RUHR UNIVERSITÄT BOCHUM RUB MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY

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

Yan Farba
yan.farba@mpi-sp.org

Clara Schneidewind
First examiner, Group leader
clara.schneidewind@mpi-sp.org

Sebastian Holler
Second examiner, Supervisor
sebastian.holler@mpi-sp.org

A thesis submitted for the degree of
B.Sc. “Applied Computer Science”
at Ruhr-Universität Bochum in 2025.
The research was conducted at the
Max Planck Institute for Security and Privacy.

Titel meiner Abschlussarbeit / title of the final thesis

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

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich erkläre mich des Weiteren damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Statutory Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university to obtain an academic degree.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I furthermore agree that the digital version of this thesis will be used to subject the paper to plagiarism examination.

Not this English translation but only the official version in German is legally binding.

23/3/25	Farba Yan	
Datum / Date	Name, Vorname	Unterschrift / Signature

Table of contents

List of Figures

Acknowledgements

I would like to thank Clara Schneidewind, the research group leader, and Sebastian Holler, my supervisor, for their extensive support, thorough reviews, and collaboration. I am very fortunate to have had such great and caring supervisors guide me through my first research work. Thanks to Clara and Sebastian, I felt part of the group from day one and experienced the excellence of the Max Planck Institute for Security and Privacy firsthand. I was warmly welcomed by the group members and, despite my little research experience, was always invited to the engaging discussions. I am lucky to have met so many wonderful and inspiring people during my work, to have slacked at the lobby playing three-dimensional tic-tac-toe, and the three months I spent at MPI-SP made me want to stay.

I would like to express my deepest gratitude to my loving family. My whole studies would not be possible if not for my dear mother and father. I love you so much, thank you for trusting my choices and supporting me along the way! My dear grandparents ensured that I stayed disciplined along the way and encouraged me every day. And, of course, without my loving girlfriend, I cannot imagine going through the last intensive weeks.

Lastly, I would like to thank my loved ones in my native language:

> Дорогие мама, папа и Юна, бабушка Люда и дедушка Олег, прабабушка Ося и бабушка Мирра, и моя милая Настя! Я вас очень люблю! Спасибо вам за поддержку, без вас я не был бы там, где я есть!

Abstract

Smart contracts are stateful programs that run on a blockchain, hold money, and handle transactions. They can be seen as blockchain users, with behavior determined by code. Their state inherently depends on the order of transactions in which they are invoked. The order of transactions in the blockchain is determined by miners, users who collect transactions and order them in the blocks they mine. Order dependency may incentivize malicious miners to reorder transactions to smart contracts for their monetary gain. Such a malicious act is called *front-running attack*, in which the order of transactions is manipulated to make a profit at the cost of other users. Front-running attacks can be prevented: if the smart contract is invoked under conditions, in which it is order-independent, the state cannot be manipulated by transactions order and a malicious miner cannot make any profit. This thesis introduces a static-analysis procedure, which generates such order-independence conditions for a contract from its source code. We cover smart contracts in Ethereum, which are written in the high-level language Solidity.

Chapter 1

Introduction

In 2009, Bitcoin [21] introduced a novel approach to a payment system: a decentralized cryptocurrency. Since then, cryptocurrencies have gained considerable adoption and interest. What sets cryptocurrencies apart from traditional payment systems is that they are administered by users in their network, without relying on any trusted parties. Cryptocurrencies are peer-to-peer systems, and thus there is no single point of failure.

Users in cryptocurrency networks maintain and secure a shared ledger of data, called the *blockchain*. The consensus protocol ensures that most of the network agrees with the state of the blockchain. *Miners* are designated nodes of the network that extend the blockchain. They verify the transactions, which are published by users to a public pool (called *mempool*), and include them as an ordered sequence in blocks which are then appended to the ledger. One prominent use for blockchains allowed by the mining mechanism is to enable *smart contracts*.

A smart contract is a program, which can be seen as enforcement of agreements between distrusting parties. The semantically correct execution of smart contracts is enforced by the consensus protocol. Smart contracts are identified by a unique address. To invoke a smart contract, a user sends a transaction to the contract address. Smart contracts are thus said to run on-chain, which means that: when a transaction to a smart contract is published, the contract code is run by a miner to include the transaction.

Ethereum is a prominent cryptocurrency with the support of Turing-complete smart contracts [25]. The code of Ethereum smart contracts is publicly available as Ethereum Virtual Machine (EVM) bytecode. After publication to the blockchain, smart contracts remain immutable[1]. Smart contracts in Ethereum are stateful programs: they have private storage in which their state is preserved, and they also implicitly contain the balance in the Ethereum currency, Ether. Most contracts in Ethereum are written in Solidity, a Java-like high-level language [15].

The expressiveness of smart contracts in Ethereum has led to a wide range of real-world applications. These include financial instruments (*e.g.*, sub-currencies, financial derivatives, savings wallets, wills) and self-enforcing or autonomous governance applications (*e.g.*, outsourced computation, decentralized gambling) [19].

[1] Although in Ethereum exists a *Proxy contract* pattern, which allows patching of a contract without changing its address, this pattern requires Ethereum assembly and is out of the scope of this thesis.

Nevertheless, Ethereum, much like the traditional financial system, is susceptible to certain vulnerabilities. One such vulnerability, which has carried over from traditional finance to Ethereum, is *"front-running"*. Front-running is an illegal practice, in which unethical brokers leverage knowledge of upcoming transactions to make trades and benefit their portfolios [18]. Similarly, since all information about future transactions is known to miners before these transactions are executed, miners can inspect them and manipulate their order for monetary gain [13, 19, 26].

Consider a simple smart contract, called `HashPuzzle`, in Listing 1.1. `HashPuzzle` is a game, in which users need to guess the preimage of the hashed challenge, with which the smart contract is created. The first user to find a solution gets all the funds of the smart contract as a reward.

Lines marked with ① and ② contain `require` function calls. The `require` function is a Solidity built-in mechanism to model preconditions. If a condition in a `require` block is not satisfied, the transaction is aborted and the state of a smart contract is reversed. Thus, the invocation with failed `require` has no effect on the smart contract.

Now, imagine an honest user finds and submits her solution, calling the `submitSolution` function. She thus publishes a transaction to the `HashPuzzle` smart contract. A malicious miner sees this transaction in the mempool and verifies that the solution is indeed correct. He then calls `submitSolution` with the same correct solution, publishing his transaction. Lastly, the miner creates a block in which his transaction is ordered before the honest user's. The malicious miner would thus be the one to redeem the whole reward, despite the honest user finding a solution first.

Listing 1.1 Smart contract written in Solidity. The function pair ⟨`submitSolution`, `submitSolution`⟩ is vulnerable against front-running attack.

```
contract HashPuzzle {
 uint public challenge;
 uint private balance;

 constructor(uint _challenge) public payable {
 challenge = _challenge;
 balance = msg.value;
 }

 function addFunds(uint _funds) public {
 require(balance + _funds ⩾ balance);                    ①
 balance += _funds;
 }

 function submitSolution(uint _solution) public {
 require(sha256(_solution) == challenge);                ②
 payable(msg.sender).transfer(balance);
 balance = 0;
 }
}
```

A similar attack can be performed by a regular user of the blockchain, which could bribe a miner into ordering his transaction first. A malicious user would then need to increase the miner rewards of his transaction (called *gas fees*). This would incentivize miners to include a tampered transaction over an honest one [13].

Maximal extractable value (MEV) [13] is a metric used in the Ethereum community to quantify the profits that are possible in all conceivable front-running attacks at the current point of blockchain execution. MEV is an incomplete metric since it captures only the direct monetary reward of miners. Yet, MEV showed that between 2020 and 2022, the losses in Ethereum due to front-running attacks surpassed 675 million USD [1].

There exist tools for identifying front-running vulnerability in smart contracts, such as [19], that use *Transaction Order Dependence (TOD)* [2] as their verification target. TOD describes a smart contract, which changes its state depending on the order of transactions at which it is invoked. The smart contracts with TOD are marked as vulnerable to front-running by such tools.

The TOD-based analysis suffers from two limitations. Firstly, it implies that safe smart contracts should practically commute, *i.e.* the result of any two invocations of a contract should not depend on the order. This however may lead to false-positive results for smart contracts, in which order-dependency may be intentional and not offer an exploitation revenue [26]. Moreover, studies reveal an inherent similarity between smart contracts and threads in concurrent environments. Making a smart contract order-independent is thus as hard as making an arbitrary data type concurrency-safe [23].

Secondly, tools that analyze TOD often use heuristics which are based on assumptions about the incentives of rational attackers. Such heuristic may be the direct monetary gain [24]. However, if the assumptions in heuristics are wrong, the TOD analysis is not sound anymore. For example, there is no direct monetary profit by front-running a voting. Yet, knowledge of changes to maximum purchasable votes enables adversaries to front-run other voters and gain an unfair advantage later.

Modern tools for front-running analysis tackle the former limitation by refining the verification target and improving their performance by pruning the functions without monetary exploitability [11, 26]. Yet, they still focus solely on monetary gains.

Motivation. TOD is an over-approximation for a front-running resistance. Instead of rendering a smart contract as vulnerable directly, it would be beneficial to first analyze under which conditions the invocations of a smart contract depend on transaction order. This provides a more fine-grained insight into the TOD and has a twofold goal.

On the one hand, generating conditions in which smart contracts are order-dependent allows us to analyze whether these conditions are known and intentional, and thus can help to disregard false-positives when searching for front-running vulnerabilities. On the other hand, an inverse of order-dependent conditions yields independence conditions (or *independence constraints*), which can be interpreted as states, in which a smart contract is invulnerable to the front-running attack. This induces a notion of conditional order-independence, which can be used as a guarantee of smart contracts' safety against front-running.

Contribution. The contributions of this thesis can be summarized as follows:

- We formally define the notion of independence constraints in the context of blockchain;
- We devise a static-analysis procedure, which models smart contracts and symbolically executes them to infer commutativity conditions of functions, and later the independence constraints;

- We implement this procedure in a fully-automated end-to-end toolchain utilizing:
  - Solidity's built-in Model Checker to encode Solidity smart contracts;
  - `Servois2` tool [12] to symbolically execute encoded smart contracts and infer independence constraints.

Chapter 2

Background

The generation of independence constraints is a static analysis procedure. This chapter introduces the theory, needed to discuss its steps. We first discuss the importance of logical modeling for program verification. Building upon that we explain how the smart contracts are modeled using Solidity SMT Encoder. We then introduce the formal definition of independence constraints, which can be thought of as commutativity conditions. Lastly, in Section 2.6 we discuss the algorithm for the automatic synthesis of commutativity conditions, implemented in the `Servois2` tool.

2.1 Solidity

In this subsection, we give a brief overview of Solidity, a language specifically designed to write smart contracts that run on the Ethereum Virtual Machine. We also discuss security concerns that arise when developing Solidity smart contracts.

Solidity is a statically-typed curly-braces language similar to Java. *Contracts* are the main elements of the Solidity source code and are similar to classes in object-oriented languages. Contract-level variables in Solidity, also called *state variables*, are persisted in the smart contracts storage, whereas local variables and function parameters have a temporary lifetime.

Solidity smart contracts are prone to errors that arise from the misconception of smart contract execution semantics [19]. In contrast to standard distributed applications, smart contracts are deployed to a public network and are immutable and irreversible after their deployment, making it impossible to patch them in case of a bug. Since smart contracts hold large amounts of virtual coins, sometimes hundreds of dollars apiece, the financial incentives are high enough to attract adversaries to exploit them. Proving the correctness of the smart contract prior to its deployment is thus a necessity for the developers.

The Solidity language includes features that aim to aid correct execution of smart contracts. If the execution of a Solidity contract is interrupted with special built-in functions, the state of the contract is reverted. One such function is the `require` statement, as shown in Listing 1.1. It is usually used to restrict smart contract invocations

to those with correct inputs. The second such function is the `assert`, used to validate the results of the correct execution of the contract itself. Both the `require` and `assert` functions help developers dynamically validate the execution of the smart contracts.

Nevertheless, not all bugs can be captured solely by `require` and `assert` statements without adding additional program logic. *Reentrancy bugs*, for example, require checking a multi-transactional execution of the smart contract. Similarly, TOD can only be identified by comparing two sequences of function executions. Additionally, `require` and `assert` capture errors dynamically, but don't provide any insight into the general correctness of a smart contract prior to the deployment, *i.e.* statically.

Formal verification of smart contracts is a promising method of statically ensuring the correctness of smart contracts since it provides a high level of guarantees [20]. The correctness of the formal verification results relies on the correctness of smart contract modeling and underlying reasoning techniques. Nevertheless, the demand for formal verification is so high, that multiple techniques are now natively supported by the Solidity compiler [6, 22].

2.2 Satisfiability Modulo Theories

Logical modeling and reasoning is the basis of formal program verification [5, 6, 17, 22]. The first step in this approach is to model (or encode) the problem as a formula in a predefined theory or combinations of theories, preserving the structure of the original problem. Theories define restrictions and interpretations of logic. Depending on the encoding, the **satisfiability** of the created formula can be interpreted as a solution to the problem (or absence of one).

The term **Satisfiability Modulo Theories** (SMT) is used to describe the satisfiability problem for an arbitrary theory or combinations thereof. The variables in the theories have sorts, *i.e.* types. The **SMT solvers** are highly optimized programs for solving the satisfiability problem. They implement decision procedures for a set of theories, where a decision procedure is an algorithm that terminates with a correct "Yes/No" answer.

The SMT Solvers needed a common language to create a set of standardized benchmarks and drive their adoption in the industry. The **SMT-LIB** standard [8] specifies such language. The standard also includes the formalization of theories and (sub-)logics, which are expressive enough to model programs in a straightforward manner. The SMT Solvers, which adhere to the SMT-LIB standard, implement decision procedures for the theories specified in the standard. In this thesis, we encode smart contracts in the SMT-LIB language and use Z3 [3] as the SMT solver to infer independence constraints.

2.3 Symbolic Execution

The method of logical reasoning we use is called *symbolic execution* [16]. Symbolic execution does not use exact values in the program and represents inputs as symbolic variables or expressions instead. One can intuitively think of it as solving a mathematical expression without using exact values and operating directly on variables.

The execution of the program is represented as a symbolic execution tree. Each node is associated with an executed statement, and the directed edges between nodes are transitions between statements. For each forking statement, such as *"if-then-else"*, two edges are leaving the node. The forking condition φ is labeled on the edges, where one edge is labeled with φ and another with $\neg\varphi$ depending on the control flow.

One can translate a symbolic path to the logical formula, called *path condition*, by accumulating the statements and forking conditions along the path. A path is called *infeasible* if its path condition is unsatisfiable.

The main goal of the symbolic execution is to verify that a given program is free from a given error. Intuitively, this would mean checking if any path of the program has an error. Since path conditions describe paths of the program, one can directly encode an error as a logical formula and verify if any of the path conditions imply it. The logical formula which expresses an error is called *verification target*. The modern procedures of symbolic execution are highly efficient and check paths in parallel using SMT Solvers [17].

2.4 Solidity SMT Encoder

The formal verification in Solidity is supported through its compilation process [6]. First, the Solidity compiler symbolically encodes smart contracts to SMT formulae. It then verifies them against predefined and user-specific verification targets using SMT solvers as the reasoning backend. We refer to the SMT generation procedure as **SMT encoding**, and the module of Solidity compiler which implements it as **SMT Encoder**.

The predefined targets include type constraint checks, which prevent overflow, underflow, array-out-of-bounds exceptions, and other local bugs. The user-specific verification targets are defined using `require` and `assert`. The SMT encoder assumes that conditions in `require` statements always hold, and tries to verify that conditions in `assert` statements are never reached.

We leverage the built-in SMT-based verification procedure of Solidity to extract the SMT encoding of smart contracts. There are currently two implementations of smart contracts encoders, also called reasoning engines: **Bounded Model Checker** (BMC) and **Constrained Horn Clauses** (CHC). Both of the reasoning engines traverse the abstract syntax tree (AST) of the smart contract and accumulate encodings of each Solidity statement inside. However, the encodings produced by the engines differ in their expressive power.

CHC [22] is a more powerful engine in terms of properties it can encode. It models a smart contract's control flow graph as a system of Horn clauses. The contract's lifecycle is represented as a cyclic transition system in which any public function can be visited non-deterministically. The smart contract's state after its invocation is preserved. This allows to model properties over an unbounded number of transactions. Internal function calls and loops are fully supported by the engine, and external function calls are assumed to be non-deterministic. Because of its expressive power, CHC also requires more computing resources.

BMC [6] follows the generic bounded model checking procedure [10] and encodes smart contract functions in isolation. The state after the function invocation is erased and loops are unrolled to a finite number of iterations. It inlines function calls if the implementation of the called functions is known, otherwise, it abstracts the call as an uninterpreted function with a non-deterministic result. BMC encodings are local, *i.e.* each block is encoded in

terms of variables that are known in its scope. Thus, BMC does not support the verification of multi-transactional properties. That makes BMC a lightweight engine, which can be used to prove local properties of functions and may yield false-positive results because of state erasure.

Although our procedure verifies the commutativity of a given smart contract, which is a multi-transactional property, we decided to choose BMC as the SMT encoder. The reason for that is the locality of BMC encodings. To embed the generated encoding to the input scheme of a commutativity synthesis algorithm discussed in Section 2.6, we needed a self-contained encoding of the smart contract's functions. Although each function of a smart contract would be encoded in its corresponding Horn clause by CHC, it would include a different clause as a predicate and would not be correctly parsed and used by the algorithm.

2.4.1 BMC Encoding on `HashPuzzle` Example

```
1  contract HashPuzzle {
2   uint public balance;
3   bytes32 public challenge;
4   ...
5
6   function submitSolution
7   (uint _solution) public {
8   require(sha256(_solution)
9   != challenge);
10  balance = 0;
11  }
12 }
```

```
1  (and
2   (= challenge_3_new challenge_3_1)
3   (= balance_5_new balance_5_2)
4   (= balance_5 balance_5_1)
5   (= balance_5_2 expr_39_1)
6   (= expr_39_1 expr_38_0)
7   (= expr_37_0 balance_5_1)
8   (= expr_38_0 0)
9   (not expr_32_3)
10  (not (= (= expr_30_3 expr_31_1)
11  expr_32_3))
12  (= expr_31_1 challenge_3_1)
13  (= expr_30_3 (select (sha256 crypto_0)
14  expr_29_length_pair_3))
15  (= expr_29_length_pair_3
16  (select (|abiencodepacked_pure| abi_0)
17  expr_28_1))
18  (= expr_28_1 _solution_22_0))
```

Figure 2.1: The `submitSolution` function of the `HashPuzzle` contract and its corresponding SMT-LIB encoding. For simplicity of the encoding, we omit the balance transfer and set the global variable `balance` to zero instead.

We will now briefly discuss the BMC encoding on the example in Figure 2.1. The figure shows the slightly altered `submitSolution` function of the `HashPuzzle` smart contract and its corresponding SMT encoding in the SMT-LIB language.

The local and state variables in the original program are assigned in SMT encoding using *Single Static Assignment* (SSA). In SSA each assignment and modification of the program variable introduces a new variable, which is assigned only once. For example, the state variable `balance` corresponds to the SSA-indexed variable `balance_5_1` on Line 7 at the beginning of the function, and introduces the new variable `balance_5_2` on Line 5 when the assignment `balance = 0` is reached.

Each variable in BMC encoding matches the `(<varname|"expr">_<id1>_<id2>)` pattern. The `id2` is the SSA index of the variable, which is increased after each assignment. The `id1` is a unique identifier of each symbolic variable which is used by BMC internally and which gets incremented when a new symbolic variable is introduced, that is when an assignment of a new variable or expression is modeled.

The variables that have the `expr` prefix in the SMT encoding are intermediate variables. These variables are assigned to the subexpressions which are traversed by BMC and are used to optimize the solving of the SMT encoding. One may observe that the number of intermediate variables exceeds the number of SSA-indexed program variables significantly.

The path conditions of symbolic execution are accumulated when the if-statements are traversed. When encoding the if-then-else statement of the form `if (`c`)` S_1 `else` S_2, the c and $\neg c$ are added to the path conditions of S_1 and S_2 respectively. This is encoded in the negated expression in Lines 10-11 in the SMT example, where `sha256(_solution) == challenge` is the path condition for the `balance = 0` assignment.

Path conditions also model conditional termination in `require` and `assert` calls. By semantics of Solidity, the statements `require(`r`)` and `assert(`r`)` terminate if r evaluates to $false$. The statements are thus encoded as implication $b \rightarrow r$ where b is the conjunction of the current path conditions, meaning that if the `require` or `assert` statement is reached, r should evaluate to $true$.

2.5 Independence Constraints

We now formally define the notion of the independence constraints. Let a smart contract $\mathcal{C} = (G, F)$ be a tuple of global variables G and methods F. We say that a state $\sigma : \textbf{Varname} \rightarrow V$ of a contract $\mathcal{C}$ is a mapping from the names of the global variables G of a contract to their values. A method $\alpha(\mathbf{u})/\mathbf{v} \in F$ is an atomic function with the name α, inputs $\mathbf{u}$ and outputs $\mathbf{v}$. We say that $\alpha_1; \beta_2$ is a sequence of invocations, where method α is invoked first and method β is invoked second. The sequence of invocations $\alpha_1; \alpha_2$ is defined analogously with only method α being invoked. We now define when two methods of a contract commute.

Definition 2.1 (State Commutativity of Methods)**.** Let $\alpha(\mathbf{u})/\mathbf{v}, \beta(\mathbf{m})/\mathbf{n} \in F$ be two methods of a contract $\mathcal{C} = (G, F)$. Consider two sequences of invocations in the same initial state σ of $\mathcal{C}$:

$$\alpha_1(\mathbf{u})/\mathbf{v}_1; \beta_2(\mathbf{m})/\mathbf{n}_2 \qquad \beta_1(\mathbf{m})/\mathbf{n}_1; \alpha_2(\mathbf{u})/\mathbf{v}_2$$

Let $\sigma_{\alpha.\beta}$ and $\sigma_{\beta.\alpha}$ be the state after the first and second invocations sequences respectively. The symbol "$\hat{=}$" denotes the equality relation of states. We say that α and β *commute in* σ, if $\sigma_{\alpha.\beta} \hat{=} \sigma_{\beta.\alpha}$, $\mathbf{v}_1 = \mathbf{v}_2$ and $\mathbf{n}_1 = \mathbf{n}_2$ holds. We denote this with $\textbf{Comm}(\alpha, \beta, \sigma)$ predicate.

Definition 2.2 (Commutativity of Methods)**.** We say that two methods $\alpha, \beta \in F$ of a contract $\mathcal{C} = (G, F)$ *commute*, if α and β commute in all initial states σ.

Since commutativity is a property that makes smart contracts resistant to front-running, we can define restrictions on the initial states of the smart contract to enforce commutativity for methods that don't commute. We denote a boolean formula ψ, which holds in a state σ, as $\psi(\sigma)$.

Definition 2.3 (Independence Constraints)**.** Let $\alpha(\mathbf{u})/\mathbf{v}, \beta(\mathbf{m})/\mathbf{n} \in F$ be two methods of a contract $\mathcal{C} = (G, F)$. The *independence constraints* $I_{\alpha,\beta}$ is a set of conditions expressed in terms of global variables G and input variables $\mathbf{m}$ and $\mathbf{n}$, for which holds:

$$\forall \psi \in I_{\alpha,\beta}.\ \forall \sigma.\ \psi(\sigma) \implies \textbf{Comm}(\alpha, \beta, \sigma)$$

In other words, independence constraints $I_{\alpha,\beta}$ restrict the initial states of the smart contract $\mathcal{C}$ to those in which α and β commute.

2.5.1 Expected Independence Constraints of `HashPuzzle`

Recall the `submitFunction` method in Figure 2.1. In the notation defined above, `submitSolution`($\{$_solution$\}$)/$\emptyset$ is a method of a contract HashPuzzle $= (G = \{balance\}, F = \{submitSolution\})$. For brevity, we denote the name of the `submitSolution` method as sS. The independence constraints $I_{sS,sS}$ are then:

$$I_{sS,sS} = \left\{ \begin{array}{c} (1)\ \texttt{balance} = 0; \\ (2)\ \texttt{solution}_{sS_1} \neq \texttt{solution}_{sS_2}; \\ (3)\ \texttt{solution}_{sS_1} \neq sha256(\texttt{challenge}); \\ (4)\ \texttt{solution}_{sS_2} \neq sha256(\texttt{challenge}) \end{array} \right\}$$

The first condition is an independence constraint since if the `HashPuzzle` does not hold any balance, there is no difference in who solves the puzzle first – the reward stays zero.

The second condition implies commutativity since there is only one correct solution.

The last two conditions are symmetrical and also suggest that if at least one of the users published the wrong solution, at most one of them will succeed.

The state, in which both of the users could get a reward and which would make invocations of `submitSolution` not commutative, is thus restricted by these independence constraints.

2.6 Inference of Commutativity Conditions

The abstraction-refinement technique of [7] provides an algorithm, which infers commutativity conditions for logically encoded abstract data type (ADT). Such ADT $\mathcal{O}$ has the following scheme:

$$\mathcal{O} = \left\{ \begin{array}{ll} \text{state} : (Var, Type)\ \text{list}; & \text{methods} : \text{Method list}; \\ eq : \text{state} \times \text{state}; & \text{spec} : \text{Method} \to (Pre, Post); \end{array} \right\}$$

$\mathcal{O}$ contains a state as a list of typed variables, an equality relation on states, methods signatures, and logical specification for each method. A logical specification of methods is defined as methods pre- and post-conditions logical formulae.

The algorithm of [7] solves the *commutativity condition synthesis* problem, given an ADT $\mathcal{O}$ and two methods $\alpha(\mathbf{u})/\mathbf{v}$, $\beta(\mathbf{m})/\mathbf{n}$ of it. The commutativity condition φ of two methods α and β is a formula expressed in terms of the object's $\mathcal{O}$ state and the inputs $\mathbf{u}$ and $\mathbf{m}$, such that α and β commute in every state satisfying φ. The

commutativity is defined according to the state equality relation of $\mathcal{O}$. The problem is then to generate such formula φ.

Intuitively, the algorithm computes subregions within the state space, in which α and β commute. The state subregions are described as logical formulae with state and input variables. The generated commutativity condition is the disjunction of all such formulae.

To find such subregions, the algorithm partitions the state space using the relations from the pre- and postcondition specifications of α and β. Such relations are called **predicates**. The arguments which form the predicates are called **terms**. We refer to predicates with terms as **formed predicates**.

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

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

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

The actual algorithm is shown in Figure 2.2. We denote that α commutes with β as $\alpha \bowtie \beta$. A subregion H is described as a (possibly infinite) conjunction of predicates that hold in H. The resulting commutativity condition φ is a disjunction of such subregions H. The algorithm initializes with the trivial commutativity condition $\varphi =$ **false**. `Refine` performs a symbolic search through the current state space H, starting from the entire state space $H =$ **true**. The (possibly infinite) set of formed predicates that the algorithm uses is provided in $\mathcal{P}$.

On each step, the algorithm checks if α and β always commute in the current region H. Since region H is expressed as a boolean formula, it has to verify that $H \implies \alpha \bowtie \beta$ is a tautology, *i.e.* is always true. If so, H can be added to the growing commutativity condition φ. If not, `Valid` returns counterexamples χ_c of commutativity in H.

The counterexamples χ_c are used to choose the next predicate $p \in \mathcal{P}$, which will determine the partition of H. The exact `Choose` method is not important, as shown below in Theorem 2.3. Most importantly, the counterexamples navigate the search, as they show which conditions can be constrained. Lastly, the search continues in both partitions $H \wedge p$ and $H \wedge \neg p$. At each step, the algorithm chooses one predicate at a time.

For example, the inferred commutativity condition φ of a method pair $\alpha = \beta =$ `submitSolution` in Section 2.5.1 would be:

$$\varphi = \bigvee \begin{cases} \texttt{balance} = 0 \\ (\texttt{balance} \neq 0) \wedge (\texttt{solution}_1 \neq \texttt{solution}_2) \\ (\texttt{balance} \neq 0) \wedge (\texttt{solution}_1 = \texttt{solution}_2) \wedge (\texttt{solution}_1 \neq sha(...)) \\ (\texttt{balance} \neq 0) \wedge (\texttt{solution}_1 = \texttt{solution}_2) \wedge (\texttt{solution}_1 = sha(...)) \wedge (\texttt{solution}_2 \neq sha(...)) \end{cases} \tag{2.1}$$

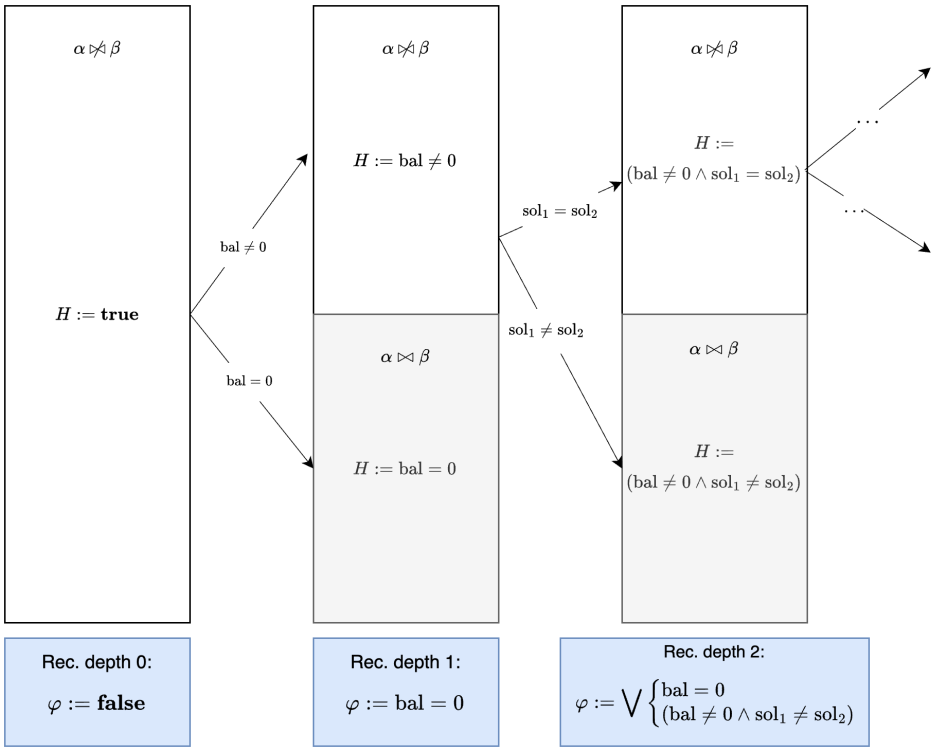

Figure 2.3: Visualization of the inference of commutativity conditions for $\alpha = \beta = \texttt{submitSolution}$. White rectangles show current regions of the state space with their logical representation (in the center) and commutativity of methods in them (at the top). Blue rectangles contain the commutativity condition inferred at the given recursion depth of the algorithm.

The inference steps are schematically shown in Figure 2.3. The predicate in this example would be equality relation "=" since it is the only relation used in the `submitSolution` method. The terms of this predicate would be expressions of arbitrary type, for which equality relation is defined. Here, for example, the algorithm uses `balance` (`bal` for short), $\texttt{solution}_1$ ($\texttt{sol}_1$), $\texttt{solution}_2$ ($\texttt{sol}_2$) and 0 as terms.

In Figure 2.3 both formed predicates $\texttt{bal} \neq 0$ and $\texttt{sol}_1 \neq \texttt{sol}_2$ directly led to a region with commutativity. Generally, this is not the case, and thus algorithm continues the search in both partitions. The complexity of the algorithm is exponential in a number of predicates.

2.6.1 Soundness, Completeness, and Termination

The commutativity condition synthesis algorithm of [7] provides the following guarantees:

Theorem 2.1 (Soundness). *For each `Refine` iteration:* $\varphi \implies \alpha \bowtie \beta$.

Theorem 2.2 (Completeness). *If* `Refine` *terminates with the commutativity condition* φ*, then* $\forall \psi \in \neg\varphi.\ \psi \implies \alpha \not\bowtie \beta$*. Here,* $\neg\varphi$ *is the CNF formula and* ψ *are clauses.*

Theorem 2.3 (Termination Conditions). `Refine` *terminates if:*

1. ***(expressiveness)*** *The state space is partitionable into a finite set of regions each described by a finite conjunction of predicates* ψ_i*, such that either* $\psi_i \implies \alpha \bowtie \beta$ *or* $\psi_i \implies \alpha \not\bowtie \beta$*;*
2. ***(fairness)*** *For every* $p \in \mathcal{P}$*,* `Choose` *eventually picks* p*.*

Remark 2.1. The proofs of the theorems are the subject of the 5th Section of [7].

From Theorem 2.1 follows that for the given ADT $\mathcal{O}$, the algorithm can be interrupted at any point and the commutativity condition found so far will be an over-approximation of the real commutativity condition. An over-approximation here is a more restrictive formula φ', such that $\varphi' \implies \varphi$ holds.

Chapter 3

Key ideas

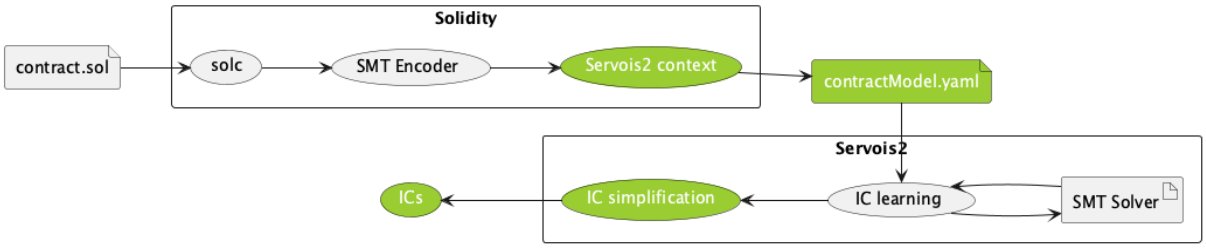

Figure 3.1: IC generation procedure. The contribution of the thesis is marked green.

Having laid the theoretical background, we can now summarize our *independence constraints generation* procedure:

> For a given smart contract, the procedure extracts the SMT encoding of the contract and embeds it to the ADT defined in Section 2.6. The ADT serves as an input to the commutativity conditions synthesis algorithm, implemented in the `Servois2` tool, with which we generate the commutativity condition φ for each pair of methods of the contract. Lastly, the commutativity conditions are transformed into sets of independence constraints.

The modular design of the procedure is shown in Figure 3.1. The source code of a smart contract is processed by the **Solidity** module, which generates a YAML representation of the smart contracts ADT. The representation is then processed by **Servois2** to generate independence constraints. In this chapter we discuss the functionalities of both modules, the embedding of smart contracts to ADTs, and the transformation of the commutativity conditions to independence constraints.

3.1 Extraction and Embedding of Smart Contracts Encoding

The purpose of the **Solidity** module is to extract the SMT encoding from a smart contracts source code. During the compilation of a smart contract, the Solidity compiler invokes the model checker, which encodes the smart contract using built-in SMT engines, in our case the Bounded Model Checker.

BMC gets the abstract syntax tree of the contract and traverses all its nodes to generate the SMT model. As described in Section 2.4, the BMC encodes functions as self-contained SMT formulas. Each Solidity statement is mapped to its SMT equivalent, types of Solidity are mapped to corresponding SMT sorts, and control flow and branching are encoded by accumulating branch conditions and restricting the encoded execution with them. We extract these encodings when the contracts and functions nodes of AST are traversed and accumulate them in the `Servois2 context`.

The `Servois2 context` handles the embedding of the smart contracts SMT encoding to the abstract data type defined in Section 2.6. The ADT is the input to the commutativity condition synthesis algorithm. The attributes of the `Servois2 context` class match the ADT attributes and the SMT encodings are grouped according to the ADT's scheme. `Servois2 context` lifecycle is managed by the Solidity model checker.

The embedding, which `Servois2 context` performs, is formally defined as follows:

Definition 3.1 (Embedding of a smart contract to ADT)**.** For a given smart contract $\mathcal{C} = (G, F)$ (as defined in Section 2.5), the `Servois2 context` creates an embedding to the ADT $\mathcal{O}$ where:

- $\mathcal{O}$.state is the list of all the smart contracts state variables $v \in G$.
- $\mathcal{O}$.eq: both states σ_1 and σ_2 are equal provided that they agree on the values of all the variables in $\mathcal{O}$.state.
- $\mathcal{O}$.methods are the signatures of the methods $m(\mathbf{u})/\mathbf{v} \in F$ of the smart contract.
- $\mathcal{O}$.spec associates the signatures of the methods with **true** preconditions and the SMT encoding of BMC as post-conditions, *i.e.*: $m_{\mathcal{O}} \mapsto (\textbf{true}, enc(m))$, where $m \in F$. We treat the preconditions as trivial since in our model any function of a smart contract can be invoked at any point in time.

As soon as the AST of a smart contract is completely traversed, the `Servois2 context` dumps its state to the YAML representation, which models the ADT $\mathcal{O}$ and serves as an input to `Servois2`. This procedure is repeated to encode multiple smart contracts within a single Solidity file. In that case, for each contract, a YAML representation is created.

3.1.1 ADT Representation of `HashPuzzle`

As an example, let's consider the generated ADT for the `HashPuzzle` smart contract. In the example we use `<submitSolution encoding>` as the shorthand for the SMT encoding, shown in Figure 2.1.

The `name` of the ADT matches the name of the given smart contract. In the `preamble`, we dump the type and variable declarations which are needed for the methods SMT encoding. The `state` is the list of typed state variables, as defined in the ADT. The value of `methods` is a list of methods representations, which contains both their signatures and logical specification (precondition as `requires` and postcondition as `ensures` blocks). Notice that methods also contain `terms`, which are used to form the `predicates`, as long as their types match. We omit the `intermediate` mapping for now. Lastly, the `states_equal.definition` contains the state equality relation.

Listing 3.1 The ADT representation of the HashPuzzle smart contract

```
name: HashPuzzle
preamble: |
 <datatypes and variables declarations>
state:
 - name: challenge_3
    type: Int
 - name: balance_5
    type: Int
states_equal:
  definition: |
 (and (and true (= challenge_3_new challenge_3)) (= balance_5_new balance_5))
methods:
 - name: submitSolution
    args:
 - name: _solution_22_0
        type: Int
    return: []
    requires: |
 true
    ensures:  |
 <submitSolution encoding>
    terms:
        Int: <terms of type Int>
predicates:
 - name: =
      type: [Int, Int]
intermediate:
    <intermediate variables assignments>
```

3.2 Inference of Independence Constraints

We invoke the `Servois2` tool when the ADT of a smart contract is created and dumped to the YAML model. `Servois2` is an OCaml library/module [12], which implements the commutativity synthesis algorithm of [7]. Additionally, it defines a distilled version of SMT-LIB language in order to parse SMT-LIB expressions and represent them as OCaml algebraic data types.

Even though `Servois2` supports the automatic extraction of predicates and terms from ADT methods specifications, our procedure relies only on embedded terms and predicates, which are extracted by the Solidity module. We discuss the limitation that drove this decision in the next chapter.

Our procedure invokes the commutativity synthesis algorithm for each combination and reflexive pair of methods. For each pair of methods, we thus get a DNF formula φ generated, as discussed in Section 2.6. To transform a formula to the set of independence constraints, we treat each term in the formula (here meaning the conjuncts, which are connected by disjunction) as separate independence constraints.

For example, the formula in Equation 2.1 would be transformed to:

$$I_{\texttt{sS,sS}} = \left\{ \begin{array}{c} \texttt{balance} = 0; \\ (\texttt{balance} \neq 0) \wedge (\texttt{solution}_1 \neq \texttt{solution}_2); \\ (\texttt{balance} \neq 0) \wedge (\texttt{solution}_1 = \texttt{solution}_2) \wedge (\texttt{solution}_1 \neq sha256(...)); \\ (\texttt{balance} \neq 0) \wedge (\texttt{solution}_1 = \texttt{solution}_2) \wedge (\texttt{solution}_1 = sha256(...)) \wedge (\texttt{solution}_2 \neq sha256(...)) \end{array} \right\} \tag{3.1}$$

3.3 Simplification of Independence Constraints

We now shortly touch upon one challenge that we encountered when trying to directly generate independence constraints from methods SMT encodings. As discussed in Section 2.4.1, the SMT encodings that BMC produces contain *intermediate* variables.

We extract the intermediate variables as valid terms since they are part of the methods logical. The commutativity synthesis algorithm then uses them to form predicates, as discussed in Section 2.6. The resulting commutativity condition formula thus may contain the predicates over intermediate variables.

Such a formula is valid according to the method's logical encoding but breaks the guarantee of the algorithm to express the commutativity condition in terms of state and input variables of methods. To express the commutativity condition without intermediate variables, we use an important invariant of BMC:

Definition 3.2 (Invariant of intermediate expressions assignment)**.** Let `expr_`m`_`n be an intermediate expression with the unique internal index m and the SSA index of the expression n. It then holds, that `expr_`m`_`n is assigned the formula over:

- local variables;
- state variables;
- other intermediate expressions `expr_`i`_`j, where $m > i$.

To replace the intermediate variables from the commutativity conditions, we replace all occurrences of them with their assignments. The intermediate variables nested in the assignments are also similarly replaced.

Definition 3.2 guarantees that *(a)* intermediate variables in assignments of others have a strictly lower unique index, and *(b)* the intermediate expression with the lowest unique index is not assigned to other intermediate expressions. Thus, this recursive replacement of each expression is bound to terminate. This idea suggests a recursive algorithm, which we discuss in detail in Section 4.3.

Chapter 4

Implementation

In this chapter, we discuss the technical details of our procedure. We begin by explaining how we extract the smart contracts encoding. We then discuss the limitation and the solution of the predicate extraction in the generated smart contracts embedding. Lastly, we touch on the algorithm for the elimination of intermediate expressions and discuss its properties.

4.1 Extraction of a Smart Contract's Encoding

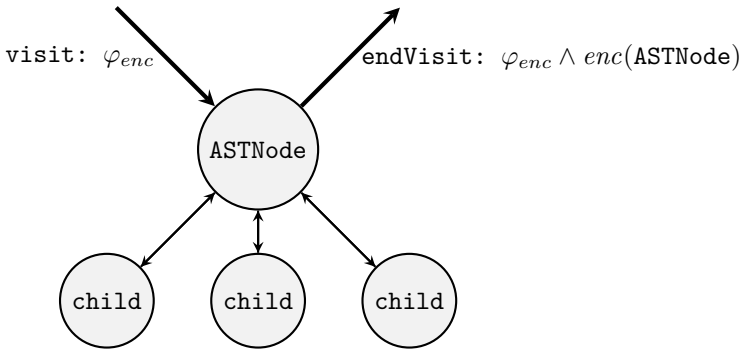

Figure 4.1: The visualization for the `Visitor` pattern, implemented in the Solidity SMT Encoder. Before visiting the `ASTNode`, the SMT Encoder has some smart contracts encoding φ_{enc}. After visiting the `ASTNode`, its encoding is added to φ_{enc}.

To verify a smart contract using an SMT solver, the Solidity compiler implements the SMT encoding. The class responsible for handling the encoding is `SMTEncoder`, located in the `libsolidity/formal` package of the Solidity source. The procedure in `SMTEncoder.cpp` follows a `Visitor` pattern, with which an abstract syntax tree (AST) of a compiled smart contract is folded to the SMT formula. Each type of AST node has the corresponding `visit` and `endVisit` hooks, which are called during the AST traversal. The `Visitor` pattern is schematically shown in Figure 4.1.

The `BMC` class inherits the `SMTEncoder` and also implements the AST traversal algorithm, additionally tweaking some encodings. We leverage the SMT encoding of `BMC` and focus only on two types of AST nodes: `FunctionDefinition`

and `ContractDefinition`. These nodes represent the defined (root) functions and contracts in the source code respectively.

The extracted encoding is accumulated in the `Servois2 context`. The name, state variables, and preamble of a contract are extracted when the contract node is completely visited. At this moment, all SSA-indexed symbolic variables and intermediate expressions are declared, and we can safely extract their declarations into the preamble. This is a vital step since the preamble is always included in queries to the SMT solver, and missing a declaration of a variable would mean breaking the procedure.

The methods encodings are added to the `Servois2 context` when a root function (*i.e.*, a method) is visited. We save the whole function encoding to the post-condition block, as described in Definition 3.1. The name, input, and output parameters of the encoded method are extracted using the `EncodingContext`, an inner helper class in which smart contract attributes are saved. Lastly, the terms and predicates are extracted from the encoding of the method, as explained in Section 4.2.

The state equality relation $\widehat{=}$ is generated as a formula over the extracted state variables of a smart contract $\mathcal{C} = (G, F)$:

$$\varphi_{\widehat{=}} := \bigwedge_{v \in G} v_1 = v_2 \tag{4.1}$$

Here, for each pair of methods $\alpha, \beta \in F$ and state variable $v \in G$, v_1 and v_2 denote the state variables $v_{\alpha.\beta}$ and $v_{\beta.\alpha}$ respectively.

4.1.1 Binding SSA-Indexed Variables to ADT State Variables

As explained in Definition 3.1, the state equality relation is expressed as the equality of ADT state variables in both sequences of invocations. However, in the logical encoding of methods multiple SSA-indexed variables represent the same state variable. For example, in Figure 2.1 the `balance_5_1` and `balance_5_2` both correspond to the `balance_5` state variable[1].

Recall that the SSA index corresponds to the number of modifications a given program variable had before the assignment. The symbolic variables with the minimal and maximal SSA indices thus correspond to the initial and final assignments of a program variable respectively.

To link the state variables in the ADT representation of a smart contract $\mathcal{C} = (G, F)$ with the SSA-indexed state variables in a method $\alpha \in F$, we modify a method encoding φ_α by binding the initial and final state variables to the corresponding SSA-indexed variables:

$$\varphi'_\alpha := \varphi_\alpha \wedge \bigwedge_{v \in G} v = v_\alpha^{\min} \wedge v_{new} = v_\alpha^{\max} \tag{4.2}$$

In the `Servois2` representation, v_{new} denotes the final modification of a state variable $v \in G$. Identifiers $v_\alpha^{\min}$ and $v_\alpha^{\max}$ represent the SSA-indexed variables of v in the encoding of $\alpha \in F$ with minimal and maximal SSA indices

[1] Currently, unique indices of the symbolic variable are not trimmed in the names of the state variables. Although it does not affect the procedure, it could be changed in the future to improve readability.

respectively. We only bind the $v_{\alpha}^{\max}$ with v_{new}, if the variable v is not modified in α, that is minimal and maximal SSA indices are equal.

4.2 Extraction of Terms and Predicates

The algorithm in Figure 2.2 requires predicates to refine the computed commutativity condition and terms to form the predicates. The `Servois2` tool implements the algorithm to automatically extract terms and predicates from method logical encoding. To infer types of terms and predicates, however, the extraction algorithm requires that each variable in the encoding of the method exists in the ADT specification as a typed binding. That is, each variable in the method encoding should either be a state variable or the parameter of the method.

As discussed in Section 2.4.1, the BMC encoding of smart contracts functions creates multiple intermediate variables. These variables are neither state variables nor parameters of the functions, and their declarations are introduced only in the `preamble`. In the current state of the `Servois2` tool, the `preamble` block is not parsed and is treated as a multiline string. Thus, the declarations of intermediate variables are not included in the ADT specification and the automatic extraction fails to infer their types.

To overcome this limitation, we devised two different approaches: we could either extend the logic of `Servois2` to parse `preamble`, or extract the terms and predicates in the **Solidity** module of the procedure. We chose the latter alternative since it was easier to implement and enabled manual debugging of the results by tweaking the terms and predicates in the YAML representation directly. In the following, we discuss our extraction algorithm.

In Solidity, the `smtutil::Expression` type is used as an internal representation for SMT formulae. The variables of the `smtutil::Expression` type are trees, in which every node is labeled with the SMT-LIB keyword it represents and its SMT sort. The leaves of the tree are typed variables. The inner nodes are functions, operators, or relation symbols with child nodes being their arguments.

Given a method encoding of the `smtlib::Expression` type, we fold it as a tree and accumulate terms and predicates. We use the following conditions to distinguish terms and predicates from arbitrary nodes:

Definition 4.1 (Terms as nodes)**.** The nodes with the SMT sort other than `Bool` are terms.

Definition 4.2 (Predicates as nodes)**.** Predicates are the nodes with the SMT sort `Bool` of arity 2, the arguments of which are terms.

Remark 4.1 (Predicates arity). The arity of predicates is based on the `Servois2` implementation, which currently supports only predicates of arity 2.

For example, the tree representation of the SMT-LIB formula `(and (= bal_1 (+ bal_0 1)) (< bal_0 bal_1))` is shown in Figure 4.2. Here, the terms are `1`, `bal_0`, `bal_1`, and `(+ bal_0 bal_1)` all of the sort `Int`. The predicates are "`+`" and "`<`" of sort `Bool` with arguments of sort `Int`.

To form predicates from terms, `Servois2` would create expressions filling the predicates with all possible terms as arguments. In the example above, `(= (+ bal_0 bal_1) 1)` would be a formed predicate, although it is not a sub-expression of the original formula.

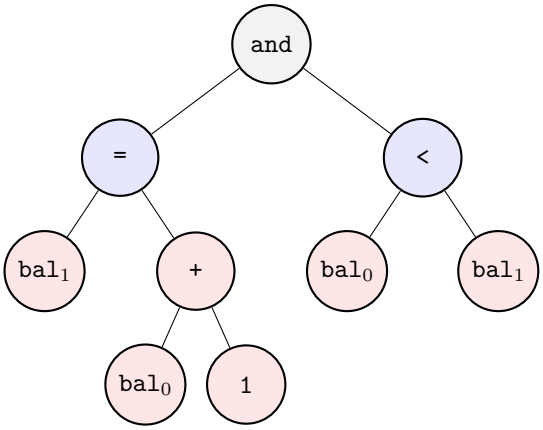

Figure 4.2: The `smtutil::Expression` tree representation of the SMT-LIB formula `(and (= bal_1 (+ bal_0 1)) (< bal_0 bal_1))`. The blue and red nodes are the predicates and terms respectively.

4.3 Simplification of Expressions over Intermediate Variables

To simplify expressions over intermediate variables in the independence constraints of a method, we needed a mapping from an intermediate variable to its symbolic assignment without other intermediate variables nested in it. We call these assignments *clean*. Using such a mapping, we could eliminate intermediate variables from a smart contract's encoding by replacing them with their clean assignments. In this section, we discuss how such a mapping is created.

First, we extract the initial assignments of the intermediate variables from the methods encoding in the **Solidity** module. As discussed in the previous section, it would mean simply folding over the methods encoding tree. We use the following condition on `smtutil::Expression` nodes to distinguish these assignments:

Definition 4.3 (Assignments of intermediate variables as nodes)**.** For all nodes labeled with equality symbol "`=`", the first argument of which are nodes labeled with the prefix "`expr`", are assignments of intermediate variables.

The first and second arguments of an assignment node in Definition 4.3 are the intermediate variable name and its assignment respectively.

We accumulate these values in a mapping from variable names to their corresponding assignments. The initial assignments still include nested intermediate variables. In BMC, each intermediate variable is unique and is assigned once over the whole smart contract's encoding. We thus merge the accumulated assignment mappings of each method to a contract-level mapping. When generating the YAML representation, we include it under the `intermediate` block.

For example, the `intermediate` mapping in the `HashPuzzle` YAML representation is shown in Listing 4.1.

The assignments in the `intermediate` mapping are then read and parsed by `Servois2`. We modified the parsing of SMT-LIB expressions to distinguish intermediate variables from arbitrary identifiers. `Servois2` includes a rich variety of OCaml algebraic data types which represent a subset of SMT-LIB grammar definitions. The `var` type, shown in Listing 4.2, is used for SMT-LIB variables. We added the `Intermediate` constructor for intermediate variables, which is applied by the parser on variables with the "`expr`" prefix.

Listing 4.1 The mapping of intermediate variables assignments in the HashPuzzle YAML representation.

```yaml
intermediate:
  expr_51_1: expr_50_0
  expr_49_0: balance_5_1
  expr_50_0: 0
  expr_47_1: expr_46_1
  expr_41_0: winner_7_1
  expr_46_1: expr_45_1
  expr_45_1: (|msg.sender| tx_0)
  expr_36_3: (not (= expr_34_1 expr_35_1))
  expr_35_1: challenge_3_1
  expr_34_1: _solution_31_0
```

Listing 4.2 The Servois2 data type used for SMT-LIB variables.

```
type var =
 | Intermediate of string (* intermediate variables *)
 | ...
```

Lastly, we clean the initial assignments. A simplified elimination algorithm is shown in Listing 4.3 (written in OCaml). Here, we represent a mapping from intermediate variables names to their clean assignments as a function `lookup : string → exp`, where the `exp` type models the SMT-LIB expressions in `Servois2`. In the actual implementation, the mapping is an OCaml `Hashtable`.

Listing 4.3 The simplified solution for the intermediate variables elimination.

```
let rec eliminate (lookup : string → exp) (assignment : exp) : exp =
 match assignment with
 | EVar (Intermediate s) → lookup s                                          ①
 | _ → (* recurse `eliminate` on each sub-expression of `assignment` *)

let modify_lookup (lookup : string → exp) ((var, assignment): string * exp) =
 let clean_assignment = eliminate lookup assignment in                       ②
 Map.add lookup var clean_assignment                                         ③

let create_clean_intermediate_lookup (assignments : (string * exp) list) : (string → exp) =
 let sorted_assignments = List.sort unique_id assignments in                 ④
 let lookup = Map.empty_map in
 List.iter (modify_lookup lookup) sorted_assignments;                        ⑤
 lookup
```

The solution includes three functions:

- `eliminate` replaces intermediate variables nested in the `assignment` expression using `lookup` mapping of clean assignments (①).
- `modify_lookup` cleans the `assignment` of the `var` variable using the `lookup` mapping (②). It then adds the cleaned assignment to the `lookup` in-place (③). We assume that the nested intermediate variables in `assignment` are already included in the `lookup`.
- `create_clean_intermediate_lookup` creates the mapping of clean assignments from the initial assignments, represented here as a list of variable-assignment pairs. It sorts the list using unique indices of variables as a key (④). Sorting is necessary for the invariant in Definition 3.2 to hold, which satisfies the assumption

of `modify_lookup`. It then initializes an empty mapping and populates it by iterating over sorted initial assignments (⑤).

As a result, by calling the `create_clean_intermediate_lookup` function with initial assignments from the `intermediate` YAML representation, one gets a mapping from intermediate variables' names to their clean assignments. This mapping can be then used to eliminate intermediate variables from other SMT-LIB expressions in the smart contract encoding.

Because of the invariant in Definition 3.2, we clean each intermediate variable and add it to the mapping only once. Assuming that the size of the intermediate variable assignments and the mapping access are constant, the complexity of the elimination is determined by the sorting algorithm. With `MergeSort`, the complexity of the elimination algorithm would be $O(n \log n)$, where n is the number of intermediate variables. However, since unique indices of the variables are small natural numbers which can be bounded by some large enough constant, using `BucketSort` one can achieve linear complexity on average.

A side-effect of intermediate variables elimination is the optimization of the commutativity condition refinement algorithm (Figure 2.2). By eliminating the intermediate variables, we reduce the number of terms used by `Servois2`. The number of terms is proportional to the number of predicates, and thus the overall complexity is also reduced.

Chapter 5

Related Work

Understanding front-running attacks. In the past decade, extensive research has been conducted to understand and defend against front-running attacks in smart contracts. [14], [9], and [27] give a systematic overview of attacks in Decentralized Finance (DeFi) platforms, including front-running attacks. [14] additionally propose defense mechanisms on different blockchain layers, including smart contracts, to prevent front-running attacks. [13] research the effects of *gas auctions*, a phenomenon that emerges when users try to bribe miners and front-run others. [13] propose *Maximum Extractable Value (MEV)* as a metric to measure profits an attacker can get by manipulating the order of transactions. [19] formalize the notion of *Transaction Order Dependence* in the blockchain context.

[23] first consider smart contracts as threads in the concurrent environment and explore the root cause of a variety of vulnerabilities, including front-running, from the concurrency perspective. [4] introduce the notion of conditional independence for two concurrently executed transitions which holds under certain conditions, called *independence constraints*. The intuition behind our definition of independence constraints unfolds if one considers a smart contract as a system with concurrent transitions, as [23] suggest. [4] also introduce an SMT-based approach to synthesize independence constraints automatically, similar to the `Servois2` tool. However, the synthesis is defined for a limited number of program statements and is not generalized for an arbitrary encoding of a program.

Identifying front-running vulnerability. In our research we focused on static analysis techniques since dynamic analysis does not provide high enough guarantees. [20] discuss opportunities for formal methods in smart contracts and give an overview of existing verification tools, including those that identify front-running vulnerabilities. The tools `Oyente` [19] and `Securify` [24] symbolically encode smart contracts EVM bytecode and search for front-running vulnerability. `Oyente` analyzes differences in Ether flow depending on the functions' order and treats such differences as front-running vulnerability. `Securify` analyzes a pattern similar to MEV to detect front-running. Both these tools focus solely on the direct monetary gains of an adversary and thus are prone to both false-negative and false-positive results.

[11] devise the `SAILFISH` tool which uses a storage dependency graph (SDG) to model the execution flow as if it was subverted by an attacker. `SAILFISH` prunes functions without hazardous access using the SDG and then

symbolically executes a smart contract. [26] implement a similar procedure in `Nyx` and refine the verification target for detecting front-running vulnerability based on empirical studies.

To our knowledge, independence constraints as a means for front-running resistance is a novel idea and has not been explored in other research yet.

Chapter 6

Conclusion & Future Work

In conclusion, we formalize a notion of independence constraints that are conditions, under which a smart contract is resistant to front-running attacks. We devise a static-analysis procedure to generate independence constraints, in which we leverage the SMT encoding capabilities of the Solidity compiler and use `Servois2` to symbolically execute an encoded smart contract. Our procedure is modular and, since `Servois2` is used as a standalone tool, can be included in the Solidity model checking process.

Two future work directions may improve the procedure:

- The current implementation of our procedure manually extracts terms and predicates from the encoding of the methods. Since the algorithm of `Servois2` is exponential in the number of predicates, the optimized extraction will improve the scalability of the procedure. Such optimization can be achieved by using automatic terms and predicates extraction implemented in `Servois2`. To enable this, one would need to additionally parse the `preamble` block as a sequence of SMT-LIB expressions and include the parsed variables to the `Servois2` smart contracts specification, as discussed in Section 4.2.
- Currently, we do not eliminate the intermediate variables from the methods encodings completely, and only clean the terms of methods. Eliminating the intermediate variables from the methods encodings included simplifying control-flow `ite` expressions, which could break the correctness of encodings. We evaluated the simplification procedures implemented in Z3 [3] and did not achieve the desired results. Yet, the simplification worked for simple smart contracts, and one could imagine a more complex algorithm in `Servois2` which would leverage Z3 simplification capabilities.

References

[1] MEV Explore.

[2] SWC-114 - Smart Contract Weakness Classification (SWC).

[3] Z3. *Microsoft Research.*

[4] Albert, E. et al. 2018. Constrained Dynamic Partial Order Reduction. *Computer Aided Verification* (Cham, 2018), 392–410.

[5] Albert, E. et al. 2020. Taming callbacks for smart contract modularity. *Proc. ACM Program. Lang.* 4, OOPSLA (Nov. 2020), 209:1–209:30. DOI:https://doi.org/10.1145/3428277.

[6] Alt, L. and Reitwiessner, C. 2018. SMT-Based Verification of Solidity Smart Contracts. *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice.* T. Margaria and B. Steffen, eds. Springer International Publishing. 376–388.

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

[8] Barrett, C. et al. 2010. The smt-lib standard: Version 2.0. *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)* (2010), 14.

[9] Baum, C. et al. 2021. SoK: Mitigation of Front-running in Decentralized Finance.

[10] Biere, A. et al. 1999. Symbolic Model Checking without BDDs. *Tools and Algorithms for the Construction and Analysis of Systems.* G. Goos et al., eds. Springer Berlin Heidelberg. 193–207.

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

[12] Chen, A. et al. 2022. Veracity: Declarative multicore programming with commutativity. *Veracity: Declarative Multicore Programming with Commutativity.* 6, OOPSLA2 (Oct. 2022), 186:1726–186:1756. DOI:https://doi.org/10.1145/3563349.

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

[14] Eskandari, S. et al. 2020. SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain. *Financial Cryptography and Data Security* (Cham, 2020), 170–189.

[15] etherscan.io Contract Statistics | Etherscan. *Ethereum (ETH) Blockchain Explorer.*

[16] King, J.C. 1976. Symbolic execution and program testing. *Commun. ACM.* 19, 7 (Jul. 1976), 385–394. DOI:https://doi.org/10.1145/360248.360252.

[17] Kroening, D. et al. 2016. *Decision Procedures: An Algorithmic Point of View.* Springer Nature.

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

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

[20] Miller, A. et al. 2018. Smart Contracts and Opportunities for Formal Methods. *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice* (Cham, 2018), 280–299.

[21] Nakamoto, S. 2009. Bitcoin: A peer-to-peer electronic cash system. (May 2009).

[22] Otoni, R. et al. 2023. A Solicitous Approach to Smart Contract Verification. *ACM Trans. Priv. Secur.* 26, 2 (Mar. 2023), 15:1–15:28. DOI:https://doi.org/10.1145/3564699.

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

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

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

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

[27] Zhou, L. et al. 2022. SoK: Decentralized Finance (DeFi) Attacks.