Oral Communication: Detection of De-Fi Profitable Scenarios through History-Based Policies

Margherita Renieri[1], Letterio Galletta[1]

[1]IMT School for Advanced Studies Lucca, Italy

Abstract

Smart contracts, deployed and executed within blockchain environments, operate autonomously, eliminating the need for central authorities. Typically written in Solidity, they interact with other contracts on Ethereum through external calls. However, external calls provide no mechanism to ensure that the called code satisfies some predefined behavioral policies. We propose a framework to specify and enforce security policies to address this issue, enhancing smart contract integrity and security. More precisely, these policies are specified by developers at the contract code level to monitor the execution of a part of the code while enforcing the desired behavior. We describe how our approach can be used to detect and prevent flash loan-based arbitrage scenarios.

Keywords

Security policies, Decentralized Finance, Formal methods

1. Talk proposal

Context and motivation Smart contracts are computer programs deployed and executed within a blockchain environment. On the Ethereum blockchain, they are commonly written in the Solidity language and compiled into the EVM bytecode. The definition of a smart contract in Solidity looks like a class in any OOP language: contracts have an internal mutable state and a set of procedures to manipulate it. The public functions of a contract can be invoked by users directly through transactions or by other contracts through the external calls mechanism. Although this external call mechanism is powerful in enabling interactions between smart contracts, it provides no means to ensure that the invoked code satisfies some predefined behavioral policies. This is even more critical when the smart contract address implementing the function to be invoked is a parameter of the current function, so the caller controls the code that will be executed. This could have severe security consequences as many attacks on smart contracts exploit external calls to run attacker-controlled code [1].

To address this, we propose a methodology allowing developers to specify and enforce prior security policies on internal and external contract calls at the code level, checking that the called code is complainant with such policies at run-time. In this way, after developers have encoded the allowed behavior in the policies, they can be sure that the called code does not deviate from it. In our proposal, a policy consists of two parts: the first is a regular expression that represents the *history* and describes the sequence of calls the invoked external call is allowed to do; the second is an assertion on the state the smart contract reaches after the call execution. Therefore, an external call to be compliant with a policy must produce a history that belongs to the language of the policy's regular expression, and a contract state that satisfies its assertion.

An overview of the framework In the talk, we outline the design and the formalization of our approach. More precisely, we start from TinySol, a core calculus for smart contracts introduced by Bartoletti et al. [2], and extend it with a policy framing construct $\phi[S]$ for guarding the execution of statements S with a developer-defined policy ϕ. We define the syntax of our policies ϕ and the operational semantics of a TinySol program S. Intuitively, a policy ϕ is a pair (re_ϕ, E_ϕ), where re_ϕ is a regular expression and E_ϕ is a boolean expression that must evaluate to true. Since E_ϕ may predicate on values that will be only known at run-time, we allow variables to occur inside re_ϕ that will be bound to concrete values during the history check. The semantics is a transition system that describes the result of a computation, namely the reaching state, and collects its *history*, namely the sequence of function calls performed at run-time together with their actual parameters. More precisely, configurations consist of triples of the form $\langle S, \sigma, \eta \rangle$, where S is the sequence of statements to execute, σ is the starting state, and η is the history; and of pair $\langle \sigma', \eta' \rangle$ representing final configurations. Transitions are defined by a set of inference rules providing the semantics to each construct of the language. We briefly illustrate how we evaluate the policy framing $\phi[S]$, given a state σ and a history

DLT 2024 : 6th Distributed Ledger Technology Workshop, May 14-15, 2024, Torino, TO

margherita.renieri@imtlucca.it (M. Renieri);
letterio.galletta@imtlucca.it (L. Galletta)
0000-0001-6987-1881 (M. Renieri); 0000-0003-0351-9169 (L. Galletta)

η. First, we run the statements S through a transition $\langle S, \sigma, \eta\rangle \rightarrow \langle \sigma', \eta'\rangle$; if the execution of S terminates, we check that the produced history η' is compliant with ϕ. To check the compliance, we rely on the formalism of symbolic automata [3]. More precisely, we transform the regular expression re_ϕ (which may have variables) into a symbolic automata A_ϕ, and check that the history η' belongs to the language of A_ϕ for some assignment ρ of variables of re_ϕ. Then, we evaluate the assertion E_ϕ in the reached state σ' extended with the assignment ρ: If E_ϕ evaluates to true, the policy ϕ is satisfied, and the execution continues. Otherwise, when the history η' does not belong to the language of A_ϕ or E_ϕ evaluates to false, the execution is aborted because of a policy violation.

An Illustrative Scenario We now show how our policy framework can detect flash loan-based arbitrage behavior. Flash loans allow any user to obtain loans without up-front collateral. Such loans are valid only within a single transaction and rely on the atomicity of blockchain transactions, specifically within a single block. If a user is unable to repay the loan, the flash loan transaction automatically fails and reverts, rendering the entire mechanism risk-free for the loaning contract. This is made possible by the EVM's capability to revert state changes. Moreover, they enable users to capitalize on several financial opportunities, one of which is arbitrage. Let us briefly introduce what an arbitrage is. The value of a token is determined by market demand and supply across various Decentralized Exchanges (DEXs).[1] Due to the lack of instantaneous synchronization among DEXs, identical tokens may be traded at slightly different prices on different exchanges. Arbitrage involves exploiting these price differences among exchanges for financial gain. A user executes a series of token swaps to capitalize on the difference in exchange rates, ultimately generating profits from the price discrepancy between the two tokens. Through flash loans, a user can perform arbitrage on different on-chain markets without an upfront amount of tokens and without the risk that the prices in the DEX would immediately change since the arbitrage occurs in a single transaction. This behavior has been linked to various real-world arbitrage exploits, such as those discussed in [4].

Below, we consider a flash-loan implementation similar to the one of Aave protocol [5]. To perform a flash loan, we require a user first to deploy on the blockchain her smart contract, call it `Flash`, following a given interface (it has to implement the `executeOperation()` function), and then to call the function `flashLoan` of the protocol, call it `Pool`, passing the address of the smart contract `Flash` and the amount of tokens to be loaned. After some sanity checks, the `flashLoan` function transfers the requested amounts to the user's contract and calls its `executeOperation()` function. Since the `executeOperation()` is completely controlled by the user, she can perform any financial operations she desires, provided that the loan amount plus some fees are paid back to the protocol when `executeOperation()` terminates its execution. For example, in a flash-loan-based arbitrage, a user may exploit the price discrepancies between tokens `t_a` and `t_b` on two different Automated Market Makers, `AMM1` and `AMM2`, to achieve a profit. Let be `User`, `Pool`, `Flash`, `AMM1`, and `AMM2` the addresses of five contracts that interact with each other. A flash loan-based arbitrage may involve the execution of the following sequence of function calls while executing `flashLoan`:

1. `Pool.balance()`
2. `Pool.transfer(Flash, z : t_a)`
3. `Flash.executeOperation()`
4. `AMM1.swap(User, t_a, t_b, x)`
5. `AMM2.swap(User, t_b, t_a, y)`
6. `Flash.payback(Pool)`

First, the `Pool` contract performs some sanity checks, during these checks, it invokes the `Pool.balance()` function. Then, the requested amounts of tokens are transferred to the contract `Flash` via a call to `Pool.transfer(Flash, z : t_a)`. This transfer of tokens is followed by the invocation of `executeOperation()` on the `Flash` contract. Once the `Flash` contract has available the flash-loaned amount, it executes various swaps to manage the borrowed assets. It perform a first swap (denoted as `x`) on one asset (`t_a`) to acquire another asset (`t_b`) on an `AMM1` contract (`AMM1.swap(User, t_a, t_b, x)`), followed by the reverse swap (denoted as `y`) on a different `AMM2` contract (`AMM2.swap(User, t_b, t_a, y)`). Once the `Flash` contract completes these operations, it repays its debt to the `Pool` contract, including additional fees via the call to `Flash.payback(Pool)` function.

Our policy mechanism can prevent the execution of arbitrage by monitoring the function calls performed inside the `executeOperation()` and checking the profit performed by the user. The idea is to define a policy $\phi = (re_\phi, E_\phi)$ as follows: the re_ϕ checks the sequence of function invocations to catch all the executions that involve two consecutive `swap`; the assertion E_ϕ checks that the price discrepancies are not too high. Protecting the invocation of `executeOperation()` with a policy framing $\phi[\mathit{executeOperation}()]$ enables us to identify and potentially mitigate profitable unwanted scenarios on the blockchain. More precisely, the policy would validate the execution of swap actions involving sets of tokens of opposing types across different AMMs (steps 5 and 6), i.e., ϕ would verify if the utilized AMMs (permitting the trading of identical tokens) have a relevant token ratio.

[1] DEXs are smart contracts that create a liquidity pool of ERC20 tokens, which are automatically traded by an algorithm.

References

[1] OWASP, Reentrancy attack, https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC01-reentrancy-attacks.html, 2024.

[2] M. Bartoletti, L. Galletta, M. Murgia, A minimal core calculus for solidity contracts, in: Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Luxembourg, September 26–27, 2019, Proceedings 14, Springer, 2019, pp. 233–243.

[3] S. Drews, L. D'Antoni, Learning symbolic automata, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2017, pp. 173–189.

[4] K. Qin, L. Zhou, B. Livshits, A. Gervais, Attacking the defi ecosystem with flash loans for fun and profit, in: International conference on financial cryptography and data security, Springer, 2021, pp. 3–32.

[5] A. D. V3, Flash loans, 2023. URL: https://docs.aave.com/developers/guides/flash-loans, accessed: March 08, 2024.