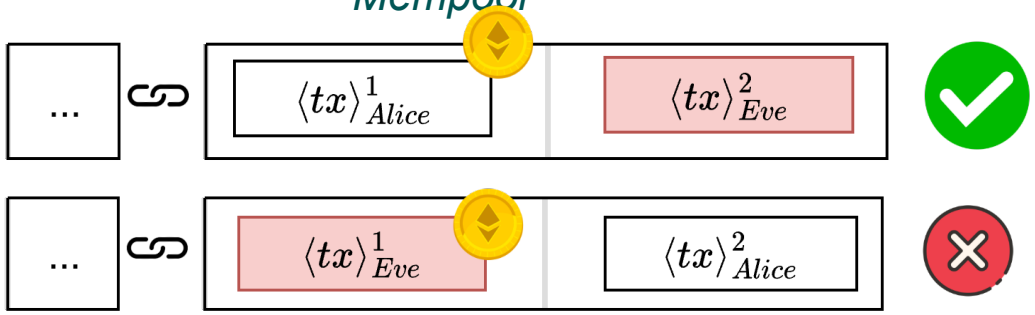
Eve

Frontrunning *HashPuzzle*



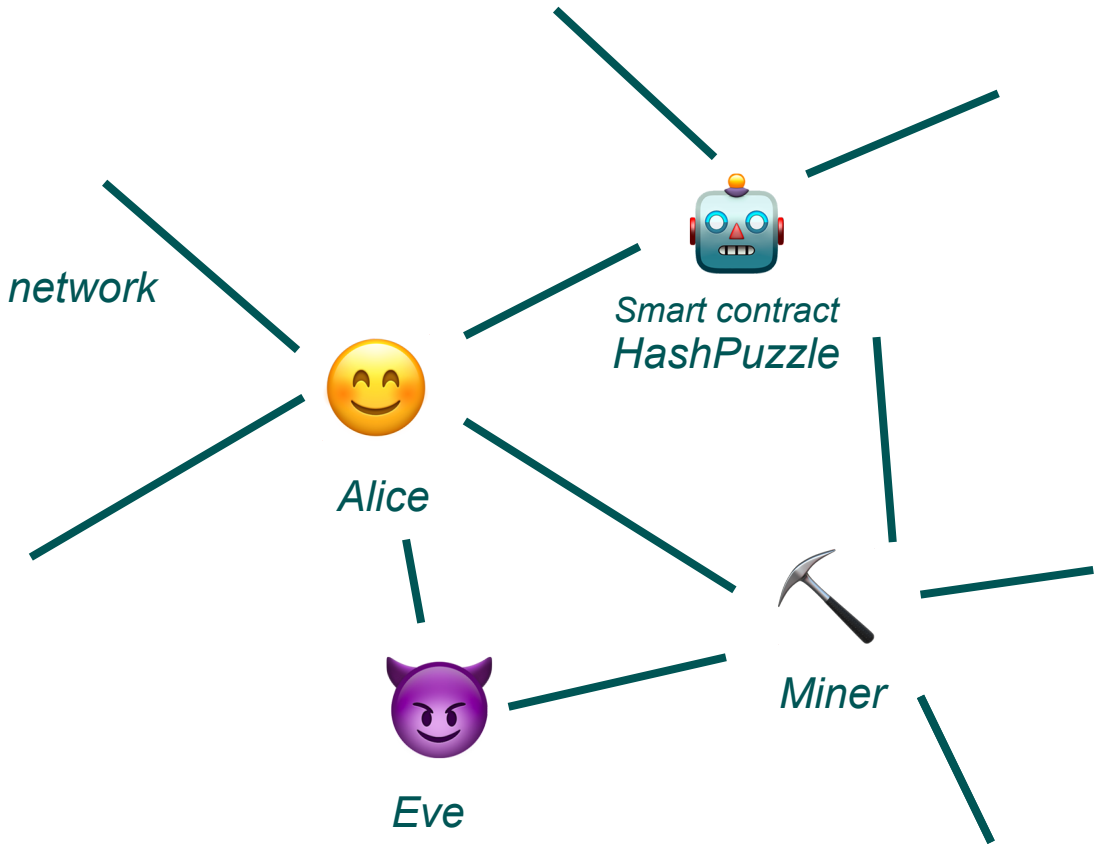


Mempool

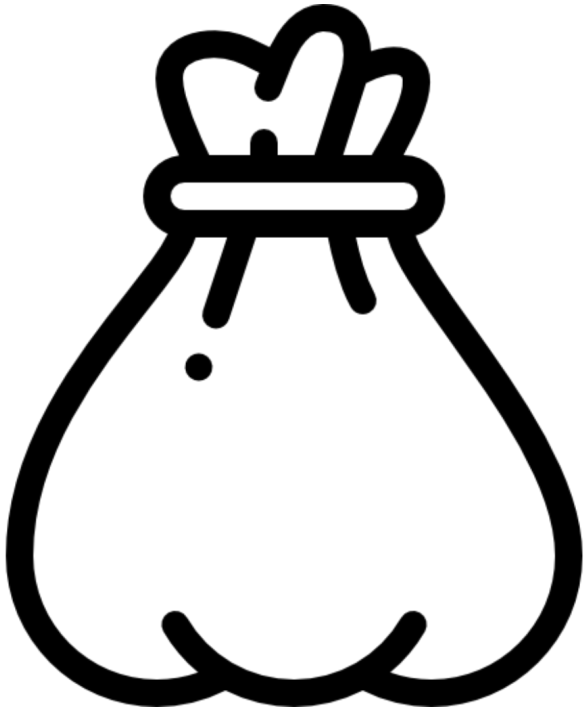


It is now for the miner to decide the order of two transactions:

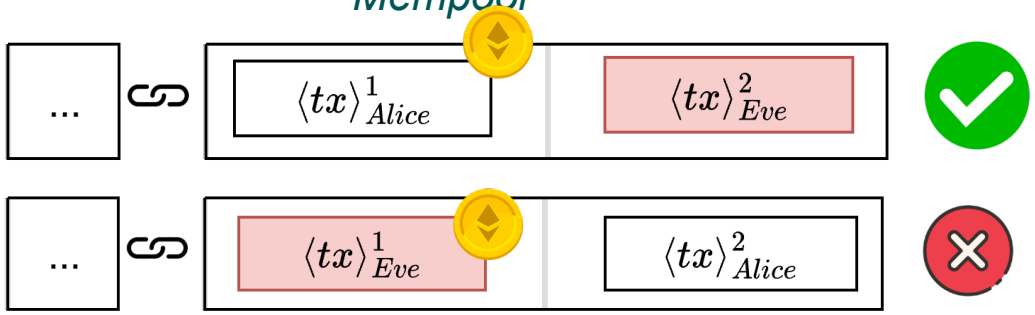
Frontrunning *HashPuzzle*



It is now for the miner to decide the order of two transactions:



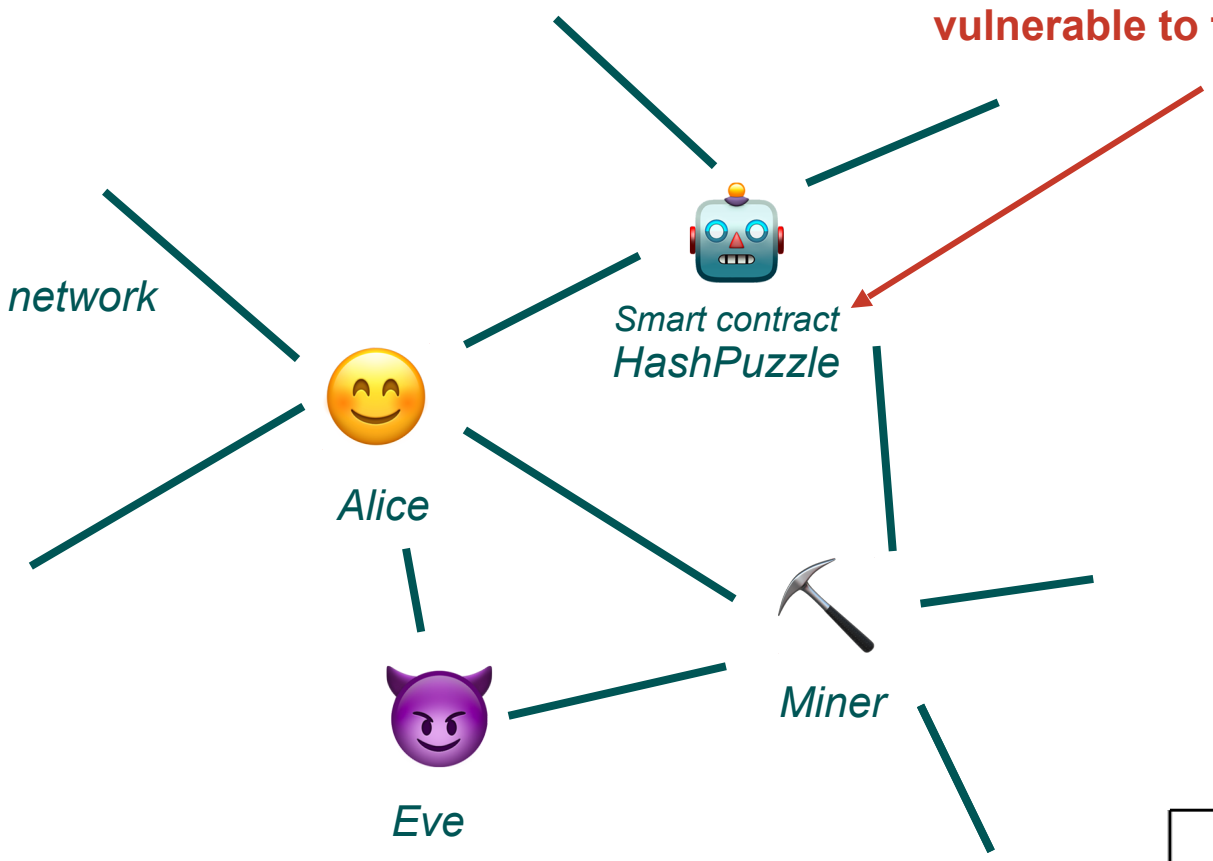
Mempool

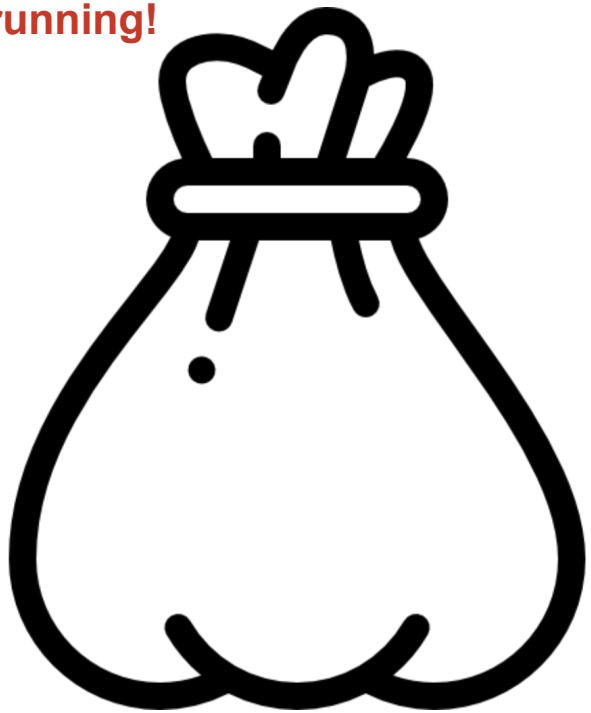


Eve can bribe the miner and ensure they steal the reward! [1]

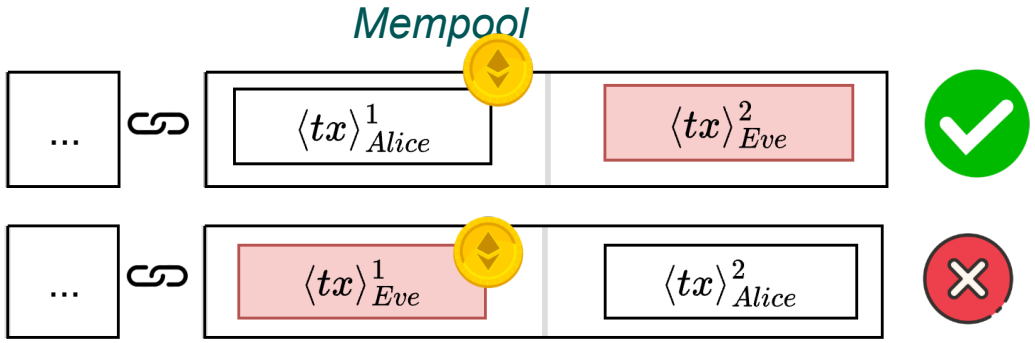
[1] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In 2022 IEEE Symposium on Security and Privacy (SP). IEEE, 198–214.

Frontrunning *HashPuzzle*





Eve can bribe the miner and ensure they steal the reward! [1]



It is now for the miner to decide the order of two transactions:

[1] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In 2022 IEEE Symposium on Security and Privacy (SP). IEEE, 198–214.

Frontrunning Attacks [1, 2]

Frontrunning [1, 2] is an attack in which an attacker uses the publicly available information about pending transactions to reorder them profitably, potentially harming users.
Attacker model: strong attacker that observes mempool and can arbitrarily insert and reorder transactions.

[1] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In 2022 IEEE Symposium on Security and Privacy (SP). IEEE, 198–214.
[2] Daian, Philip, et al. "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges." arXiv preprint arXiv:1904.05234 (2019).

Frontrunning Attacks [1, 2]

Frontrunning [1, 2] is an attack in which an attacker uses the publicly available information about pending transactions to reorder them profitably, potentially harming users.
Attacker model: strong attacker that observes mempool and can arbitrarily insert and reorder transactions.

From 2018 to 2021: **540.54M USD** extractable from Ethereum due to frontrunning. **[1]**

[1] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In 2022 IEEE Symposium on Security and Privacy (SP). IEEE, 198–214.
[2] Daian, Philip, et al. "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges." arXiv preprint arXiv:1904.05234 (2019).

Frontrunning Attacks [1, 2]

Frontrunning [1, 2] is an attack in which an attacker uses the publicly available information about pending transactions to reorder them profitably, potentially harming users.
Attacker model: strong attacker that observes mempool and can arbitrarily insert and reorder transactions.

From 2018 to 2021: **540.54M USD** extractable from Ethereum due to frontrunning. **[1]**

Frontrunning **undermines incentives of rational miners**.

[1] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In 2022 IEEE Symposium on Security and Privacy (SP). IEEE, 198–214.
[2] Daian, Philip, et al. "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges." arXiv preprint arXiv:1904.05234 (2019).

Transaction Order Dependence (TOD) [3]

If there are two transaction which, after invoking a smart contract, show different effects on blockchain state **depending on the their order**, we say a smart contract is **transaction-order dependent.**

[3] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

Transaction Order Dependence (TOD) [3]

If there are two transaction which, after invoking a smart contract, show different effects on blockchain state **depending on the their order**, we say a smart contract is **transaction-order dependent.**

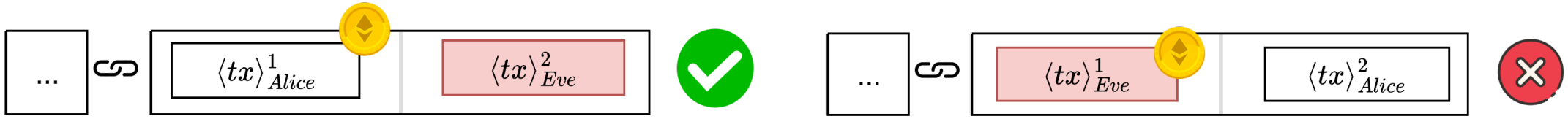
→ *HashPuzzle is transaction-order dependent, as* tx_{Alice} *and* tx_{Eve} *lead to different blockchain states depending on their order*

[3] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

Transaction Order Dependence (TOD) [3]

If there are two transaction which, after invoking a smart contract, show different effects on blockchain state **depending on the their order**, we say a smart contract is **transaction-order dependent.**

→ *HashPuzzle is transaction-order dependent, as* tx_{Alice} *and* tx_{Eve} *lead to different blockchain states depending on their order*

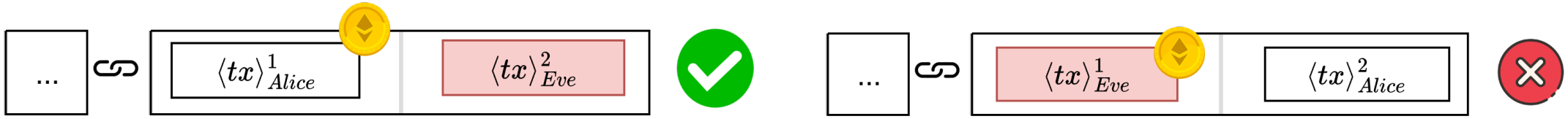


[3] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

Transaction Order Dependence (TOD) [3]

If there are two transaction which, after invoking a smart contract, show different effects on blockchain state **depending on the their order**, we say a smart contract is **transaction-order dependent.**

→ *HashPuzzle is transaction-order dependent, as tx_{Alice} and tx_{Eve} lead to different blockchain states depending on their order*



TOD is a **necessary**, but **insufficient** condition for a smart contract to be vulnerable to frontrunning!

[3] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

Key idea

Counterexample to TOD being sufficient: Commit-Reveal

Counterexample to TOD being sufficient: Commit-Reveal

- We can fix *HashPuzzle* by introducing two phases in which it works: **commit** and **reveal**.

Counterexample to TOD being sufficient: Commit-Reveal

- We can fix *HashPuzzle* by introducing two phases in which it works: **commit** and **reveal**.
- In the end of the phase, the user's action **is surely executed**.

Counterexample to TOD being sufficient: Commit-Reveal

- We can fix *HashPuzzle* by introducing two phases in which it works: **commit** and **reveal**.
- In the end of the phase, the user's action **is surely executed**.

Counterexample to TOD being sufficient: Commit-Reveal

- We can fix *HashPuzzle* by introducing two phases in which it works: **commit** and **reveal**.
- In the end of the phase, the user's action **is surely executed**.

```
contract HashPuzzleFix {
        …
    function commit(bytes32 commitment) {
            require(phase == Commit);
        …
    }
    function reveal(bytes32 solution) {
            require(phase == Reveal);
        …
    }
}
```

Counterexample to TOD being sufficient: Commit-Reveal

```
contract HashPuzzleFix {
        …
    mapping(address => bytes32) public commitments;
    function commit(bytes32 commitment) {
        require(phase == Commit);
        commitments[msg.sender] = commitment;
    }
    function reveal(bytes32 solution) {
            require(phase == Reveal);
        require(commitments[msg.sender] == sha256(solution, msg.sender));
        …
    }
}
```

Counterexample to TOD being sufficient: Commit-Reveal

- The fix is to get hashed solutions as commitments.

```
contract HashPuzzleFix {
        …
    mapping(address => bytes32) public commitments;
    function commit(bytes32 commitment) {
        require(phase == Commit);
        commitments[msg.sender] = commitment;
    }
    function reveal(bytes32 solution) {
            require(phase == Reveal);
        require(commitments[msg.sender] == sha256(solution, msg.sender));
        …
    }
}
```

Counterexample to TOD being sufficient: Commit-Reveal

- The fix is to get hashed solutions as commitments.
- This is secure, as after **commit** phase no new solution can be proposed.

```
contract HashPuzzleFix {
        …
    mapping(address => bytes32) public commitments;
    function commit(bytes32 commitment) {
            require(phase == Commit);
            commitments[msg.sender] = commitment;
    }
    function reveal(bytes32 solution) {
                    require(phase == Reveal);
            require(commitments[msg.sender] == sha256(solution, msg.sender));
            …
    }
}
```

Counterexample to TOD being sufficient: Commit-Reveal

```
contract HashPuzzleFix {
        …
    mapping(address => bytes32) public commitments;
    function commit(bytes32 commitment) {
        require(phase == Commit);
        commitments[msg.sender] = commitment;

    }
    function reveal(bytes32 solution) {
            require(phase == Reveal);
        require(commitments[msg.sender] == sha256(solution, msg.sender));
        …
    }
}
```

Counterexample to TOD being sufficient: Commit-Reveal

- However, an **irrational** honest user may leak their solution!

```
contract HashPuzzleFix {
        …
    mapping(address => bytes32) public commitments;
    function commit(bytes32 commitment) {
        require(phase == Commit);
        commitments[msg.sender] = commitment;

    }
    function reveal(bytes32 solution) {
            require(phase == Reveal);
        require(commitments[msg.sender] == sha256(solution, msg.sender));
        …
    }
}
```

Counterexample to TOD being sufficient: Commit-Reveal

- However, an **irrational** honest user may leak their solution!
- TOD **checks all smart contracts states**, so it renders this fix vulnerable.

```
contract HashPuzzleFix {
        …
    mapping(address => bytes32) public commitments;
    function commit(bytes32 commitment) {
        require(phase == Commit);
        commitments[msg.sender] = commitment;

    }
    function reveal(bytes32 solution) {
            require(phase == Reveal);
        require(commitments[msg.sender] == sha256(solution, msg.sender));
        …
    }
}
```

Key idea

The **honest user behavior** is the missing piece for the precise analysis, as it defines which **states of the smart contracts** are **reachable** when interacting with users.

Property

$$\forall R.\ \text{😇}\vdash R \wedge \text{😇}(R) = \langle \text{😇} \rangle \wedge R_n = \Gamma \Rightarrow$$

Key idea

The **honest user behavior** is the missing piece for the precise analysis, as it defines which **states of the smart contracts** are **reachable** when interacting with users.

Property

$$\forall R. \; \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Rightarrow$$

We check if in all blockchain executions R that conform to the honest user protocol

Key idea

The **honest user behavior** is the missing piece for the precise analysis, as it defines which **states of the smart contracts** are **reachable** when interacting with users.

Property

$$\forall R.\ \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Rightarrow$$

We check if in all blockchain executions R that conform to the honest user protocol and honest user transactions yielded by the protocol,

Key idea

The **honest user behavior** is the missing piece for the precise analysis, as it defines which **states of the smart contracts** are **reachable** when interacting with users.

Property

$$\forall R.\ \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Rightarrow$$

$$\forall \langle \rangle .\ \forall \Gamma', \Gamma''.\ \Gamma \rightarrow \Gamma' \wedge \Gamma \rightarrow \Gamma'' \implies \Gamma' \approx \Gamma''$$

We check if in all blockchain executions R that conform to the honest user protocol and honest user transactions yielded by the protocol,

Key idea

The **honest user behavior** is the missing piece for the precise analysis, as it defines which **states of the smart contracts** are **reachable** when interacting with users.

Property

$$\forall R.\ \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Rightarrow$$
$$\forall \langle \rangle.\ \forall \Gamma', \Gamma''.\ \Gamma \to \Gamma' \wedge \Gamma \to \Gamma'' \implies \Gamma' \approx \Gamma''$$

We check if in all blockchain executions R that conform to the honest user protocol and honest user transactions yielded by the protocol,
an attacker cannot extract value (due to commutativity of attacker/honest user transactions).

Caveat

Property

$$\forall R.\ \text{😇}\vdash R \wedge \text{😇}(R) = \langle \text{😇} \rangle \wedge R_n = \Gamma \Longrightarrow$$
$$\forall \langle \text{😈} \rangle.\ \forall \Gamma', \Gamma''.\ \Gamma \overset{\text{😈}}{\rightarrow}\overset{\text{😇}}{\rightarrow} \Gamma' \wedge \Gamma \overset{\text{😇}}{\rightarrow}\overset{\text{😈}}{\rightarrow} \Gamma'' \Longrightarrow \Gamma' \approx \Gamma''$$

Caveat

Property

$$\forall R.\ \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Longrightarrow$$

$$\forall \langle \rangle.\ \forall \Gamma', \Gamma''.\ \Gamma \rightarrow\!\!\rightarrow \Gamma' \wedge \Gamma \rightarrow\!\!\rightarrow \Gamma'' \implies \Gamma' \approx \Gamma''$$

Caveat

Property

$$\forall R.\ \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Rightarrow$$

$$\forall \langle \rangle.\ \forall \Gamma', \Gamma''.\ \Gamma \rightarrow\!\!\rightarrow \Gamma' \wedge \Gamma \rightarrow\!\!\rightarrow \Gamma'' \implies \Gamma' \approx \Gamma''$$

Problem: Verification of hyperproperty is computationally complex.

Caveat

Property

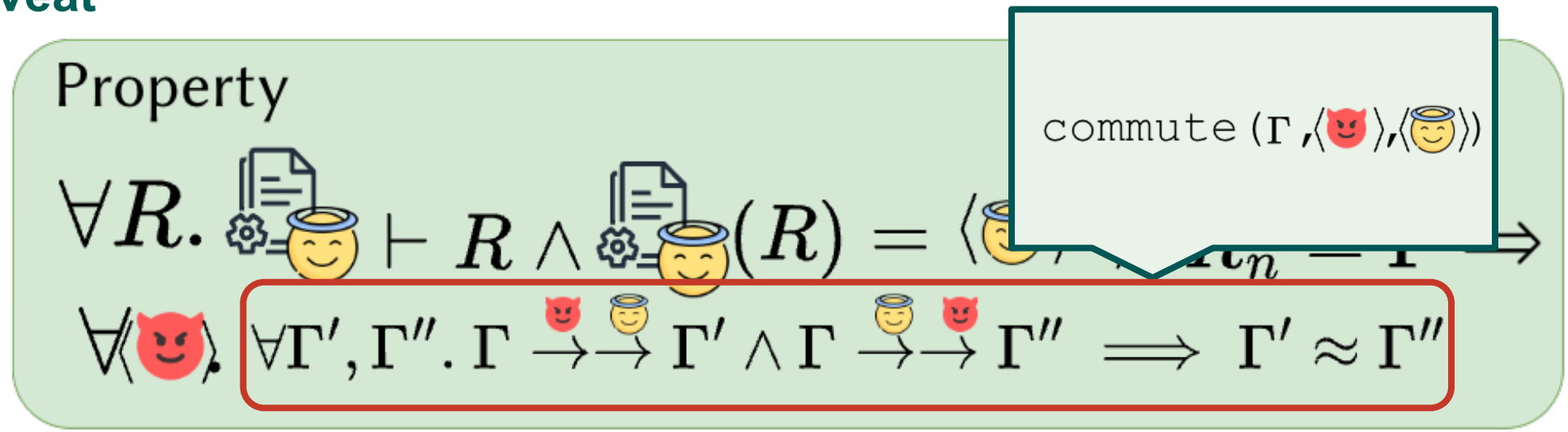
$$\forall R.\ \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Longrightarrow$$

$$\forall \langle \rangle.\ \forall \Gamma', \Gamma''.\ \Gamma \rightarrow\!\!\rightarrow \Gamma' \wedge \Gamma \rightarrow\!\!\rightarrow \Gamma'' \Longrightarrow \Gamma' \approx \Gamma''$$

Problem: Verification of hyperproperty is computationally complex.

Solution: reduce the verification to **reachability analysis** by precomputing **commutating states.**

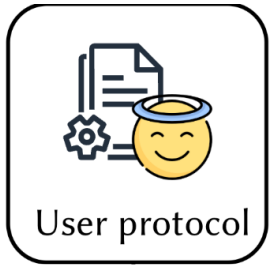
Caveat



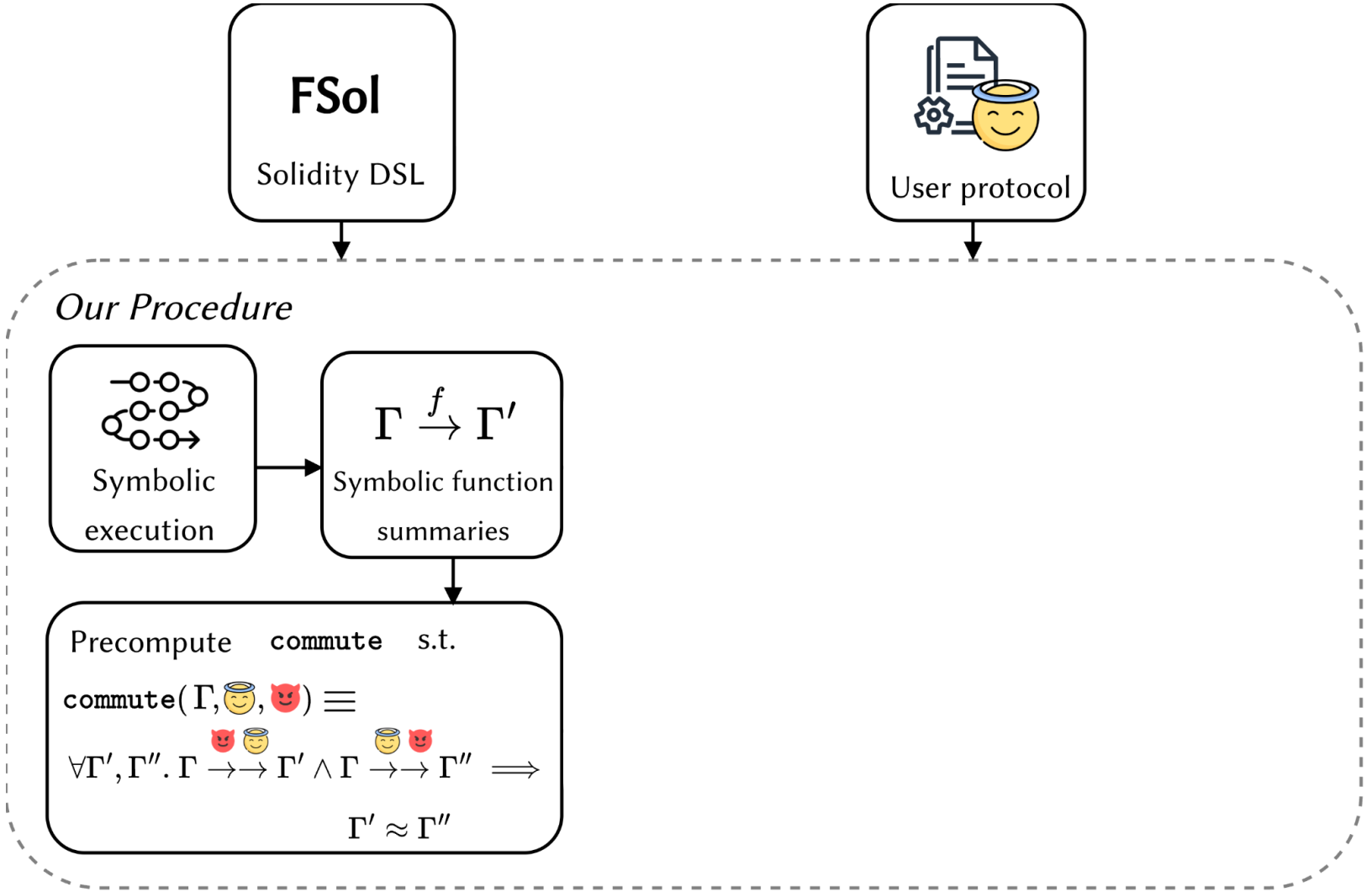
Problem: Verification of hyperproperty is computationally complex.

Solution: reduce the verification to **reachability analysis** by precomputing **commutating states.**

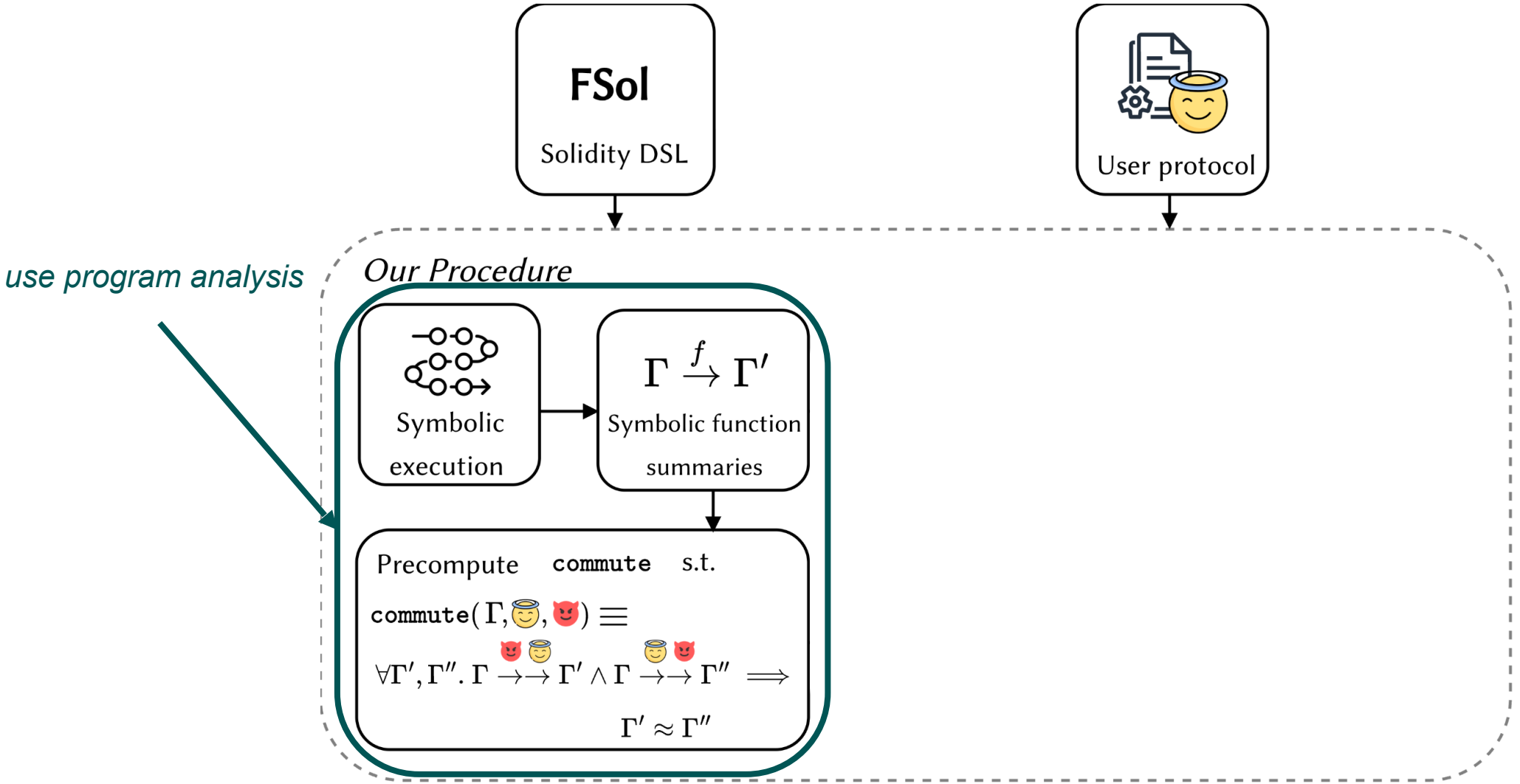
Overview



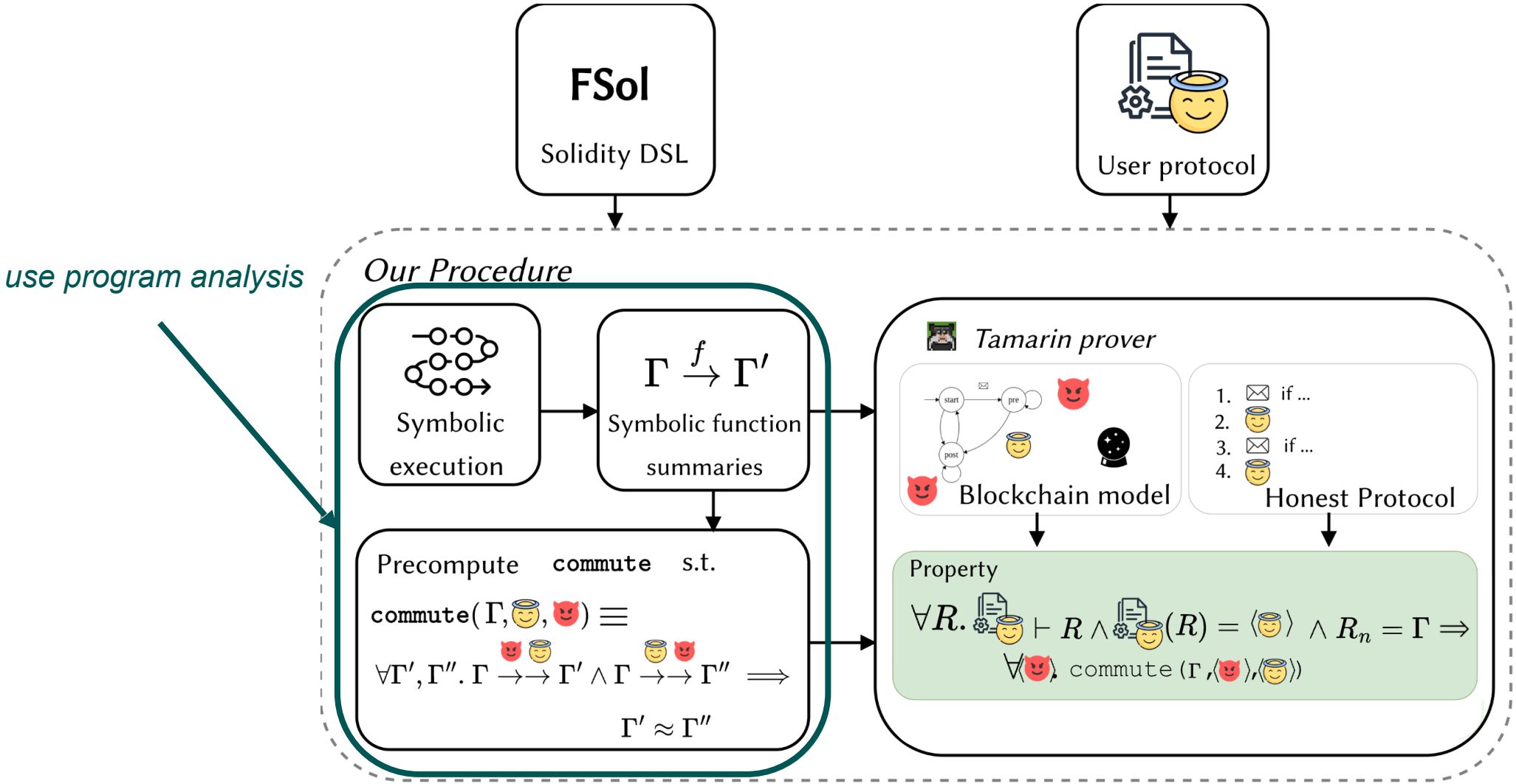
Overview


FSol
Solidity DSL
User protocol
Our Procedure
Symbolic execution
$\Gamma \xrightarrow{f} \Gamma'$
Symbolic function summaries
Precompute commute s.t.
commute(Γ, 😇, 😈) ≡
$\forall \Gamma', \Gamma''.\ \Gamma \overset{😈\,😇}{\rightarrow\!\rightarrow} \Gamma' \wedge \Gamma \overset{😇\,😈}{\rightarrow\!\rightarrow} \Gamma'' \implies$
$\Gamma' \approx \Gamma''$

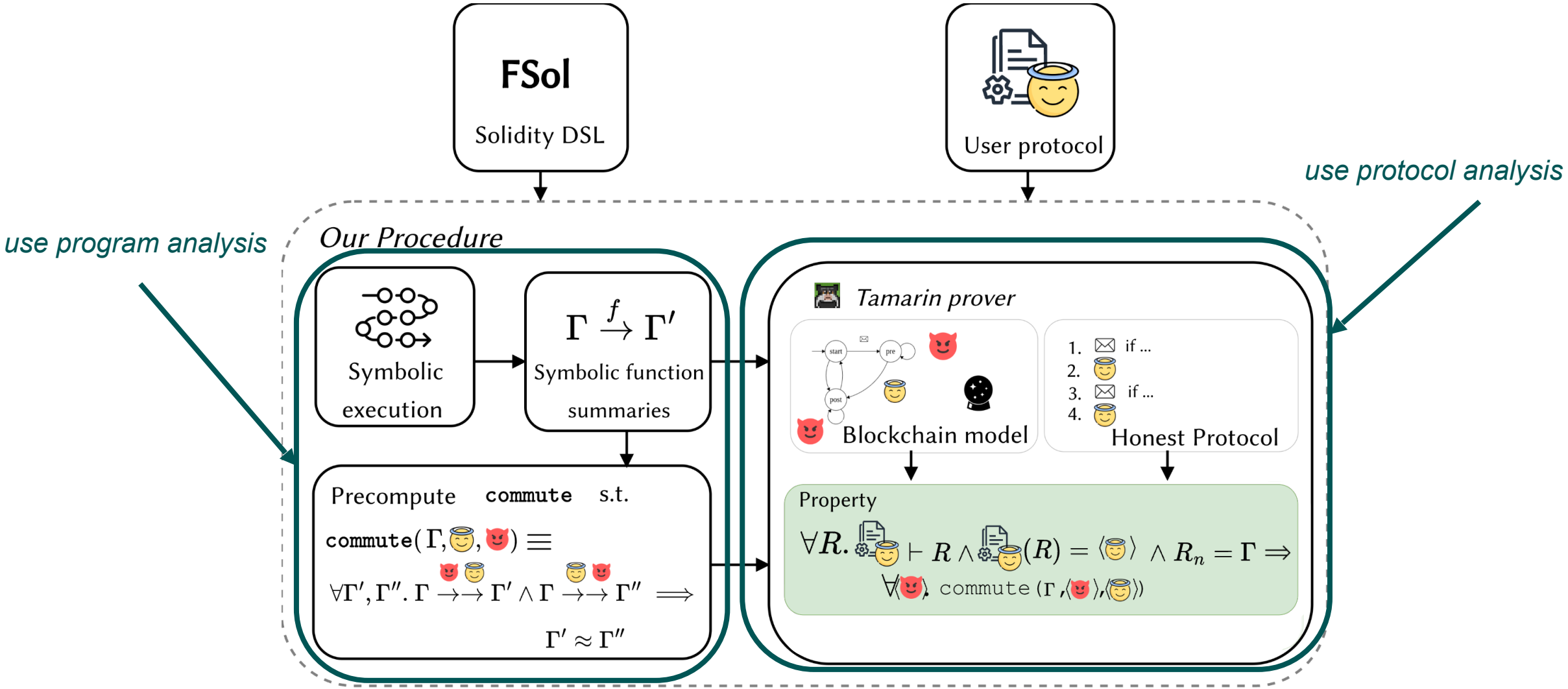
Overview


FSol
Solidity DSL
User protocol
use program analysis
Our Procedure
Symbolic execution
$\Gamma \xrightarrow{f} \Gamma'$
Symbolic function summaries
Precompute commute s.t.
commute$(\Gamma, 😇, 😈) \equiv$
$\forall \Gamma', \Gamma''.\ \Gamma \overset{😈😇}{\rightarrow\!\!\rightarrow} \Gamma' \wedge \Gamma \overset{😇😈}{\rightarrow\!\!\rightarrow} \Gamma'' \implies$
$\Gamma' \approx \Gamma''$

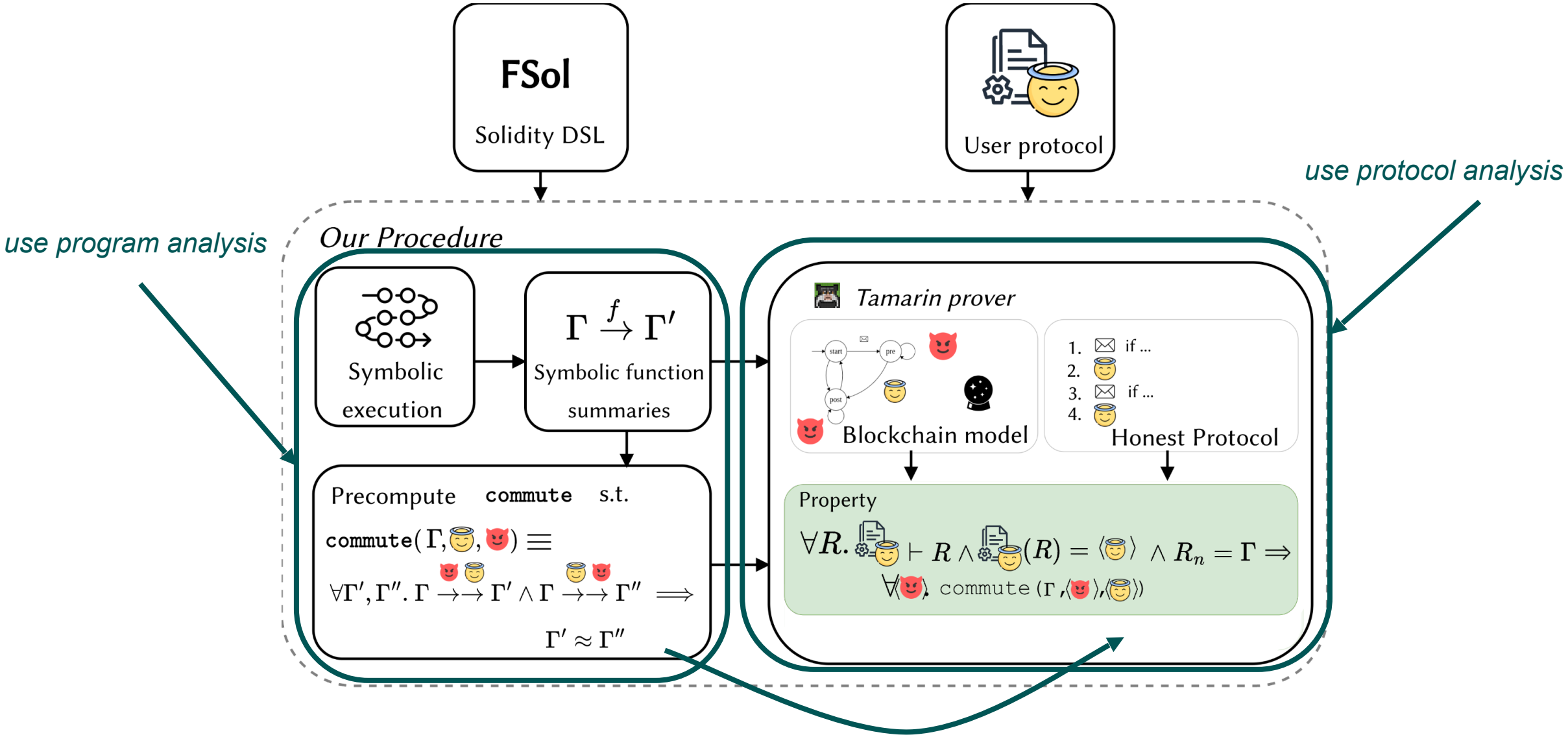
Overview


FSol
Solidity DSL
User protocol
use program analysis
Our Procedure
Symbolic execution
$\Gamma \xrightarrow{f} \Gamma'$
Symbolic function summaries
Precompute commute s.t.
$\forall \Gamma', \Gamma''. \Gamma \to\!\!\to \Gamma' \wedge \Gamma \to\!\!\to \Gamma'' \implies \Gamma' \approx \Gamma''$
Tamarin prover
start
pre
post
Blockchain model
1. if ...
2.
3. if ...
4.
Honest Protocol
Property
$\forall R. \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Rightarrow$
commute

Overview


FSol
Solidity DSL
User protocol
use protocol analysis
use program analysis
Our Procedure
Symbolic execution
$\Gamma \xrightarrow{f} \Gamma'$
Symbolic function summaries
Precompute commute s.t.
$\texttt{commute}(\Gamma, \text{😇}, \text{😈}) \equiv$
$\forall \Gamma', \Gamma''. \Gamma \to\!\!\to \Gamma' \wedge \Gamma \to\!\!\to \Gamma'' \implies \Gamma' \approx \Gamma''$
Tamarin prover
start
pre
post
Blockchain model
1. ✉ if ...
2. 😇
3. ✉ if ...
4. 😇
Honest Protocol
Property
$\forall R. \text{📄😇} \vdash R \wedge \text{⚙😇}(R) = \langle \text{😇} \rangle \wedge R_n = \Gamma \implies$
$\forall \langle \text{😈} \rangle, \texttt{commute}(\Gamma, \langle \text{😈} \rangle, \langle \text{😇} \rangle)$

Overview


FSol
Solidity DSL
User protocol
use protocol analysis
use program analysis
Our Procedure
Symbolic execution
$\Gamma \xrightarrow{f} \Gamma'$
Symbolic function summaries
Precompute commute s.t.
$\texttt{commute}(\Gamma, \text{😇}, \text{😈}) \equiv$
$\forall \Gamma', \Gamma''.\ \Gamma \to\!\!\to \Gamma' \wedge \Gamma \to\!\!\to \Gamma'' \implies \Gamma' \approx \Gamma''$
Tamarin prover
start
pre
post
Blockchain model
1. ✉ if ...
2. 😇
3. ✉ if ...
4. 😇
Honest Protocol
Property
$\forall R.\ \text{⚙😇} \vdash R \wedge \text{⚙😇}(R) = \langle \text{😇} \rangle \wedge R_n = \Gamma \implies$
$\forall \langle \text{😈} \rangle, \texttt{commute}(\Gamma, \langle \text{😈} \rangle, \langle \text{😇} \rangle)$

FSol: DSL for frontrunning resistance

FSol by example

```
contract HashPuzzleFix {
    phases [] [Commit, Reveal];
    state address winner = None;
    state mapping(address => bytes32) commitments;
    state bytes32 solution;

                …
}
```

FSol by example

```
contract HashPuzzleFix {
    phases [] [Commit, Reveal];
    state address winner = None;
    state mapping(address => bytes32) commitments;
    state bytes32 solution;

            function commit(bytes32 commitment) [Commit] {
            commitments[msg.sender] = commitment;
        }
            …
}
```

FSol by example

```
contract HashPuzzleFix {
    phases [] [Commit, Reveal];
    state address winner;
    state mapping(address => bytes32) commitments;
    state bytes32 solution;

        function commit(bytes32 commitment) [Commit] {
                commitments[msg.sender] = commitment;
        }

                function reveal(bytes32 res) [Reveal] {
                require(winner == None);
                require(hash(res) == sol);

                …
        }
}
```

Compiling FSol to Solidity

```
function reveal(bytes32 res) [Reveal] {
    require(winner == None);
    require(hash(res) == sol);

    winner = msg.sender;
    payable(msg.sender)
            .transfer(this.balance);
}
```

Function in FSol

Annotation-based translation

```
/// @custom:symex public-function
/// @custom:symex phase-indicator Reveal
/// @custom:symex safe-input-args 1
/// @custom:symex mapping none
function reveal(bytes32 res) public {
        /// injected function
    require(phase == Reveal);

    require(winner == address(0));
    require(hash(res) == sol);

    winner = msg.sender;

        payable(msg.sender)
                .transfer(this.balance);
}
```

Translated function in Solidity

Program analysis:

computing commutativity conditions

Goal: precompute commuting states

Step 1: symbolically execute each function to encode **state updates**

```
function reveal(bytes32 res) [Reveal] {
    require(winner == None);
    require(hash(res) == sol);

    winner = msg.sender;
    payable(msg.sender)
            .transfer(this.balance);
}
```

Function in FSol

Goal: precompute commuting states

Step 1: symbolically execute each function to encode **state updates**

```
function reveal(bytes32 res) [Reveal] {
    require(winner == None);
    require(hash(res) == sol);

    winner = msg.sender;
    payable(msg.sender)
            .transfer(this.balance);
}
```

Function in FSol


```
winner ↦
        If(And(
                hash(res) == solution,
                winner == None,
                phase == Reveal),
           sender,
           winner)
```

*Symbolic summary for **reveal***

Goal: precompute commuting states

Step 2: symbolically execute each pair of smart contract's functions to encode **sequential state updates**.

Goal: precompute commuting states

Step 2: symbolically execute each pair of smart contract's functions to encode **sequential state updates**.

$$\Gamma \xrightarrow{f} \Gamma' \xrightarrow{g} \Gamma''$$

Goal: precompute commuting states

Step 2: symbolically execute each pair of smart contract's functions to encode **sequential state updates**.

$$\Gamma \xrightarrow{f} \Gamma' \xrightarrow{g} \Gamma''$$

Similar to step 1 by reusing computed function summaries.

Goal: precompute commuting states

Goal: precompute commuting states

- Have: state updates of **f; g** for all functions **f, g** of a smart contract.

Goal: precompute commuting states

- Have: state updates of **f; g** for all functions **f, g** of a smart contract.
- Step 3: synthesize sound commutativity conditions, parametrized by the state, such that:

Goal: precompute commuting states

- Have: state updates of **f; g** for all functions **f, g** of a smart contract.
- Step 3: synthesize sound commutativity conditions, parametrized by the state, such that:

$$\forall \psi \in CC_{\alpha,\beta}.\ \forall \Gamma.\ \psi(\Gamma) \implies \mathbf{Comm}(\Gamma, \alpha, \beta)$$

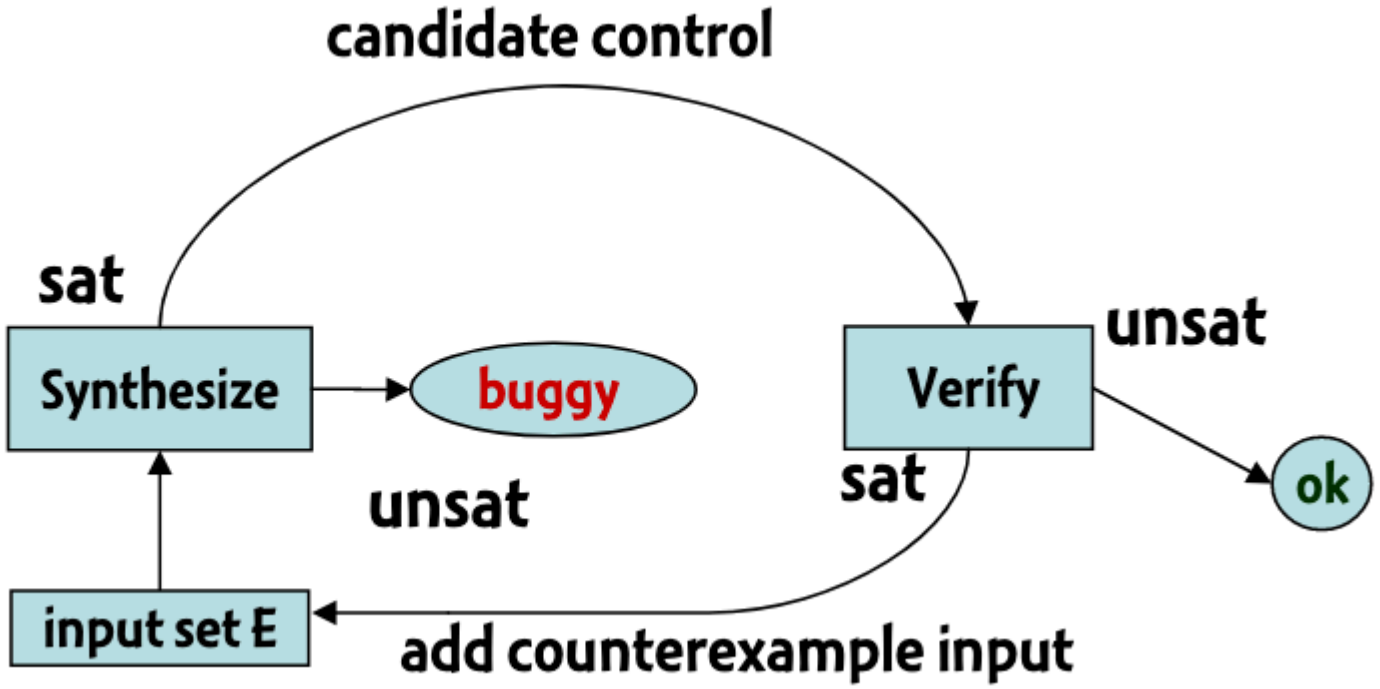
Goal: precompute commuting states

- Current approach: use **counterexample-guided abstraction refinement [4].**
- Refine the conditions for commutativity.
- The **Servois2 [5]** tool implements it, with heuristic for smaller conditions.

[4] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H. (2000). Counterexample-Guided Abstraction Refinement . In: Emerson, E.A., Sistla, A.P. (eds) Computer Aided Verification. CAV 2000. Lecture Notes in Computer Science, vol 1855. Springer, Berlin, Heidelberg
[5] Adam Chen, Parisa Fathololumi, Mihai Nicola, Jared Pincus, Tegan Brennan, and Eric Koskinen. 2023. Better Predicates and Heuristics for Improved Commutativity Synthesis. In Automated Technology for Verification and Analysis: 21st International Symposium, ATVA 2023, Singapore, October 24–27, 2023, Proceedings, Part II. Springer-Verlag, Berlin, Heidelberg, 93–113.

Goal: precompute commuting states

- Current approach: use **counterexample-guided abstraction refinement [4].**
- Refine the conditions for commutativity.
- The **Servois2 [5]** tool implements it, with heuristic for smaller conditions.



[4] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H. (2000). Counterexample-Guided Abstraction Refinement . In: Emerson, E.A., Sistla, A.P. (eds) Computer Aided Verification. CAV 2000. Lecture Notes in Computer Science, vol 1855. Springer, Berlin, Heidelberg
[5] Adam Chen, Parisa Fathololumi, Mihai Nicola, Jared Pincus, Tegan Brennan, and Eric Koskinen. 2023. Better Predicates and Heuristics for Improved Commutativity Synthesis. In Automated Technology for Verification and Analysis: 21st International Symposium, ATVA 2023, Singapore, October 24–27, 2023, Proceedings, Part II. Springer-Verlag, Berlin, Heidelberg, 93–113.

Commutativity conditions generation

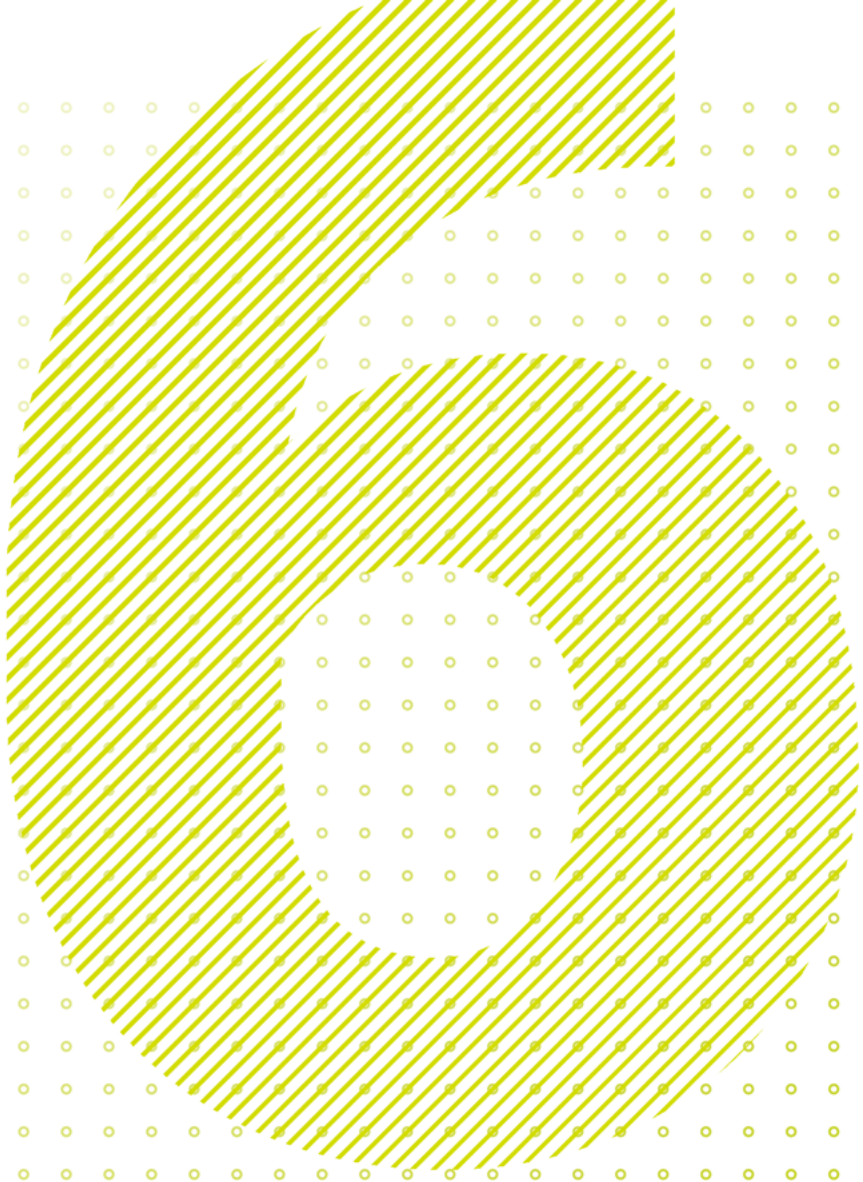
- *HashPuzzle* commutativity conditions for **(solve, solve)** pair are:

```
contract HashPuzzle {
    bytes32 public challenge;
    address public winner;
    function solve(bytes32 solution) {
        require(
                        sha256(solution) == challenge);
                payable(msg.sender)
                        .transfer(this.balance);
        this.winner = msg.sender;
    }
}
```

*Vulnerable **HashPuzzle** smart contract.*

```
(or (= attackerAddress honestAddress)
    (not (= winner None))
    (= attackerAddress None)
    (= honestAddress None)
    (not (= challenge
            (sha256 solution_honest))))
    (not (= challenge
            (sha256 solution_attacker)))))
```

*Conditions on initial state, under which function pair **solve, solve** commutes.*



Protocol analysis using Tamarin

Goal: construct conforming traces

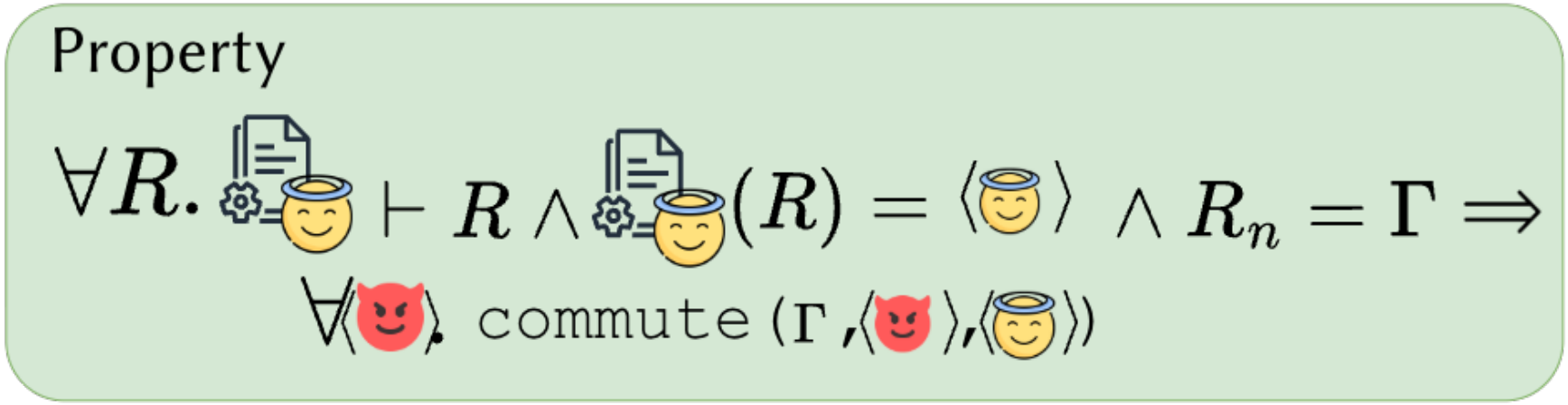
Property

$$\forall R.\ \text{📄😇} \vdash R \wedge \text{📄😇}(R) = \langle \text{😇} \rangle \wedge R_n = \Gamma \Rightarrow$$

$$\forall \langle \text{😈} \rangle, \texttt{commute}(\Gamma, \langle \text{😈} \rangle, \langle \text{😇} \rangle)$$

Goal: construct conforming traces

- So far: we have sound approximation of commuting states.



Goal: construct conforming traces

- So far: we have sound approximation of commuting states.
- Now: use protocol analysis tools to construct traces that *conform to honest user protocol.*

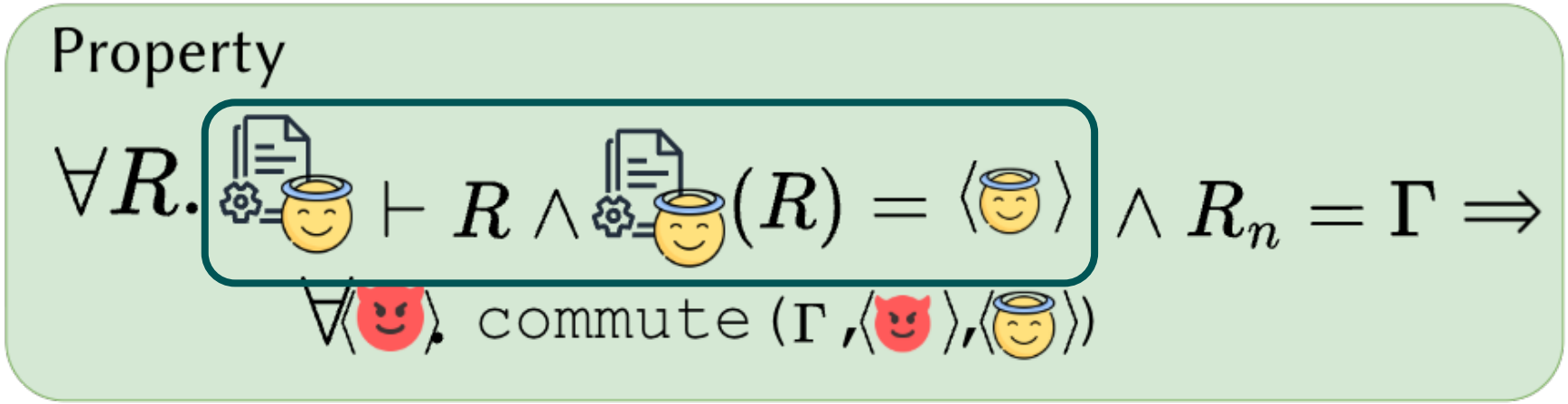
Property

$$\forall R.\ \vdash R \wedge (R) = \langle \rangle \wedge R_n = \Gamma \Rightarrow$$

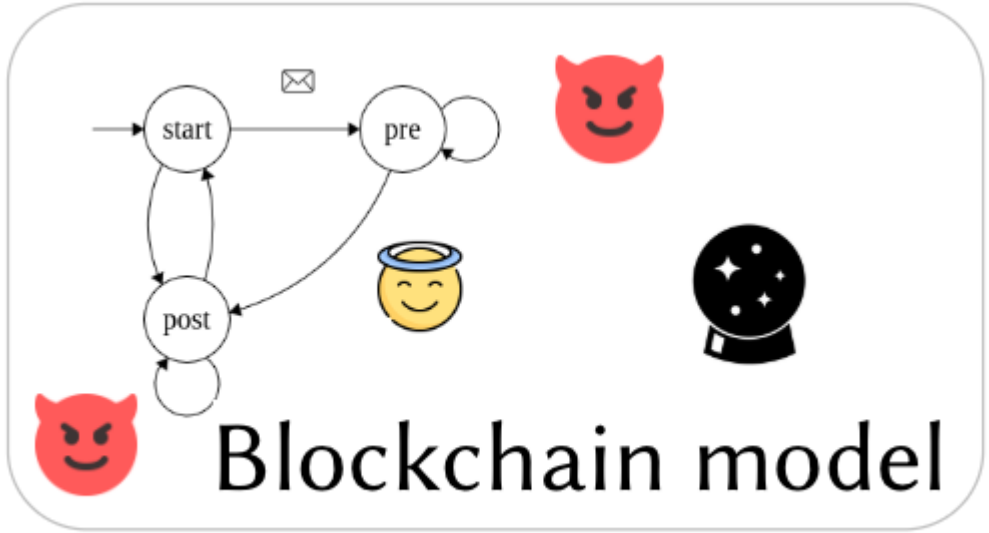
$$\forall \langle \rangle,\ \texttt{commute}(\Gamma, \langle \rangle, \langle \rangle)$$

Goal: construct conforming traces

- So far: we have sound approximation of commuting states.
- Now: use protocol analysis tools to construct traces that *conform to honest user protocol.*

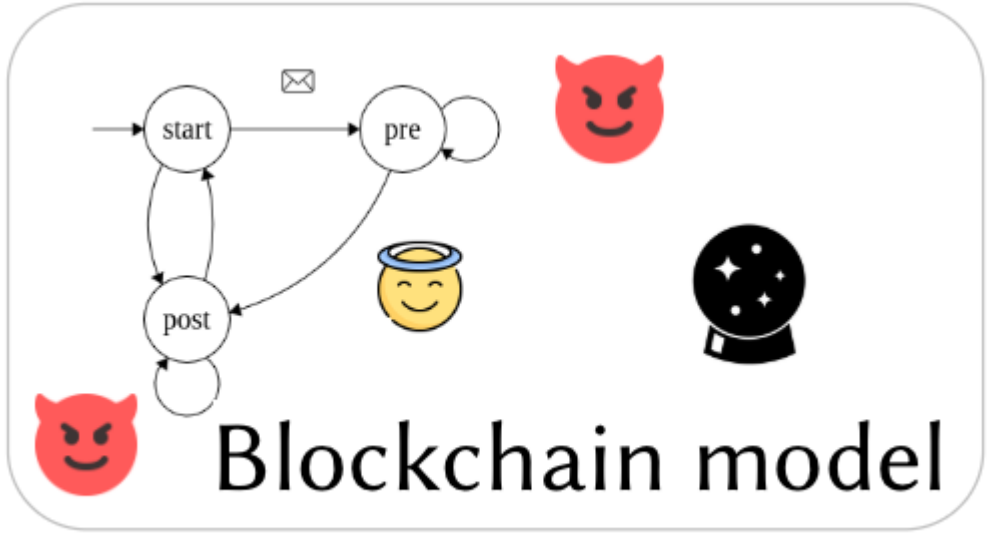


Blockchain model



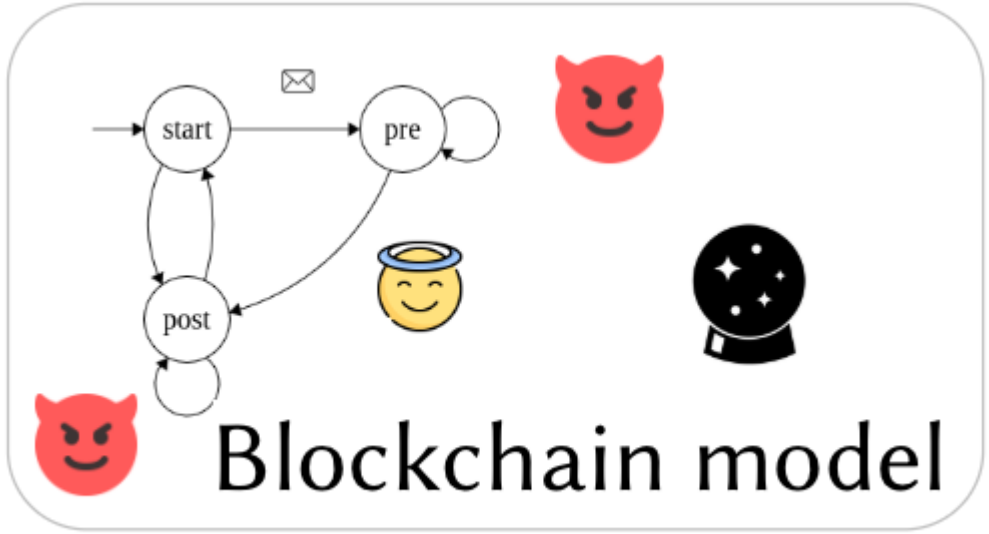
Blockchain model

- We provide a model of blockchain in Tamarin. We assume a **phase-based execution** in it.



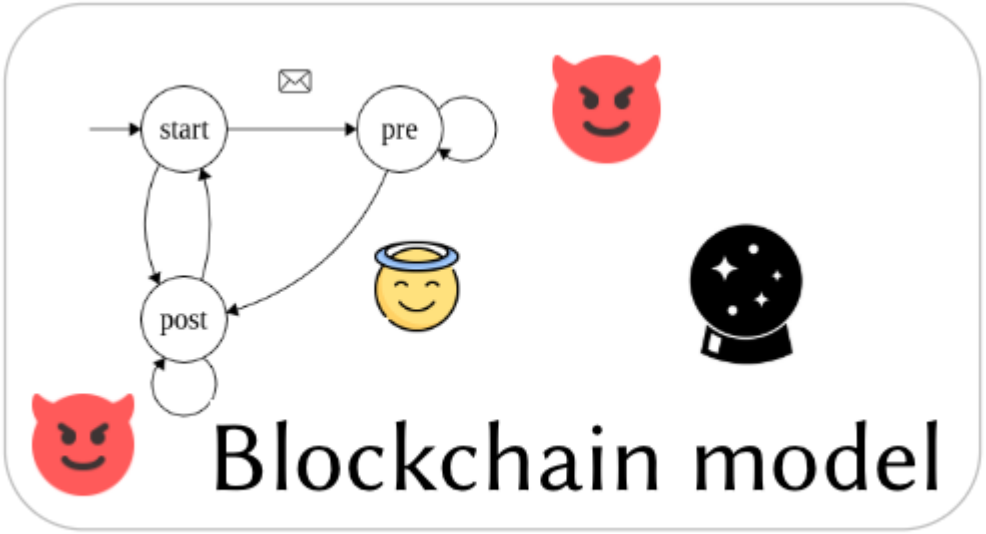
Blockchain model

- We provide a model of blockchain in Tamarin. We assume a **phase-based execution** in it.
- With phases, honest users only do one action per phase, and then wait until this action is executed on blockchain.

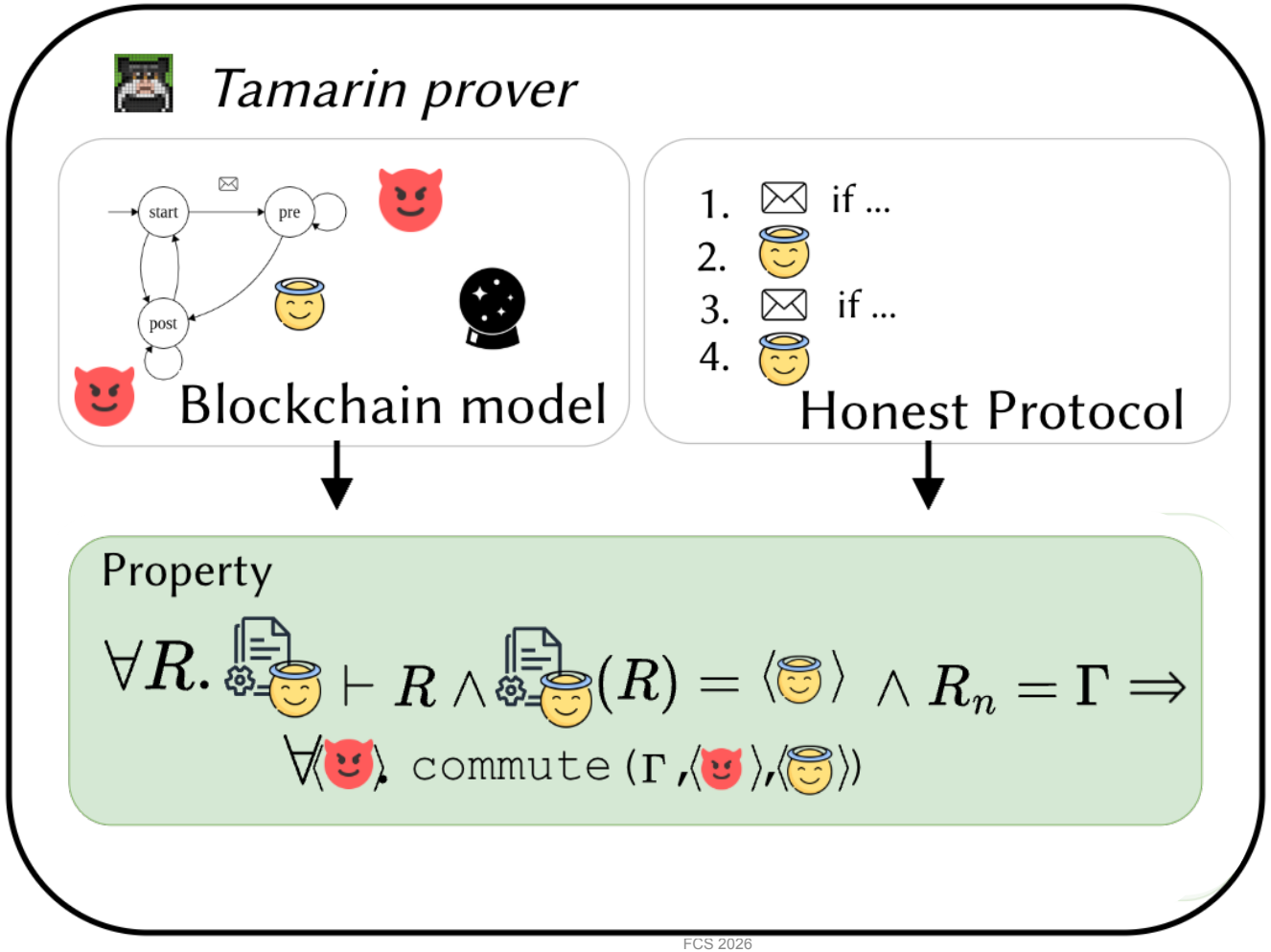


Honest user protocol

- We embed the provided honest user protocol to the blockchain model.
- The honest user protocol defines when the trace conforms to the trace.

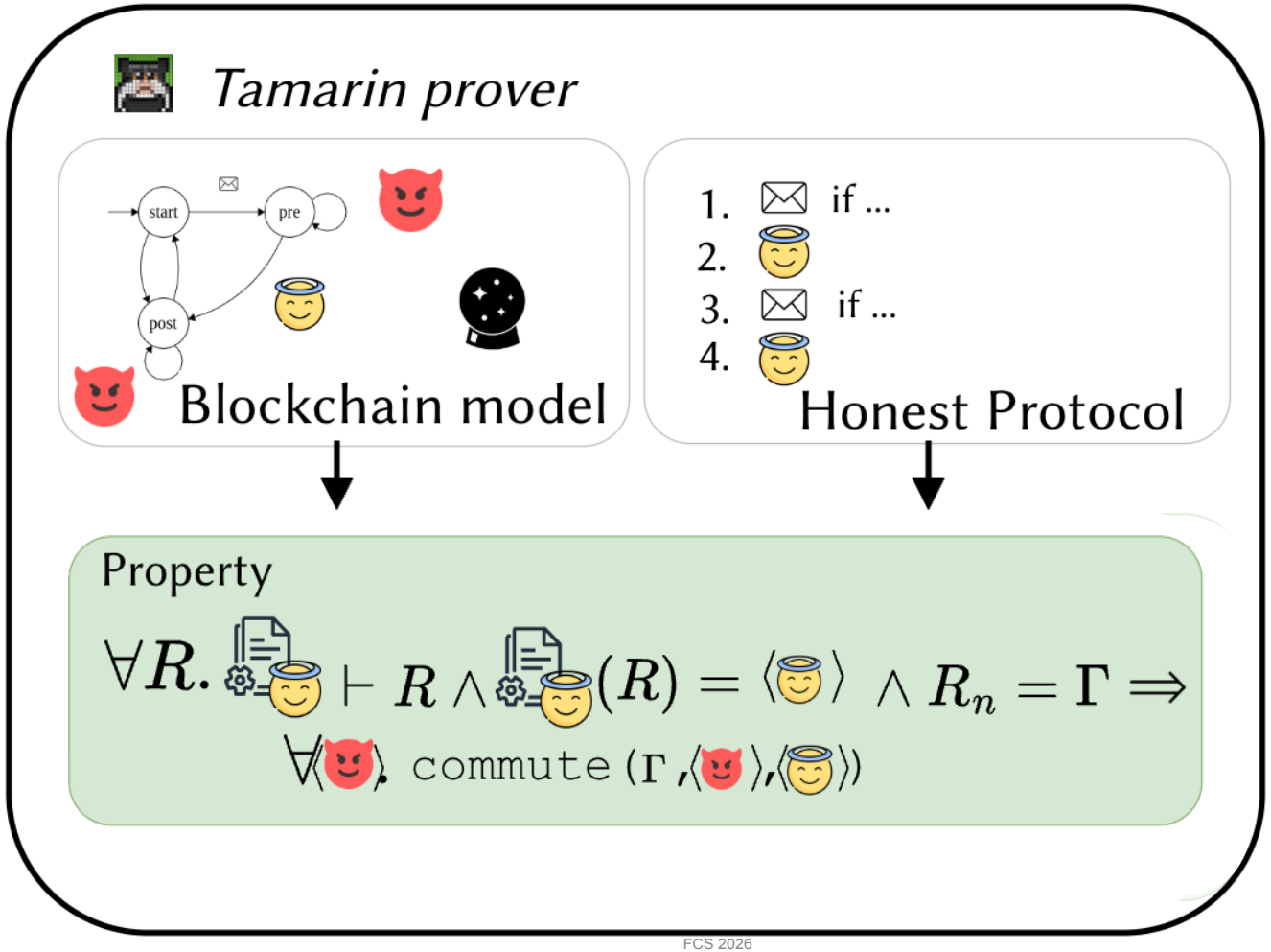


Query verification

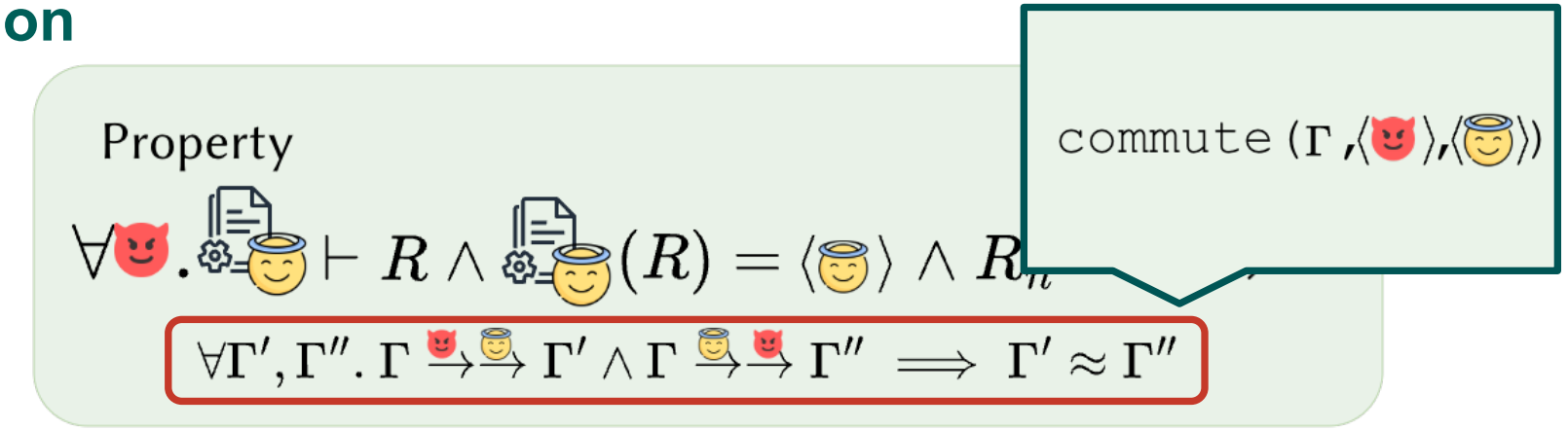


Query verification

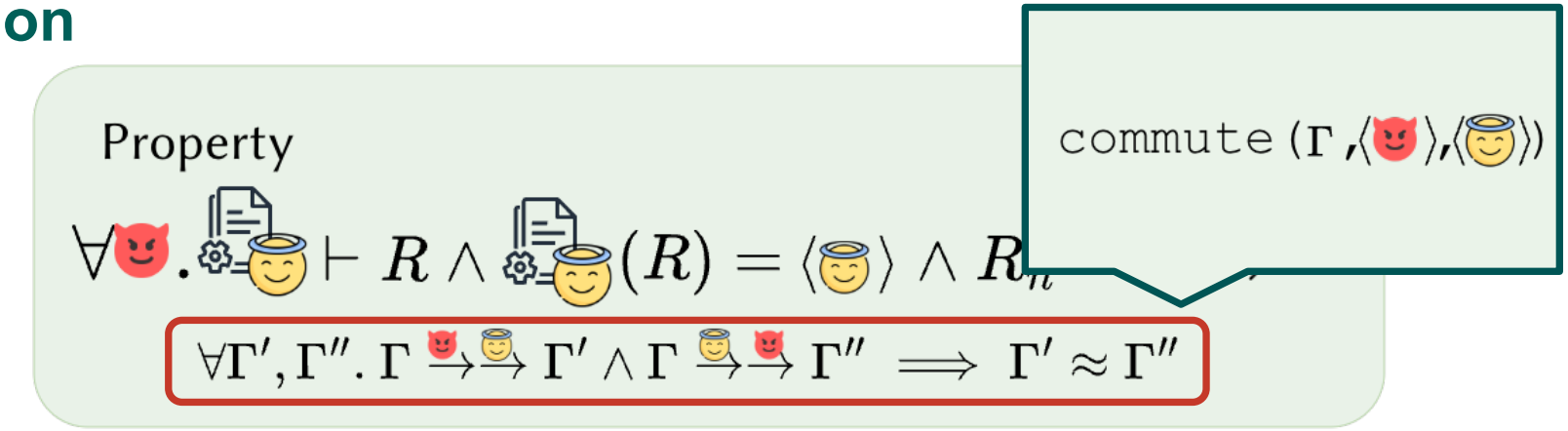
- Lastly, we use the pre-computed commutativity conditions to check frontrunning resistance.



Conclusion

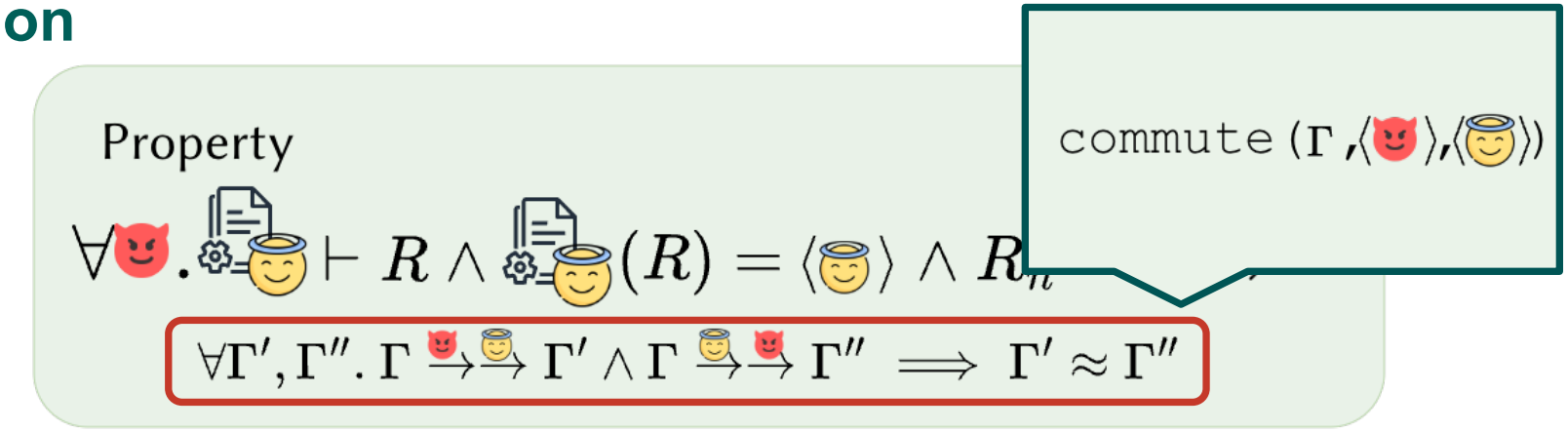

Property
commute (Γ, ⟨😈⟩, ⟨😇⟩)
$\forall$😈. 😈😇 $\vdash R \wedge$ 😈😇$(R) = \langle$😇$\rangle \wedge R_{\mathit{fl}}$
$\forall \Gamma', \Gamma''. \Gamma \overset{😈}{\rightarrow}\overset{😇}{\rightarrow} \Gamma' \wedge \Gamma \overset{😇}{\rightarrow}\overset{😈}{\rightarrow} \Gamma'' \implies \Gamma' \approx \Gamma''$

Conclusion



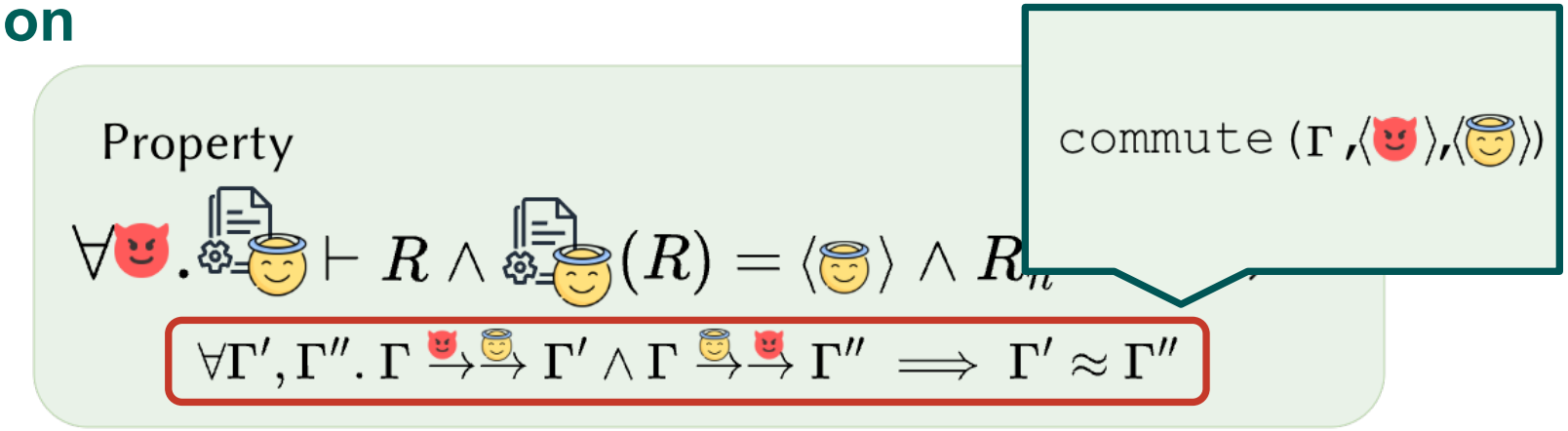
1. We identified an imprecision in the state-of-the-art frontrunning analyses, which don't account for **honest user behavior.**

Conclusion



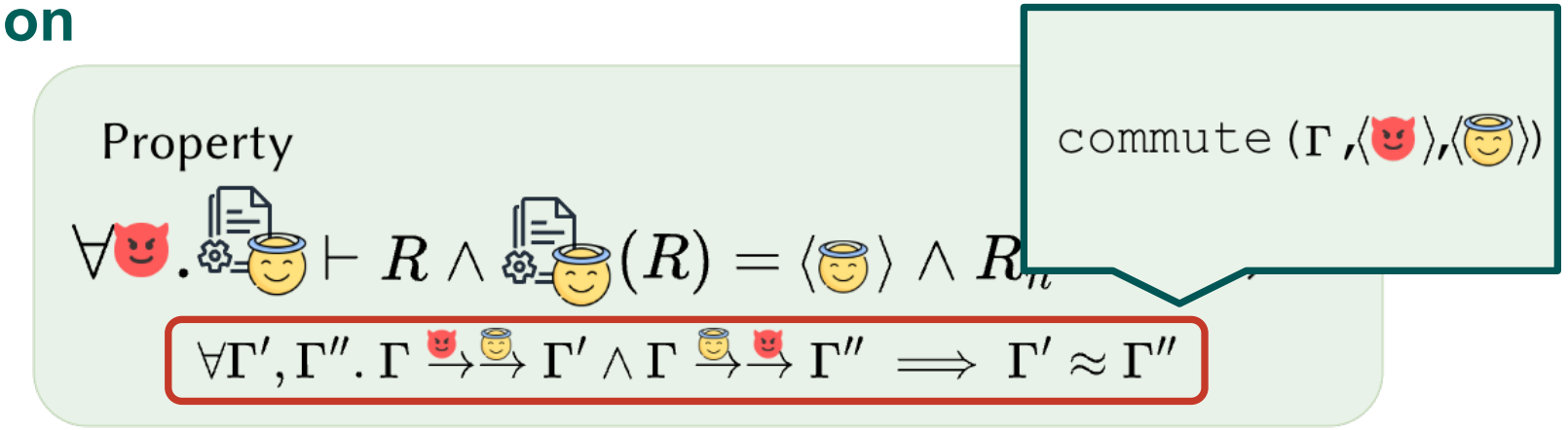
1. We identified an imprecision in the state-of-the-art frontrunning analyses, which don't account for **honest user behavior.**
2. We propose a procedure, which fixes this imprecision.

Conclusion



1. We identified an imprecision in the state-of-the-art frontrunning analyses, which don't account for **honest user behavior.**
2. We propose a procedure, which fixes this imprecision.
3. We combine **program and protocol analysis** with **lightweight DSL** to make analysis tractable.

Conclusion



1. We identified an imprecision in the state-of-the-art frontrunning analyses, which don't account for **honest user behavior.**
2. We propose a procedure, which fixes this imprecision.
3. We combine **program and protocol analysis** with **lightweight DSL** to make analysis tractable.

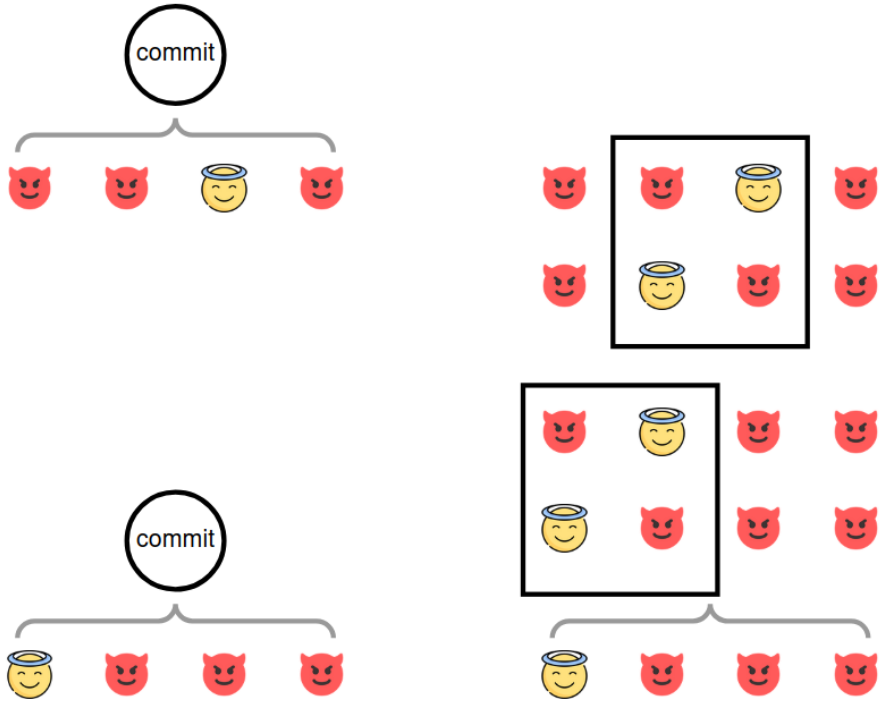
Thank you!

If you have further questions or want to collaborate, contact:

Email: sebastian.holler@mpi-sp.org, yan.farba@mpi-sp.org

I'm happy to connect on LinkedIn!

Backup slide: Soundness of protocol analysis



- Idea: We symbolically execute the user strategy S for every possible attacker and check that honest user transaction commutes