Generation of Independence Constraints in Smart Contracts for Front-running Resistance

Bachelor thesis defense

Yan Farba

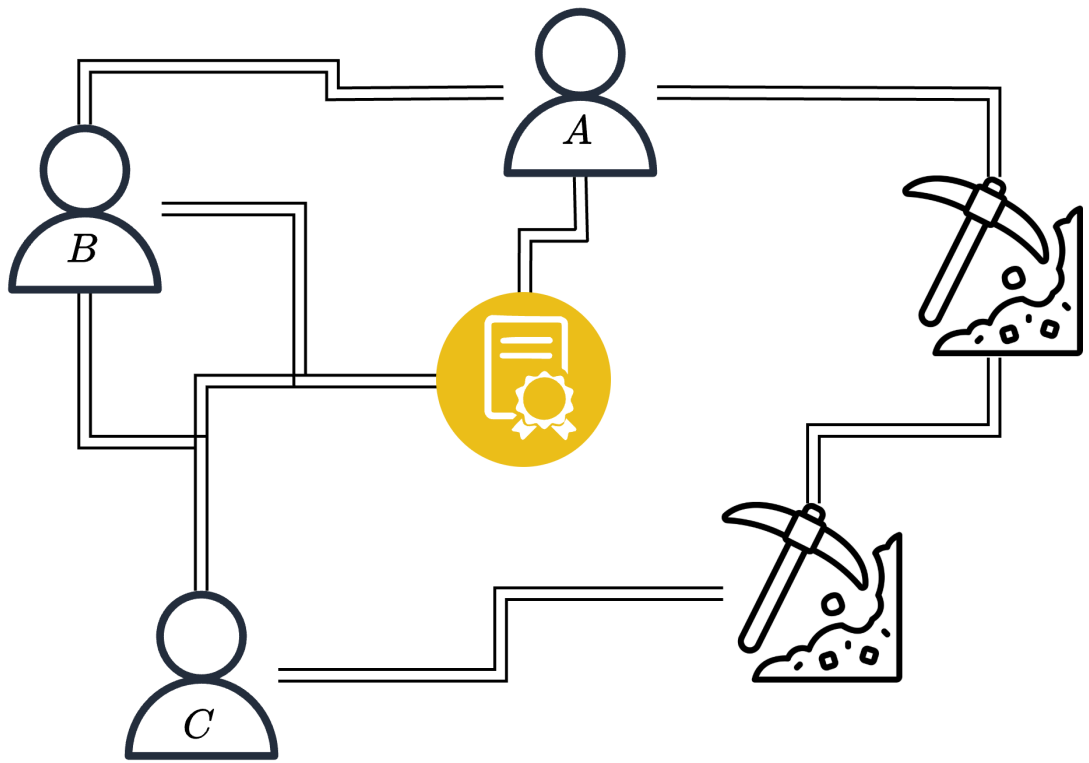
yan.farba@mpi-sp.org

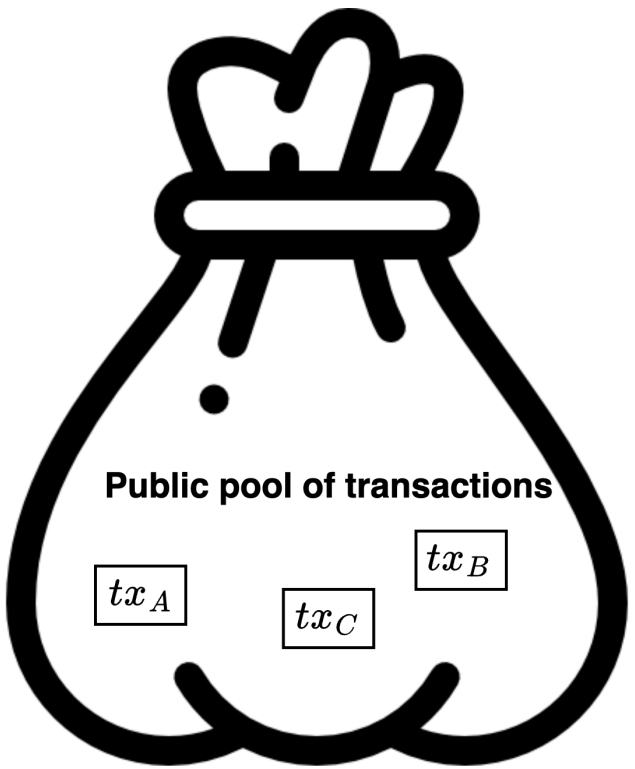
Max Planck Institute for Security and Privacy

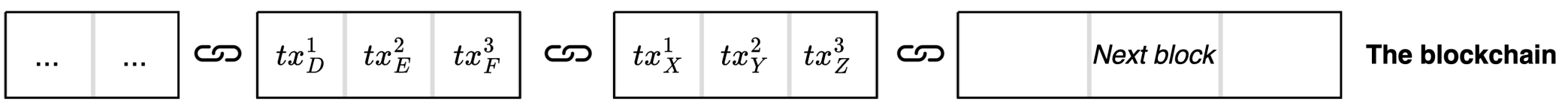
April 4, 2025

Introduction

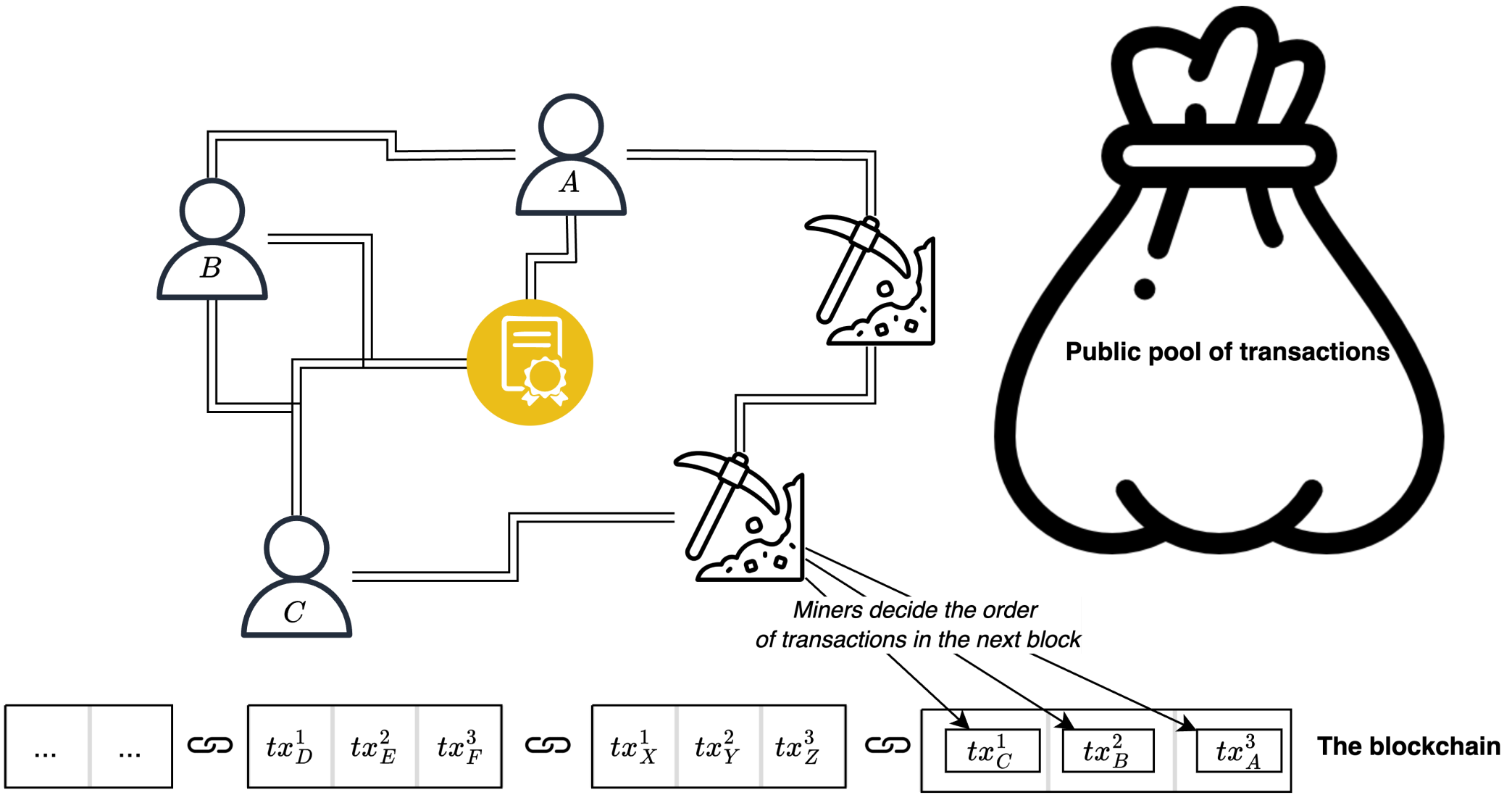
Blockchain & Smart Contracts



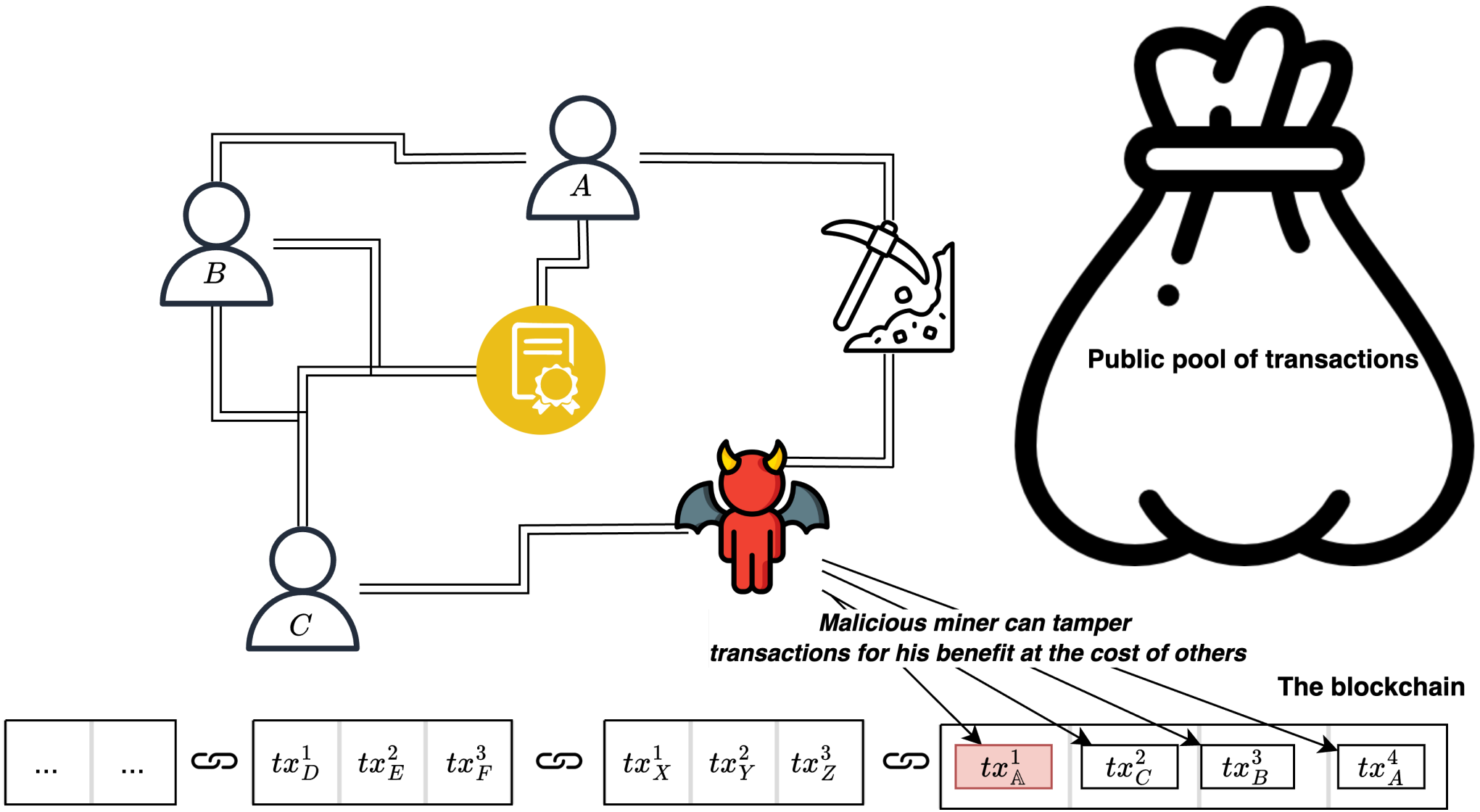




Blockchain & Smart Contracts



Blockchain & Smart Contracts



Front-running attacks

Definition from traditional finance

Front-running is an illegal practice, in which unethical brokers leverage knowledge of upcoming transaction to benefit their portfolios [5].

Front-running attacks

Definition from traditional finance

Front-running is an illegal practice, in which unethical brokers leverage knowledge of upcoming transaction to benefit their portfolios [5].

- In DeFi: miners re-order/tamper transactions to smart contracts for their benefit.

Front-running attacks

Definition from traditional finance

Front-running is an illegal practice, in which unethical brokers leverage knowledge of upcoming transaction to benefit their portfolios [5].

- In DeFi: miners re-order/tamper transactions to smart contracts for their benefit.

  → Regular users can front-run by bribing miners [4].

Front-running the `HashPuzzle`

Consider the `HashPuzzle`, a game in which the winner redeems the whole contracts balance!

```
1 contract HashPuzzle {
2     bytes32 public challenge;
3
4     function submitSolution(bytes32 _solution) public {
5         require(sha256(_solution) == challenge);
6         // transfer balance
7     }
8 }
```

Front-running the `HashPuzzle`

Consider the `HashPuzzle`, a game in which the winner redeems the whole contracts balance!

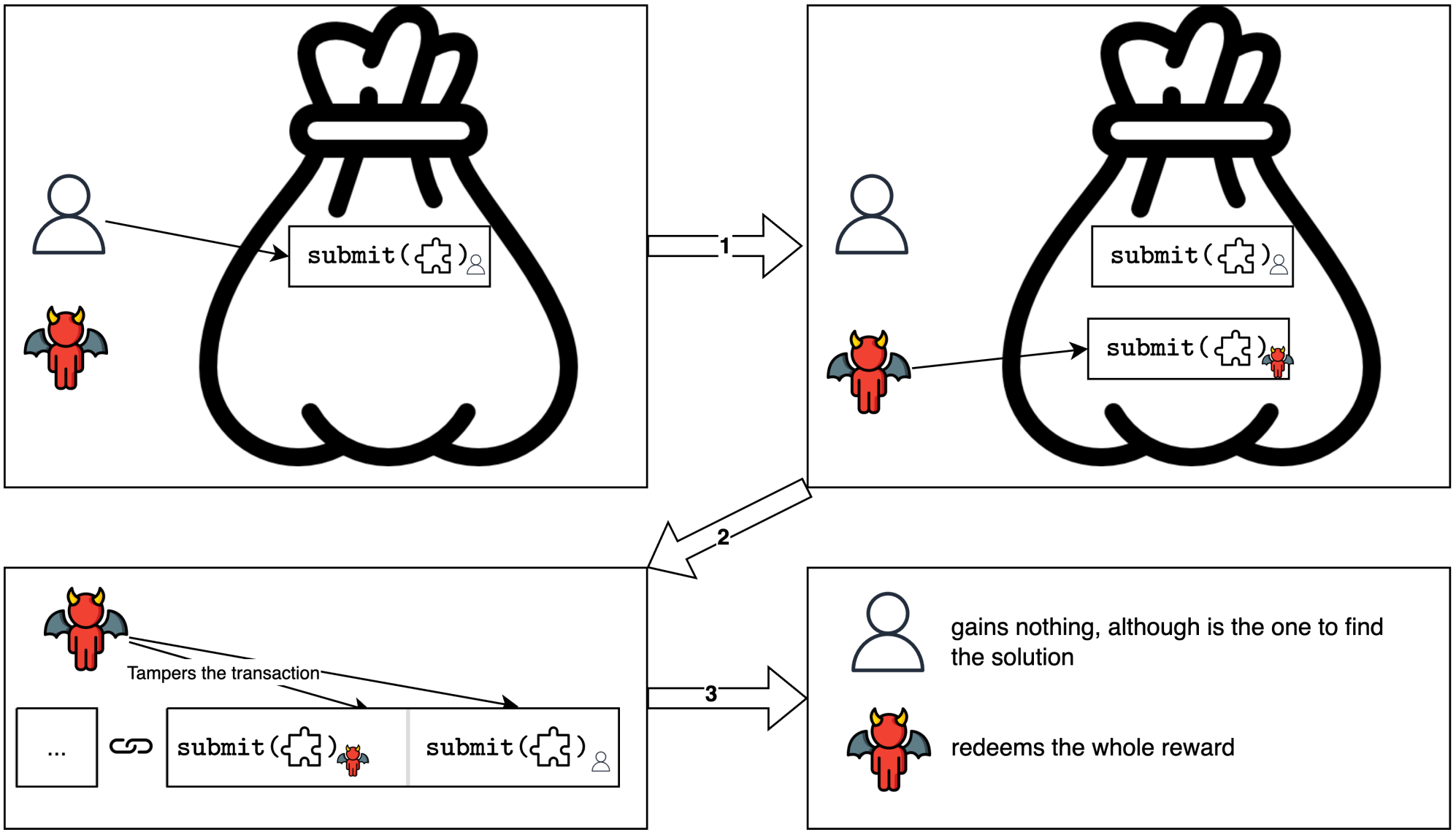
```
1 contract HashPuzzle {
2     bytes32 public challenge;
3
4     function submitSolution(bytes32 _solution) public {
5         require(sha256(_solution) == challenge);
6         // transfer balance
7     }
8 }
```

Front-running the `HashPuzzle`

Consider the `HashPuzzle`, a game in which the winner redeems the whole contracts balance!

```
1 contract HashPuzzle {
2     bytes32 public challenge;
3
4     function submitSolution(bytes32 _solution) public {
5         require(sha256(_solution) == challenge);
6         // transfer balance
7     }
8 }
```

An adversary can front-run `HashPuzzle` as follows:



Threats of Front-running

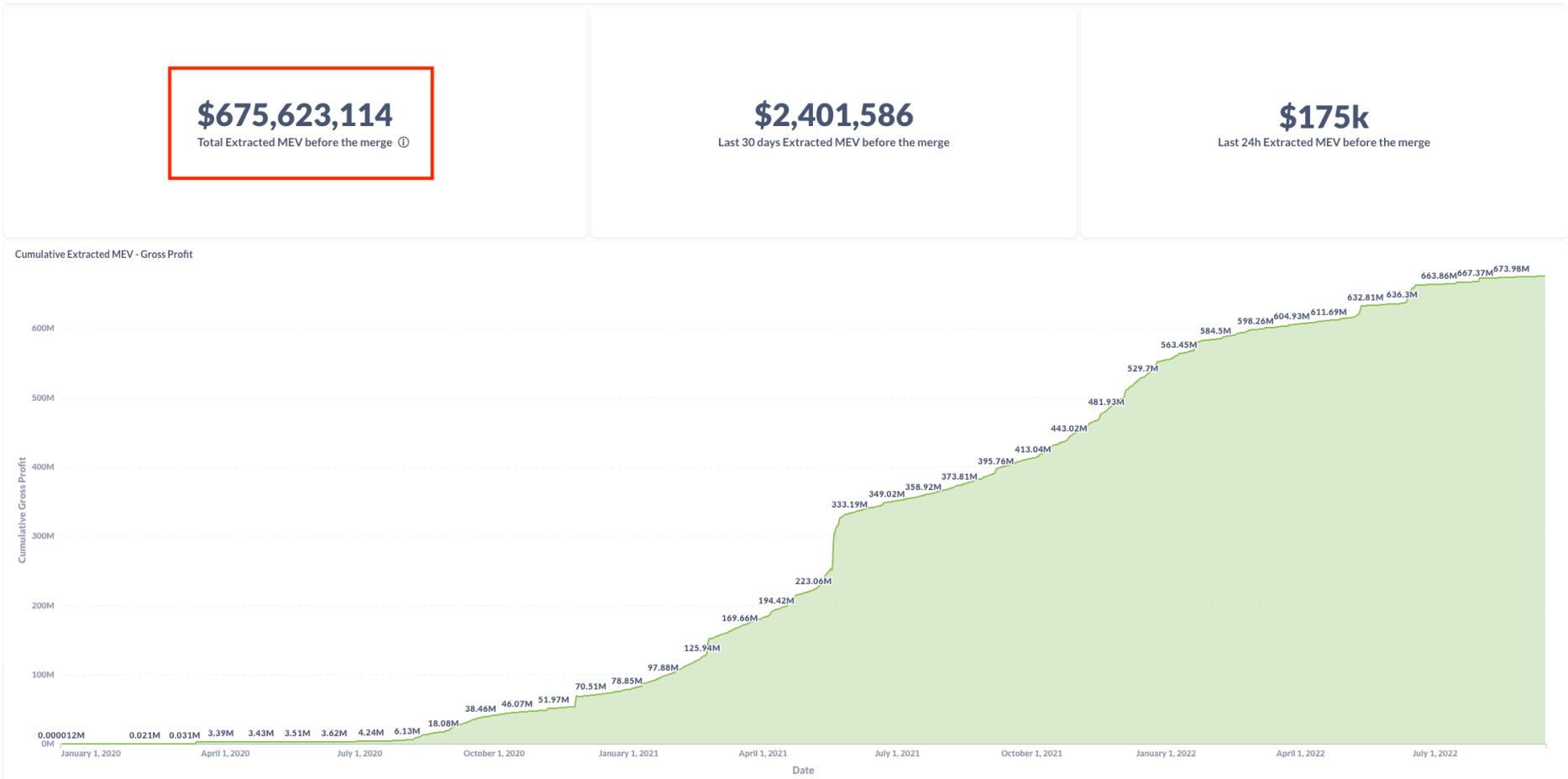


Figure 1: Extraction from Ethereum due to front-running [1]

Threats of Front-running

- Front-running undermines the fairness of the consensus protocol [4].

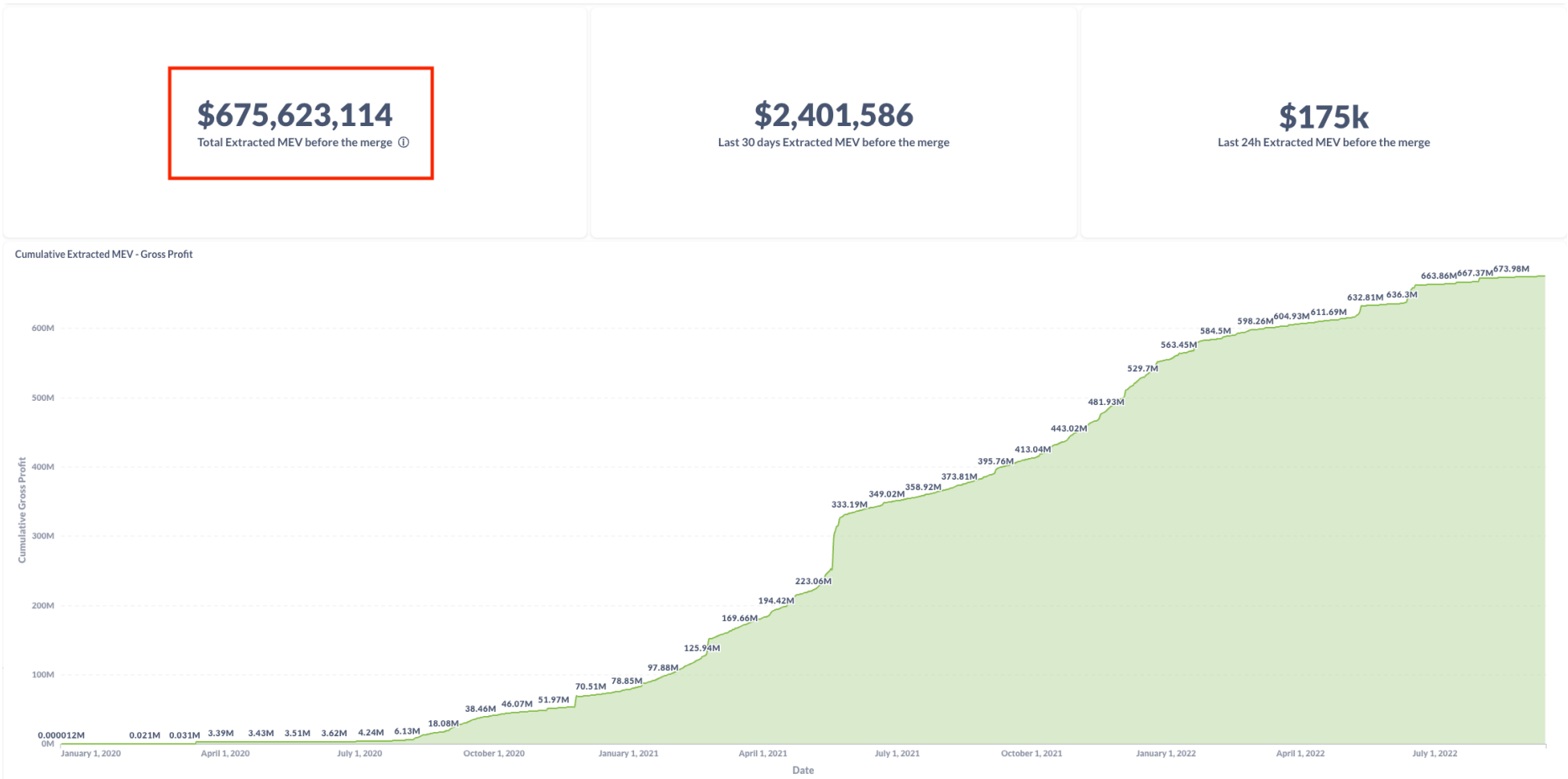
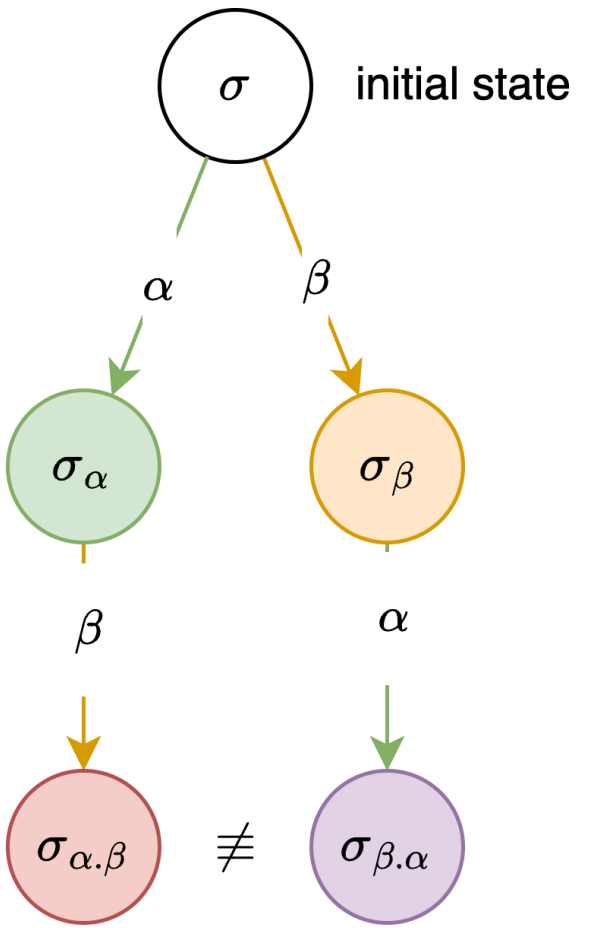


Figure 1: Extraction from Ethereum due to front-running [1]

Transaction Order Dependence

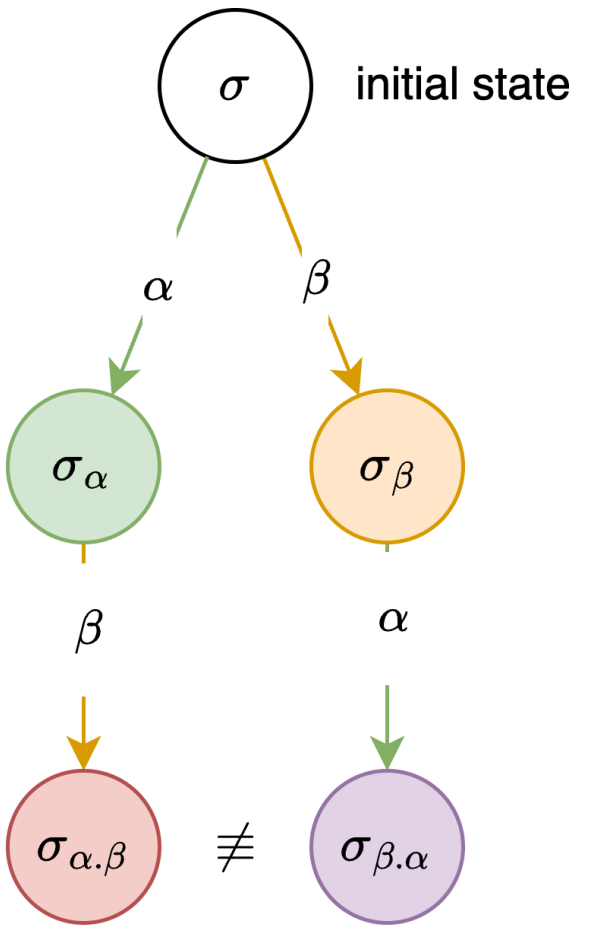
A property on which state-of-the-art front-running analysis is based



Transaction Order Dependence

Transaction Order Dependence

A property on which state-of-the-art front-running analysis is based



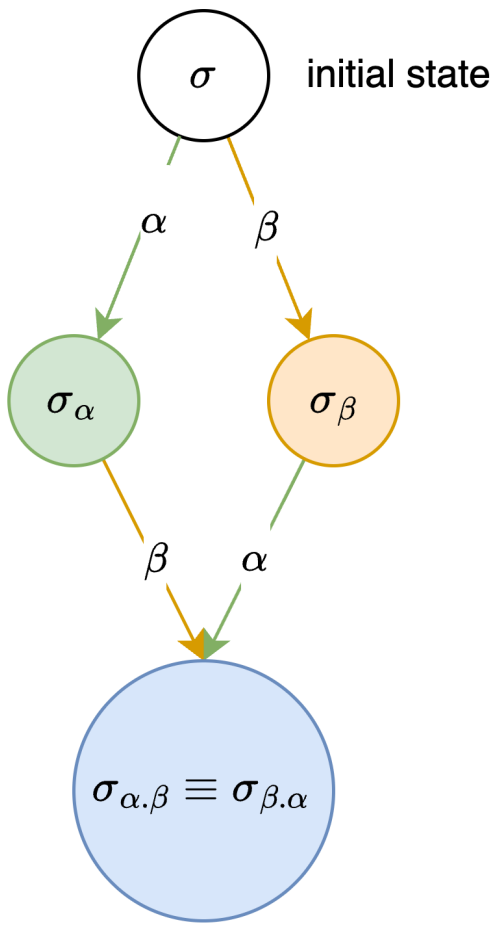
Transaction Order Dependence

- TOD is a primary verification target for identifying front-running [3, 6, 8, 9].

Motivation

Limitations of TOD analysis

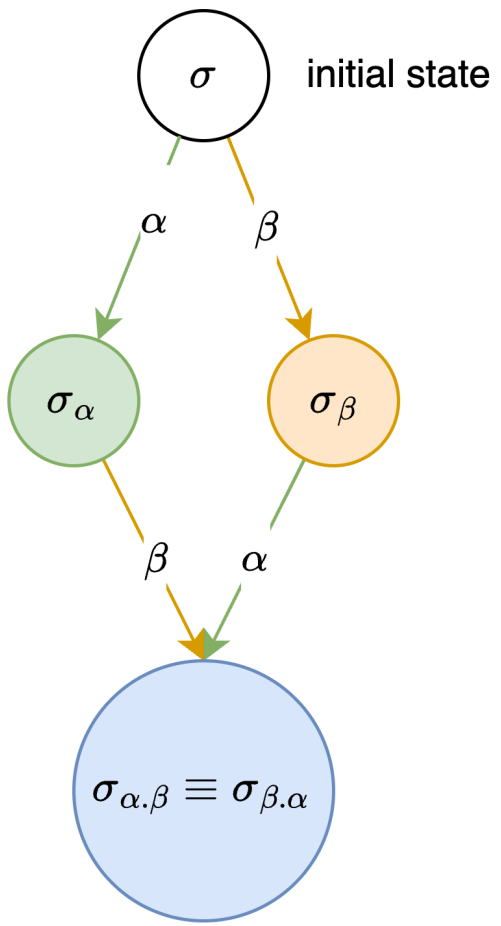
1. No TOD $\Longrightarrow$ $\forall \alpha, \beta :$



Motivation

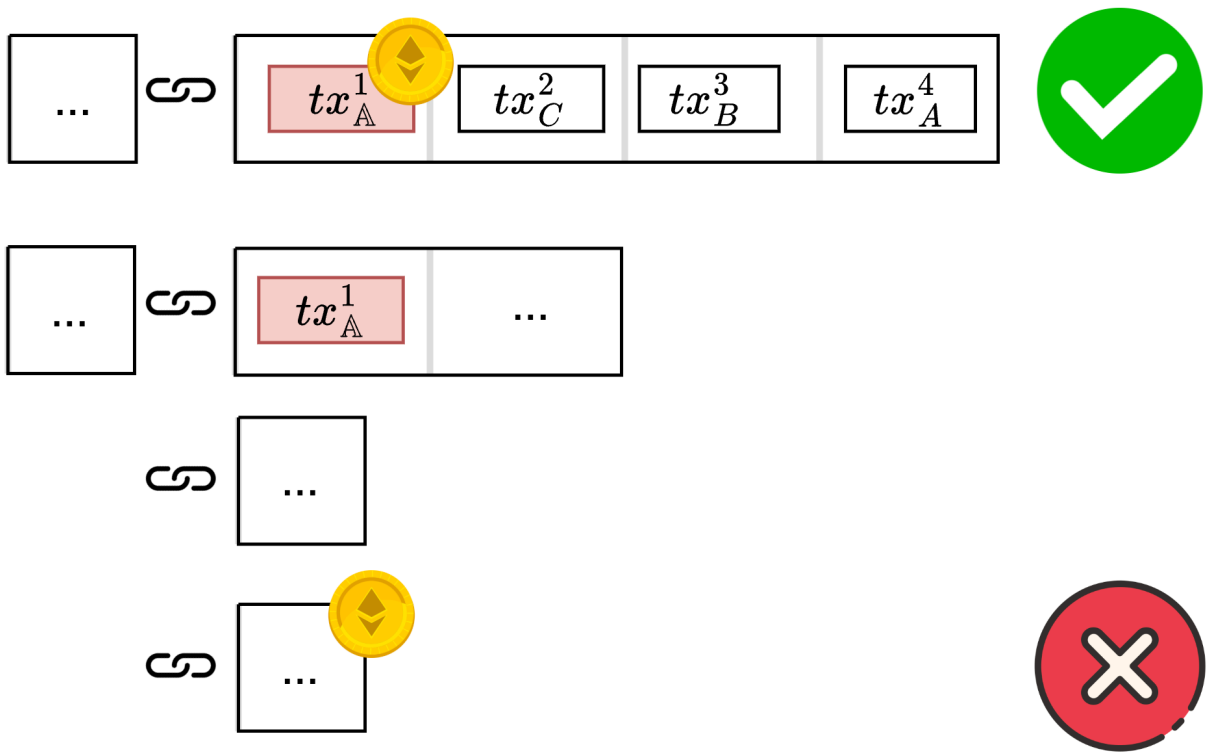
Limitations of TOD analysis

1. No TOD $\Longrightarrow$ $\forall \alpha, \beta :$



2. Using heuristics to express TOD

***Example**: MEV as TOD* is a profit adversary makes



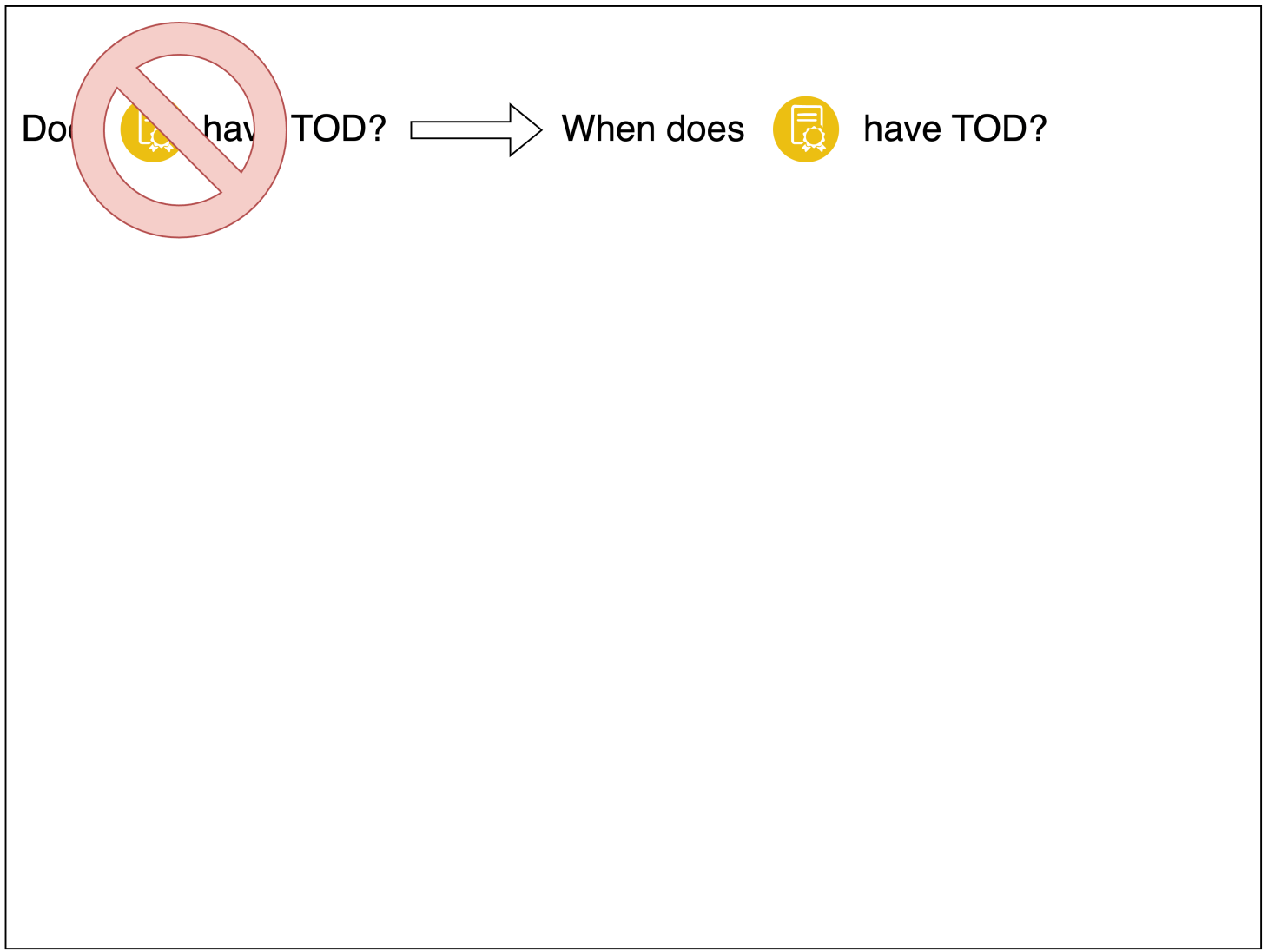
Contribution

Independence constraints as a means against front-running

Does  have TOD?

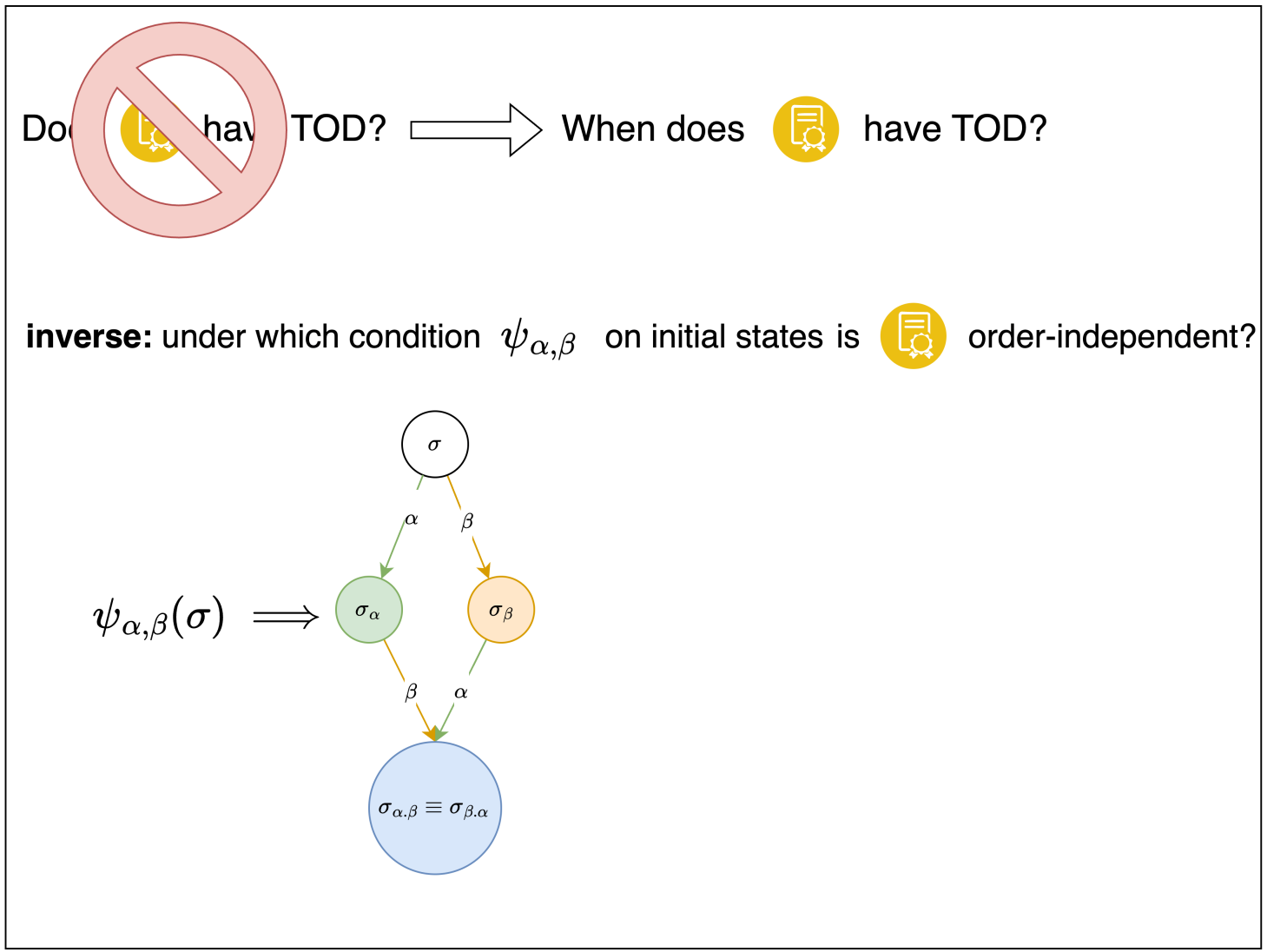
Contribution

Independence constraints as a means against front-running



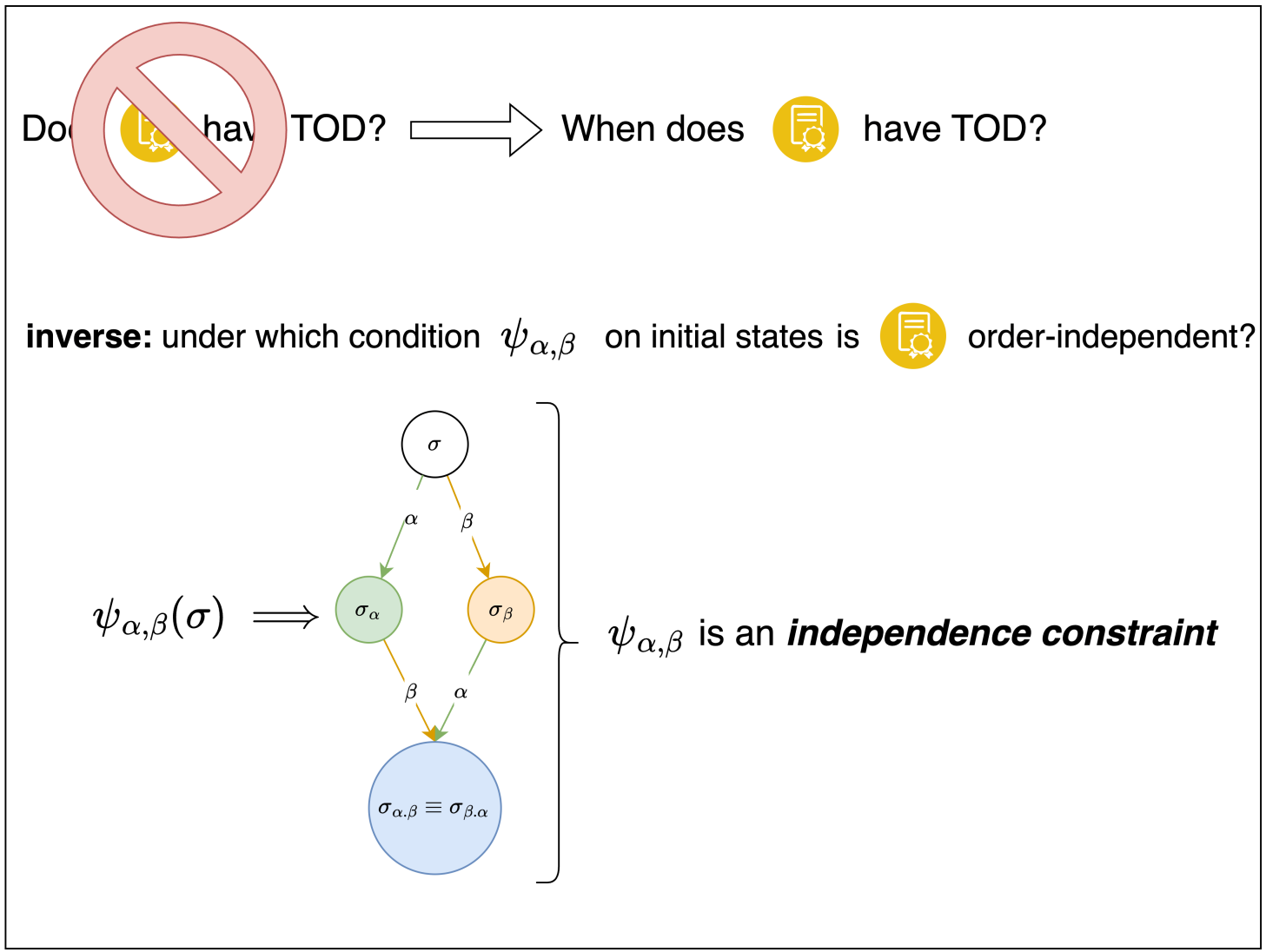
Contribution

Independence constraints as a means against front-running



Contribution

Independence constraints as a means against front-running



Contribution

Independence constraints as a means against front-running

- Our procedure **generates** independence constraints for a smart contract.
- We implement it as an end-to-end toolchain.

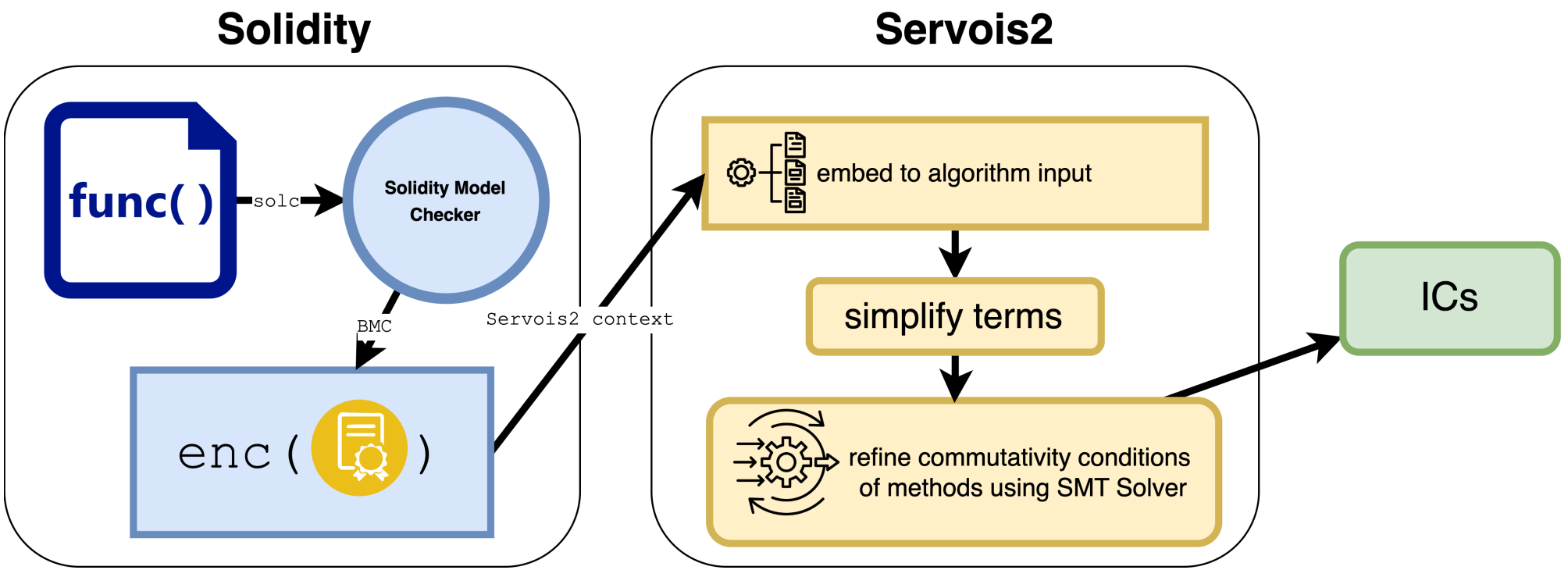
Full Procedure

Design

- Static-analysis procedure.
- Executes smart contracts *symbolically*.

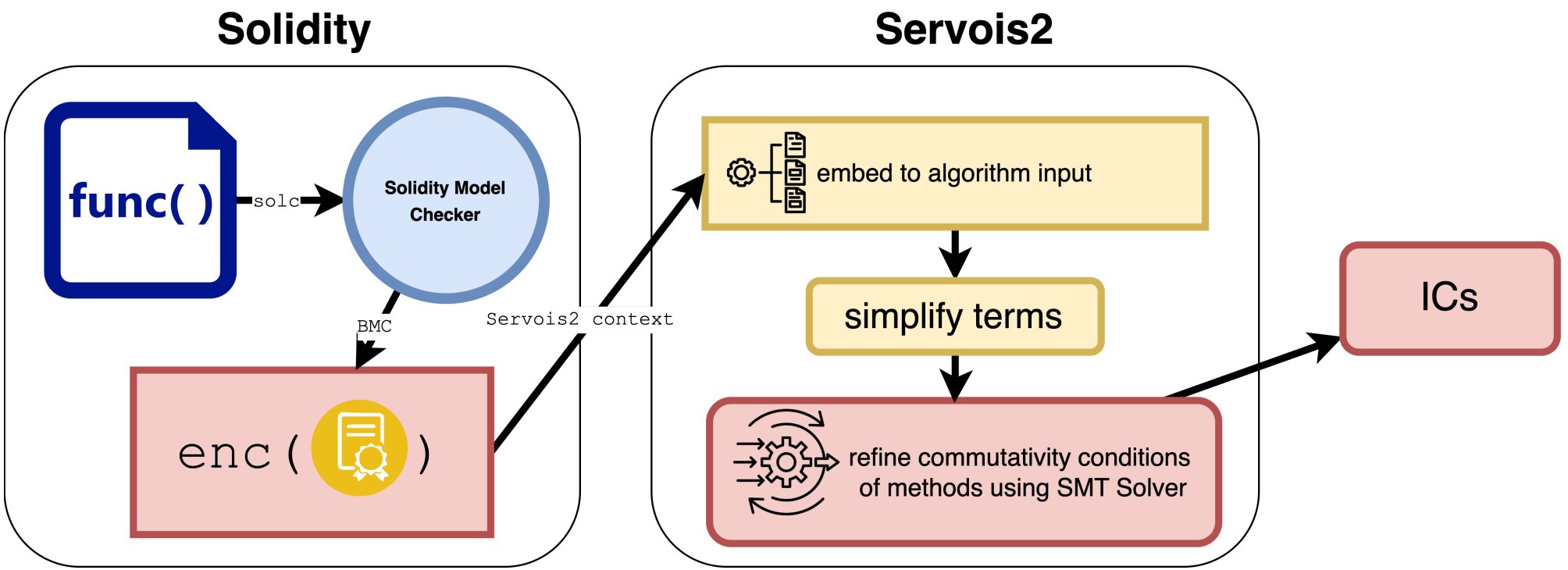
Design

- Static-analysis procedure.
- Executes smart contracts *symbolically*.



Design

- Static-analysis procedure.
- Executes smart contracts *symbolically*.



Independence Constraints Generation

Example: ICs of `HashPuzzle`

```
1 function submitSolution(bytes32 _solution) public {
2     require(sha256(_solution) == challenge);
3     // transfer balance
4 }
```

$$\psi^1_{\text{👤},\text{😈}} := \texttt{bal} = 0$$

Example: ICs of `HashPuzzle`

```
1 function submitSolution(bytes32 _solution) public {
2     require(sha256(_solution) == challenge);
3     // transfer balance
4 }
```

$$\psi^1_{\text{👤},\text{😈}} := \texttt{bal} = 0$$

$$\psi^2_{\text{👤},\text{😈}} := \texttt{sol}_{\text{👤}} \neq \texttt{sol}_{\text{😈}}$$

$$\psi^3_{\text{👤},\text{😈}} := \texttt{sol}_{\text{👤}} \neq sha256(\texttt{chal})$$

$$\psi^4_{\text{👤},\text{😈}} := \texttt{sol}_{\text{😈}} \neq sha256(\texttt{chal})$$

Example: ICs of `HashPuzzle`

```
1 function submitSolution(bytes32 _solution) public {
2     require(sha256(_solution) == challenge);
3     // transfer balance
4 }
```

$$\psi^1_{\text{👤},\text{😈}} := \texttt{bal} = 0$$

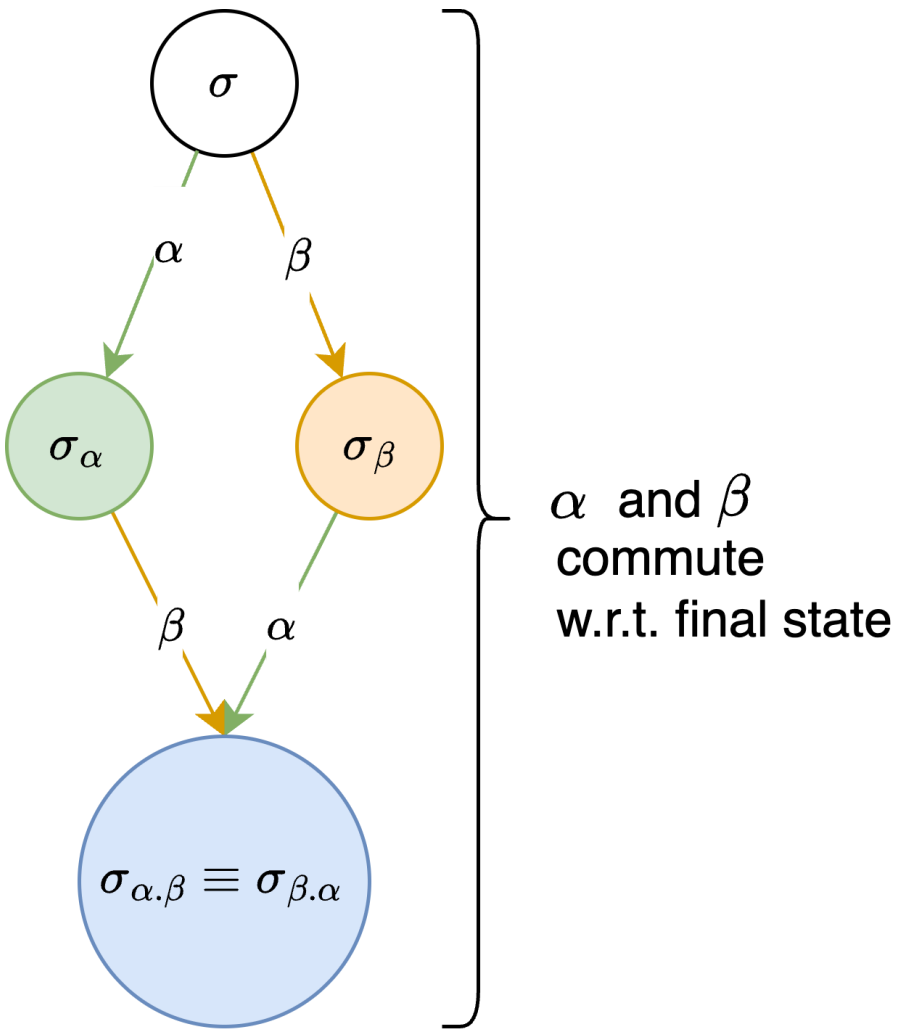
$$\psi^2_{\text{👤},\text{😈}} := \texttt{sol}_{\text{👤}} \neq \texttt{sol}_{\text{😈}}$$

$$\psi^3_{\text{👤},\text{😈}} := \texttt{sol}_{\text{👤}} \neq sha256(\texttt{chal})$$

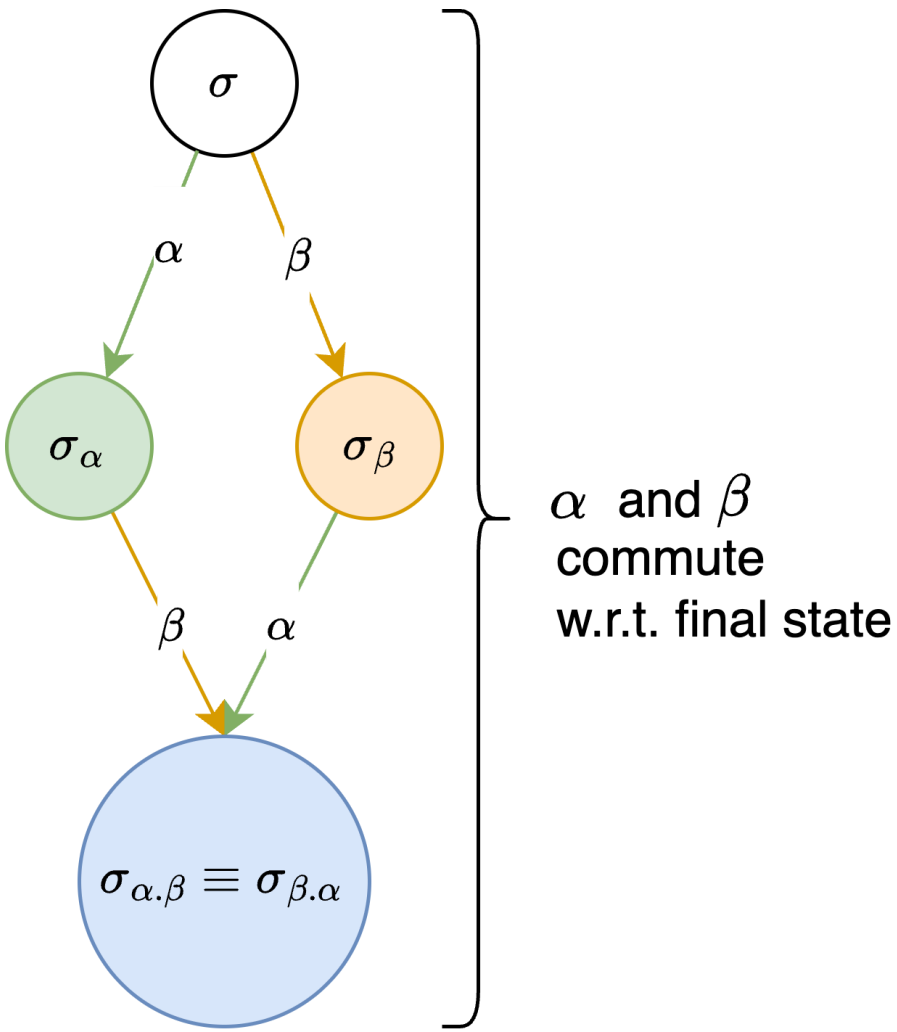
$$\psi^4_{\text{👤},\text{😈}} := \texttt{sol}_{\text{😈}} \neq sha256(\texttt{chal})$$

} under these conditions **HashPuzzle** is invulnerable to front-running

Independence Constraints & Commutativity Conditions

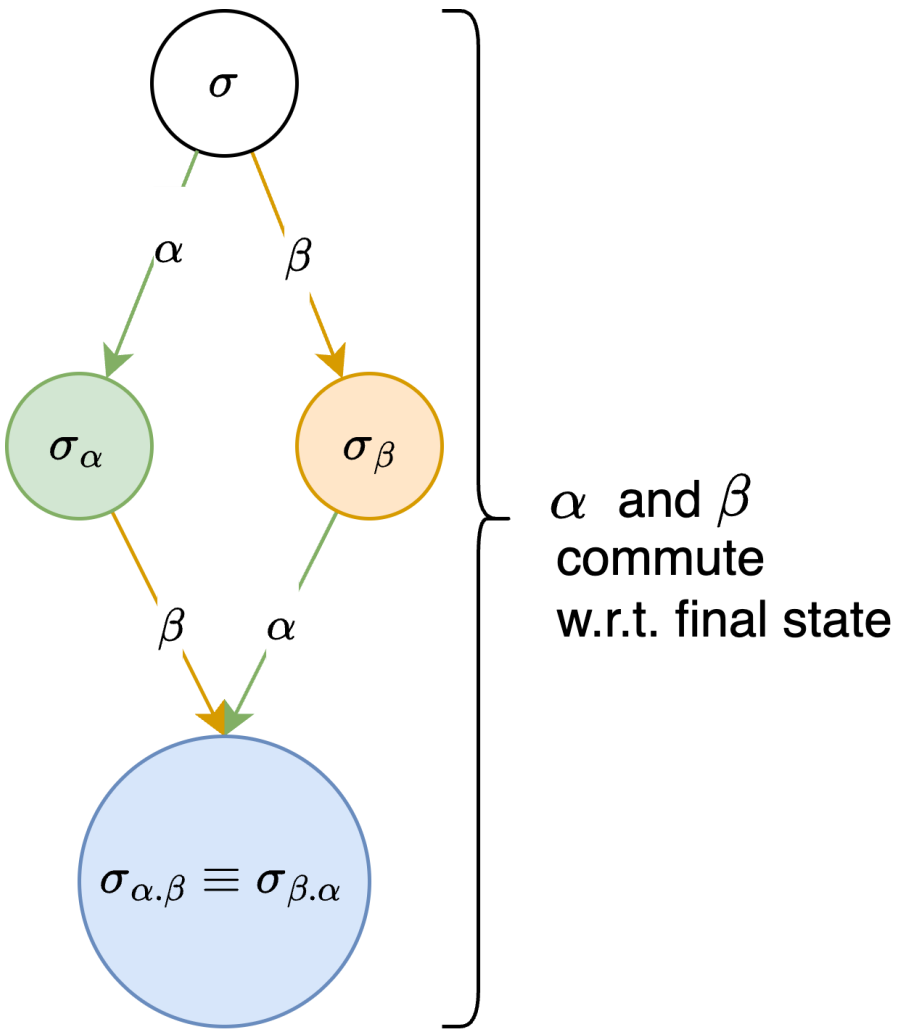


Independence Constraints & Commutativity Conditions



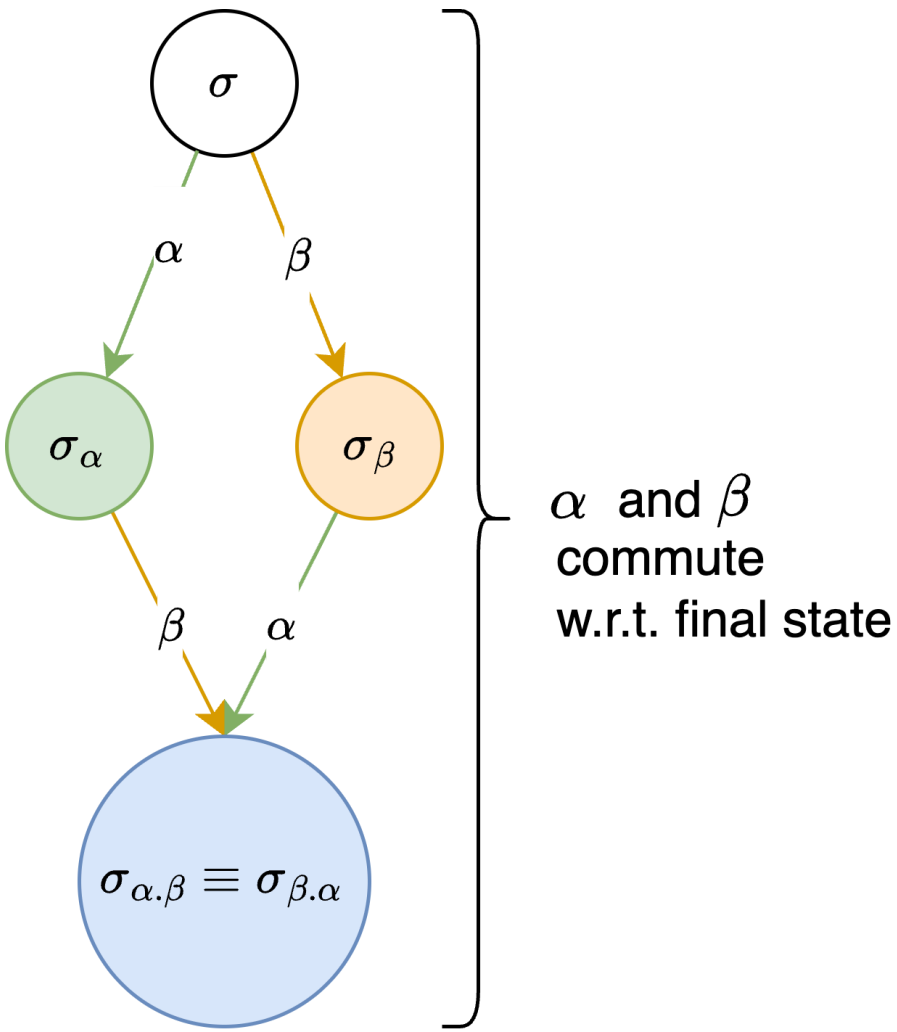
- Smart contracts are similar to threads in the concurrent environment [7].

Independence Constraints & Commutativity Conditions



- Smart contracts are similar to threads in the concurrent environment [7].
- Apply commutativity condition refinement algorithm [2].

Independence Constraints & Commutativity Conditions



- Smart contracts are similar to threads in the concurrent environment [7].
- Apply commutativity condition refinement algorithm [2].
  - → Computes subregions of a state space, in which α and β commute.

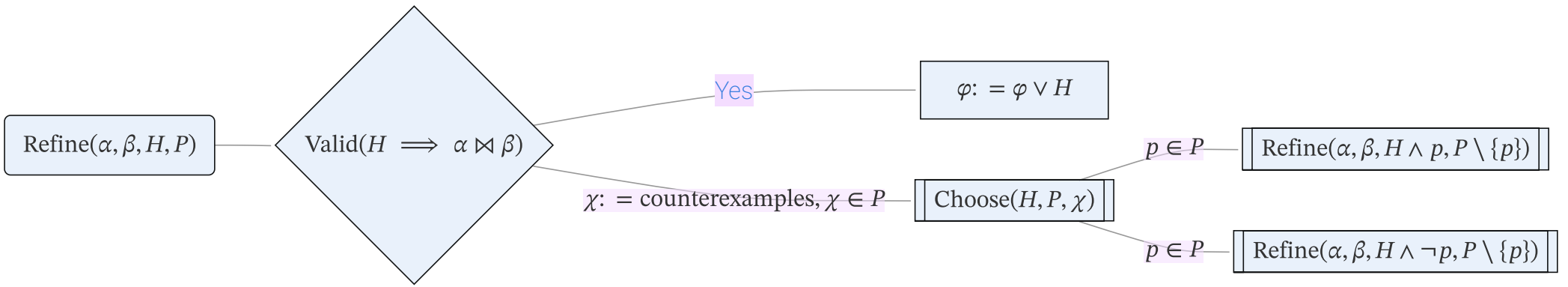
Commutativity Condition Refinement Algorithm [2]

Goal: compute subregions, in which α and β commute.

Commutativity Condition Refinement Algorithm [2]

Goal: compute subregions, in which α and β commute.

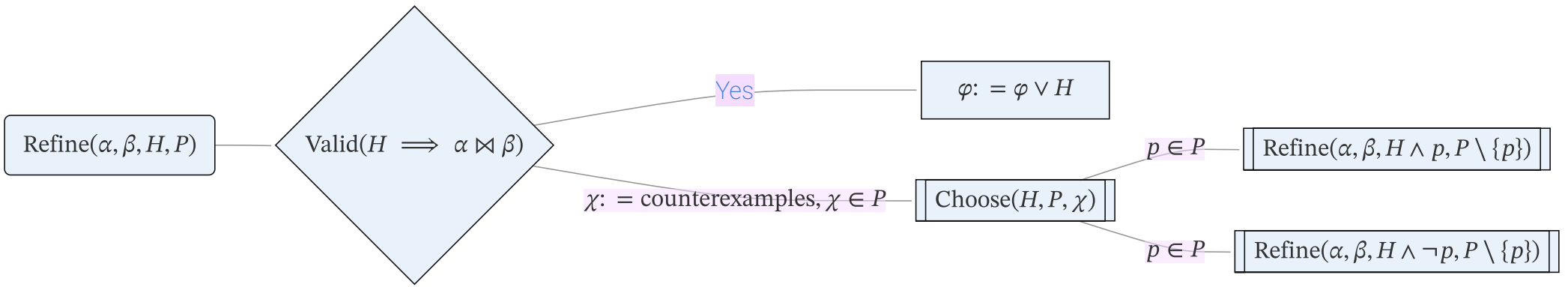
Figure 2: Flowchart of the commutativity condition refinement algorithm.



Commutativity Condition Refinement Algorithm [2]

Goal: compute subregions, in which α and β commute.

Figure 2: Flowchart of the commutativity condition refinement algorithm.

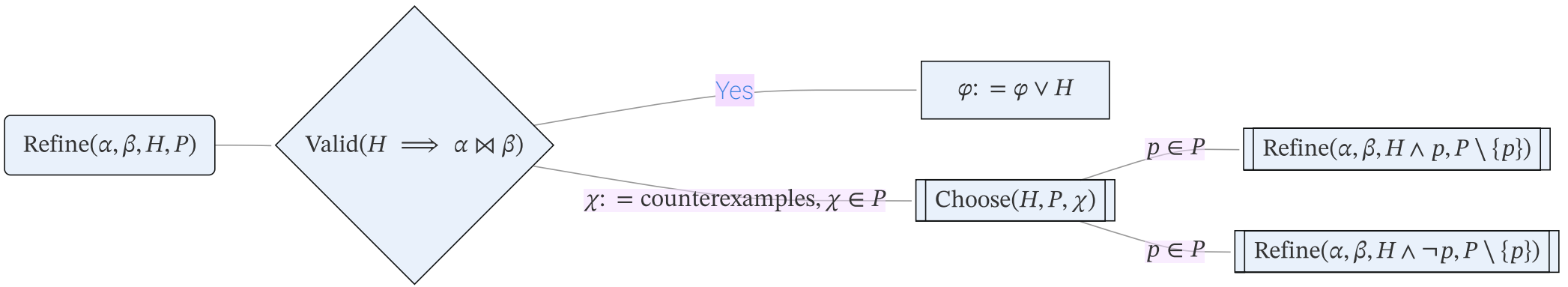


- "$\bowtie$" is the commutativity relation.

Commutativity Condition Refinement Algorithm [2]

Goal: compute subregions, in which α and β commute.

Figure 2: Flowchart of the commutativity condition refinement algorithm.

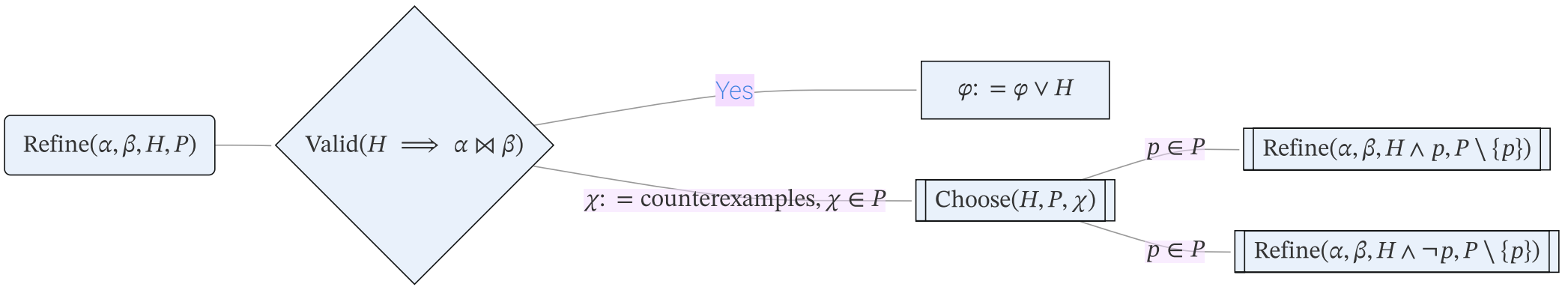


- "$\bowtie$" is the commutativity relation.
- Subregion H is a (possibly infinite) conjunction of conditions on states, that hold in it.

Commutativity Condition Refinement Algorithm [2]

Goal: compute subregions, in which α and β commute.

Figure 2: Flowchart of the commutativity condition refinement algorithm.

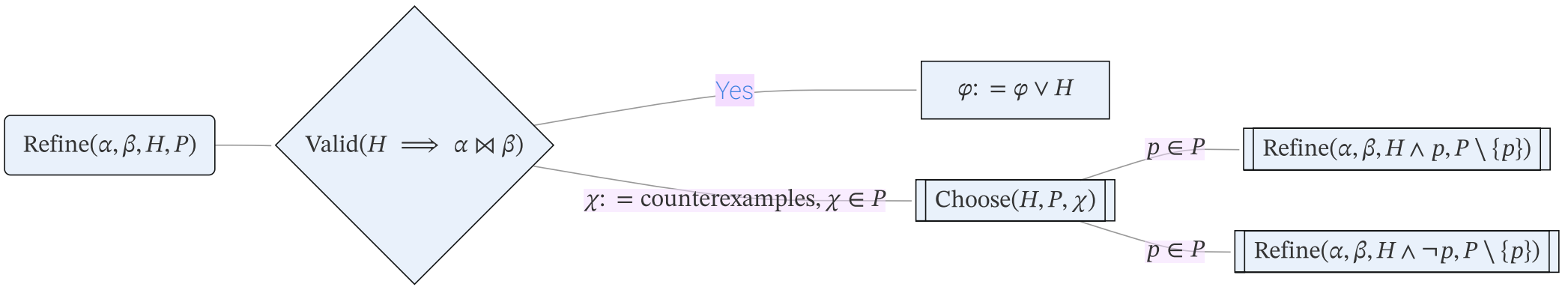


- "$\bowtie$" is the commutativity relation.
- Subregion H is a (possibly infinite) conjunction of conditions on states, that hold in it.
- Predicates $\mathcal{P}$ are building blocks of the algorithm.

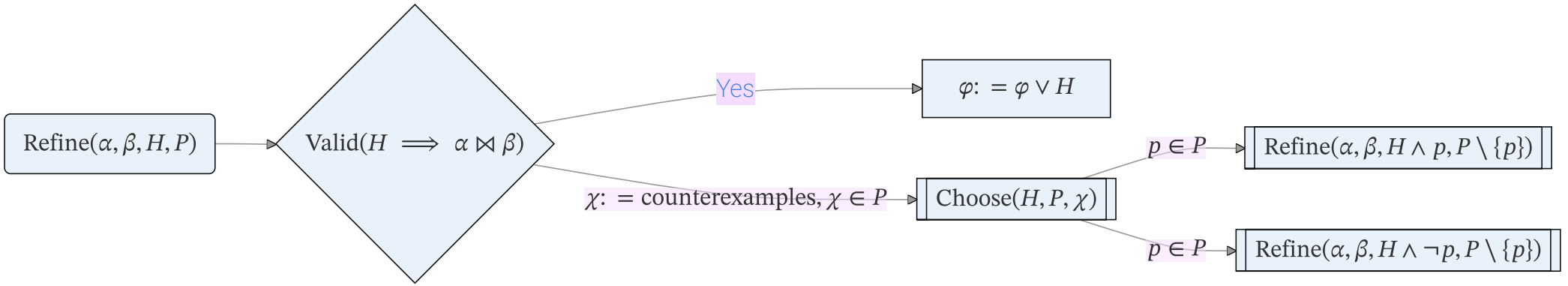
Commutativity Condition Refinement Algorithm [2]

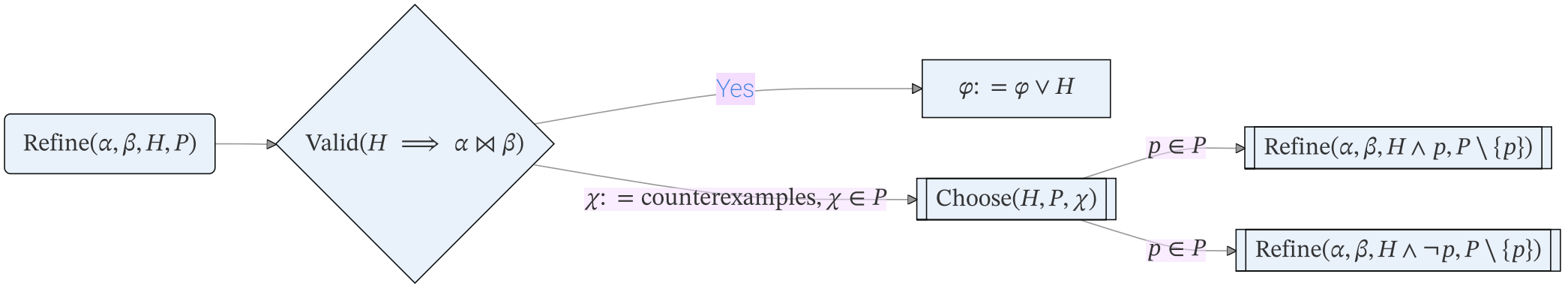
Goal: compute subregions, in which α and β commute.

Figure 2: Flowchart of the commutativity condition refinement algorithm.

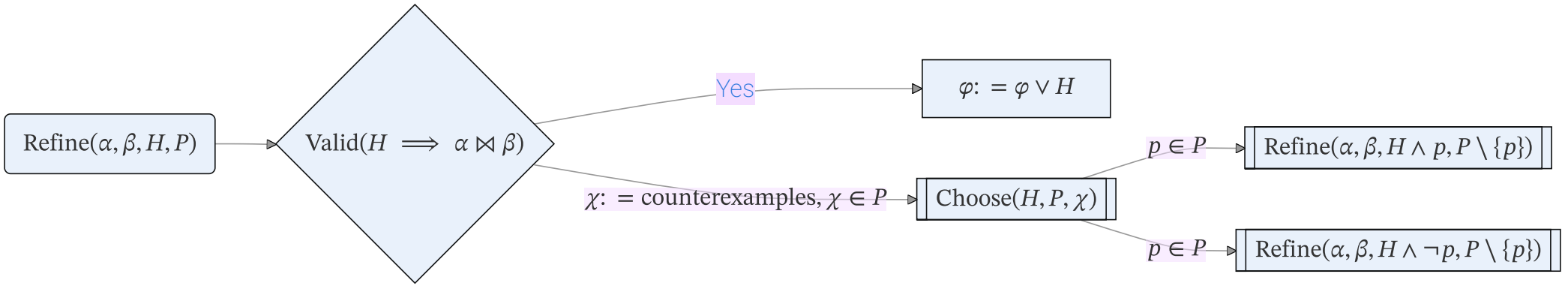


- "$\bowtie$" is the commutativity relation.
- Subregion H is a (possibly infinite) conjunction of conditions on states, that hold in it.
- Predicates $\mathcal{P}$ are building blocks of the algorithm.
  - → Built from relations and terms from the methods.

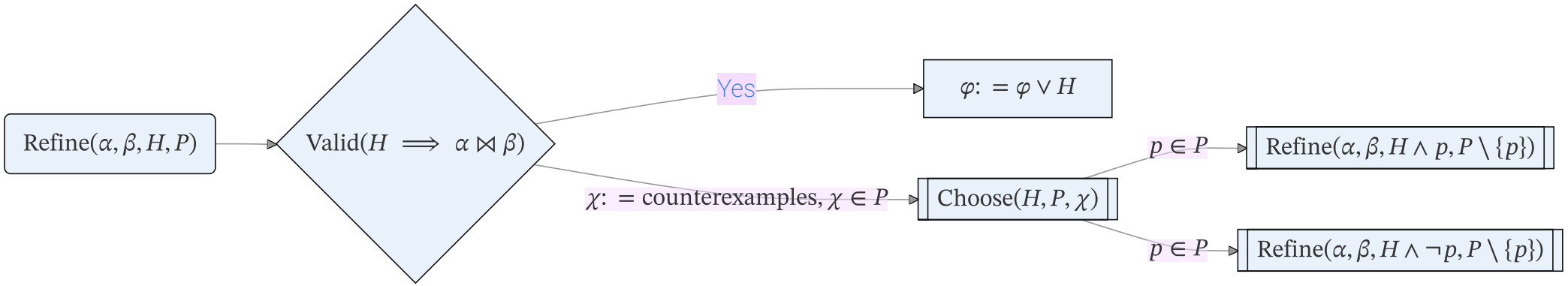

Refine(α, β, H, P)
Valid$(H \implies \alpha \bowtie \beta)$
Yes
$\varphi := \varphi \vee H$
$\chi :=$ counterexamples, $\chi \in P$
Choose(H, P, χ)
$p \in P$
Refine$(\alpha, \beta, H \wedge p, P \setminus \{p\})$
$p \in P$
Refine$(\alpha, \beta, H \wedge \neg p, P \setminus \{p\})$



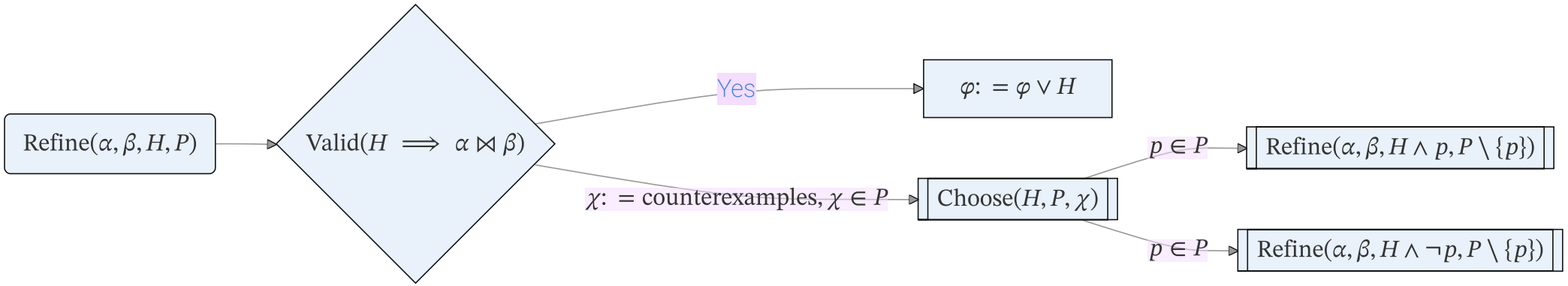
- Observation: when $\mathbf{Valid}(H \implies \alpha \bowtie \beta)$, then:



- Observation: when $\mathtt{Valid}(H \implies \alpha \bowtie \beta)$, then:
  - → in H the smart contract is order-independent,

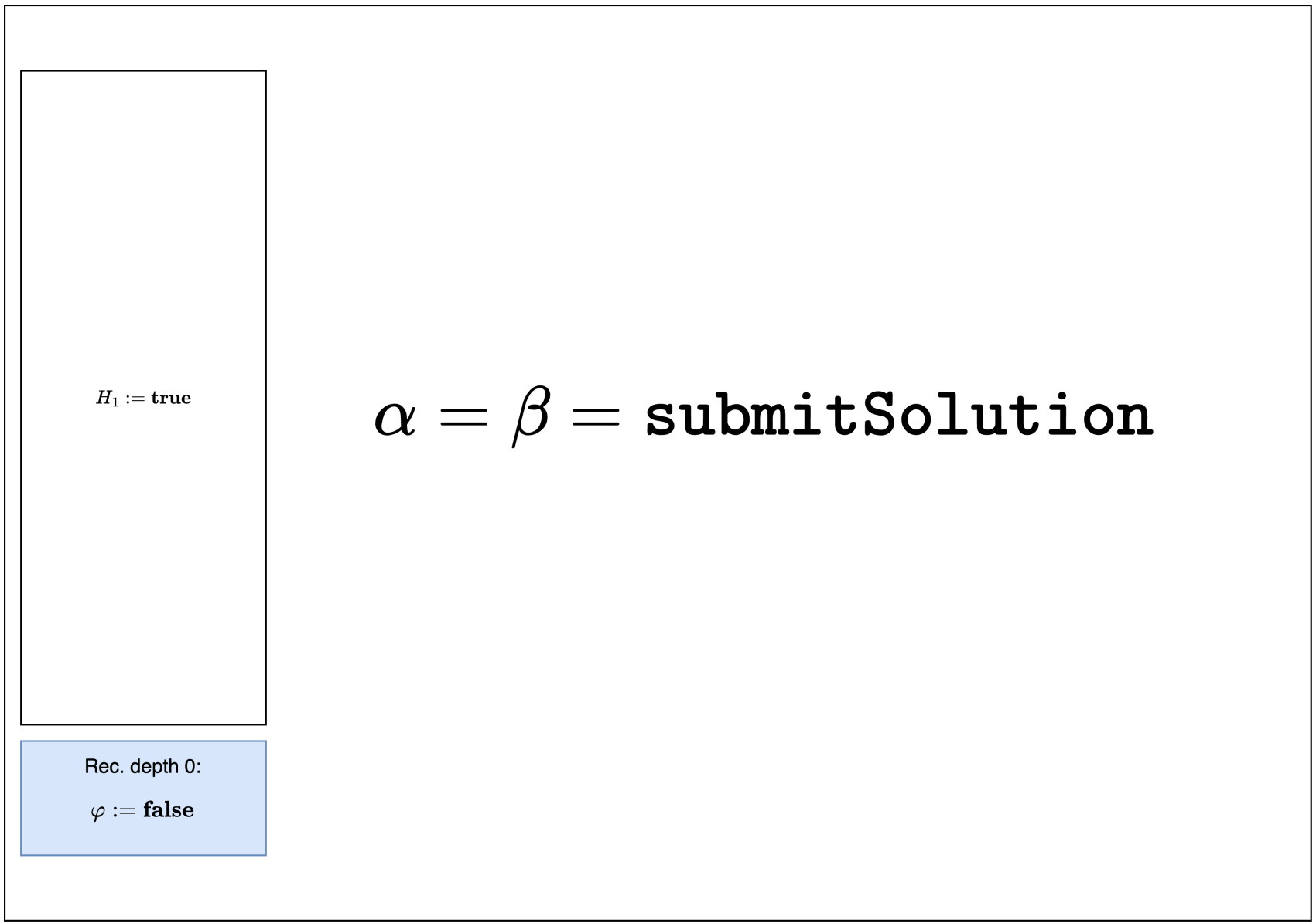


- Observation: when $\texttt{Valid}(H \implies \alpha \bowtie \beta)$, then:
  - → in H the smart contract is order-independent,
  - → $H = \psi_1 \wedge \psi_2 \wedge \ldots$ is **an independence constraint**.

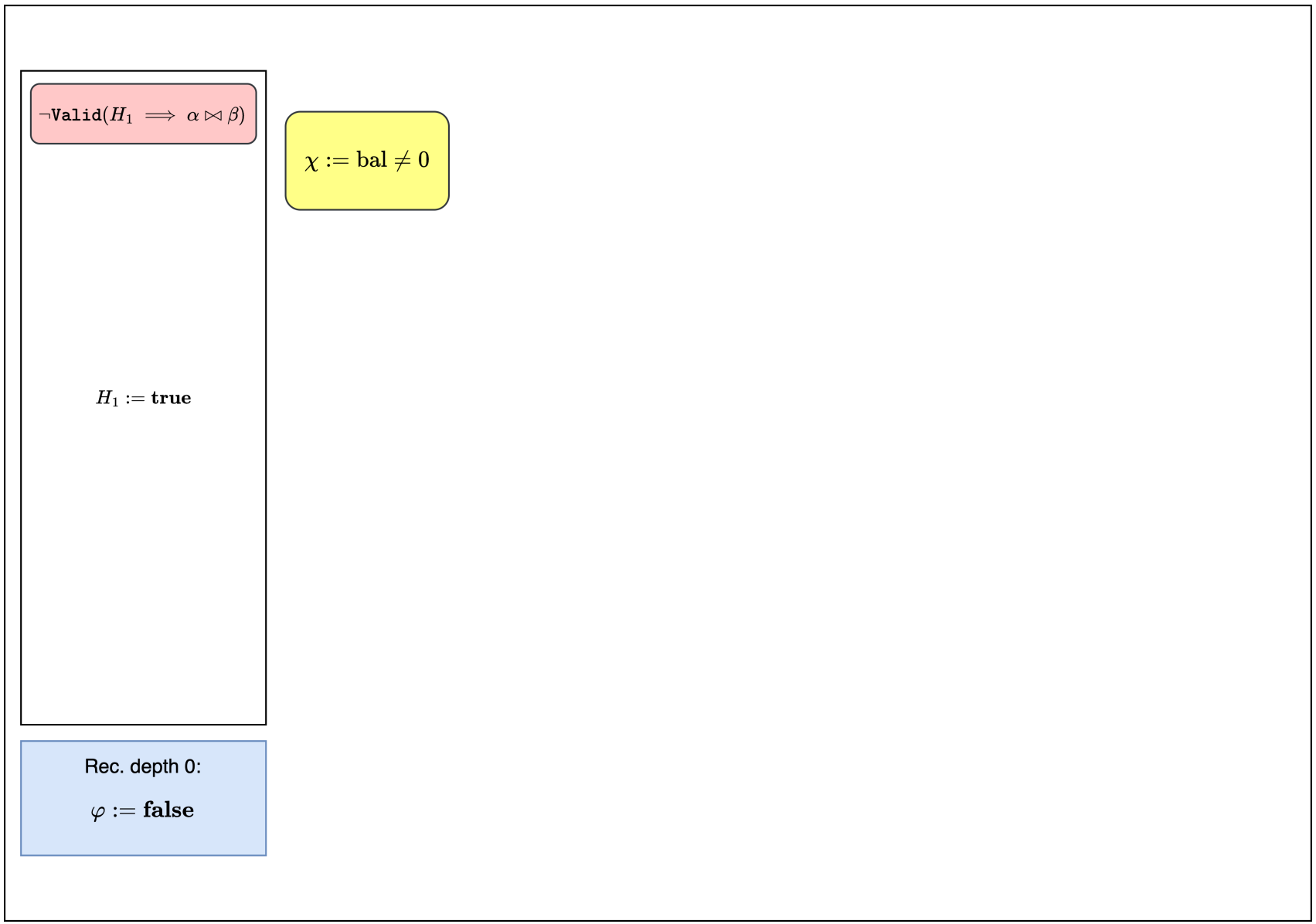


- Observation: when $\texttt{Valid}(H \implies \alpha \bowtie \beta)$, then:
  - → in H the smart contract is order-independent,
  - → $H = \psi_1 \wedge \psi_2 \wedge \ldots$ is **an independence constraint**.
- φ is a disjunction of all independence constraints.

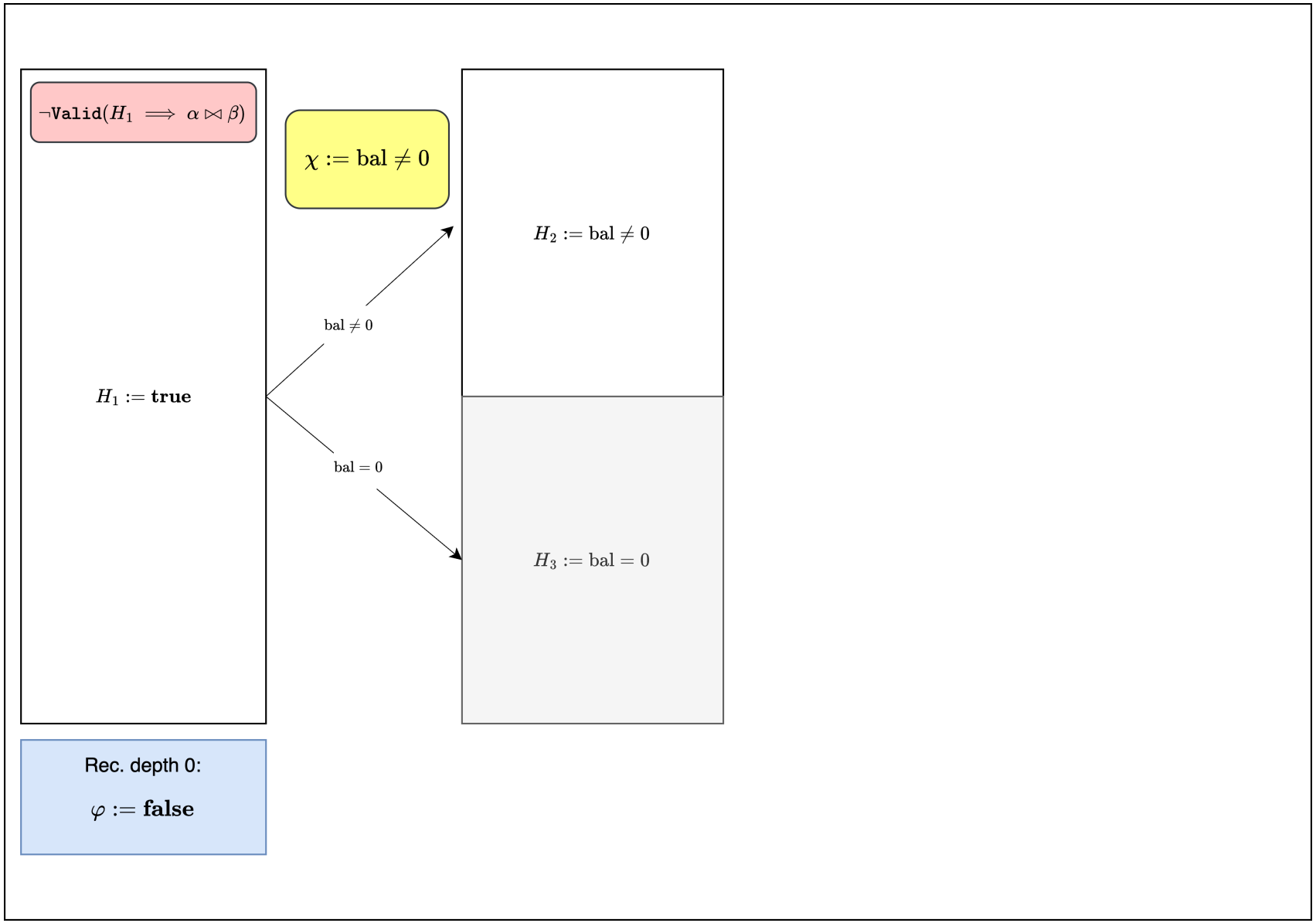
Algorithm on `HashPuzzle`



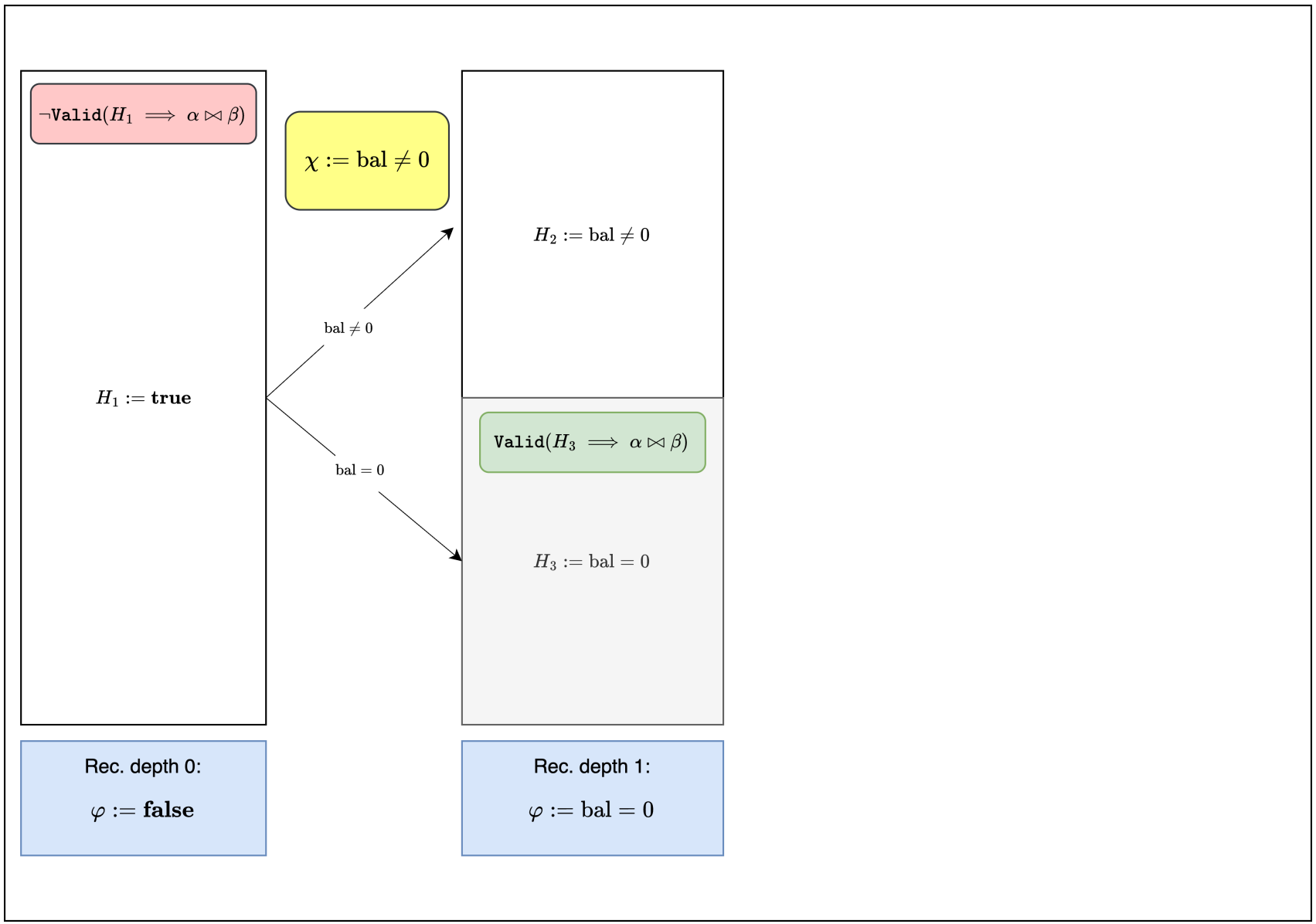
Algorithm on `HashPuzzle`



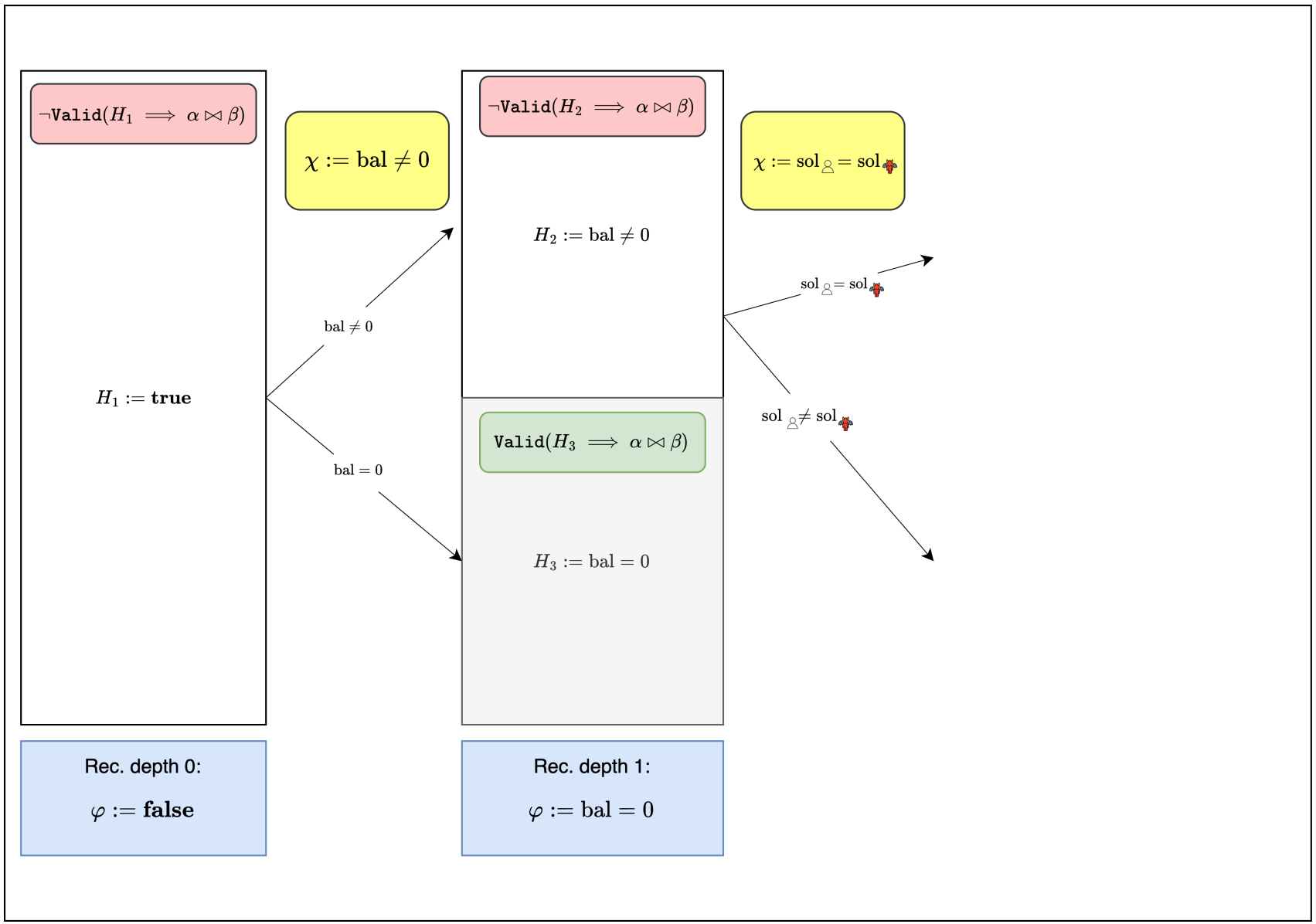
Algorithm on HashPuzzle



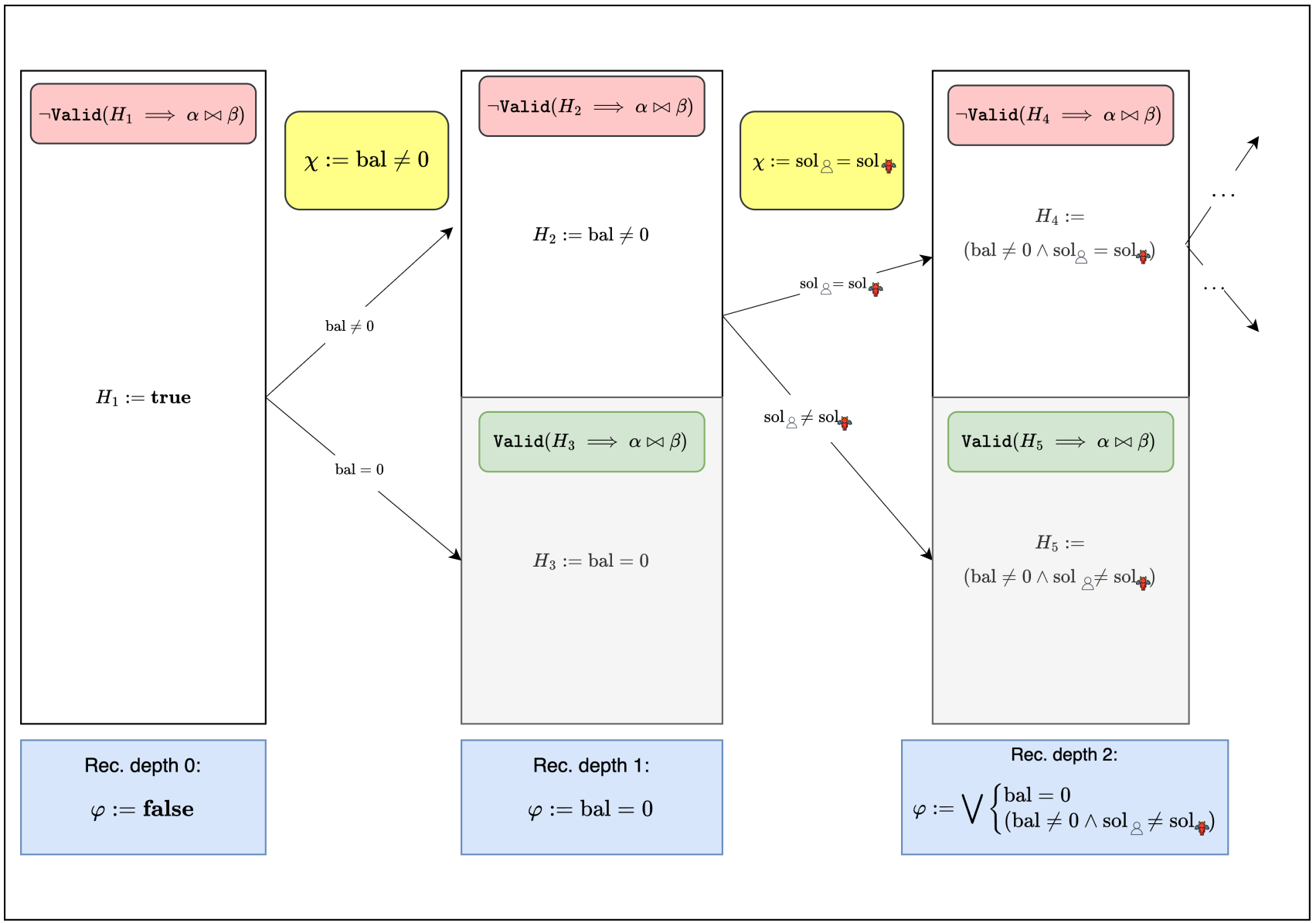
Algorithm on HashPuzzle



Algorithm on HashPuzzle

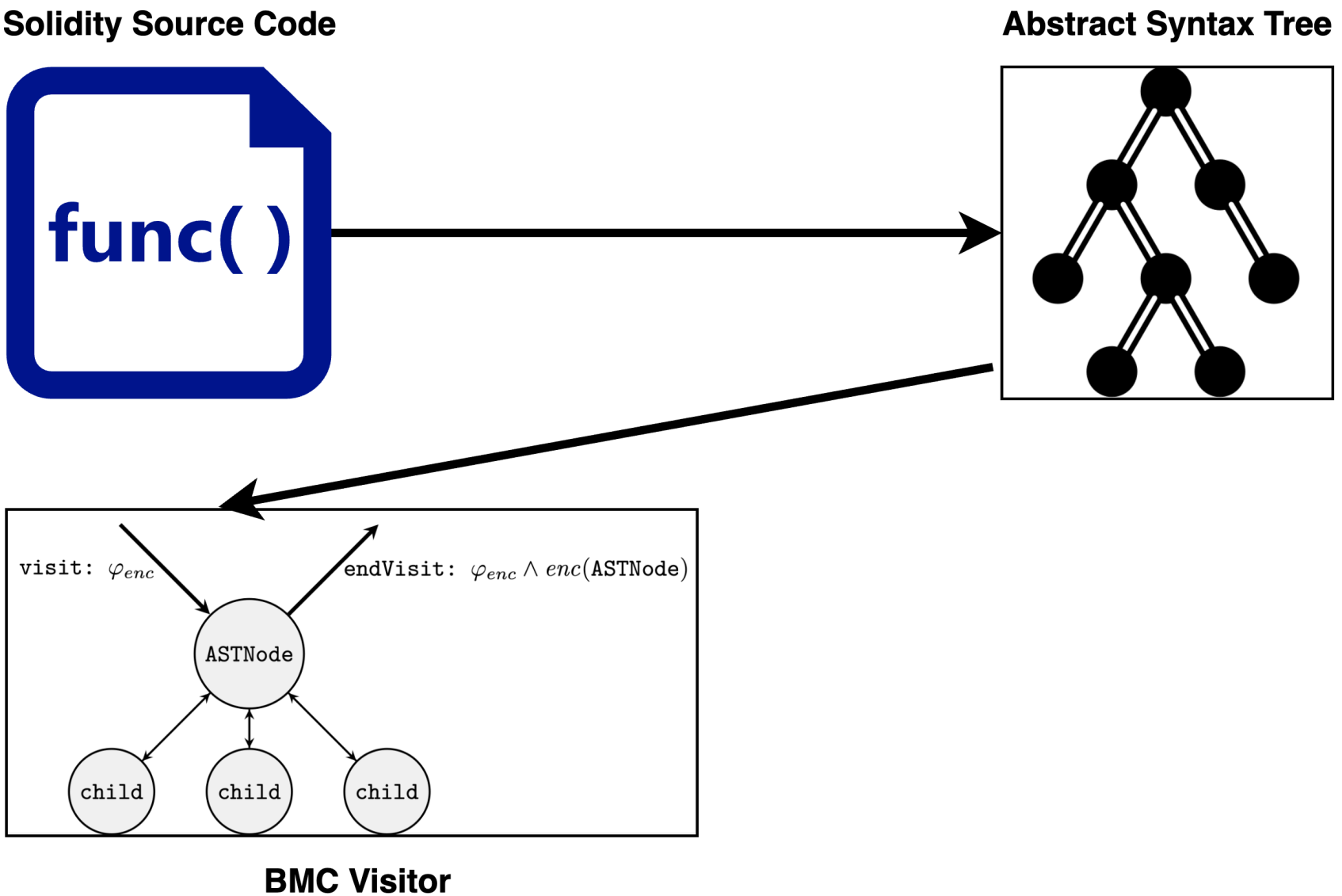


Algorithm on HashPuzzle

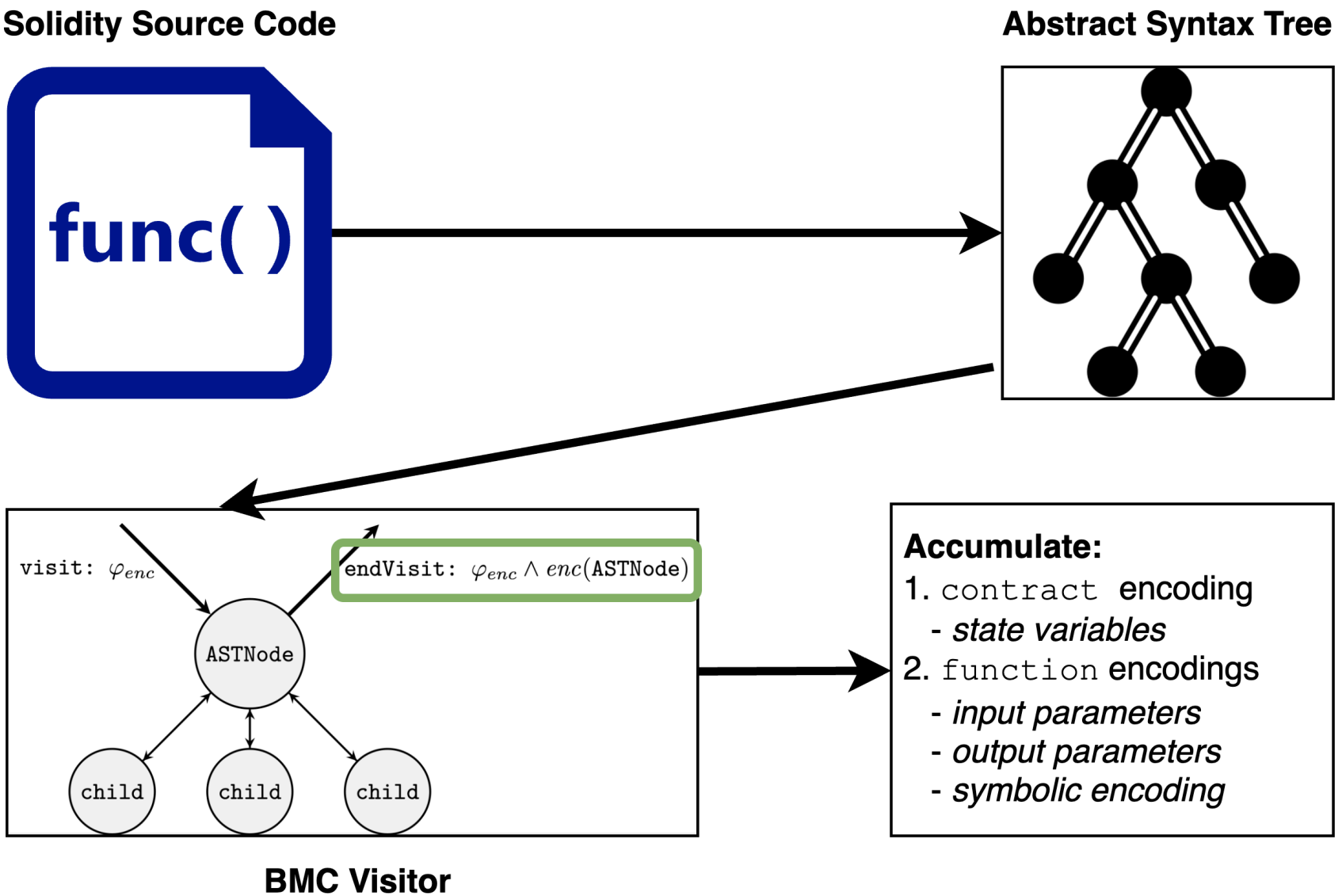


Smart Contracts' Encoding

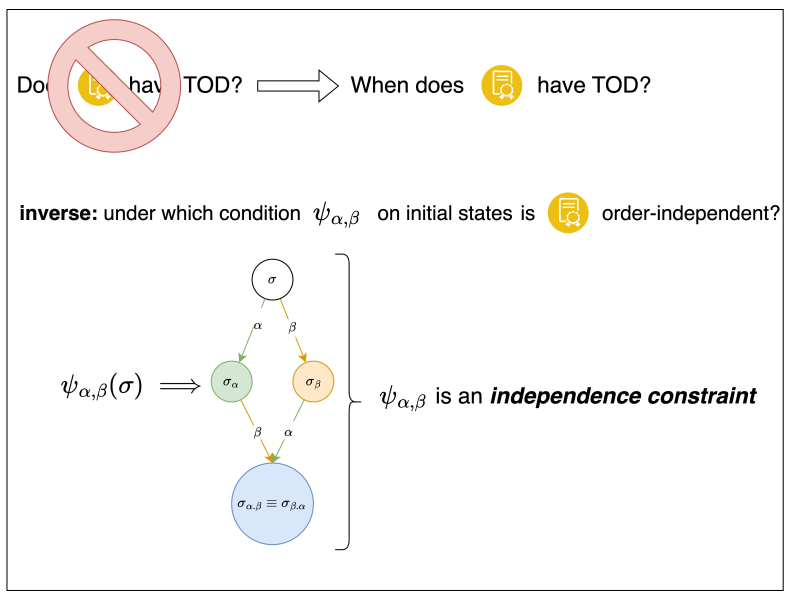
BMC Encoding

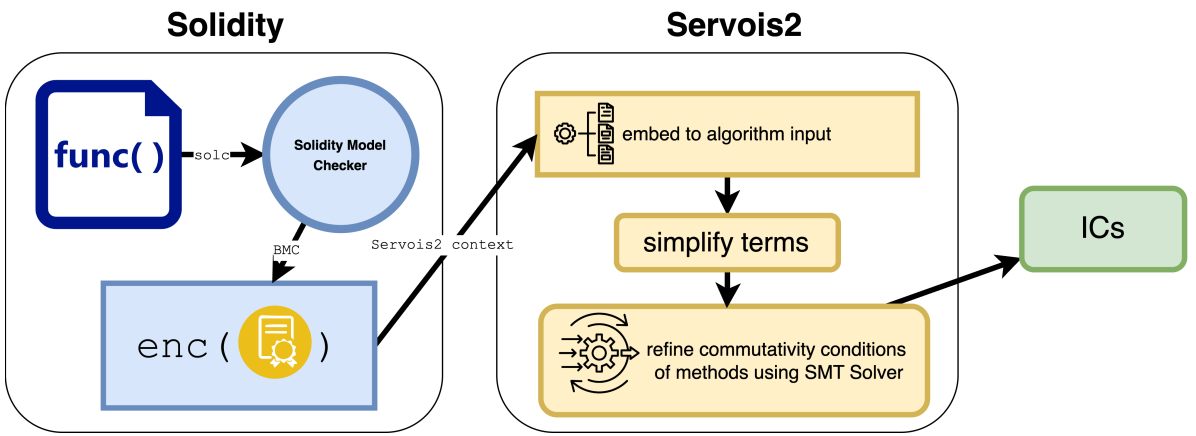


BMC Encoding



Conclusion


Do hav TOD?
When does have TOD?
inverse: under which condition $\psi_{\alpha,\beta}$ on initial states is order-independent?
$\psi_{\alpha,\beta}(\sigma)$
$\psi_{\alpha,\beta}$ is an ***independence constraint***


Solidity
Servois2
func( )
solc
Solidity Model Checker
BMC
enc( )
Servois2 context
embed to algorithm input
simplify terms
refine commutativity conditions of methods using SMT Solver
ICs

Back-up Slides

Commutativity Condition Refinement Algorithm

Full version

Input: Methods α and β, state region $H = \psi_1 \wedge \psi_2 \wedge \ldots$ (initially **true**) and set of predicates $\mathcal{P}$
Output: A commutativity condition φ for α and β (here a global variable)

1: **procedure** REFINE($\alpha, \beta, H, \mathcal{P}$)
2: **if** VALID($H \implies \alpha \bowtie \beta$) **then** ▷ Query an SMT solver to verify that $\alpha \bowtie \beta$ is a tautology in H
3: $\varphi := \varphi \vee H$
4: **else**(χ_c := counterexamples to VALID)
5: $p :=$ CHOOSE($H, \mathcal{P}, \chi_c$) ▷ $p \in \mathcal{P}$ is a predicate partitioning the state region H
6: REFINE($\alpha, \beta, H \wedge p, \mathcal{P} \setminus \{p\}$)
7: REFINE($\alpha, \beta, H \wedge \neg p, \mathcal{P} \setminus \{p\}$)
8: **end if**
9: **end procedure**

Figure 2.2: The algorithm for iterative refinement of commutativity conditions introduced in [7] (the simplified version for inferring only commutativity conditions is shown).

▶ Back

References

[1] MEV Explore.

[2] Bansal, K. et al. 2018. Automatic Generation of Precise and Useful Commutativity Conditions. *Tools and Algorithms for the Construction and Analysis of Systems* (Cham, 2018), 115–132.

[3] Bose, P. et al. 2021. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. arXiv.

[4] Daian, P. et al. 2019. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. arXiv.

[5] Lewis, M. 2014. *Flash Boys: A Wall Street Revolt*. W. W. Norton.

[6] Luu, L. et al. 2016. Making Smart Contracts Smarter. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna Austria, Oct. 2016), 254–269.

[7] Sergey, I. and Hobor, A. 2017. A Concurrent Perspective on Smart Contracts. arXiv.

[8] Tsankov, P. et al. 2018. Securify: Practical Security Analysis of Smart Contracts. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, Oct. 2018), 67–82.

[9] Zhang, W. et al. 2024. Nyx: Detecting exploitable front-running vulnerabilities in smart contracts. *2024 IEEE Symposium on Security and Privacy (SP)* (2024), 2198–2216.