Combining Program and Protocol Analysis for Frontrunning Resistance in Smart Contracts

Sebastian Holler
MPI-SP
Bochum, Germany
sebastian.holler@mpi-sp.org

Yan Farba
MPI-SP
Ruhr-University Bochum
Bochum, Germany
yan.farba@mpi-sp.org

Clara Schneidewind
MPI-SP
Bochum, Germany
clara.schneidewind@mpi-sp.org

ABSTRACT

Frontrunning is a smart contract vulnerability that stems from the non-synchronous nature of blockchain transaction processing. Traditional approaches to detecting frontrunning in smart contracts rely on the notion of Transaction Order Dependence (TOD). A smart contract is considered to have TOD if the result of two contract calls may vary with their execution order. However, TOD does not serve as an accurate predictor for frontrunning vulnerabilities, as it disregards that honest users may purposefully limit contract calls to situations where reordering cannot be harmful. Following this observation, we develop the first frontrunning analysis tool that combines program analysis of a smart contract with protocol analysis for modeling honest user strategies.

KEYWORDS

Transaction Order Dependence, TOD, Frontrunning resistance, Protocol Analysis, Program Analysis, Commutativity.

1 INTRODUCTION

Frontrunning attacks. Smart contracts are decentralized programs that can be invoked through blockchain transactions. Ethereum [10] is the most prominent blockchain platform with support for smart contracts. Ethereum smart contracts are stateful: they can hold tokens as their *balance* and have persistent storage that can be accessed with *storage variables*. If the logic of a smart contract depends on its state, the processing order of contract-invoking transactions may change its intermediate and final states.

Smart contracts with dependency on transaction order create an exploitation avenue for blockchain attackers. Such exploits are called *frontrunning attacks*. When frontrunning, an attacker tries to maliciously impact the results of transaction execution by crafting and reordering transactions. This is possible in the blockchain context, since staging transactions for execution requires users to publish those transactions in a public *mempool*, from which specific network nodes, so-called *miners*, create new blocks that extend the blockchain. This leaves miners with the ability to inspect, insert, and reorder messages before finalizing them through publication on the blockchain. Consider, for example, the following `HashPuzzle` (short `HP`) smart contract written in Solidity – a high-level Java-like smart contract language:

```
1 contract HashPuzzle {
2     bytes32 public challenge;
3     function solve(bytes32 solution) {
4         require(sha256(solution) == challenge);
5         payable(msg.sender).transfer(this.balance);
6     }
7 }
```

The `HP` contract has a public `challenge` storage variable (accessed on line 2), whose preimage users should guess. Whoever's call to the `solve` function with the correct solution is processed first gets the smart contract's balance as a reward (lines 4 and 5). If the solution is wrong, the **`require`** statement aborts the execution.

Suppose an honest user publishes a transaction $\langle \texttt{HP.solve}(s) \rangle_h$ to `HP` with the correct solution s for inclusion in the next block. Here, $\langle p \rangle_h$ denotes that the user h publishes a transaction with a payload p, where p contains a contract, function and parameters of a call. Since the code and storage of `HP` and the solution `s` can be observed by an attacker before the block is finalized, they can verify whether the solution is indeed correct and use it to publish their own transaction $\langle \texttt{HP.solve}(s) \rangle_a$. The order in which these transactions are processed determines whether the attacker or the honest user receives the reward. In practice, it has been shown that attackers can effectively bribe miners to ensure that their transaction will (with high probability) be scheduled first [7].

Identifying frontrunning. Existing detection approaches for frontrunning vulnerabilities center around the notion of Transaction Order Dependence (TOD) [5]. A smart contract exhibits TOD if there exist two contract-invoking transactions that, when processed in different orders, show different relevant effects on the final blockchain state. E.g., in the `HashPuzzle`, the final states after execution differ by the recipient of the reward.

However, while the absence of TOD is sufficient to ensure a contract's frontrunning resistance, secure contracts may still exhibit TOD: a TOD-free smart contract ensures transaction order independence in *any possible blockchain state* for *all kinds of transactions*. However, a smart contract user can also evade frontrunning attacks by ensuring that states where their *own transactions* are susceptible to TOD are *unreachable*. Consider the following version of the `HashPuzzle` contract that implements a commit-reveal scheme, a common frontrunning protection [2], where the `solve` function can only be called after secretly committing to the solution by calling `commit` before a certain timeout (notice the **`require`** call on lines 7 and 8 that verifies the commitment).

```
1  function commit(bytes32 commitment) {
2      require(timeout >= this.blockstamp);
3      commitments[msg.sender] = commitment;
4  }
5  function solve(bytes32 solution) {
6      require(timeout < this.blockstamp);
7      require(commitments[msg.sender]
8              == sha256(solution, msg.sender));
9      require(sha256(solution) == challenge);
10     payable(msg.sender).transfer(this.balance);
11 }
```

For users that do not reveal their solutions before the timeout, interactions with this contract are secure against frontrunning. An attacker would only learn the solution when the honest user is calling the `solve` function once the time for publishing commitments is over. At this point, the attacker cannot commit in retrospect to this solution as any call to `commit` would revert (line 2). However, the contract clearly exhibits TOD since calls to the `solve` functions would not commute in contract states where both the attacker and the user committed to the correct solutions. This is as TOD disregards the honest user behavior, which (by not publishing the solution early) ensures that such a state would not be reachable.

Improving Frontrunning Analysis. State-of-the-art static analysis tools use TOD to identify frontrunning vulnerabilities [3, 8, 9, 11]. However, by disregarding that smart contracts make assumptions about the behavior of honest users, [3, 11], these tools cannot identify which states a smart contract may actually reach. To improve on state-of-the-art, we develop the first push-button frontrunning verification toolchain that explicitly models honest user behavior. In particular, we propose a verification pipeline that combines program analysis to identify TOD-inspired commutativity conditions with protocol analysis to model the interaction between an honest user and the smart contract. Constructions, proofs, and implementation in this paper are work in progress.

2 TOOL OVERVIEW

Analysis Idea and Challenge. At its core, our analysis checks whether an honest user protocol (which models the interaction strategy of an honest user with the smart contract) can enforce that only such states are reached where the honest user transactions commute with attacker transactions. To make this precise, we assume (1) A formal blockchain semantics $\Gamma \xrightarrow{\langle p \rangle} \Gamma'$ that describes how a blockchain state Γ evolves to state Γ' when executing transaction $\langle p \rangle$ (see **??**); (2) A user protocol Σ_h, which maps blockchain execution history $R = \Gamma_1 \xrightarrow{\langle p_1 \rangle} \Gamma_2 \ldots \xrightarrow{\langle p_{n-1} \rangle} \Gamma_n$ to the next transaction $\langle p \rangle_h = \Sigma_h(R)$ that the user h aims to conduct; (3) A formal model of (adversarial) blockchain execution $\Sigma_h \vdash R$ describing the multi-step blockchain executions R that are possible in the presence of a user h following protocol Σ_h. With that, we aim to check whether

$$\forall R, \langle p_a \rangle, \langle p_h \rangle, \Gamma,\ \Gamma'.\ \Sigma_h \vdash R\ \wedge\ \langle p_h \rangle_h = \Sigma_h(R) \\ \wedge\ R \xrightarrow{\langle p_a \rangle_a} \xrightarrow{\langle p_h \rangle_h} \Gamma\ \wedge\ R \xrightarrow{\langle p_h \rangle_h} \xrightarrow{\langle p_a \rangle_a} \Gamma' \Rightarrow \Gamma \approx \Gamma'$$

where $\approx$ denotes the equivalence of the relevant part of the blockchain state (e.g., the contract state variables and the balances of users).

Checking this property comes with several challenges: (i) The relation $\Sigma_h \vdash R$ models the protocol execution, which is subject to interference by a powerful blockchain attacker. Since such an attacker controls the transaction order and inserts transactions, exploring all relevant states that are reachable in such executions is hard; (ii) As the protocol may make use of cryptography (e.g., the commit-reveal-scheme in the `HashPuzzle` example), the protocol execution needs to model the properties of basic cryptographic

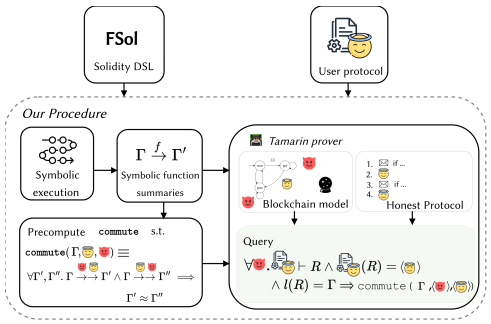

Figure 1: Procedure overview. Here, $l(R) = \Gamma$ denotes that Γ is the final state of the run R.

primitives; (iii) The property finally requires checking *commutativity* of two contract operations p_h and p_a, a complex property that requires the comparison of two different operation sequences.

We approach these challenges as follows: Our core insight is that we separate our analysis into aspects that we can tackle with *program analysis techniques* and such that we can tackle with *protocol analysis techniques*. At a high-level, we will use the symbolic protocol analysis tool Tamarin [6] to check whether all states reachable by the user protocol satisfy a set of *commutativity conditions* that we infer with program analysis techniques.

We provide an overview of our analysis pipeline in Figure 1: The input to our analysis tool is (a) a smart contract written in a light-weight DSL, called FSol, on top of the Solidity language and (b) a user protocol modeled in Tamarin. FSol can easily be translated into standard Solidity, which allows us to leverage existing Solidity-related tooling, in particular, symbolic execution engines [4, 11]. We use those engines to obtain *function summaries* for the different smart contract functions, which provide logical descriptions of the state updates triggered by transactions that invoke those functions. These state update descriptions are then used (i) as input to the commutativity condition generation, which derives conditions under which calls to contract functions commute; (ii) as input to the Tamarin model of blockchain execution, which describes the reachable blockchain states when interacting with the honest user protocol. The final verification query is then executed on top of the Tamarin model to check whether the blockchain execution can ever reach a state where the honest user issues a transaction that does not commute with possible attacker transactions in this state.

Computing Commutativity Conditions. Commutativity conditions have been mainly studied for concurrent data structures, where they characterize states in which data- structure-operations commute. We build upon existing work in this domain [1] that shows how to effectively synthesize sound commutativity conditions from logical descriptions of the data structure operations. We obtain such a logical description from the summaries of the smart contract functions, calculated through symbolic execution.

Blockchain Model. Tamarin provides support for modeling and analyzing cryptographic protocols in the presence of strong attackers, which are constraint by cryptographic assumptions in a symbolic model of cryptography. We devise a blockchain model in Tamarin that provides a concise characterization of the attacker's

reordering and transaction insertion abilities. Here, we take advantage of the fact that the attacker's ability to delay honest user transactions is bounded. This limitation allows honest user protocols to proceed within *rounds* such that their transactions are guaranteed to be included within a single round. By enforcing that also smart contracts respect this round structure (as we do with the help of the FSol DSL), we can significantly simplify our blockchain attacker model, since the ability of the attacker boils down to freely arrange a specific user transaction within a round.

The final toolchain enables developers to write smart contracts in FSol, conduct automated frontrunning verification, and to then compile their contracts into deployable Solidity code.

REFERENCES

[1] Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2018. Automatic Generation of Precise and Useful Commutativity Conditions (Extended Version). *CoRR* abs/1802.08748 (2018). arXiv:1802.08748 http://arxiv.org/abs/1802.08748

[2] Carsten Baum, James Hsin-yu Chiang, Bernardo David, Tore Kasper Frederiksen, and Lorenzo Gentile. 2022. Sok: Mitigation of front-running in decentralized finance. In *International Conference on Financial Cryptography and Data Security*. Springer, 250–271.

[3] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178.

[4] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[5] 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*. 254–269.

[6] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer, 696–701.

[7] 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.

[8] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[9] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 189 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360615

[10] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[11] Wuqi Zhang and et al. 2024. Nyx: Detecting Exploitable Front-Running Vulnerabilities in Smart Contracts. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society.