A Policy Framework for Regulating External Calls in Smart Contracts

Margherita Renieri and Letterio Galletta

IMT School for Advanced Studies Lucca, Lucca, Italy
{margherita.renieri,letterio.galletta}@imtlucca.it

Abstract. Smart contracts, functioning autonomously within blockchain ecosystems, obviate the necessity for central oversight. Typically programmed in Solidity, these contracts interact with others on Ethereum via external calls that allow a contract to invoke a function of another contract. However, external calls lack mechanisms to ensure that the called code satisfies some predefined behavioral policies. In this paper, we propose a formal framework to specify and enforce security policies to address this issue. Specifically, we present a core calculus for smart contracts equipped with constructs for specifying policies at the code level, allowing for monitoring and enforcing desired behaviors. We provide the formal semantics of this calculus and describe how our approach can be used to detect and prevent flash loan-based arbitrage scenarios.

Keywords: Security policies, Decentralized Finance, Arbitrage Attacks

1 Introduction

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 functions to manipulate it. These public functions can be invoked by users directly through transactions or other contracts through the external calls mechanism. Although this 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 we pass the address of a smart contract as an argument to a function that, in turn, invokes a function within the passed contract, so letting the caller control the code that is actually executed. For example, a contract `Bank` may implement a function `calculateInterest(RateProvider pro)` by invoking the function `getCurrentRate()` on the parameter `pro` of type `RateProvider` to obtain the current rate of a given crypto-asset in exchange for some fees. The trustworthiness of the computation depends on the code of `getCurrentRate()`. Relying on external code could have severe security consequences as many attacks on smart contracts exploit external calls to run attacker-controlled code, e.g., reentrancy attacks [12,8].

To address this issue, we propose a methodology to specify and enforce security policies at contract code level, inspired by Schneider [14] and Kozen [10]. The methodology involves (1) extending the programming language with constructs for expressing security policies to guard pieces of code; and (2) introducing inside the semantics of the language mechanisms to check that the code is compliant with such policies at run-time.

More precisely, we start from TinySol [4], a core calculus for smart contracts, and we extend it with a policy framing construct $\phi[S]$ for guarding the execution of statements S with a developer-defined policy ϕ. Intuitively, a policy ϕ is a predicate that specifies the set of program executions that are acceptable. The predicate ϕ is a pair (re, E), where re is a regular expression and E is an assertion (boolean expression). The regular expression re predicates on the *execution history*, namely the sequence of function calls made by the contract. While the expression E predicates on the state of the computation, i.e., the values of contract and function variables. Since the values of these variables will be only known at run-time, we allow them to occur also inside re. These variables will be bound to concrete values while evaluating re against the execution history. We define the semantics of TinySol as a transition system that describes the result of a computation (the reaching state), and that keeps track of the sequence of function calls performed at run-time together with their actual parameters (the *history*). Moreover, we define a run-time procedure to check when the execution satisfies the policies of a contract. This procedure works as follows: given a policy $\phi = (re, E)$, we first obtain a symbolic automaton from re that accepts all the histories that may be compliant for some assignment of variables occurring in re; then, we evaluate the assertion E to check its validity. A history is compliant if it passes both these checks, otherwise it is not, and the execution is reverted.

In the rest of the paper, we proceed as follows. We provide some background information and motivations for our work in Section 2. Section 3 presents the formalization of our methodology. Section 4 shows our use cases on DeFi protocols. Section 5 compares our work with the relevant literature, and Section 6 concludes and discusses future work.

2 Background and Motivation

Decentralized Finance (DeFi) is an innovative financial system built on blockchain technology, utilizing smart contracts to enable decentralized trading, lending, and investment. While Ethereum is a dominant platform, DeFi ecosystems have also developed on other platforms like Solana. However, DeFi is characterized by significant fragmentation and inefficiencies, with prices for identical financial instruments varying considerably across different venues and each venue responding inconsistently to market movements. These discrepancies offer profitable opportunities for speculative actions to manipulate the price of assets to take some advantage or to cause damage to other users.

Flash loans are financial instruments that allow users to borrow crypto-assets without needing up-front collateral, leveraging the atomicity of blockchain trans-

actions. This atomicity ensures that the entire transaction either completes successfully or is aborted. Intuitively, they work as follows. At the beginning of a transaction, a user requests a loan of some crypto-assets to a DeFi service offering flash loans. The user performs various financial operations with the borrowed crypto-assets, including arbitrage, collateral swaps, etc., that can be completed within the scope of a single transaction. At the end of the transaction, the user pays back the loan plus a small fee. If the user cannot repay the loan, the transaction fails and reverts: all actions taken during the transaction are undone as if they never happened. This mechanism is risk-free for the lending contract and it is possible due to the Ethereum Virtual Machine's (EVM) ability to revert state changes. Qin et al. [13] provide a detailed introduction to flash loans, their common uses, and potential attacks facilitated by them.

Listing 1.1 shows a prototypical implementation of a `flashLoan` function, similar to the one of Aave protocol [1].

```
function flashLoan(receiverAddress, asset, amount, params):
    // Calculate fees for the asset
    // Check if there is enough liquidity
    if liquidity[asset] < amount:
        raise error "Insufficient liquidity"

    // Update the liquidity to reflect the loan
    liquidity[asset] -= amount

    // Transfer the assets to the receiver
    transfer(asset, receiverAddress, amount)

    // Execute the operation in the receiver contract
    if not receiverAddress.executeOperation(asset, amount, fee, msg.sender,
         params):
        raise error "FlashLoan execution failed"

    // Collect the borrowed amount plus fee
    totalDebt = amount + fee

    // Check if receiver has enough tokens to repay the loan
    receiverBalance = getBalance(receiverAddress, asset)
    if receiverBalance < totalDebt:
        raise error "Insufficient token balance to repay the loan"

    transferFrom(receiverAddress, self, totalDebt, asset)
    liquidity[asset] += totalDebt

    // Emit the flash loan event
    emit FlashLoan(receiverAddress, asset, amount, fee)
```

Listing 1.1: Code of a `flashloan` function.

The function above allows a calling smart contract to borrow an asset, execute arbitrary code using that asset via the external call mechanism, and then return the borrowed amount with a small fee. The parameter `receiverAddress` specifies the contract that receives the borrowed assets and executes a custom code via the function `executeOperation` called within the `flashLoan` function. The parameter `asset` contains the address of the asset to be borrowed, while `amount` specifies its quantity. The parameter `params` provides additional data required by the receiver contract for its operations. Once the fees for the borrowed asset are calculated and some sanity checks verify that the pool has enough liquidity,

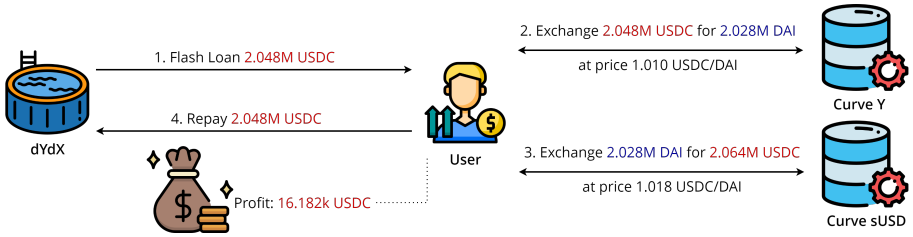


Fig. 1: High-level execution of a flash loan-based arbitrage transaction: (1) flash loan requested to the dYdX liquidity pool; (2) swap exchange from USDC to DAI in the AMM Curve Y; (3) swap exchange from DAI to USDC in the AMM Curve sUSD;(4) repay the flash loan.

the requested assets are transferred to the receiver contract. Then, the function `flashLoan` invokes the external function `executeOperation`, passing as parameters the borrowed asset, the amount, the requested fees, the caller's address (`msg.sender`), and additional parameters. The receiver contract returns true to signal successful execution; otherwise, the transaction raises an error. After the receiver contract executes its code, it must repay the borrowed amount plus the fees. When the debt is paid back to the lending contract, the liquidity pool is updated to reflect the repayment, restoring the previous amount of crypto-assets.

Flash loans offer several financial opportunities for price attacks and speculative actions, including arbitrage. Milionis et al. [11] provide an in-depth analysis of arbitrage focusing on the differences between Automated Market Maker[1] (AMM) and market prices. Based on their strategies, arbitrageurs can be classified into opportunistic and myopic. The first category waits for larger mispricings, hoping for greater future gains rather than immediate profits, and prefers missing opportunities for faster, more short-term focused arbitrageurs. The second one, on the other hand, aims to maximize immediate profit: users execute trades that make a profit by buying (or selling) crypto-assets from an AMM if their price, adjusted for fees, is below (or above) the market one, and then selling (or buying) them to another AMM at the market price. The generic flow of flash loan-based arbitrage transactions of a myopic arbitrageur is the following: the user borrows some crypto-assets through a flash loan on a AMM and uses the loaned amount to execute a series of exchanges with other AMMs. She generates profits from the price discrepancies between the assets.

Figure 1 illustrates the steps of a flash loan-based arbitrage involving the liquidity pool dYdX and the two AMMs, CurveY and Curve sUSD. The transaction dated back to July 31, 2020,[2] the arbitrageur borrowed 2.048 million USDC,[3]

[1] A financial instruments that allow users to exchange crypto-assets (tokens).
[2] Its address is 0xf7498a2546c3d70f49d83a2a5476fd9dcb6518100b2a731294d0d7b9f79f754a.
[3] USD Coin is a digital currency entirely backed by U.S. dollar assets: each USDC token represents a tokenized U.S. dollar, with a value closely mirroring that of one U.S. dollar.

executed two exchanges, one from USDC to DAI[4] from Curve Y, and the other one in the opposite directions from Curve sUSD. At the end of them, she paid its debt with dYdX but realized a profit of 16.182 thousand USDC.

Arbitrage, while beneficial for synchronizing asset prices across different DeFi markets, can be exploited to perform attacks on the market as described above. The capability of composing DeFi protocols through external calls is essential due to the interconnected nature of DeFi platforms. However, this composability not only facilitates chained trading and arbitrage opportunities but also introduces opportunities for malicious actors. Therefore, regulating external calls through robust policies is imperative to mitigate potential attacks and maintain market integrity. Here, we propose history-based policies as a means to regulate how the various protocols are composed, enforcing good behavior among the involved parties. More precisely, our policy framework addresses security concerns by analyzing transaction patterns and protocol interactions to identify potential vulnerabilities and proactively strengthen defenses against malicious activities. Protecting the invocation of the function `executeOperation` with a policy ϕ in the function `flashLoan` enables us to identify and potentially mitigate profitable unwanted scenarios like the one described above (see Section 4).

3 TinySol with Policies

In this section, we introduce the core calculus for smart contracts, building upon the work of Bartoletti et al. [4]. First, we present the syntax and semantics of the enriched language, with a particular focus on the introduced policy component (Sections 3.1 and 3.2). Following this, we formalize the execution of transactions using a labeled transition system (LTS). Finally, we review symbolic automata and show how we use them to define our policy checking mechanism (Section 3.3).

3.1 Syntax

We assume a set **Val** of *values* ranged by $v, k, \ldots$, a set **Const** of *constant names* $x, y, \ldots$, a set of *procedure names* $\mathtt{f}, \mathtt{g}, \ldots$. and a set **Addr** of *addresses* $\mathcal{X}, \mathcal{Y}, \ldots$, partitioned into *account addresses* $\mathcal{A}, \mathcal{B}, \ldots$ and *contract addresses* $\mathcal{C}, \mathcal{D}, \ldots$. As a notation, we write sequences in bold, e.g. $\boldsymbol{v}$ is a sequence of values, while ϵ denotes the empty sequence. We use $n, n', \ldots$ to range over $\mathbb{N}$, and $b, b', \ldots$ to range over boolean values.

In TinySol, a *contract* is a finite set of terms of the form $\mathtt{f}(\boldsymbol{x})\{S\}$, where S is a *statement*. Each term $\mathtt{f}(\boldsymbol{x})\{S\}$ is a contract function, where $\mathtt{f}$ is its name, $\boldsymbol{x}$ are its formal parameters (omitted when empty), and S is the function body. Moreover, we represent the internal state of a contract as a key-value store, formally, a partial function from keys $k \in$ **Val** to values $v \in$ **Val**.

The syntax of TinySol is in Figure 2. Many statements (Figure 2, left) are standard in common imperative languages, so we only comment on the most

[4] DAI is an Ethereum stablecoin that maintains a value close to USD using supply-controlling smart contracts, regulated by MakerDAO's MKR token holders.

relevant: `throw` raises an uncatchable exception, rolling-back the state; $\mathcal{X}.\mathtt{f}(\boldsymbol{v})\$n$ implements the external call mechanism calling the function `f` of the contract at address $\mathcal{X}$, passing the arguments $\boldsymbol{v}$, and transferring n units of currency to $\mathcal{X}$; $\phi[S]$ executes the statements S guarded by the policy ϕ, aborting the execution if the history generated by S is not compliant with ϕ.

Also, the expressions (Figure 2, right) are quite standard. We assume all the usual constants, arithmetic, logic, and cryptographic operators. Peculiar of TinySol is the expression $!k$ that evaluates to *true* if the key k is bound in the contract store, *false* otherwise. Then, the expression $?k$ looks up the value of the key k in the contract store. Finally, the expression $E_1.E_2$ evaluates E_1 to produce the address of a contract $\mathcal{X}$ and then evalutes E_2 in that address.

A *policy* ϕ (Figure 2, left) is a pair (re, E) where re is a regular expression that represents the sequence of function invocations, and E is a boolean expression. The regular expression syntax is standard: $\cdot$ denotes concatenation, $+$ choices, and * zero or more repetitions. The alphabet is represented by patterns *patt* capturing function invocation and may contain variables. More precisely, $\mathcal{C}.\mathtt{f}(\boldsymbol{v})$ denotes an invocation to the function `f` of the contract with address $\mathcal{C}$ with argument $\boldsymbol{v}$; whereas $x.\mathtt{f}(\boldsymbol{v})$ is a call to function `f` with argument $\boldsymbol{v}$ where x is a placeholder for the address of the contract the function belongs to; $\mathcal{C}.\mathtt{f}(\boldsymbol{x})$ denotes a call where the arguments are variables; finally, $x.\mathtt{f}(\boldsymbol{x'})$ is a call where the address and the arguments are denoted by variables. The variables occurring in a pattern can be used by the expression E to express some conditions. These variables will be bound to concrete values while checking the policy compliance.

Finally, we assume a mapping Γ from addresses to contracts that we use to retrieve the definition of a contract. In order to allow for a uniform treatment of account and contract addresses, we require that $\Gamma(\mathcal{A}) = \{\mathtt{f_{skip}}()\{\mathtt{skip}\}\}$ for all account addresses $\mathcal{A}$. Also, we use the following syntactic sugar: we write $\mathcal{X}.\mathtt{f}(\boldsymbol{v})$ to denote a call $\mathcal{X}.\mathtt{f}(\boldsymbol{v})\$n$, when there is no money transfer (i.e., $n = 0$). We write `if` E `then` S for `if` E `then` S `else skip`.

3.2 Semantics

We formalize the execution of TinySol contracts by providing operational semantics for both transactions and function contracts. In our semantics, the execution of a transaction is not deterministically started by a user account, and it triggers the execution of a contract function. A successful transaction results in a change of the blockchain state. We first present the semantics of TinySol statements and then the semantics of transactions. Note that our policy mechanism is activated during statement execution but affects transaction semantics since the corresponding transaction is reverted when a policy is violated.

Statements The semantics of a contract is given in terms of a transition system where configurations have the form:

$$\langle S, \sigma, \rho, \eta \rangle$$

$S ::=$	statement	$E ::=$	expression
$\texttt{skip}$	skip	v	value
$\mid \texttt{throw}$	exception	$\mid x$	const name
$\mid E := E'$	store update	$\mid \mathcal{X}$	address
$\mid S; S'$	sequence	$\mid \texttt{op}\ \boldsymbol{E}$	operator
$\mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S'$	conditional	$\mid ?E$	key lookup
$\mid \texttt{while}\ E\ \texttt{do}\ S$	loop	$\mid !E$	key bound?
$\mid \phi[S]$	security policy	$\mid E_1.E_2$	context
$\mid E_0.\texttt{f}(\boldsymbol{E_1})\$E_2$	call		
		$\mathit{patt} ::=$	function calls
$\phi ::= (re, E)$	policy	$\mathcal{C}.\texttt{f}(\boldsymbol{v})$	ground
$re ::=$	regular expression	$\mid x.\texttt{f}(\boldsymbol{v})$	addr. variable
patt	basic event	$\mid \mathcal{C}.\texttt{f}(\boldsymbol{x})$	arg. variables
$\mid \epsilon$	empty string	$\mid x.\texttt{f}(\boldsymbol{x'})$	addr. and arg. variables
$\mid re \cdot re$	sequential composition		
$\mid re + re$	choice		
$\mid re^\star$	iteration		

Fig. 2: Syntax of the language.

where S is the sequence of statements to be executed; σ is the starting state; ρ is the variable environment; η is the history recording the sequence of function calls, made during the execution.

In detail, a *blockchain state* σ: **Addr** $\to$ (**Val** $\rightharpoonup$ **Val**) is a map from addresses to key-value stores. A key-value store is, in turn, a partial function from keys to values and represents the internal state of a smart contract. We use brackets to represent finite maps, e.g., $\{^{v_1}/_{x_1}, \cdots, ^{v_n}/_{x_n}\}$ maps x_i to v_i, for $i \in 1..n$. As usual, when a key k is not bound to any value in $\sigma\,\mathcal{X}$, we write $\sigma\,\mathcal{X}\,k = \bot$. We postulate that for each address, there exists the distinct key `balance` recording the balance of that address. Moreover, when we refer to the key of an address $\mathcal{X}$, we use the notation $\mathcal{X}.k$ and call it a qualified key. Consequently, we write $\sigma(\mathcal{X}.k)$ for $\sigma\mathcal{X}k$. Below, we use the auxiliary operators $\sigma + \mathcal{X} : n$ and $\sigma - \mathcal{X} : n$ on states, to, respectively, increase/decrease the `balance` of $\mathcal{X}$ of n currency units:

$$\sigma \circ \mathcal{X} : n \;=\; \sigma\{^{(\sigma\,\mathcal{X}\,\texttt{balance}) \circ n}/_{\mathcal{X}.\texttt{balance}}\} \qquad (\circ \in \{+, -\})$$

A variable environment ρ: **Const** $\rightharpoonup$ **Val** is a partial function from constant name to values used to provide values to function parameters (that once assigned cannot be changed) and to special names such as `value` and `sender` used inside the body of functions.

A history η is a sequence of function calls given by the following grammar:

$$\eta ::= \epsilon \mid \mathcal{X}.\texttt{f}(\boldsymbol{v}) \mid \eta \cdot \eta$$

$$[\![v]\!]^{\mathcal{X}}_{\sigma,\rho} = v \qquad [\![x]\!]^{\mathcal{X}}_{\sigma,\rho} = \rho\, x \qquad [\![\mathcal{Y}]\!]^{\mathcal{X}}_{\sigma,\rho} = \mathcal{Y} \qquad [\![\mathtt{op}\boldsymbol{E}]\!]^{\mathcal{X}}_{\sigma,\rho} = \mathrm{op}\,[\![\boldsymbol{E}]\!]^{\mathcal{X}}_{\sigma,\rho} \qquad [\![E_1.E_2]\!]^{\mathcal{X}}_{\sigma,\rho} = [\![E_2]\!]^{[\![E_1]\!]^{\mathcal{X}}_{\sigma,\rho}}_{\sigma,\rho}$$

$$[\![?E]\!]^{\mathcal{X}}_{\sigma,\rho} = \sigma\,\mathcal{X}\,([\![E]\!]^{\mathcal{X}}_{\sigma,\rho}) \qquad [\![!\,E]\!]^{\mathcal{X}}_{\sigma,\rho} = \begin{cases} \mathit{true} & \text{if } [\![E]\!]^{\mathcal{X}}_{\sigma,\rho} \neq \bot \text{ and } \sigma\,\mathcal{X}\,([\![E]\!]^{\mathcal{X}}_{\sigma,\rho}) \neq \bot \\ \mathit{false} & \text{if } [\![E]\!]^{\mathcal{X}}_{\sigma,\rho} \neq \bot \text{ and } \sigma\,\mathcal{X}\,([\![E]\!]^{\mathcal{X}}_{\sigma,\rho}) = \bot \end{cases}$$

Fig. 3: Semantics of expressions.

where ϵ is the empty history, $\mathcal{X}.\mathtt{f}(\boldsymbol{v})$ denotes the invocation of the function `f` of the contract with address $\mathcal{X}$ passing the values $\boldsymbol{v}$ as arguments.

We give the operational semantics of statements in a big-step style, where transitions have the form

$$\langle S, \sigma, \rho, \eta\rangle \xrightarrow{\mathcal{X}} \langle \sigma', \eta'\rangle$$

meaning that the statement S executed in the contract $\mathcal{X}$, in the state σ, with environment ρ, and history η terminates and produces the final state σ' and history η'. Note that our semantics is parameterized by the contract address in which the evaluation is taking place.

Figure 3 shows the semantics of expressions where we use `op` to denote syntactic operators and op for their semantic counterparts. The semantics of expression results in a value and its definition is quite standard: the environment ρ is used to evaluate constant names x, while the state σ is used to evaluate keys in $!\,E$ and $?E$. The semantics of $E_1.E_2$ first evaluates E_1 within a contract with address $\mathcal{X}$ producing an address $\mathcal{Y}$, then evaluates E_2 within the contract $\mathcal{Y}$. Note that expressions have no side effects, and all the semantic operators are *strict*, i.e. their result is $\bot$ if some operand is $\bot$.

The semantics of statements is in Figure 4, and it is mostly standard, except for the last two rules. The rule [POLICY] evaluates the policy framing $\phi[S]$ construct: it executes the statement S that (if it terminates) produces a state σ' and a history η'; then, it checks that η' is compliant with the policy ϕ through the predicate $\eta' \vDash_{\sigma',\rho,\mathcal{X}} \phi$ defined in Section 3.3. If the history meets the policy, the computation results in the pair $\langle \sigma', \eta'\rangle$, otherwise it is aborted.

The rule [PROCEDURE CALL] handles a procedure call $E_0.\mathtt{f}(\boldsymbol{E_1})\$E_2$ within the contract $\mathcal{X}$. The premise of the rule requires that (i) E_0 evaluates to an address $\mathcal{Y}$; (ii) E_2 evaluates to a non-negative number n, not exceeding the balance of $\mathcal{X}$; (iii) the contract at $\mathcal{Y}$ has a procedure named `f` with formal parameters $x_1 \cdots x_h$; (iv) $\boldsymbol{E_1}$ evaluates to a sequence of values of length h. If all these conditions hold, then the procedure body S is executed in a state where $\mathcal{X}$'s balance is decreased by n, $\mathcal{Y}$'s balance is increased by n, and in an environment where the formal parameters are bound to the actual ones, and the special names `sender` and `value` are bound, respectively, to $\mathcal{X}$ (the caller) and n (the value transferred to $\mathcal{Y}$). Executing S may affect both the store of $\mathcal{X}$ and, in case of procedure calls, also the store of other contracts.

$$\frac{}{\langle \texttt{skip}, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma, \eta\rangle}\ [\textsc{skip}] \qquad \frac{[\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} = k \quad [\![E']\!]^{\mathfrak{X}}_{\sigma,\rho} = v}{\langle E := E', \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma\{^{v}/_{\mathfrak{X}.k}\}, \eta'\rangle}\ [\textsc{update}]$$

$$\frac{[\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} = b \quad b \in \{true, false\}}{\langle \texttt{if } E \texttt{ then } S_{true} \texttt{ else } S_{false}, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle S_b, \eta'\rangle}\ [\textsc{condition}] \qquad \frac{\langle S_0, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}{\langle S_0; S_1, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}\ [\textsc{sequence}]$$

$$\frac{[\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} = false}{\langle \texttt{while } E \texttt{ do } S, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma, \eta\rangle}\ [\textsc{while false}] \qquad \frac{[\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} = true \quad \langle S, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}{\langle \texttt{while } E \texttt{ do } S, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}\ [\textsc{while true}]$$

$$\frac{\langle S, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle \quad \eta' \vDash_{\sigma',\rho,\mathfrak{X}} \phi}{\langle \phi[S], \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}\ [\textsc{policy}]$$

$$\frac{\begin{array}{ll} [\![E_0]\!]^{\mathfrak{X}}_{\sigma,\rho} = \mathcal{Y} & \texttt{f}(x)\{S\} \in \Gamma(\mathcal{Y}) \\ [\![\boldsymbol{E_1}]\!]^{\mathfrak{X}}_{\sigma,\rho} = \boldsymbol{v} & \sigma' = \sigma - \mathfrak{X} : n + \mathcal{Y} : n \\ [\![E_2]\!]^{\mathfrak{X}}_{\sigma,\rho} = n & \rho' = \{^{\mathfrak{X}}/_{\texttt{sender}}, {}^{n}/_{\texttt{value}}, {}^{v}/_{x}\} \end{array} \qquad \begin{array}{l} \eta' = \eta :: \mathcal{Y} : f(v) \\ \langle S, \sigma', \rho', \eta'\rangle \xrightarrow{\mathfrak{X}} \langle \sigma'', \eta''\rangle \end{array}}{\langle E_0.\texttt{f}(\boldsymbol{E_1})\$E_2, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma'', \eta''\rangle}\ [\textsc{procedure call}]$$

Fig. 4: Semantics of statements.

Transactions A *transaction* T is a term of the form $\mathcal{A} \xrightarrow{n} \mathcal{C} : \texttt{f}(\boldsymbol{v})$, where $\mathcal{A}$ is the address of the caller, $\mathcal{C}$ is the address of the called contract, $\texttt{f}$ is the called procedure, n is the value transferred from $\mathcal{A}$ to $\mathcal{C}$, and $\boldsymbol{v}$ is the sequence of actual parameters. We formalize the execution of a transaction T through a transition system where configurations have the form $\langle \sigma, \eta\rangle$ describing the blockchain state where T takes place producing a new configuration $\langle \sigma', \eta'\rangle$. Executing the transaction causes the execution of the contract function $\texttt{f}$. The call of $\texttt{f}$ is defined by the following rules:

$$\frac{\texttt{f}(\boldsymbol{x})\{S\} \in \Gamma(\mathcal{C}) \quad \sigma\, \mathcal{A}\, \texttt{balance} \geq n \qquad \langle S, \sigma - \mathcal{A} : n + \mathcal{C} : n, \{^{\mathcal{A}}/_{\texttt{sender}}, {}^{n}/_{\texttt{value}}, {}^{\boldsymbol{v}}/_{\boldsymbol{x}}\}, \eta\rangle \xrightarrow{\mathcal{C}} \langle \sigma', \eta'\rangle}{\langle \sigma, \eta\rangle \xrightarrow{\mathcal{A} \xrightarrow{n} \mathcal{C}:\texttt{f}(\boldsymbol{v})} \langle \sigma', \eta'\rangle}\ [\textsc{Tx1}]$$

$$\frac{\texttt{f}(\boldsymbol{x})\{S\} \in \Gamma(\mathcal{C}) \qquad \big(\sigma\, \mathcal{A}\, \texttt{balance} < n \quad \text{or} \quad \langle S, \sigma - \mathcal{A} : n + \mathcal{C} : n, \{^{\mathcal{A}}/_{\texttt{sender}}, {}^{n}/_{\texttt{value}}, {}^{\boldsymbol{v}}/_{\boldsymbol{x}}\}, \eta\rangle \not\rightarrow\big)}{\langle \sigma, \eta\rangle \xrightarrow{\mathcal{A} \xrightarrow{n} \mathcal{C}:\texttt{f}(\boldsymbol{v})} \langle \sigma, \eta\rangle}\ [\textsc{Tx2}]$$

Rule [Tx1] handles the case where the transaction is successful: this happens when $\mathcal{A}$'s balance is at least n, and the procedure call terminates in a non-error state, meaning that all policies defined in $\texttt{f}$ are satisfied. Note that n units of currency are transferred to $\mathcal{C}$ *before* starting to execute $\texttt{f}$, and that the names $\texttt{sender}$ and $\texttt{value}$ are set, respectively, to $\mathcal{A}$ and n. Instead, [Tx2] applies either when $\mathcal{A}$'s balance is not enough, or the execution of $\texttt{f}$ fails (this also covers the case when $\texttt{f}$ does not terminate or during the execution of the function body is not compliant with all the policies of function $\texttt{f}$). In these cases, T does not alter the state $\langle \sigma, \eta\rangle \xrightarrow{\mathsf{T}} \langle \sigma, \eta\rangle$.

We naturally extend the semantics above when we have a sequence of transactions $[\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_n]$, formally:

$$\langle \sigma_0, \eta_0 \rangle \xrightarrow{\mathsf{T}_1} \langle \sigma_1, \eta_1 \rangle \xrightarrow{\mathsf{T}_2} \langle \sigma_2, \eta_2 \rangle \cdots \xrightarrow{\mathsf{T}_n} \langle \sigma_n, \eta_n \rangle$$

Note that erroneous transactions occur in the execution of the sequence $[\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_n]$ does not effect the semantics, since rule [Tx2] leaves the state and the history unchanged.

3.3 Checking policies

In this section, we describe how we ensure that a history η complies with a policy $\phi = (re, E)$. Intuitively, our verification procedure consists of two steps: (i) we check that η belongs to the language of re, by transforming re into a symbolic automaton; (ii) we evaluate the assertion E in the current state and check that it evaluates to true. If both steps succeed, η complies with ϕ, otherwise, it does not. Below, we first review the basic notions of symbolic regular expressions and automata [16], and then we detail the two steps above.

Symbolic automata are generalizations of finite automata where the alphabet is a Boolean algebra, and transitions are labeled by predicates over such algebra. This allows symbolic automata to operate over infinite alphabets, such as the set of rational numbers, while retaining the decidability properties of their finite counterparts. Similarly, symbolic regular expressions are a generalization of regular expressions operating over a Boolean algebra.

An *effective Boolean algebra* $\mathcal{A}$ is a tuple

$$(\mathfrak{D}, \Psi, [\![\_]\!], \top, \bot, \vee, \wedge, \neg) \tag{1}$$

where $\mathfrak{D}$ is a (possibly infinite) set of *domain elements*; Ψ is a set of *predicates* over $\mathfrak{D}$ closed under the logical connectives $\vee, \wedge, \neg$; the component $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ is a *denotation function* such that $[\![\top]\!] = \mathfrak{D}$, $[\![\bot]\!] = \emptyset$, and $\forall \psi_1, \psi_2 \in \Psi, [\![\psi_1 \vee \psi_2]\!] = [\![\psi_1]\!] \cup [\![\psi_2]\!]$, and $[\![\neg\psi_1]\!] = \mathfrak{D} \setminus [\![\psi_1]\!]$. Moreover, when $[\![\psi_1]\!] \neq \emptyset$, then the predicate ψ_1 is *satisfiable*. We require that checking satisfiability is decidable.

Example 1 (Linear Integer Arithmetic). As an example of effective Boolean algebra, consider $(\mathbb{Z}, \Psi, [\![\cdot]\!], \bot, \top, \vee, \wedge, \neg)$, where $\mathbb{Z}$ is the set of integer numbers. The set Ψ contains all quantifier-free formulas on integer linear arithmetic, namely, formulas such as $\psi_{>0}(x) \triangleq x > 0$ and $\psi_{\text{odd}}(x) \triangleq x\%2 = 1$. Given a formula ψ, $[\![\psi]\!]$ denotes the set of integer satisfying it; $\bot, \top, \vee, \wedge, \neg$ represent the always false, true predicates and the standard logical connectives. It is noteworthy that satisfiability is decidable for such a Boolean algebra. Indeed, given a formula ψ, the interpretation function $[\![\psi]\!]$ can use an SMT solver for checking the satisfiability and model generation. For example, $[\![\psi_{>0}(x)]\!]$ is true if there exists an integer x such that $x > 0$, and $[\![\psi_{\text{odd}}(x)]\!]$ is true if there exists an integer x such that $x\%2 = 1$. Boolean operations allow for the combination and manipulation of these predicates within algebra.

Given an effective Boolean algebra $(\mathfrak{D}, \Psi, [\![\_]\!], \top, \bot, \vee, \wedge, \neg)$, a *symbolic regular expression* is defined as follows:

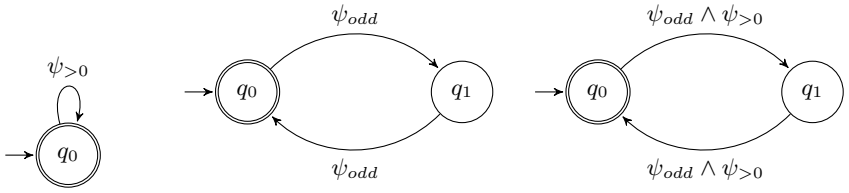
- The constants ε and $\emptyset$ are symbolic regular expressions denoting the languages $\{\varepsilon\}$ and $\emptyset$, respectively.
- Any predicate $\psi \in \Psi$ is a symbolic regular expression that accepts the language defined as $\mathcal{L}(\psi) = [\![\psi]\!]$.
- Given symbolic regular expressions sre_1 and sre_2, the expressions $sre_1 + sre_2$, $sre_1 \cdot sre_2$, and sre_1^* are symbolic regular expressions.

Elements of $\mathfrak{D}$ are *characters*. A *word* over $\mathfrak{D}$ is a sequence $a_1 \dots a_m$, where $a_i \in \mathfrak{D}$, $i = 1, \dots, m$. If $m = 0$, then we have the *empty* word, denoted by ε. The set of all words over $\mathfrak{D}$ is denoted by $\mathfrak{D}^\star$. Any language defined by a symbolic regular expression is a *symbolic regular language*.

A *symbolic nondeterministic finite automaton with epsilon transitions*, *s-ϵNFA* is a quintuple

$$\mathcal{N} = (\mathcal{A}, Q, \Delta, I, F)$$

where the alphabet $\mathcal{A}$ is an effective Boolean algebra; Q is a finite set of *states*; $\Delta \subseteq Q \times (\Psi \cup \{\epsilon\}) \times Q$ is a finite set of *transitions*; $I \subseteq Q$ is the set of *initial states*; and $F \subseteq Q$ is the set of *final states*. The language of $\mathcal{N}$, denoted by $\mathcal{L}(\mathcal{N})$ is the set of words $w \in \mathfrak{D}^\star$ such that either $w = \varepsilon$ or $w = a_0 \dots a_k$ and there exist a sequence of transitions $(q_i, \psi_i, q_{i+1}) \in \Delta$ and $a_i \in [\![\psi_i]\!]$ for $i \in [0, k]$, with $q_0 \in I$ and $q_k \in F$.



(a) Automaton $\mathcal{M}_{pos}$. (b) Automaton $\mathcal{M}_{ev/odd}$. (c) Automaton $\mathcal{M}_{ev/odd} \times \mathcal{M}_{pos}$.

Fig. 5: Symbolic automata built on the Boolean algebra of Example 1.

Example 2. Consider again the Boolean algebra of Example 1. Figure 5 illustrates three examples of *s-ϵNFA*s, $\mathcal{M}_{\text{pos}}$, $\mathcal{M}_{\text{ev/odd}}$, and $\mathcal{M}_{\text{ev/odd}} \times \mathcal{M}_{\text{pos}}$, built on this algebra and where $\psi_{>0}(x) \wedge \psi_{\text{odd}}(x)$ denote the formula from Example 1. The automaton $\mathcal{M}_{\text{pos}}$ accepts all strings consisting only of positive numbers, while $\mathcal{M}_{\text{ev/odd}}$ accepts all strings of even length consisting only of odd numbers. For example, $\mathcal{M}_{\text{ev/odd}}$ accepts the string $2, 4, 6, 2$ (we separate characters with a comma for clarity, but the comma is not in the language) and rejects the strings $2, 4, 6$ and $51, 26$. The automaton $\mathcal{M}_{\text{ev/odd}} \times \mathcal{M}_{\text{pos}}$ is obtained by the product of $\mathcal{M}_{\text{pos}}$ and $\mathcal{M}_{\text{ev/odd}}$, and accepts the language $\mathcal{L}(\mathcal{M}_{\text{pos}}) \cap \mathcal{L}(\mathcal{M}_{\text{ev/odd}})$.

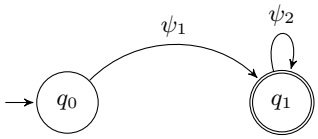


Fig. 6: Automaton $\mathcal{N}$ for the symbolic regular expression *sre* of Example 3.

Tamm and Veanes [16] proved that we can apply the counterpart of Thompson's construction [17,20] to a symbolic regular expression to derive an *s-ϵNFA*. For further details on symbolic regular expressions and automata, we refer the interested reader to [16].

Now we define $\mathcal{P} = (\mathfrak{D}, \Psi, [\![\_]\!], \top, \bot, \vee, \wedge, \neg)$ the Boolean algebra for our policies, as follows:

- $\mathfrak{D}$ is the set of ground patterns generated by the first production *patt* of Figure 2, namely, $\mathfrak{D} = \{p \in patt \mid p = \mathcal{C}.\mathtt{f}(\boldsymbol{v})\}$.
- Ψ is the set of pattern predicates defined as follows. For each pattern $p \in patt$ we introduce a predicate $\psi_p(x) \triangleq x = p$ that holds when the argument x (a ground pattern from $\mathfrak{D}$) matches p for some substitution θ of variables in p. Therefore, we define the set of predicates $\Psi = \{\psi_p \mid \psi_p \text{ for } p \in patt\}$ and we close it under logical connectives $\vee, \wedge, \neg$.
- for each pattern predicate ψ_p, $[\![\psi_p]\!] = \{q \mid \psi_p(q) \text{ holds for some } \theta\}$.

We note that the Boolean algebra $\mathcal{P}$ is effective because $[\![\psi_p]\!] \neq \emptyset$ is decidable. Indeed, at least two approaches exist to check the satisfiability of a pattern predicate ψ_p. The first approach involves a straightforward matching between the argument x (a ground pattern) and the pattern p, finding a possible assignment to the variables of p that makes them syntactically equal. The second approach instead involves encoding our Boolean algebra in the theory of the uninterpreted function with equality and using a standard SMT solver to check satisfiability.

To ensure that a history η complies with a policy $\phi = (re, E)$ within the contract $\mathcal{X}$ and in the state σ', in symbols $\eta' \vDash_{\sigma',\rho,\mathcal{X}} \phi$, we proceed as follows. First, we transform re into a symbolic regular expression predicating on the pattern of the policy and build the corresponding *s-ϵNFA* $\mathcal{N}$. Proposition 1 of [16] ensures that the *s-ϵNFA* accepts the same language of the symbolic regular expression. Then we check if the history η' belongs to the language $\mathcal{L}(\mathcal{N})$: this means that if there exists a sequence of transitions $q_0 \xrightarrow{p_1} \dots \xrightarrow{p_n} q_f$ for $\eta' = p_1 \dots p_n$ where each transition is possible for some substitution θ_i. Therefore, η' belongs to the language $\mathcal{L}(\mathcal{N})$ holds for some $\theta' = \theta_1 \dots \theta_n$. Then, we evaluate the assertion $[\![E]\!]_{\sigma',\rho\cdot\theta'}$ in the state σ' and in the environment ρ extended with the substitution θ' and check that it evaluates to true. If all the steps above succeed, η' complies with ϕ, otherwise, it does not: when the history η' does not belong to the language of $\mathcal{N}$ ($\eta' \notin \mathcal{L}(\mathcal{N})$) or E evaluates to false, the execution is aborted because of a policy violation.

Example 3. Consider a policy $\phi = (re, E)$ where $re = x.\mathtt{f}_1(y) \cdot z.\mathtt{f}_2(w)^\star$ and $E = (y > 0 \wedge w = 13)$. We first transform re into a symbolic regular expression sre as follows: take the predicates $\psi_1 = x.\mathtt{f}_1(y)$ and $\psi_2 = z.\mathtt{f}_2(w)$, then $sre = \psi_1 \cdot \psi_2^\star$ by simply lifting the patterns of re in the corresponding predicates. From this symbolic regular expression, we build the corresponding automaton $\mathcal{N}$ shown in Figure 6. Given a history $\eta = c.\mathtt{f}_1(18) \cdot d.\mathtt{f}_2(13) \cdot d.\mathtt{f}_2(13)$, it is easy to check that $\eta \in \mathcal{L}(\mathcal{N})$ with variables assignment $\theta = \{y \mapsto 18, x \mapsto c, z \mapsto d, w \mapsto 13\}$, so η satisfies re. Then, we evaluate E in a state σ and environment ρ extended with θ, namely $[\![E]\!]_{\sigma,\rho\cdot\theta} = (y > 0 \wedge w = 13)$ that evaluates to true. Therefore, history η is compliant with the policy ϕ.

4 Detecting Arbitrage

This section shows how our policy framework can prevent the arbitrage shown in Figure 1 and illustrates how our policy checking mechanism works. We introduce a policy to protect the call to `executeOperation()` in Listing 1.1 for detecting sequences of function calls that may indicate the attempt of an arbitrage. The idea is that our policy detects sequences of invocations to the `swap` method to transfers of tokens in opposing directions involving the same token types and different AMMs. The policy will be parametric on the contracts performing the transfers and on the types of tokens. Below, we denote with `User`, `dYdX`, `CurveY`, and `CurvesUSD` the addresses of the contracts involved in Figure 1 and with `Flash` the address of the contract created by the user implementing the `executeOperation` function. Moreover, we use τ_a and τ_b to indicate the token types USDC and DAI, respectively. Also, we assume that the market price for token types τ_a and τ_b is provided by an Oracle contract with address `O`. Assume `User` requests a `loan` of 2048 million tokens of type τ_a and invokes the `flashLoan` function of Listing 1.2 passing the address of `Flash` as first parameter:

```
function executeOperation(
  asset, amount, fee, address, memory
)  {
  // First Swap on CurveY
  CurveY.swap(User, USDC, DAI, 2048M);

  // Get New Token Balance after the first swap
  uint256 toBalance = IERC20(DAI).balanceOf(address(this));
  require(toBalance > 0, "Swap on CurveY failed to return any tokens");

  // Second Swap on CurvesUSD
  CurvesUSD.swap(User, DAI, USDC, 2028M);

  // Get New Token Balance after the second swap
  uint256 finalBalance = IERC20(USDC).balanceOf(address(this));
  require(finalBalance > 0, "Swap on CurvesUSD failed to return any tokens");

  // Ensure that final balance is enough to repay the loan with fee
  require(finalBalance >= amount.add(fee), "Arbitrage trade did not return
      enough tokens to repay the loan");
```

Listing 1.2: Function implementing an arbitrage to be used with `flashLoan`.

The execution of `flashLoan()` results in the following sequence of calls:

1. `dYdX.balance()`
2. `dYdX.transfer`$(\tau_a, \texttt{Flash}, \texttt{loan})$
3. `Flash.executeOperation()`
4. `CurveY.swap`$(\texttt{User}, \tau_a, \tau_b, \texttt{amount}_1)$
5. `CurvesUSD.swap`$(\texttt{User}, \tau_b, \tau_a, \texttt{amount}_2)$
6. `Flash.payback(dYdX)`

First, the `dYdX` contract performs some sanity checks, during these checks, it invokes the `dYdX.balance()` function. Then, the requested amounts of tokens are transferred to the contract `Flash` via a call to `dYdX.transfer`$(\tau_a, \texttt{Flash}, \texttt{loan})$. This transfer of tokens is followed by the invocation of `executeOperation()` of the `Flash` contract that executes the two swaps: the first swap concerns $\texttt{amount}_1$ (2048 million) of tokens of type τ_a to acquire $\texttt{amount}_2$ (2028 million) tokens of type τ_b on `CurveY` (`CurveY.swap`$(\texttt{User}, \tau_a, \tau_b, \texttt{amount}_1)$). Then, the function performs the reverse swap on `CurvesUSD` (`CurvesUSD.swap`$(\texttt{User}, \tau_b, \tau_a, \texttt{amount}_2)$). After these swaps, the `Flash` contract repays its debt to the `dYdX`, including additional fees, via the call to the function `Flash.payback(dYdX)`.

```
// Execute the operation in the receiver contract
φ { if not receiverAddress.executeOperation(asset, amount, fee, msg.sender,
     params):
    raise error "FlashLoan execution failed" }
```

Listing 1.3: Policy introduction in the `flashloan`.

We prevent the execution of the arbitrage by enclosing the call to the function `executeOperation` inside a policy framing construct. The amended line of code of the `flashLoan` function is shown in Listing 1.3. The policy ϕ blocks the execution when it detects a sequence of swaps of opposing directions involving the same token types across different AMMs (steps 4 and 5 of the flow described above), that AMMs have a different token ratio, and that the token price significantly differs from their market price provided by the oracle `O`. Formally, the policy $\phi = (re, E)$ is defined as follows:

$$
\begin{aligned}
re =& a.\texttt{swap}(x, y, z) \cdot b.\texttt{swap}(x', z', y') \\
E =& (a \neq b \wedge x = x' \wedge \frac{a.y}{a.z} = \frac{b.z'}{b.y'} \wedge \left|\frac{a.y - \texttt{O}.y}{a.z - \texttt{O}.z}\right| \leq \epsilon \wedge \left|\frac{b.z' - \texttt{O}.z'}{b.y' - \texttt{O}.y'}\right| \leq \epsilon)
\end{aligned}
$$

where the regular expression re detects the sequence of the two consecutive swaps, while the assertion E holds when the involved AMMs are different ($a \neq b$), the calling contract is the same ($x = x'$), the ratio between the exchanged token is the same ($\frac{a.y}{a.z} = \frac{b.z'}{b.y'}$), and the ratio of the difference between the exchanged tokens of the AMMs and the market price of each asset as given by contract Oracle `O` to is below a small threshold ϵ. Note that in the assertion E we assume that the amount of the token type y exchanged through a swap on the AMM a is accessed using the syntax $a.y$. We use the same syntax for the other token types and AMMs, and for accessing the market price of

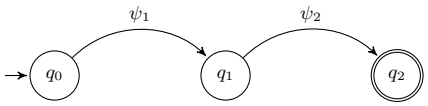


Fig. 7: Automaton $\mathcal{A}$ for the policy preventing flash-loan-based arbitrages.

each token type via the oracle `O`. We first transform re into the symbolic regular expression $\psi_1 \cdot \psi_2$ where $\psi_1 = a.\texttt{swap}(x, y, z)$ and $\psi_2 = b.\texttt{swap}(x', z', y')$, and then build the corresponding automaton $\mathcal{A}$ shown in Figure 7. Given $\eta = \texttt{CurveY.swap}(\texttt{User}, \tau_\texttt{a}, \tau_\texttt{b}, \texttt{amount}_1) \cdot \texttt{CurvesUSD.swap}(\texttt{User}, \tau_\texttt{b}, \tau_\texttt{a}, \texttt{amount}_2)$, it is easy to check that $\eta \in \mathcal{L}(\mathcal{A})$ with variables assignment $\theta = \{a \mapsto \texttt{CurveY}, b \mapsto \texttt{CurvesUSD}, x \mapsto \texttt{User}, x' \mapsto \texttt{User}, y \mapsto \tau_\texttt{a}, z \mapsto \tau_\texttt{b}, y' \mapsto \tau_\texttt{b}, z' \mapsto \tau_\texttt{a}\}$. Then, we evaluate E in a state σ and environment ρ extended with θ, namely $[\![E]\!]_{\sigma,\rho\cdot\theta}$, that evaluates to false because the ratio of between the exchanged tokens ($\frac{\texttt{CurveY.}\tau_\texttt{a}}{\texttt{CurveY.}\tau_\texttt{b}} \neq \frac{\texttt{CurvesUSD.}\tau_\texttt{a}}{\texttt{CurvesUSD.}\tau_\texttt{b}}$) differs. Indeed, as shown in Figure 1, the price ratio for `CurveY` is 1.010 USDC/DAI, while for `CurvesUSD` is 1.918 USDC/DAI. This discrepancy confirms the attempt of an arbitrage, so the history η is not compliant with the policy ϕ, and the execution of `flashLoan` is reverted.

5 Related Work

We compare our work with the relevant literature. Over the past few decades, the formalism of symbolic automata has been extensively studied [16,6,7], and applied to solve different verification tasks. Here, we rely on the results of Tamm and Veanes in [16] concerning the theory of symbolic regular languages and on the relation between symbolic automata and symbolic regular expression. To the best of our knowledge, this is the first paper that uses symbolic automata to express and enforce security policy following Schneider [14].

To formally present our policy mechanism, we extend TinySol, the minimal calculus for the Solidity contract introduced by Bartoletti et al. [4]. We enhanced the language by introducing a way to express policy over the execution history and the construct of policy framing to enforce them. Moreover, we extended the semantics accordingly and defined a procedure for checking policies at run-time.

The concept of history-based policies has been explored in previous works, notably by Skalka and Smith [15] and Bartoletti et al. [2,3]. Skalka and Smith [15] combine type and effect systems with model-checking to extract a history expression from the code automatically and to verify that a higher-order program satisfies a policy ϕ. Their approach allows the static enforcement of history- and stack-based security mechanisms. Bartoletti et al. [2] uses a similar mechanism to express access control policies over resources and to verify that objects and resources are created and manipulated according to such policies. The main difference between these papers and ours is that their verification procedure occurs statically and on the code of programs, whereas our verification occurs at run-time. In particular, their verification procedures require the code of the whole

program or at least an abstract description of the behavior of all called functions, given in terms of types and effects. Here, we do not have this assumption, so we can easily deal with open-world scenarios like DeFi, where a contract can be called by other unknown contracts.

Several papers address price manipulation vulnerabilities in DeFi and propose detection tools for smart contracts, such as DeFiRanger [21], DeFiScanner [19], DeFiTainter [9], BLOCKEYE [21], and FlashSyn [5]. DeFiRanger [21] builds a cash flow tree (CFT) from Ethereum transaction data to reflect token transfers and accounts, giving a DeFi meaning to blockchain transactions. Wang et al. [19] uses taint analysis to uncover price manipulation vulnerabilities in DeFi protocols. BLOCKEYE [18] performs automated and accurate real-time attack detection of Ethereum-based DeFi protocols by symbolic reasoning techniques. FlashSyn [5] automatically synthesizes adversarial transactions exploiting DeFi protocols via flash loans, overcoming complex DeFi logic and search space challenges with a synthesis-via-approximation technique. Unlike the above papers, our work focuses on formal verification, ensuring run-time security by validating histories produced by smart contract execution. This methodology guarantees the absence of security violations during execution, setting it apart from existing solutions. Finally, Zhou et al. in [22] introduced a method based on analyzing internal transactions to detect and prevent attacks within DeFi platforms. They emphasize policy enforcement and vulnerability management amidst DeFi's composability, especially concerning arbitrage scenarios utilizing external calls. Although the goal of this paper is similar to ours, their focus is primarily on automated revenue extraction through trading strategies. In contrast, our smart contract calculus emphasizes formal methods and run-time verification for robust policy enforcement and vulnerability management.

6 Conclusion

In this paper, we introduced a formal framework for specifying and enforcing security policies to regulate external calls in smart contracts. We enhanced TinySol, a core calculus for smart contracts, by incorporating constructs that allow developers to specify their policies at the code level, and a run-time verification mechanism to check that the called code is compliant with the policy.

In future work, we plan to study how our mechanism can be implemented. We envisage two possible directions: implementing the policy checking mechanism inside the EVM or encoding it directly within Solidity. Moreover, we want to explore other profitable scenarios within the DeFi ecosystem that could leverage our policy framework to prevent speculative actions. Finally, we will study if we can verify that an external call satisfies the policies up-front the transaction without actually running it.

Acknowledgments Work partially supported by projects SERICS (PE00000014) and PRIN AM∀DEUS (P2022EPPHM) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

1. Aave Documentation V3: Flash loans. https://docs.aave.com/developers/guides/flash-loans (2023), accessed: August, 2024
2. Bartoletti, M., Degano, P., Ferrari, G.L.: History-based access control with local policies. In: Proceedings of 8th International Conference on Foundations of Software Science and Computational Structures, FOSSACS 2005. pp. 316–332. Springer (2005)
3. Bartoletti, M., Degano, P., Ferrari, G., Zunino, R.: Local policies for resource usage analysis. ACM Trans. Program. Lang. Syst. 31(6), 23:1–23:43 (2009)
4. Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for solidity contracts. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Proceedings. pp. 233–243. Springer (2019)
5. Chen, Z., Beillahi, S.M., Long, F.: Flashsyn: Flash loan attack synthesis via counter example driven approximation. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–13 (2024)
6. Drews, S., D'Antoni, L.: Learning symbolic automata. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 173–189. Springer (2017)
7. Fisman, D., Frenkel, H., Zilles, S.: Inferring symbolic automata. Logical Methods in Computer Science 19 (2023)
8. John Fáwọlé: A broad overview of reentrancy attacks in solidity contracts. https://www.quicknode.com/guides/ethereum-development/smart-contracts/a-broad-overview-of-reentrancy-attacks-in-solidity-contracts (2023), accessed: August, 2024
9. Kong, Q., Chen, J., Wang, Y., Jiang, Z., Zheng, Z.: Defitainter: Detecting price manipulation vulnerabilities in defi protocols. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 1144–1156 (2023)
10. Kozen, D.: Language-based security: Invited lecture. In: International Symposium on Mathematical Foundations of Computer Science. pp. 284–298. Springer (1999)
11. Milionis, J., Moallemi, C.C., Roughgarden, T.: Automated market making and arbitrage profits in the presence of fees. arXiv preprint arXiv:2305.14604 (2023)
12. OWASP Project: Reentrancy attack. https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC01-reentrancy-attacks.html (2024), accessed: August, 2024
13. Qin, K., Zhou, L., Livshits, B., Gervais, A.: Attacking the defi ecosystem with flash loans for fun and profit. In: International conference on financial cryptography and data security. pp. 3–32. Springer (2021)
14. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security (TISSEC) 3(1), 30–50 (2000)
15. Skalka, C., Smith, S.: History effects and verification. In: Asian Symposium on Programming Languages and Systems. pp. 107–128. Springer (2004)
16. Tamm, H., Veanes, M.: Theoretical aspects of symbolic automata. In: International Conference on Current Trends in Theory and Practice of Informatics. pp. 428–441. Springer (2017)
17. Thompson, K.: Programming techniques: Regular expression search algorithm. Communications of the ACM 11(6), 419–422 (1968)

18. Wang, B., Liu, H., Liu, C., Yang, Z., Ren, Q., Zheng, H., Lei, H.: Blockeye: Hunting for defi attacks on blockchain. In: 2021 IEEE/ACM 43rd international conference on software engineering: companion proceedings (ICSE-companion). pp. 17–20. IEEE (2021)
19. Wang, D., Wu, S., Lin, Z., Wu, L., Yuan, X., Zhou, Y., Wang, H., Ren, K.: Towards a first step to understand flash loan and its applications in defi ecosystem. In: Proceedings of the Ninth International Workshop on Security in Blockchain and Cloud Computing. pp. 23–28 (2021)
20. Watson, B.W.: A taxonomy of finite automata construction algorithms. Computing science notes, Technische Universiteit Eindhoven (1993)
21. Wu, S., Yu, Z., Wang, D., Zhou, Y., Wu, L., Wang, H., Yuan, X.: Defiranger: Detecting defi price manipulation attacks. IEEE Transactions on Dependable and Secure Computing PP, 1–15 (2023)
22. Zhou, L., Qin, K., Cully, A., Livshits, B., Gervais, A.: On the just-in-time discovery of profit-generating transactions in defi protocols. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 919–936. IEEE (2021)