# Poster: Blockchain Mempool Security: Provable Designs, Formal Proofs, and Automated Verification

Wanning Ding
*Syracuse University*
Syracuse, New York, USA
wding04@syr.edu

Yibo Wang
*Syracuse University*
Syracuse, New York, USA
ywang349@syr.edu

Yuzhe Tang
*Syracuse University*
Syracuse, New York, USA
ytang100@syr.edu

## I. Introduction

In permissionless blockchains with open membership, denial of service (DoS) attacks remain a fundamental threat. In such attacks, an adversary can excessively consume blockchain services without being held accountable. Although extensive protocols have been proposed to mitigate various forms of DoS (e.g., Sybil resistance through mining protocols [1]–[3], and execution fees to deter resource-exhausting smart contracts [4], [5]), securing a blockchain system against DoS attacks remains an open problem. This gap arises due to undefined behaviors and the wide divergence between protocol design and real-world implementation.

One particularly vulnerable component that has received limited attention from protocol designers is mempool. On each blockchain node, the mempool stores unconfirmed transactions before they are processed by downstream services such as mining or transaction propagation. A successful denial of mempool service can cascade into broader disruptions, including preventing validators from including transactions in blocks. As shown in recent work [6], such attacks can cause validators to produce empty blocks, resulting in global network degradation. The only known Ethereum mempool DoS attacks, namely DETER attacks, were discovered through manual code inspection [6], indicating a lack of systematic tools for exposing these vulnerabilities.

**Problem Statement:** In this work, we tackle the automatic exploit generation for understanding Ethereum mempool DoS security. While manual efforts have found isolated issues, there is no systematic framework to comprehensively and efficiently explore the space of potential mempool DoS attacks. Automatic exploit generation is essential for enabling Ethereum developers to proactively test their clients offline and for defenders to build more effective online detection mechanisms.

To this end, we define a specific class of mempool DoS attacks, which we call *Asymmetrical DeniAl of Mempool Service* (ADAMS). In these attacks, the adversary aims to fill the victim's mempool with crafted transactions at a significantly lower cost than the cost incurred by the displaced or rejected victim transactions. The asymmetry in cost makes such attacks particularly dangerous and cost-effective for the adversary.

**Threat Model:** We assume an ADAMS attacker controls a node in the Ethereum network and connects to several normal nodes, selecting a specific one as the victim. The attacker aims to disable the mempool service of the victim node by (1) evicting existing transactions and (2) occupying the mempool to block new transactions. As demonstrated in prior work [6], attackers can select high-value victims, such as nodes offering RPC services or mining pool infrastructure, to maximize the impact. Importantly, an ADAMS attack is considered successful only when the attacker achieves mempool denial at a cost significantly lower than that of the victim transactions displaced or prevented from entering.

## II. Approach

### A. Design Rationale and Overview

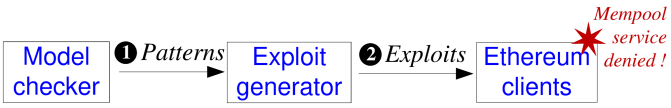



Fig. 1. Workflow overview of attack discovery

Given an Ethereum client, our goal is to generate crafted transaction sequences that cause successful ADAMS attacks on the mempool. Exhaustively searching the vast input space (e.g., $2^{5120}$ in Geth) is impractical. To address this, we propose a two-level framework: (1) discover abstract attack patterns via model checking on a simplified mempool model; and (2) concretely validate and mutate them against real clients.

### B. Step 1: Find Patterns by Model Checking

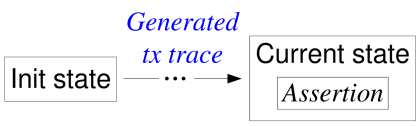



Fig. 2. Checking the mempool model by exploring transactions.

We construct a lightweight mempool model that captures key admission rules shared across Ethereum clients, such as transaction replacement and fee-based eviction. The model operates on Ether-transfer transactions with fixed gas, and all accounts start with equal balances. Using TLA+ [7], we apply

model checking to explore transaction sequences under two initial mempool states: one full of normal transactions, and one empty.

We define safety using three assertions: at least one original transaction should remain, the total fee value should remain high, or new normal transactions should still be accepted. A violation of all three indicates a successful ADAMS attack pattern.

From this process, we discover nine distinct attack patterns. Four of them evict existing transactions from a full mempool, including ED1 to ED4. The remaining five lock an initially empty mempool to decline subsequent valid transactions, including LD1 to LD5. These patterns cover different design choices: directly using invalid transactions, using valid ones that later become invalid, or using valid low-fee transactions strategically. Notably, seven of these patterns are new and were previously unknown in the literature.

### C. Step 2: Concrete Exploit Generation

To validate these attack patterns on real Ethereum clients, we develop a concrete exploit generation algorithm as shown in Figure 3. The algorithm takes as input a discovered pattern and a target client, then adaptively mutates implementation-specific knobs, such as how transactions are grouped into messages or which accounts are used.

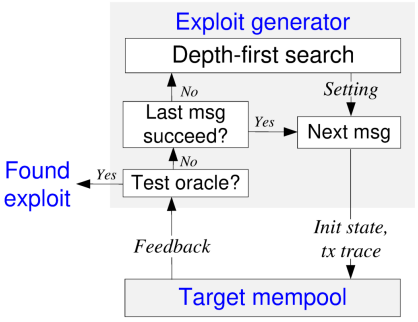



Fig. 3. Workflow of concrete exploit generation

It iteratively constructs candidate traces and replays them against the client's mempool, checking whether the result satisfies one of two success conditions: (1) the mempool evicts legitimate transactions while retaining only low-cost attacker transactions, or (2) the mempool locks out all new transactions and holds only attacker-crafted ones.

To efficiently search the space, we use a depth-first strategy. If a partial trace makes progress, such as causing a new eviction, the algorithm continues; otherwise, it backtracks and tries a different setting. This approach allows us to synthesize practical ADAMS exploits that are both effective and implementation-aware.

## III. Attack Evaluation

We evaluate the effectiveness of the discovered ADAMS attacks using both local networks and the Ethereum Rinkeby testnet. We measure success by how much the attack disrupts the inclusion of normal transactions in blocks, and how much Ether the attacker spends per disrupted block.

On the Rinkeby testnet, we launch an ED4 attack using over 6,000 crafted transactions sent from an attacker node to the public network. These transactions successfully propagate to over 90 percent of the testnet nodes. Figure 4 shows an Etherscan screenshot of the blocks generated during the attack. Gas usage in these blocks drops sharply from normal levels (above 8.9 percent) to as low as 0.34 percent. The first block includes mostly attacker replacement transactions, while the next two blocks remain underutilized and carry a mix of attacker and normal transactions. Notice that in our experiments, we didn't target our attacks to the top miner nodes as the existing DETER attacks entail.

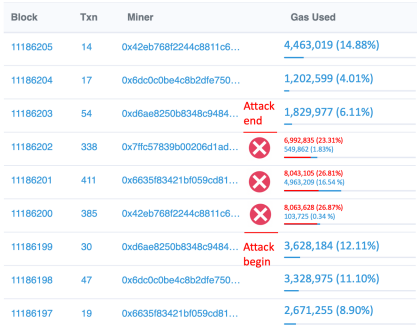



Fig. 4. Etherscan screenshot of the blocks generated during the attack on Rinkeby

## References


[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," May 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[2] A. Narayanan, J. Bonneau, E. W. Felten, A. Miller, and S. Goldfeder, *Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction*. Princeton University Press, 2016. [Online]. Available: http://press.princeton.edu/titles/10908.html

[3] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, J. Katz and H. Shacham, Eds., vol. 10401. Springer, 2017, pp. 357–388. [Online]. Available: https://doi.org/10.1007/978-3-319-63688-7_12

[4] D. Pérez and B. Livshits, "Broken metre: Attacking resource metering in EVM," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/broken-metre-attacking-resource-metering-in-evm/

[5] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks," in *ISPEC 2017*, 2017, pp. 3–24. [Online]. Available: https://doi.org/10.1007/978-3-319-72359-4_1

[6] K. Li, Y. Wang, and Y. Tang, "DETER: denial of ethereum txpool services," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds. ACM, 2021, pp. 1645–1667. [Online]. Available: https://doi.org/10.1145/3460120.3485369

[7] L. Lamport, *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. [Online]. Available: http://research.microsoft.com/users/lamport/tla/book.html

IEEE S&P
@IEEESSP

46th IEEE Symposium on Security and Privacy

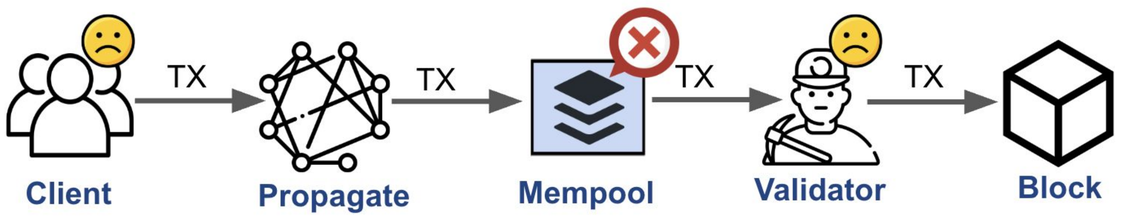
# Blockchain Mempool Security: Provable Designs, Formal Proofs, and Automated Verification


Wanning Ding  Yibo Wang  Yuzhe Tang 

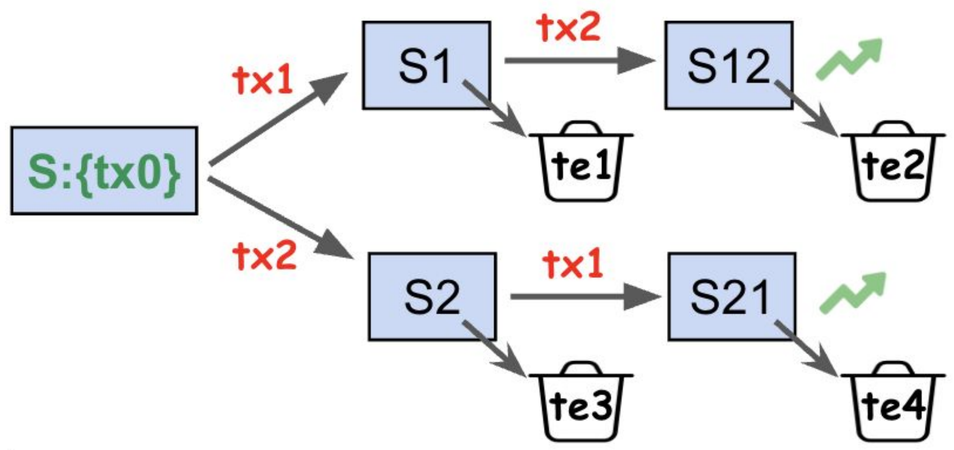
Syracuse University


## Introduction

- Mempool - critical subsystem in blockchain
- Damage of a denied mempool
  - Validator unable retrieve txs.
  - Web3 clients unable to transact.




## Security Definition



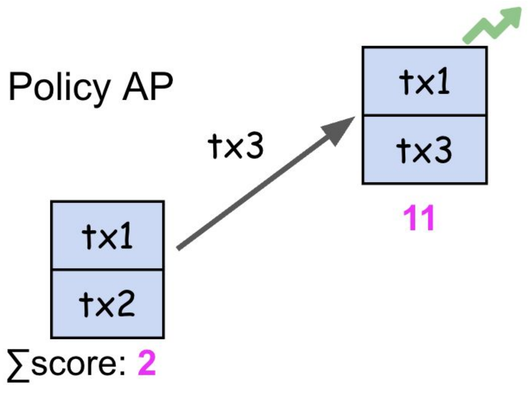

Eviction Security: $\forall S' \in \{S_{12}, S_{21}\}$, $fees(S') \geq g(S)$

Locking Security: $\forall te \in \{te_1, te_2, te_3, te_4\}$, $price(te) < h(S)$

## Provable Designs

**Admit by Ancestor Min Price (AP)**

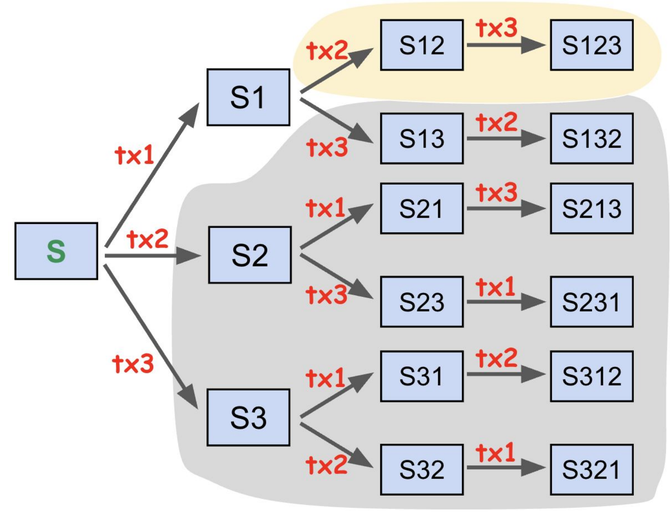
- **Score-AP (Ancestor Minimum Price):** Lowest price in a transaction's ancestors.
- **Admission policy:** Admit transaction ta, evict child transaction te of the lowest score if.f score(ta) > score(te).




| | *Sender* | *Nonce* | *Price* | $Score_{AP}$ |
|---|---|---|---|---|
| $tx_1$ | Alice | 1 | 1 | 1 |
| $tx_2$ | Alice | 2 | 100 | 1 |
| $tx_3$ | Bob | 1 | 10 | 10 |
| $tx_4$ | David | 1 | 20 | 20 |

## Proof in Pen and Paper (IEEE S&P'25)

- Naive proof by exhaustion
  - N! permutations
- Our proof
  - f(S21) = f(S12): **Order insensitivity**
    - Order of tx doesn't affect mempool end state
    - Reducing N! permutations to 1.
  - f(S12) > f(S1): **Monotonic score increase**
    - f(S123) > f(S12) > f(S1) > f(S): **Induction**
  - f(S123) > g(S123)




## Toward Formal Proofs

```
9  Theorem mempool_price_always_increases :
10   forall (mp_before mp_after : Mempool) (tx : Tx),
11     mp_after = admit_tx mp_before tx ->
12     total_price mp_before <= total_price mp_after.
```

Specify property and proof

- Coq supports generating constraints from code and solving them.

```
13 Proof.
14   intros mp_before mp_after tx Heq.
15   unfold admit_tx in Heq.
16
17   destruct (length mp_before <? mempool_size_limit) eqn:Hcheck.
18
19   - (* Case 1: mempool is not full - directly add tx *)
20     subst mp_after.
21     (*Heq : mp_after = tx :: mp_before*)
22     (*Goal: total_price mp_before <= total_price (tx :: mp_before)*)
23     simpl.
24     (* Goal: total_price mp_before <= tx.(price) + total_price mp_before*)
25     lia. (*Linear Integer Arithmetic*)
```

## Toward Automated Verification

Baseline mempool in model checker

- Small state, short input sequence
- Atomic tx
- A set of txs

```
1 MODULE main
2 VAR
3   st0: {N,P,F}; st1: {N,P,F}; ops:{P,F};
4 ASSIGN
5   init(st0) := N;   init(st1) := N;
6   TRANS  case
7     st0 = N & st1 = N & ops = P:next(st0) = P & next(st1) = st1;
8     ...
9     st0 = F & st1 = F & ops = F:next(st0) = F & next(st1) = F;
10    TRUE : next(st0) = st0 & next(st1) = st1; esac;
11 LTLSPEC G(st0 != F | st1 != F);
12 SPEC AG EF((st0 = F & st1 = F) -> EX (st0 = P & st1 = F));
```

RQ: How to scale to larger mempool

- Reduce state space without losing soundness/completeness?