Model Checking and Gas Optimization of Move Smart Contracts, and Transaction Order Dependency Detection and Rectification

by

Eric Keilty

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Graduate Department of Electrical & Computer Engineering
University of Toronto

Model Checking and Gas Optimization of Move Smart Contracts, and Transaction Order Dependency Detection and Rectification

Eric Keilty
Master of Applied Science
Graduate Department of Electrical & Computer Engineering
University of Toronto
2023

Abstract

Blockchain technology has revolutionized various industries by providing secure transaction mechanisms in a decentralized, trustless environment. In 2014, the Ethereum blockchain platform introduced smart contracts, facilitating the deployment of a wide range of decentralized applications. However, since its inception numerous vulnerabilities have been discovered in the Ethereum Virtual Machine, many resulting in significant financial loss. Consequently, a new smart contract language, Move, has been developed, where security and verifiability are first class features. As the adoption of Move increases, it necessitates robust developer tools and adherence to best practice principles, similar to the existing infrastructure present in Ethereum. This thesis contributes to the advancement of this goal. First, it introduces VeriMove, the first model checking framework for the Move language. Experiments show that model checking is a feasible method to formally verify global properties in Move smart contracts. Second, this thesis presents the first gas optimization analysis of the Move language. Experiments show that the proposed gas optimization patterns reduce gas consumption in a typical smart contract by $7-56\%$. Lastly, this thesis proposes a novel algorithm to automatically audit the transaction order dependency vulnerability present in many popular public blockchain platforms. A prototype implementation is developed on Ethereum for the Solidity smart contract language. Experiments show that the proposed methodology can be used successfully to detect and rectify such vulnerabilities, or to certify their absence.

Acknowledgements

I am extremely lucky to have met an abundance of amazing people throughout my degree, without whom this thesis would not have been possible. First and foremost I extend my gratitude towards my supervisor Professor Andreas Veneris. I am deeply thankful for his guidance both in research and in life.

Thank you to my defense committee members Professor Fan Long and Professor Hans-Arno Jacobsen for generously giving their valuable time to analyze and refine my work. Likewise, thank you my defense committee chair Professor Frank Kschischang for his time and encouragement.

I am fortunate to have been part of a stellar group, surrounded by incredible colleagues. To Keerthi Nelaturu, thank you for your mentorship and collaboration throughout my research, and for being a fun travel companion. To Panagiotis Michalopoulos, thank you for all of your help in organizing and running the courses ECE345 and ECE358, and for being a kind and reliable person. To Srisht Fateh Singh, thank you for proof reading all of my work and your advice in academic writing. Above all else, to each of you, thank you for being great friends.

Finally, to my family, partner, and friends, your love and support means so much to me. Without you all, I would not be who I am today.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Blockchain technology has revolutionized various industries by providing secure transaction mechanisms in a decentralized, trustless environment. Smart contracts, in particular, have emerged as a powerful application of blockchain technology, enabling self-executing, tamper-proof agreements without the need for intermediaries. The Ethereum platform [27] is at the forefront of the smart contract revolution, facilitating the deployment of a wide range of decentralized applications (DApps).

Since its inception in 2014, numerous vulnerabilities have been discovered in the Ethereum blockchain [82, 131, 154]. The infamous DAO hack took advantage of the reentrancy vulnerability [135], resulting in over 3.6 million ETH (60 million USD at the time) being drained from the `DAO` smart contract [159, 165]. The impact was so significant that the community hard-forked the Ethereum blockchain to revert the attack [28]. The Parity Multisig Wallet hack resulted from a misuse of the `delegatecall` function causing all public functions in the `WalletLibrary` to be callable by anyone. This gave attackers unauthorized *access control* over specific multisig wallets, allowing them to transfer over $15,000$ ETH (30 million USD at the time) to their own wallets [126, 127]. In the `BeautyChain` contract, the `batchTransfer` function contained an unchecked arithmetic operation. As a result, the attacker used a batch-overflow attack (or more generally an integer-overflow attack) to trick the smart contract into minting a large number of tokens rather than only a few [19, 125].

Given the immutable nature of blockchains, rectifying vulnerabilities in DApps is considerably more challenging and thereby increases the associated risks when compared to traditional software applications, evident by the significant financial loss in the aforementioned examples. Ethereum has made efforts to mitigate these vulnerabilities through patches, updates, and best practices guidelines [141, 157]. Furthermore, researchers have developed numerous tools to detect security vulnerabilities in Ethereum smart contracts [56, 81, 89]. Using the above attacks as case-studies, the reentrancy attack is checked by almost every vulnerability detection tool. Best practice guidelines urge developers to only use `call` when interacting with external contracts, only using `delegatecall` when absolutely necessary. Finally, OpenZeppelin [122] created the `SafeMath` library [123], which prevents arithmetic-related vulnerabilities, including integer overflow. However, even this library has been patched due to unforeseen vulnerabilities [109].

Unfortunately, these vulnerability prevention measures will never be sufficient. The design of

Ethereum and the mechanism of asset transfer via the `fallback` function is fundamentally insecure. As the adage goes: *security cannot be bolted on, it must be built in.* The security and verifiability of a smart contract should be a feature enforced by design, rather than an after-the-fact addition dependent on the developer. This is the philosophy that inspired the Move language [23].

In Move, many of the most common vulnerabilities of Ethereum are mitigated by design. The reentracy and `delegatecall` vulnerabilities do not exist in Move, as it uses static rather than dynamic dispatch. Move introduces the novel *resource* type which enforces a strict system of ownership to prevent unauthorized access control, doing away with the flawed `fallback` function mechanism for transferring digital assets. Vulnerabilities such as integer overflow/underflow are checked at compile-time by the *bytecode verifier*. Furthermore, Move includes a theorem proving formal verification tool called the Move Prover [175] for more complex property verification and vulnerability detection.

As the adoption of Move increases, it necessitates robust developer tools and adherence to best practice principles, similar to the existing infrastructure present in Ethereum. While Move's design mitigates a large number of common vulnerabilities, it does not prevent all vulnerabilities. The extensive research on formal verification, gas optimization, and vulnerability detection of Ethereum smart contracts needs to be applied to the Move language.

1.2 Contributions

1.2.1 VeriMove: A Model Checking Framework for the Move Language

The first contribution of this thesis is the development of VeriMove, the first model checking framework that supports the Move language. It provides a user-friendly interface for developers to graphically design their smart contracts. Properties about their smart contract are specified using natural language templates and automatically verified. Once the user is satisfied with the smart contract design, the corresponding Move source code is generated. VeriMove is an expansion of the VeriSolid [104] model checking tool for Solidity [144], which contains two additional features. First, VeriMove extracts VeriSolid's parsing of Solidity statements into a separate module called the language parser. This allows any language, including Move, to utilize the model checking framework given the corresponding syntax tree. Second, VeriMove implements a finite state machine generator, which creates a prototype transition system given a pre-written smart contract.

The performance of VeriMove is compared to the performance of VeriSolid on the same set of smart contracts. These contracts are implemented in both VeriMove and VeriSolid. Each contract is given a series of verification properties and verified by each tool. The experimental results show that model checking is a feasible method to formally verify global properties in Move smart contracts.

1.2.2 Gas Optimization of Move Smart Contracts

The second contribution of this thesis is to present the first work on gas optimization in the Move language. Aptos is chosen as the underlying platform for the analysis since it is the leading Move-enabled blockchain platform, and it was the first to develop a gas meter.

This thesis details Aptos's gas meter. Then, the research on gas optimization in Solidity is analyzed with its potential application explored in the Move language. This thesis proposes 11 gas optimization patterns and principles for the Move language, and provides 5 patterns that decrease the time complexity of the smart contract but have no effect on gas consumption. A sample smart contract is created for each optimization pattern, demonstrating their validity. These contracts are used to estimate the effect of each proposed optimization in a typical Move smart contract. The experimental results show that the proposed gas optimization patterns reduce gas consumption in a typical smart contract by $7 - 56\%$.

1.2.3 Automated Auditing of Price Gouging TOD Vulnerabilities

The final contribution of this thesis is to develop a novel algorithm for automatically detecting and rectifying the transaction order dependency (TOD) vulnerability in smart contracts. This vulnerability exploits the public mempool to gain information about pending transactions. An attacker utilizes the priority fee mechanism to inject their malicious transaction before honest transactions in order to change their final output. The mempool and priority fee mechanism are features of many popular public blockchains, such as Ethereum and Aptos. As a result, smart contracts submitted to these blockchains are susceptible to this attack.

This thesis proposes a static analysis based approach utilizing point-to analysis and guard statements to automatically locate and rectify such TOD vulnerabilities. In particular, for each public function, the algorithm identifies the dependent global blockchain variables. By ensuring their values do not change between the transaction's submission and execution, it is impossible for other transactions to affect the final output. This algorithm is implemented in a prototype tool using Slither [54], a static analyzer for Solidity. The empirical results on a benchmark suite containing 51 Solidity smart contracts show that the proposed methodology can be used to detect and rectify such vulnerabilities, or to certify their absence.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 provides background on blockchain fundamentals, the Ethereum blockchain platform, the Move smart contract language, and the field of formal methods. Chapter 3 presents VeriMove, the first model checking framework for the Move language. Chapter 4 presents the first work on gas optimization in the Move language. Chapter 5 presents a static analysis methodology and prototype implementation for the detection and rectification of the TOD vulnerability. Finally, Chapter 6 summarizes this thesis and outlines avenues for future work on these topics.

Chapter 2

Background

2.1 Introduction

This chapter provides a brief introduction to the fields of blockchain and formal methods, both of which are important background related to the contributions of this thesis. Section 2.2 provides an introduction to the fundamentals of blockchains. Section 2.3 describes the popular blockchain platform Ethereum including its introduction of smart contracts. Section 2.4 describes the new smart contract language Move, describing its novel design and safety features. Finally, Section 2.5 provides an introduction to the field of formal methods, in particular the techniques of theorem proving and model checking.

2.2 Blockchain

As debit and credit cards continue to be used in lieu of cash [44, 45], commerce becomes more dependent on financial institutions to authorize and settle each transaction. This is undesirable for both consumers and merchants. The merchant must hassle the consumer for personal information to avoid fraudulent transactions. Credit card companies charge merchants between 1.5% to 3.5% of each transaction [72]. Part of this fee goes to the financial institutions to settle the transaction, which can take multiple days as financial institutions wait to batch as many transactions as possible. Due to this reliance on financial institutions as the *trusted middleman*, the current commerce system contains inefficiencies, which increase the cost of goods and services and limits the circulation of money in the economy.

Blockchain refers to a class of algorithms and protocols which allow for secure transactions between individuals without the need for a trusted middleman. Its rapid adoption represents a profound paradigm shift in the world's economic systems [24]. This section describes the fundamental aspects of a blockchain.

2.2.1 Properties of Cryptocurrency and Early Systems

A *digital currency* refers to any currency that is available in electronic form. A *cryptocurrency* is a digital currency which does not require a centralized authority to facilitate transactions. Centrally controlled digital currencies, such as video game credits and customer loyalty points, are simple to

design, as the creator maintains complete control over transaction protocols. However, designing a cryptocurrency that is decentralized while maintaining security and integrity proved to be much more difficult.

The main function of a currency is to be exchanged. Ideally, the exchange of cryptocurrencies closely resembles that of physical cash. To illustrate the required attributes of a cryptocurrency, consider the interaction between two individuals: Alice and Bob. Suppose Alice wants to send a portion of her *digital tokens*, a particular instance of a cryptocurrency, to Bob. Such a transaction must satisfy the following five properties. First, Alice can send any of her tokens to any person at any time. Second, the transaction does not require the consent of any additional parties. Third, only Alice is able to spend her tokens. No one else can spend her tokens for her (unless Alice gives explicit permission). Fourth, Alice cannot spend the same tokens more than once. Fifth, the transaction is *immutable*, neither Alice nor Bob can alter nor revert the transaction once it has been sent.

Early cryptocurrency systems [13, 33, 35, 46, 49, 59, 155] utilized a peer-to-peer, Byzantine fault tolerant network in order to satisfy the first two properties. The third property is solved using public key cryptography. Each user in the network generates a public/private key pair. The public key is used as a unique identifier, called an *address*, to send and receive transactions. Each transaction must be signed by the private key of the sender, called a *digital signature*. Therefore, assuming that Alice exclusively holds her private key, only she can produce a signed transaction for the transfer of her tokens. The fifth property requires a decentralized *consensus algorithm*. This is a difficult problem, but partly solved by early systems. Consensus algorithms will be discussed in Section 2.2.4.

The main roadblock of early systems was satisfying the fourth property, called the *double spending problem*. Prior systems were susceptible to *Sybil attacks*, where a single attacker creates many accounts in order to gain a large share of the network. Thus, these protocols required all participants of the network to be known in order to ensure security, which excludes applications requiring anonymity. In 2008 Satoshi Nakamoto introduced Bitcoin [114], which was the first cryptocurrency system to solve the double spending problem, and therefore satisfied all of the aforementioned properties. Bitcoin marked the genesis of modern blockchains. Its ideas and concepts have been utilized and expanded upon by many subsequent systems [1, 7, 10, 58, 97, 137, 148, 170].

2.2.2 Blockchain Fundamentals

A blockchain is a peer-to-peer, Byzantine fault tolerant network which maintains a decentralized, transparent, and immutable *digital ledger*. The ledger records the exchange and distribution of digital assets across the network, or more generally the *global state* of the blockchain. A *transaction* is a cryptographically signed set of instructions to update the global state of the blockchain. The most common type of transaction is the exchange of cryptocurrency or other digital assets between users. However, transactions in modern blockchains with the advent of *smart contracts* can perform more abstract operations (discussed in Section 2.3). The digital ledger is structured as a series of *blocks*, which connect sequentially to each other, *i.e.* a chain [18, 68]. Each block contains many transactions and represents a transition of the blockchain from one global state to the next. Figure 2.1 shows an example of this data structure.

There are two types of network participants in a blockchain. *Users* hold a unique public/private key pair, which allows them to access the ledger as well as submit transactions. *Validators*, also called network *nodes*, facilitate the transactions between users by receiving transactions, ordering

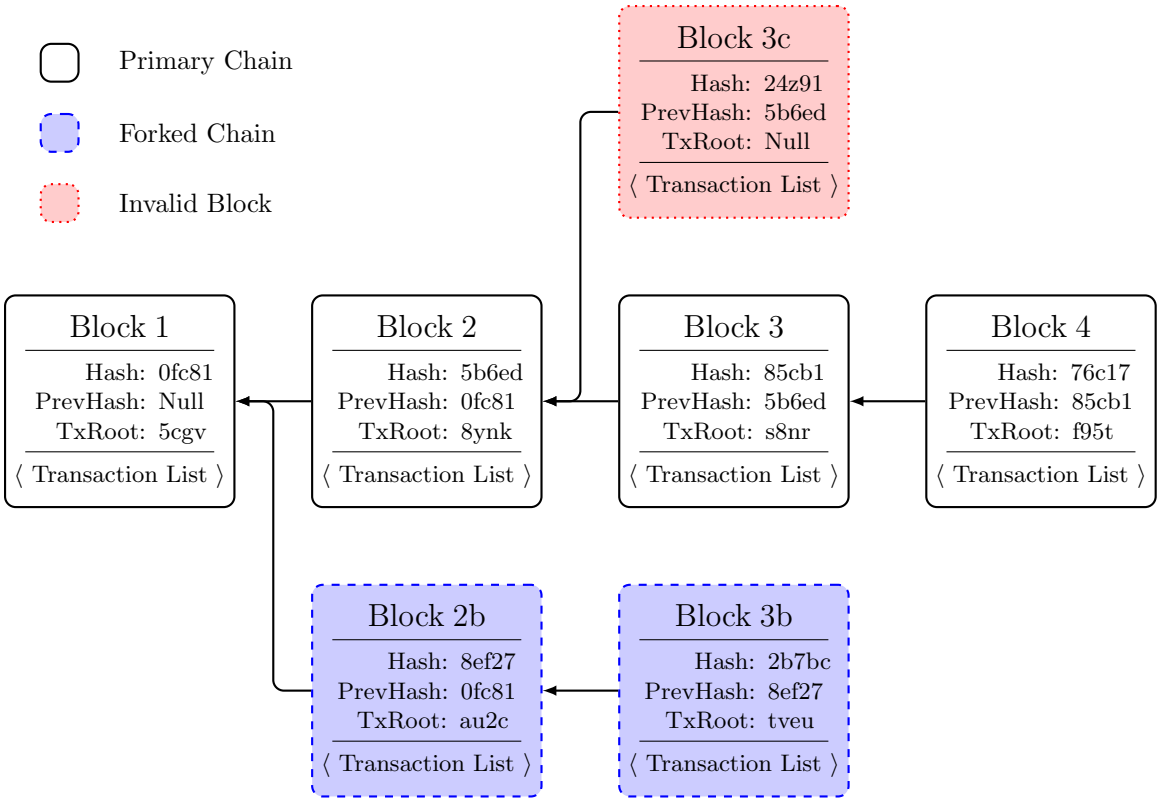

Figure 2.1: A Simplified Example of a Blockchain

them into blocks, and updating the blockchain state accordingly. Validators are given rewards for each block they add in the form of the network's native cryptocurrency. The value of the reward is designed so that validators are game-theoretically incentivized not to deviate from the intended protocol [95, 110].

The process of combining transactions into a block in accordance with the rules and protocols of the network is called *block validation*. Once a block has been validated, it is broadcast to the entire network. Upon receiving a new block, each network node checks the validity of the block, called *block verification*. There are three possible outcomes. 1) The block was not validated corrected, called an *invalid block* shown in red in Figure 2.1, so it fails verification and is rejected by the network. This results in no change to the digital ledger. 2) The block is validated correctly and passes verification upon broadcasting. In this case, the block is added to the network and the ledger state is updated accordingly. The sequence of valid blocks agreed upon by the majority of the network is called the *primary chain* shown in white in the above figure. 3) Multiple blocks are validated and submitted to the network at the same time. This results in network nodes disagreeing on the ledger state, called a *fork* shown in blue in Figure 2.1. Resolving forks such that there is vast majority agreement on the primary chain is the job of the consensus algorithm discussed in Section 2.2.4.

2.2.3 Block Contents

Each block in a blockchain contains three sections: the block number, the header, and the body. In Figure 2.1, these sections are delineated by horizontal lines. The *block number* denotes the block's position in the sequence of the primary chain. The *block header* contains meta-data about the block constructed during block validation and used for block verification. Lastly, the *block body* contains the list of transactions to be executed as well as all of the data necessary for their execution. Since transactions are non-commutative in general, the order of transactions within blocks is important.

In more detail, the block header includes the fields `PrevHash`, `Hash`, and `TxRoot`. Depending on the particular blockchain and consensus algorithm, other fields are also included. The `PrevHash` field is the hash (*e.g.* SHA-256, keccak256, etc) of the header of the previous block in the sequence. The `Hash` field is the hash of the current block's header excluding the `Hash` field. Since the `PrevHash` is included in this hash, the current block is dependent on the previous block, which is dependent on its previous block, and so on. Thus, if the header of any block in the blockchain is modified, this change will cascade down the chain through the `Hash` fields. Consensus algorithms utilize this property to ensure that block headers are immutable. Lastly, the `TxRoot` field contains the hash of the transaction list. Various data structures are used to accomplish this efficiently [90, 106]. Thus, any change to the transaction list will result in a different `TxRoot` value. Since the block header is immutable and the `TxRoot` value is included in the block header, the transaction list is also immutable.

2.2.4 Consensus Algorithms

Generally, there are two types of blockchains: *permissionless* and *permissioned*, also called *public* and *private* respectively. Permissionless blockchains allow anyone to join as a validator. The job of the consensus algorithm in these blockchains is two-fold: design a Sybil-resistant block validation process such that no single node can gain too much power in the network, and implement a fork-resolution procedure so that all network nodes agree on the state of the blockchain. There are numerous consensus algorithms [25, 96, 146, 151, 167], but the two most popular are Proof of Work [48] and Proof of Stake [87].

In *Proof of Work* (PoW) blockchains, the validation of a block requires the validator to solve a cryptographic puzzle. The solution is incorporated into the block header through an additional field called the *nonce*, which is an abbreviation for "number used once" or "no. once". As long as standard cryptographic assumptions hold [132], finding such a nonce requires random guessing and significant computation. Hence, validators in PoW blockchains are called *miners*. To resolve forks, many PoW blockchains use the *longest chain rule*, which asserts that, given a set of distinct chains, the longest sequence is the primary chain [43]. Thus, power over the network is partitioned by computational resources, so a single node gaining disproportionate power over the network becomes difficult and unlikely [8].

A significant drawback of PoW is its considerable computational waste as all miners participate in a network-wide brute force search for the nonce. *Proof of Stake* (PoS) was developed as a more efficient alternative. In this protocol, validators choose an amount of their tokens to "stake". When a new block needs to be constructed, a validator is chosen randomly, proportional to their stake. If the block is found to be improperly validated, then the validator's stake is *slashed*, *i.e.* the validator

forfeits their stake to the network, and the block is reverted. To resolve forks, PoS blockchains typically use finality and checkpoint mechanisms [29].

Permissionless blockchains often suffer from low throughput and high latency due to the computationally intensive consensus algorithms [166]. An alternative approach is to have a closed set of validators, allowing only authorized nodes to act as validators, called a permissioned blockchain. They can achieve a high throughput and low latency, but they are not a trustless environment as a central authority gatekeeps the validators. The job of the consensus algorithm in these blockchains is to design efficient and effective Byzantine fault-tolerant (BFT) algorithms to tolerate arbitrary failures [93, 173, 174]. If there are n participants in the network, PBFT [31] and HoneyBadgerBFT [108] use multi-round voting to achieve consensus in $O(n^2)$ messages. Additionally, SBFT [61], HotStuff [171], and Prosecutor [172] achieved consensus in $O(n)$ messages by optimizing voting protocols to a single round.

2.3 The Ethereum Blockchain

In 2014, Vitalik Buterin introduced Ethereum [170], whose main innovation was to expand the capabilities of the distributed ledger. Unlike Bitcoin, which only tracks token ownership within the network, Ethereum's global state allows for arbitrary computation and storage of user data. Transactions take the form of *bytecode*, which are instructions executed by the Ethereum Virtual Machine (EVM) to update the global state. A *smart contract* is a program executed by a blockchain consisting of many bytecode instructions. Thus, Ethereum can be thought of as a universal computer operating on a decentralized, immutable database shared among network participants. This generalized model of a blockchain has been adopted by almost every subsequent blockchain platform [1, 7, 10, 58, 137, 148].

2.3.1 Accounts

The global state of Ethereum is partitioned into *accounts*, which are mappings from account addresses and to an account state. An *account address* a 20-byte hexadecimal string used to interact with other accounts. The *account state* contains important account information such as the balance of ETH (Ethereum's native token) and a nonce used to count the number of transactions sent from the account. All accounts can receive, hold, and send ETH as well as interact with deployed smart contracts. Ethereum is an *accounts-based* blockchain where the ETH balance of an account is represented as an integer in the global state. The transfer of ETH simply modifies the balance of the sending and receiving accounts. This contrasts *unspent transaction output* (UTXO) blockchains, such as Bitcoin, where the native token is non-fungible and instances of tokens are explicitly transferred between accounts.

There are two types of accounts in Ethereum: externally-owned accounts and contract accounts. An *externally-owned account* (EOA) is an account created by a user. They are defined a by public/private key pair, which can be used to receive, hold, and send ETH as well as invoke deployed smart contracts. A *contract account* does not contain a private key; instead, its account activities are controlled by a smart contract. Thus, the account state of a contract account additionally contains the code of its smart contract and its storage contents. To ensure data authenticity, the storage

contents of an account contains a 256-bit hash of the root of a Merkle Patricia Trie [107]. Execution of a contract account's bytecode can be initialized by an EOA or another contract account.

2.3.2 Smart Contracts

In Ethereum, smart contracts are written in Solidity [144], a high-level JavaScript-inspired language, which is compiled into low-level bytecode and stored in the smart contract's account. Each smart contract defines methods, which can be invoked by other account via transactions submitted to the blockchain. If validated inside a block, the bytecode instructions are executed by all network nodes using the EVM and the global blockchain state is updated accordingly.

Smart contracts data can be located in memory or storage. *Memory* contains runtime data such as local variable values. This consists of 256-bit wide sectors, which can be randomly accessed. Data is packed in the order given by the contract. If runtime data exceeds the capacity of the current memory, then another 256-bit wide sector is allocated. After execution is complete, the allocated memory sectors can no longer be accessed. Note that memory data is not freed, instead it is overwritten by subsequent smart contract executions. However, this may change in the future. *Storage*, also called *state variables*, refers to data located in the contract's global storage on the blockchain. This data is stored in its contract account state, and persists after the execution of the smart contract.

2.3.3 Gas

Each network node must store the entire global state of the blockchain as well as execute each transaction. This is vulnerable to denial of service attacks where a user submits a computation-intensive transaction, causing a delay in the network. For instance, an infinite loop in the smart contract. Ethereum, and most modern blockchains, solve this problem using the concept of *gas*. Each EVM operation is given a corresponding cost, measured in gwei. One gwei is equivalent to 10^{-9} ETH. Each transaction then has a *total gas price* required for its execution, also measured in gwei, which is calculated by the *gas meter* at runtime. Upon submitting the transaction, the user specifies a `max_fee`, which is an upper bound on the total gas they are willing to spend on the transaction. If a transaction uses more gas than its gas limit allows, then the transaction will fail, causing no change to the global state. Additionally, this incentivizes non-malicious users to optimize their smart contract logic in order to reduce computation and subsequently gas cost, causing an overall decrease in network latency.

The London upgrade [157] introduced a significant change to Ethereum's block validation protocol. When a smart contract transaction is invoked and submitted for block validation, it is first added to the *mempool*, which is a waiting area for the pending transactions that have not yet been added to a block. In addition to the `max_fee`, a `priority_fee` (measured in gwei) is included in each transaction. The validator of a block receives all priority fees from every transaction included in the block. This fee acts as a tip to incentives validators to include particular transactions. The higher the tip, the more likely a validator will include the transaction in the next block. In certain smart contracts this mechanism can be exploited for financial gain via the *front-running* attack (Chapter 5).

2.4 The Move Language

Move [23] is a new smart contract language designed to ensure security and verifiability while retaining coding flexibility. It introduces the novel *resource* type with the goal of addressing the scarcity and access control issues inherent in representing digital assets on a blockchain. Additionally, Move incorporates a *bytecode verifier* for static analysis to catch common vulnerabilities, as well as a theorem proving formal verification tool called the Move Prover [175] for more complex, user-defined properties. At the time of writing, the Aptos [10], Sui [148], OpenLibra [1], and Starcoin [147] blockchains use Move as their smart contract language.

2.4.1 Global Ledger State

The global ledger state of Move consists of a set of *accounts*, each represented as a unique 256-bit value known as an *account address*. Each account consists of a set of *table entries*, which are *key-value pairs* stored in Binary Canonical Serialization format. These table entries contain smart contract bytecode published by the account and data owned by the account. An *item* of global storage is a generic term that refers to any key-value pair in global storage.

The structure of Move's global state can be interpreted as independent capsules of data, each owned by a unique account address. Due to the rigid structure of the Move language, an account cannot access data outside of its capsule. This is starkly different from other blockchains such as Ethereum, where the data of many different users may be stored under the same account address. Figures 2.2 and 2.3 compares account data storage in Move and Ethereum. In this example, account `0x123` creates a smart contract called `Coin` with a global state variable called `balance`. The account `0x123` and another account `0xABC` both create an instance of `balance`. In Move, the values of these `balance` variables are stored in the owner's account space. In Ethereum, however, the smart contract gets its own independent address space, `0xSC123` in this example, which stores both values.

The benefit of Move's global storage structure is that the program logic is separated from the digital assets of the users. This is not only more secure but is also more expressive and flexible compared to Ethereum. Due to the features of the Move language, accounts have full control over their data. No other account, including the account defining the smart contract, can access it.

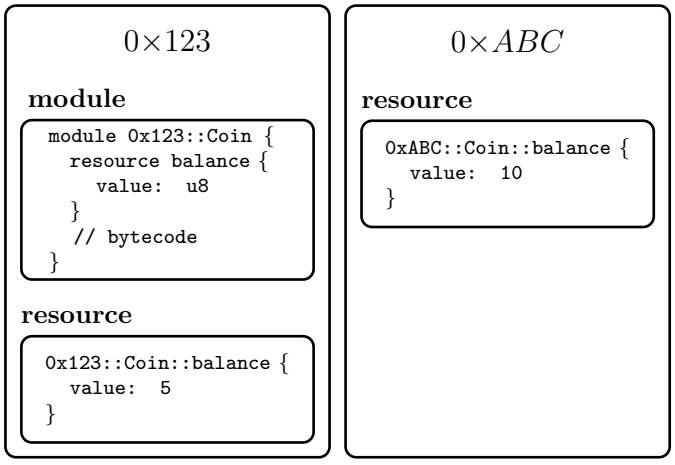

Figure 2.2: The Global Ledger State of Move

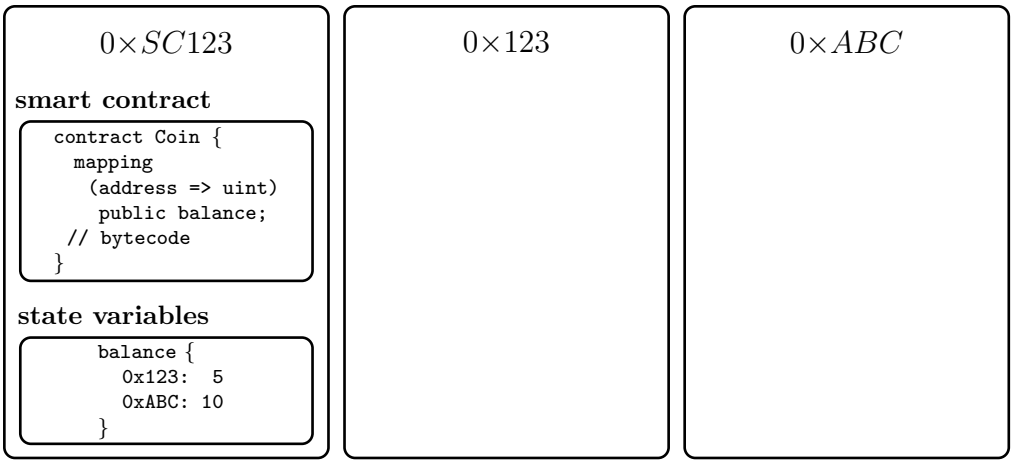

Figure 2.3: Global Ledger State of Ethereum

2.4.2 Smart Contracts

Smart contracts are written in the high-level language Move and compiled into low-level bytecode, which is executed by the Move Virtual Machine (MoveVM). The MoveVM is a stack machine containing an *operation stack* where Move values are stored and a *call stack* where active functions are stored. All registers in the MoveVM are 64-bits. Users theoretically have unlimited local memory allocation. However, the instantiation of memory requires gas and computation. A memory-intensive smart contract will result in either an out-of-gas error or an execution-time-limit error.

A Move smart contract is called a *module*, which define *structs* and *procedures*. Structs are custom data types that can be objects of global storage. Procedures are functions that define rules for state transitions. When a module is submitted to the blockchain, its high-level Move code is compiled to low-level bytecode and stored (but not executed) in the account of the publisher on the blockchain. *Transaction scripts* can import modules to utilize their structs and procedures. Similar to the `main` function in a C program, the transaction script is the code initially executed by blockchain transactions.

2.4.3 Memory Management

To manage memory, Move implements a Rust-like system of ownership [158] where each variable "owns" its stored value, and each stored value can only have one owner. A stored value can be copied to another variable (if allowed by the type) in which case the stored value is duplicated and assigned to the new variable as its sole owner; the original variable retains its stored value. Alternatively, ownership of a stored value can be transferred to another variable in which case the new variable owns the stored value, and the original variable is no longer valid to use. Values can be transferred via variable assignment or the return value of a function. Once the end of a local scope is reached, all local variables are *dropped* and their allocated memory is freed. The *borrow checker* is the compiler component that ensures the program follows these ownership rules.

Figure 2.4 shows an example of value ownership transfer in Move. Analyzing the left figure first, the variable `s1` is instantiated on line 4 with the value `"Hello, World!"`. On line 5, the ownership of this value is *transferred* from variable `s1` to `s2`. Now, variable `s2` points to the value `"Hello,`

```
module MyModule::my_module {
    use std::string;
    fun transfer_of_ownership() {
        let s1 = string::utf8(b"Hello, World!");
        let s2 = s1;
        let s3 = s1;    // error
    }
}
```

```
module MyModule::my_module {
    use std::string;
    fun transfer_of_ownership() {
        let s1 = string::utf8(b"Hello, World!");
        let s2 = copy s1;
        let s3 = copy s1;
    }
}
```

Figure 2.4: Value Ownership Transfer Example

`World!"` and variable `s1` is a `null` pointer. On line 6, variable `s3` is attempting to borrow the value stored in variable `s1`. However, since `s1` no longer owns any values, an error is thrown. In the right figure, the value in variable `s1` is *copied* to `s2` and `s3` rather than transferred. Thus, no error is thrown and all variables have different instances of the value `"Hello, World!"`.

2.4.4 Struct Abilities

In a Move module, custom data types can be defined using *structs*. A struct can have the following abilities: `key`, `store`, `copy`, and `drop`. The abilities `key` and `store` allow the struct to be used as a key and value, respectively, in key-value pairs in global storage. The ability `copy` allows an instance of the struct to be copied into another variable. The ability `drop` allows an instance of the struct to be dropped by the end of the scope. Primitive types - `u8`, `u64`, `u128`, `bool`, and `address` - have the abilities `store`, `copy`, and `drop`, which results in the standard behavior of classical program variables. These abilities are enforced in the Move language through the Rust-like ownership system and the bytecode verifier.

Figure 2.5 gives an example of a struct with no abilities, which serves to further illustrate Move's ownership and memory management rules. The top-left figure instantiates an instance of `MyStruct`. At the end of the function's local scope, Move attempts to free all local variables. However, since `MyStruct` does not have the `drop` ability, an error is thrown. This can be remedied in three ways. First, in the top-right figure, the `drop` ability is added to `MyStruct`. Second, in the bottom-left figure, the struct is manually deallocated. Third, in the bottom-right figure, the struct is passed into the return statement, and the value's ownership is transferred to the caller of the function.

```
module MyModule::my_module {
    struct MyStruct {
        value: u64
    }
    fun struct_with_no_abilities() {
        let my_struct = MyStruct { value: 0 };
        // error
    }
}
```

```
module MyModule::my_module {
    struct MyStruct has drop {
        value: u64
    }
    fun struct_with_no_abilities() {
        let my_struct = MyStruct { value: 0 };

    }
}
```

```
module MyModule::my_module {
    struct MyStruct {
        value: u64
    }
    fun struct_with_no_abilities() {
        let my_struct = MyStruct { value: 0 };
        let MyStruct{ value: _ } = my_struct;
    }
}
```

```
module MyModule::my_module {
    struct MyStruct {
        value: u64
    }
    fun struct_with_no_abilities(): MyStruct {
        let my_struct = MyStruct { value: 0 };
        return my_struct;
    }
}
```

Figure 2.5: No Ability Struct Example

2.4.5 Resources

A *resource* is a struct with only the `key` ability (and optionally the `store` ability). Thus, it cannot be created nor destroyed by code outside its declaring module and can never be copied or dropped. When a resource is initialized, it must eventually be stored globally under an account address. Like all variables in Move, resources are subject to the Rust-like ownership rules. Thus, the storing account address is the resource's sole owner. Resources may be transferred between account addresses; however, since resources cannot be duplicated, the original account address loses access to the resource as the receiving account address becomes the sole owner. While resources may seem restrictive, they allow programmers to encode safe, yet customizable digital assets that are controlled only by their owner and can neither be copied nor destroyed by code outside the declaring module.

Figure 2.6 shows each way one can access a resource in global storage. Using the `move_to` function, a resource can be added to global storage. In this case, ownership is transferred from the defining module to its global storage. Using the `move_from` function, ownership of a previously stored resource can be transferred to a module variable. Note that the resource cannot be dropped, so the module must either deallocate the resource or transfer its ownership elsewhere. Using the `borrow_global` function, a resource's fields can be read. Lastly, using the `borrow_global_mut` function a resource's fields can be updated. Note that during a read and write, ownership of these values has not been transferred, only borrowed. Thus, at the end of the local scope, ownership is transferred back to global storage. In all examples, in order to access a resource one must obtain signer privilege. A *signer* is an account address with a digital signature, which gives other accounts permission to access their global storage data.

```
module MyModule::my_module {
    struct MyResource has key {
        value: u64
    }
    fun create_resource(account: &signer) {

        let my_resource = MyResource { value: 0 };
        move_to(account, my_resource);
    }
}
```

```
module MyModule::my_module {
    struct MyResource has key {
        value: u64
    }
    fun destroy_resource(account: &signer)
    acquires MyResource {
        let my_resource =
                    move_from<MyResource>(account);
        MyResource {value: _} = my_resource;
    }
}
```

```
module MyModule::my_module {
    struct MyResource has key {
        value: u64
    }
    fun read_resource(account: &signer)
    acquires MyResource {
        let my_resource =
                    borrow_global<MyResource>(account);
        let x: u64 = copy my_resource.value;
    }
}
```

```
module MyModule::my_module {
    struct MyResource has key {
        value: u64
    }
    fun write_resource(account: &signer)
    acquires MyResource {
        let my_resource =
                borrow_global_mut<MyResource>(account);
        my_resource.value = 1;
    }
}
```

Figure 2.6: Resource Create, Destroy, Read, and Write

2.4.6 Built-In Verification

Before any Move module can be published, it must pass the *bytecode verifier*. At compile-time, the bytecode verifier statically verifies basic lightweight safety properties. These checks fall into four categories: *(i)* stack consistency checks such as the height of the operand stack is the same at the

beginning and end of each basic block; *(ii)* structural checks such as checking that statements are well-formed and module dependencies are acyclic; *(iii)* semantic checks such as incorrect procedure arguments, dangling references, duplicating a variable without the `copy` ability, and dropping a variable without the `drop` ability; and *(iv)* authorization checks such as accessing items without signer privilege [23].

For verification of more complex properties, Move has an embedded offline verifier called the Move Prover [175], which is a theorem proving formal verification tool written in Rust. The prover takes as input Move source code annotated with specifications and determines whether the code meets those specifications. Supported specifications include Floyd-Hoare pre-conditions, post-conditions, and function aborts.

2.5 Formal Methods

In traditional software applications, the typical method for verifying a program's correctness is through empirical *system testing*. This involves writing particular test-cases to verify intended behavior and mocking the production environment via simulations. Test-cases can only verify functional requirements, which may contain ambiguities that lead to inadequate testing. Simulation requires assumptions, which do not always cover all the aspects of a system [47]. The core issue is that system testing can find bugs, but cannot guarantee the absence of errors as exhaustive system testing is almost never possible practically nor theoretically.

Formal methods are mathematically-based techniques for modeling programs, unambiguously specifying system requirements, and verifying adherence to these requirements. Unlike system testing, these methods can guarantee program correctness and an absence of errors. Typically, these systems utilize an underlying first-order or higher-order logical language along with *satisfiability modulo theories* (SMT) solvers, which generalize the concepts of *boolean satisfiability* (SAT) to more complicated data structures. Two popular types of formal methods are *theorem proving* and *model checking*.

2.5.1 Theorem Proving

Theorem proving consists of modeling programs and system requirements in rigorous mathematical logic. The adherence of a program to system requirements is proved using a formal proof-system. There are three types of theorem provers. Automated theorem provers (ATP) use logical deduction and exhaustive search until either a proof or a counter-example is found [30, 111, 119, 136]. However, due to the complexity of system requirements, exhaustive search may not be feasible. Interactive theorem provers (ITP) allow for human intervention in order to complete difficult proof [20, 42, 62, 85, 152]. For example, the user may assert smaller *lemmas* which can be automatically proved. Then, using these lemmas, an automated search for a proof of the main result becomes feasible. Lastly, hybrid tools combine multiple theorem provers into a single application [112, 128, 129, 134].

The advantage of theorem proving is their theoretical foundation. If the underlying logic is sound and complete, then the theorem proving tool can formally verify any property. Moreover, theorem proving can model code of arbitrary size and systems with infinite state-spaces without state-space explosion. However, the disadvantage of theorem proving is their practical application. APTs suffer from being extremely computationally intensive. This is partially solved by IPTs; however, these

tools typically require an expert in mathematical logic in order to use them effectively. Furthermore, many theorem proving tools work on a method-by-method basis, meaning they can prove functional properties, but not necessarily system-wide properties.

Floyd-Hoare Logic

Floyd-Hoare Logic [55, 76] is one of the first axiomatic methods of formally proving properties of programs. While modern theorem provers use SMT solvers with first-order or higher-order logic as its foundation, the concepts of preconditions and postconditions are still used in the theory of formal methods and many theorem proving tools, including the Move Prover. Thus, a brief introduction is given in this section.

Let C be a program. A *precondition* is a claim about the initial state of C. A *postcondition* is a claim about the terminal state of C. The following is called a *Floyd-Hoare triple*,

$$\{P\}\ C\ \{Q\} \tag{2.1}$$

where P and Q are preconditions and postconditions of program C, respectively. A Floyd-Hoare triple is said to be *valid* if when program C is executed in a state satisfying P, it's final state satisfies Q. The goal of Floyd-Hoare logic is to determine whether a given Floyd-Hoare triple is valid.

An *axiom* is a statement that is valid by assumption. In Floyd-Hoare logic, there are two axioms: the *empty axiom* and the *assignment axiom*, given in Equations 2.2 and 2.3 below.

$$\frac{\texttt{true}}{\{R\}\ \texttt{empty}\ \{R\}} \tag{2.2}$$

$$\frac{\texttt{true}}{\{R[E/x]\}\ x := E\ \{R\}} \tag{2.3}$$

A *rule of inference* is a transformation from one Floyd-Hoare triple to another that maintains validity. Examples of such rules in Floyd-Hoare logic are the *composition rule* and *consequence rule* given in Equations 2.4 and 2.5.

$$\frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1\ ;\ C_2\ \{Q\}} \tag{2.4}$$

$$\frac{P \implies P' \qquad \{P'\}\ C\ \{Q'\} \qquad Q' \implies Q}{\{P\}\ C\ \{Q\}} \tag{2.5}$$

The notation indicates that the top set of statements entails the bottom set of statements. These rules equivocate semantic and syntactic validity; the latter of which is easily verified by computers. Applying the syntax of these rules guarantees that the result maintains semantic validity.

When attempting to prove the semantic validity of a Floyd-Hoare statement, one starts with the target statement and applies the inference rules in reverse. Iteratively applying these rules will result in a tree branching upward, rooted by the target statement. In Floyd-Hoare logic, a *proof* is a tree of inferences starting at the target statement such that each step maintains validity and all leafs are axioms. When constructing a proof, there is an element of choice. In the Equation 2.4, there is a choice of the statement R. In Equation 2.5, there is a choice of statements P' and Q'. APTs require heuristic algorithms to make these choices. ITPs require human ingenuity to suggest

intelligent statements.

This logic is sound and complete, relative to the interpretive semantics [41]. Thus, given any Floyd-Hoare triple, a proof exists if and only if the triple is valid. This logical structure provides a method of specifying program properties that is both intuitive for developers as they typically think in terms of inputs and outputs of programs, and automatically verifiable by a computer as semantic validity is reduced to syntactic validity.

2.5.2 Model Checking

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for that model [16]. Figure 2.7 shows the overview of the model checking process. A program is modeled as an abstract transition system and properties are encoded in a restricted formal logic. Then a model checker analyzes the system. If there exists a path through the transition system that violates one of the properties, the model checker will output a concrete counterexample in the form of an execution path. Otherwise, the system satisfies the given properties.

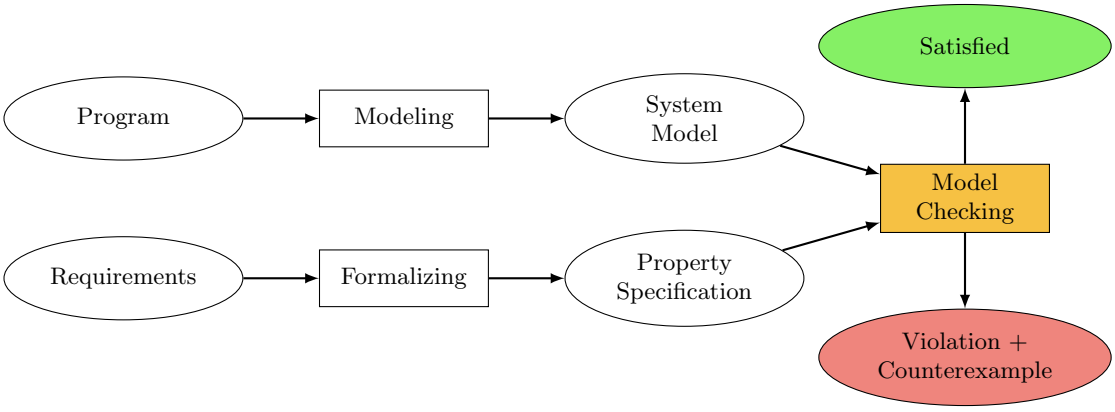

Figure 2.7: Model Checking Overview

A *transition system* is an automaton-like data structure consisting of states and transitions. Each state represents a set of configurations of the program. Each transition represents a set of sequential operations which update the program configuration. An example of a transition system is given in Figure 2.8, which models the functionality of a simplified vending machine that dispenses coke and sprite. The red node indicates the initial state of the system. A desired property of this system is "if the user inserts money, then the machine will eventually dispense a beverage". This statement can be formalized into rigorous logic and verified. In this transition system, all paths eventually lead to either `dispense_coke` or `dispense_sprite`. Thus, this property holds in this model.

There are two types of model checkers: explicit-state and symbolic. *Explicit-state model checking* algorithms directly compute program states and use graph algorithms to explore the state space starting from the initial state [71, 77, 78, 86]. The issue with this technique is state-space explosion and limited memory. *Symbolic model checking* algorithms partially address these issues using implicit representations of sets of states [6, 32, 38, 60, 150]. Binary Decision Diagrams (BDD) are used to verify the system against the property specifications [105]. Quantified Boolean Formula (QBF) are used to deal with asynchronous programs [40]. Properties are specified in a restricted formal

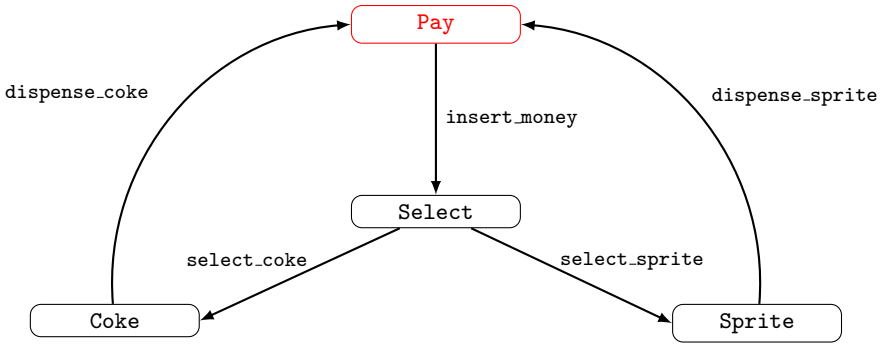

Figure 2.8: Transition System - Vending Machine Example

logic so that they are compatible with state-space search. Examples include Linear Temporal Logic (LTL) [133], Property Specification Language (PSL) [80], SystemVerilog Assertions (SVA) [169], and Computational Tree Logic (CTL) [39].

The advantages of model checking arise from its ease of use and interpretation, especially when compared to theorem proving. Model checking requires the creation of a high-level abstract representation of their program. This compels the user to focus on the overall system behavior, rather than minute details. As a result, model checking does not demand expertise in mathematical logic. Additionally, if a system does not adhere to a property, model checking provides concrete counterexamples in the form of state traces, aiding developers in locating the source of the issue and facilitating debugging efforts. Finally, unlike most theorem proving tools, model checking can verify properties concerning the interactions that span multiple function executions.

The primary challenge of model checking is state-space explosion. For large systems, many model checking tools only partially explore the state space, *e.g.* up to a certain depth. Thus, to ensure completeness of verification, the user is limited to either small systems or high-level representations of complex systems. Additionally, properties must be specified in simplified logics such as LTL and CTL. Together, this limits the expressiveness of the properties that users can verify.

Chapter 3

VeriMove: A Model Checking Framework for the Move Language

3.1 Introduction

Due to the immutability of the blockchain, it is imperative to identify and prevent vulnerabilities in smart contracts prior to deployment. One method for mitigating vulnerabilities is to use the many coding practices [81] accessible in traditional programming languages. Current smart contract coding practices include the following: *(i)* carefully understanding the semantics of the language specification [170], *(ii)* using safe coding practices highlighted by teams like OpenZeppelin [122], and *(iii)* mandatory source code auditing by qualified service providers [22]. While these practices can prevent common vulnerabilities, they offer no guarantees, and other vulnerabilities may still be present.

An alternative approach for mitigating vulnerabilities is to use *formal verification*. Formal verification tools are based on formal operational semantics and offer robust verification guarantees. They enable the formal specification and verification of attributes and can discover both typical and atypical vulnerabilities that could lead to a security property violation. Among the existing verification tools there are three common categories of techniques: theorem proving, symbolic execution, and model checking [56, 70].

Move [23] is a smart contract language designed to ensure security and verifiability while retaining coding flexibility. Currently, the bytecode verifier and Move Prover [175] can only verify *local* properties that are contained within a single Move function. However, some properties are *global* in nature, and occur as the result of many function executions. This demonstrates a need for additional verification tools. Notably, VeriSolid's correct-by-design model checking framework has shown success in prior work [104, 116] for verifying global properties in Solidity.

This chapter presents VeriMove, a model checking framework that extends VeriSolid for the Move language. The contributions of this research are as follows:

- A comparative analysis of Move and Solidity, focusing on the safety features (and lack thereof) found in both languages and the trade-offs associated with utilizing one over the other.
- VeriMove is introduced, a model checking framework that leverages VeriSolid to verify

and generate Move smart contracts automatically based on the specifications provided.

- The workflow of the VeriMove prototype implementation is outlined in-depth.
- As part of the model checking framework, the operational semantics for the new Move constructs are defined.
- VeriMove is compared with VeriSolid by comparing the experimental outcomes of three types of smart contracts, showing that VeriMove can verify global properties in Move with reasonable performance.

The remainder of the chapter is structured as follows. Section 3.2 summarizes the works that are relevant to this research. Section 3.3 compares both Solidity and Move in detail, examining their features, vulnerabilities, and trade-offs. Section 3.4 described the blind auction smart contract as a motivating example for the necessity of model checking. Section 3.5 examines workflow of the VeriMove model checking framework and introduce its components. Section 3.6 describes the additional and updated operational semantics for the Move language. Section 3.7 describes the prototype implementation in detail and examines the experimental results. Finally, Section 3.8 concludes this work.

3.2 Related Work

Due to the immutability of the blockchain, it is best to detect vulnerabilities in a smart contract before it is deployed. This can be achieved through the method of formal verification, which proves or disproves whether certain properties hold for a given smart contract. Verification tools for smart contracts can be classified as either theorem proving, symbolic execution, or model checking [56, 70].

Proof-based methods involve modeling the program and the desires properties in a formal mathematical language. A theorem prover for that language then uses well-known logical axioms and simple inference rules to prove (or disprove) that the desires properties hold in the smart contract. Tools of this type that are compatible with EVM bytecode include the $\mathbb{K}$ framework and F*. $\mathbb{K}$ [74] is a general purpose framework that uses the formal semantics of a language to generate a variety of tools. Using their semantic definition of the EVM bytecode, the $\mathbb{K}$ framework automatically generates a deductive verifier called KEVM. This uses reachability logic to evaluate program specifications expressed as reachability claims. Bhargavan et al. [21] designed a framework that converts Solidity source code and EVM bytecode into the existing language F* where it can be verified for correctness and saftey properties. The only theorem proving tool implemented for the Move language is the Move Prover discussed in Section 2.4.6.

Symbolic execution replaces program variables with symbolic expressions such that subsequent variables are expressed in terms of previous variables. The execution paths with respect to all feasible inputs are generated and searched for vulnerabilities. Tools such as Oyente [14], GASPER [36], and Osiris [162] construct a Control Flow Graph [4] to represent the state-space and use an SMT solver to detect vulnerabilities. Other tools such as Slither [54] and SmartCheck [160] utilize a semantic tree along with an intermediate representation [94] to detect vulnerabilities. Currently, there are no symbolic execution tools implemented for the Move language.

Given a finite-state model of the program and a formal specification of the desired properties, model checking verifies that the model behavior conforms to the specifications. This is often achieved through state space exploration and satisfiability solvers. Well-known model checking tools include Zeus and VeriSolid. Zeus [84] is a symbolic model checking tool that takes as input Solidity source code along with XACML policy specification. It converts these inputs to a low-level intermediate representation (LLVM bitcode [94]) and leverages SeaHorn [67] to perform the symbolic model checking. VeriSolid [104] is discussed in detail in Section 3.5.1. Due to its graphical representation of the transition system and natural language templates for property specification, it is more user-friendly than Zeus. VeriMove, the tool introduced by this thesis, is the first model checking tool implemented for the Move language.

3.3 Comparison of Move and Solidity

Solidity is a smart contract language designed to run on the EVM. To date, it is one of the most popular and widely used languages for smart contract development. Move is a recently developed smart contract language, which continues to garner support among various blockchain networks due to its unique safety features. This section compares the main differences between Move and Solidity, the trade-offs associated with each, and ultimately why Move is worthy of tooling.

3.3.1 Global Storage and Local Memory Management

During the execution of a smart contract, the compiler needs to manage three things: 1) the source code of the smart contract, 2) the local variables used during execution, and 3) global variables that remain persistent after execution.

In Solidity, each contract is given its own address space on the blockchain, where its source code (functions) and global variables (state variables) are stored. In Move, however, smart contracts are not given a separate address space. Instead, the smart contract bytecode (module) and global variables (resources) are stored in the account of their respective owners. Thus, a declared resource is not necessarily stored under the same account address as its defining module. This decoupling of data from the control flow logic is not only more secure, but also makes Move a more expressive and flexible language compared to Solidity.

For local variables in Solidity, once the execution of the function completes, the temporary memory pointers move to the next available memory slot. From the developer's perspective it looks like their temporary memory has been wiped. However, Solidity does not guarantee that this memory has been "zeroed out", and does not provide any method for developers to manually free their memory [145]. While Solidity claims this may change in the future, as it stands Solidity is incredibly susceptible to memory leaks. In contrast as discussed in Section 2.4.3, Move implements a Rust-like memory management system where each value has exactly one owner. This makes all Move variables (both global and local) completely memory safe and guarantees no memory leaks.

3.3.2 Transfers

In order to perform a transfer from one contract address to another in Solidity, the sender contract must use either `send`, `transfer`, or `call`, each of which behaves differently and are intended

for different applications. Meanwhile, the receiver contract must implement a `fallback` function. When the transfer is made, the EVM exits the sender contract and enters the `fallback` function of the receiver contract. The `fallback` function completes the transfer, but may also execute other code unbeknownst to the sender contract. Once the end of the fallback function is reached, the EVM returns to the sender contract and continues on the next line. Many of the vulnerabilities in Solidity can be attributed to unexpected behavior of and manipulating the interaction between these functions, such as reentrancy, mishandled exceptions, unchecked call return value, delegate call to untrusted callee, and denial of service from unexpected revert [34].

In Move, resources were developed to be implemented as digital assets. Recall from Section 2.4.5 that resources must follow the strict, Rust-like ownership rules and resources cannot be copied nor dropped. Thus, transferring a resource is simply a matter of transferring its ownership between account addresses, which can only be done by the resource's sole owner. This is the biggest advantage of Move; its design and implementation is centered around making the transfer of resources safe and secure. Consequently, all aforementioned vulnerabilities, including reentrancy, are mitigated from the Move language by design.

3.3.3 Trade-Offs

Move is a more restricted language compared to Solidity. Based on the results from Section 3.7, Move requires an increase of 35% in the lines of code on average for equivalent smart contract functionality. This gives Move a larger learning curve and makes it generally more difficult to use in practice, which can result in more bugs and unintentional vulnerabilities.

The restrictions in the Move language are present by design and have an important purpose. A Move smart contract that passes the bytecode verifier is completely memory safe and void of common vulnerabilities, such as integer overflow/underflow and reentrancy. Moreover, the Move language and the resource type were designed for safe and secure transactions. To date, a few vulnerabilities have been discovered in the MoveVM [120, 121], which have subsequently been fixed. However, these were minor mistakes in the implementation of the virtual machine, rather than a fundamental design flaw of the language. Thus, compared to Solidity, Move is a much safer language.

3.4 Blind Auction: A Motivating Example

In a blind auction, each participant submits a bid which is kept secret from all other participants. After the bidding period, the secret bids are revealed, and the winner is the participant with the highest bid.

The Solidity documentation provides a smart contract implementation of a blind auction [143]. During the bidding period, bidders submit the hash of their bids along with a deposit to the smart contract. After the bidding period is over, each bidder must send their secret key to the smart contract to reveal their encrypted bid. A bid is valid if the original deposit is larger than the submitted bid amount. The winner is the highest valid bid. After the bidding period, the winner withdraws the difference between their deposit and their bid. The other bidders withdraw their entire deposits. This procedure ensures that each bid is confidential during the bidding period (blind) and payment of a valid bid is irreversible after the bidding period (binding).

Consider the following statement about the blind auction procedures: "A bidder cannot submit a bid after the bidding period". Formally verifying this property using theorem proving tools is difficult as theorem provers typically can only verify the behavior of individual functions. The above statement is a high-level property, which spans multiple functions invocations. Thus, model checking is required to verify such a statement.

Figure 3.1 shows the transition system for the above blind auction smart contract implementation, which will be used in the model checking process. It has been decomposed into the following states.

- **Initialization** (`Init`): Before the bidding period has started.
- **Accepting Blinded Bids** (`ABB`): The bidding period begins. Participants submit their blinded bids and make their deposits.
- **Revealing Bids** (`RB`): After the bidding period, bids are revealed and a winner is determined.
- **Finished** (`F`): After the reveal period, the winning bidder withdraws the difference between their deposit and their bid; other bidders withdraw their entire deposits.
- **Canceled** (`C`): No winner is declared and all bidders withdraw their entire deposits.

Each transition represents a method in the smart contract. For example, during the bidding period, participants invoke the `bid` function in order to submit their bids and deposits. After the bidding period, bidders call the `reveal` function to reveal their blinded bids. Finally, after the reveal period, bidders call the `withdraw` function to get their deposits back. Between square brackets are conditions that must be satisfied in order to invoke a state transition. For example, `close` cannot be called until after the bidding period.

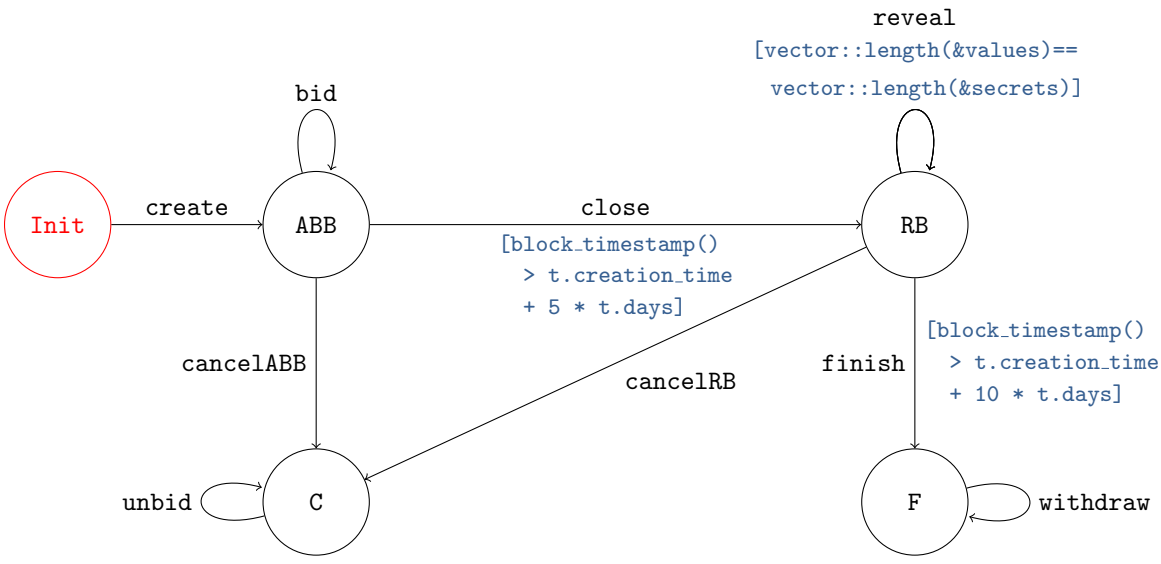

Figure 3.1: Blind Auction Transition System

3.5 VeriMove: Design and Verification Workflow

VeriMove is an open-source, web-based, model checking tool designed for collaborative development of Move smart contracts with build-in version control enabling branching, merging, and history viewing. This section describes the components of the tool and its workflow. VeriMove includes two major additions/modifications to VeriSolid: the language parser and the finite state machine (FSM) generator.

3.5.1 VeriSolid

VeriSolid [104] is an open-source, web-based, model checking framework built on top of WebGME [101] and FSolidM [102, 103]. It allows developers to specify their program functionality using an abstract, graphical representation in the form of a transition system. The desired system properties are encoded using various natural language templates, which can verify safety, liveness, and deadlock freedom properties. In order to verify a smart contract, the transition system is converted into a Behavior-Interaction-Priority (BIP) model [17], which is then translated into an NuSMV model [38]. The templated properties are used to generate Computation Tree Logic (CTL) specifications [15, 39]. State space exploration in the BIP model can verify deadlock freedom properties and the NuSMV model can verify safety and liveness properties using the nuXmv model checker [32]. Once the developer is satisfied with the model and properties, VeriSolid generates the equivalent Solidity source code. This architecture is replicated and adapted for the Move language in VeriMove.

3.5.2 Language Parser

Much of the functionality in VeriSolid and VeriMove requires complex statements to be broken up into a series of single expressions. In VeriSolid, this process was done largely manually for Solidity statements, which makes it difficult to extend to other languages, such as Move. Thus, VeriMove extracted this functionality into a modular component called the *language parser*. Given the grammar definition of a language, the language parser automatically generates a parsing tree, which is used to obtain the simplified expressions. The `move-tree-sitter` package is used to build the syntax tree for parsing Move statements. This feature makes VeriMove more flexible than VeriSolid, as only the grammar definition needs to be changed to support other languages.

3.5.3 Finite State Machine Generator

Developers using VeriSolid must adhere to a highly restricted format when writing smart contracts, which can be a time-consuming and labor-intensive task. Thus, VeriMove implements the ability to automatically create a transition system from Move source code. To accomplish this, the FSM generator creates an *initial state* and a *core state*. A transition is added from the initial state to the core state which initializes the smart contract. Then, all smart contract functions are represented as self-looping transitions in the core state. This generates a preliminary transition model that can be modified by the developer for their application. Due to the language parser, this functionality is extendable to other languages.

```
module BlindAuction::blind_auction {
    struct Bid has key, store {
        blinded_bid: vector<u8>,
        deposit: u128,
    }
    struct State has key {
        bids: Table<address, vector<Bid>>,
        pending_returns: Table<address, u128>,
        highest_bidder: address,
        highest_bid: u128,
    }
    fun withdraw() acquires State {
        let s = borrow_global_mut<State>(self());
        let mut_amount = Table::borrow_mut_with_default(&mut s.pending_returns, &sender(), 0);
        if (*mut_amount > 0) {
            if (sender() != s.highest_bidder ) {
                transfer(sender(), *mut_amount);
            } else {
                transfer(sender(), *mut_amount - s.highest_bid);
            };
            *mut_amount = 0;
        };
    }
}
```

Figure 3.2: Withdraw Function Move Implementation

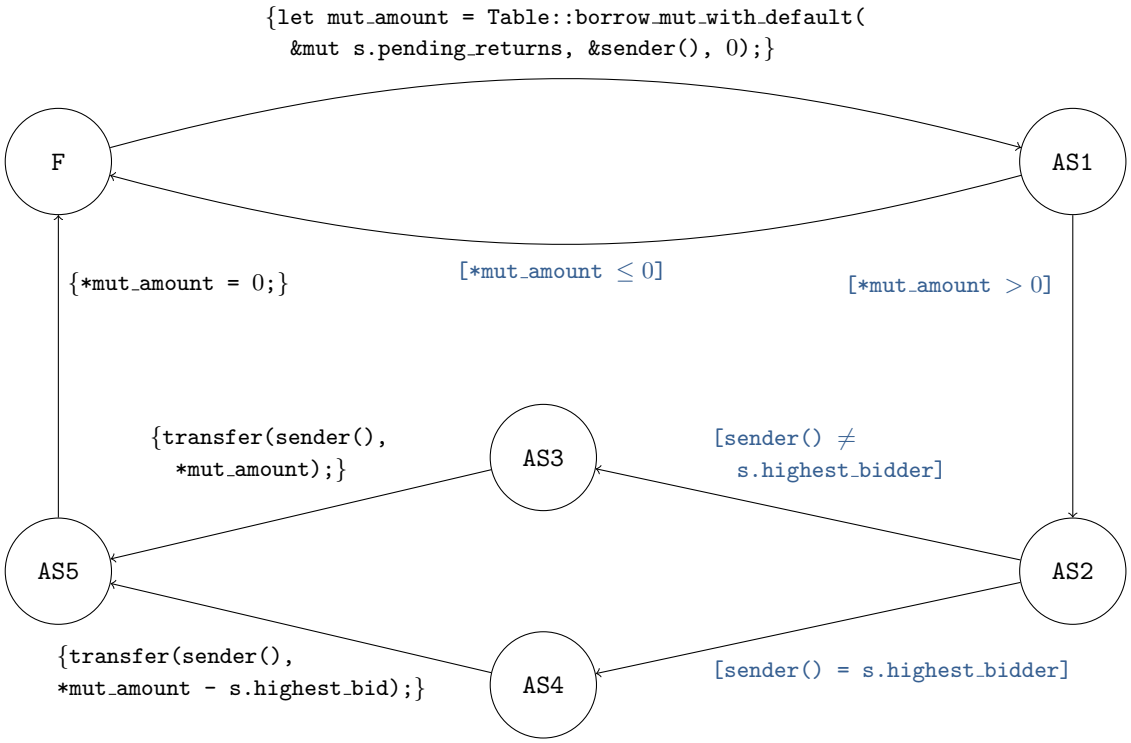

Figure 3.3: Augmented Model of the `withdraw` Transition

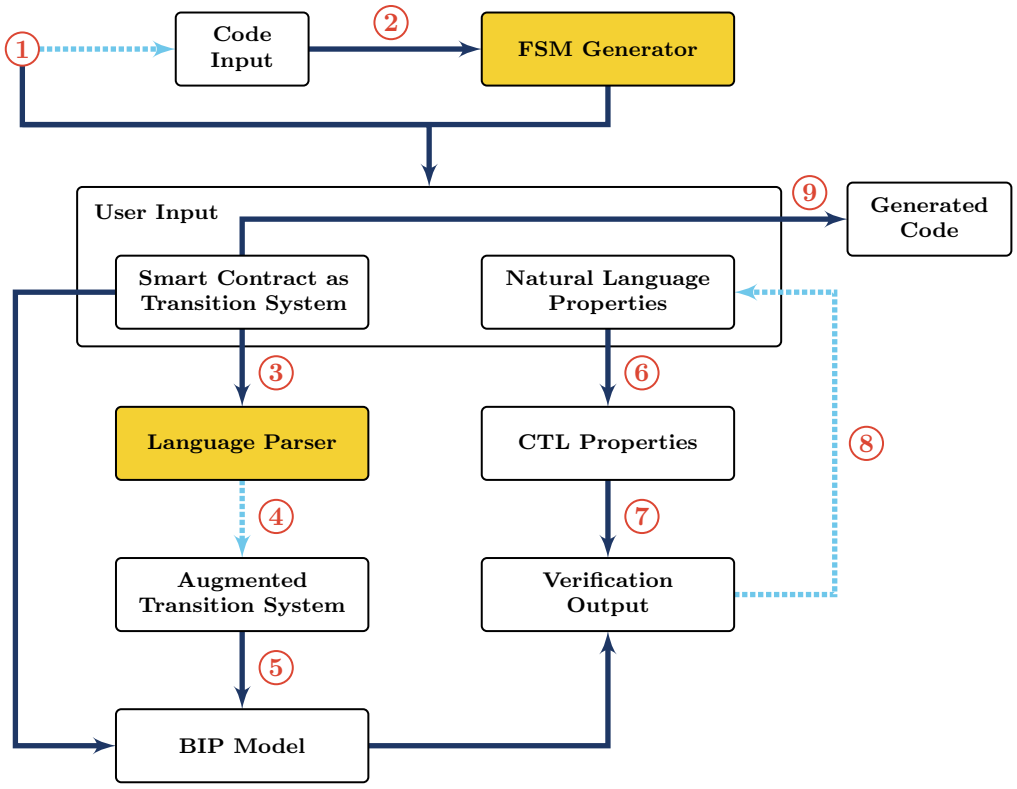

Figure 3.4: Design and Verification Workflow

3.5.4 Augmented Transition System

A transition system is a high-level, abstract representation of the smart contract. The logic of each transition is specified by a smart contract function, which is written by the user. Figure 3.2 gives the Move code for the `withdraw` transition in the `BlindAuction` smart contract. During verification of system properties, each abstract transition is broken down into its individual Move expressions. The result is called the *augmented transition system*. Figure 3.3 gives the augmented transition system generated by the `withdraw` transition code in Figure 3.2. Each Move statement that modifies the state of the smart contract is given a transition, notated by curly brackets. Likewise, each conditional statement is given a transition, notated by square brackets. Note that line 13 is ignored by the augmented transition system as it only acquires access to a global variable, which does not modify the state of the smart contract and is not a conditional check.

3.5.5 VeriMove Workflow

Figure 3.4 shows the steps of the VeriMove design flow. The components of VeriMove that were added to VeriSolid are highlighted in yellow. Mandatory steps are represented by solid arrows, while optional steps are represented by dashed arrows. In step 1), the developer input is given in the form of a transition system and system properties. Like VeriSolid, VeriMove utilizes a graphical user interface (GUI) for creating the transition system representation of the smart contract and provides natural language templates for specifying smart contract properties. If the developer already has Move source code, then they can use the FSM generator to automatically create the

transition system and use the GUI to further refine the model. In step 2), if code input is provided, the FSM generator will convert the code into a preliminary transition system. The verification loop starts at the next step. In step 3), the language parser simplifies the Move statements in each transition into a series of simple Move expressions. Steps 4-8) are identical to the workflow in VeriSolid [104]. Here, the transition system is converted to an augmented transition system, which is converted into a BIP model. The natural language properties are converted into CTL properties. The BIP model and CTL properties are given to an NuSMV solver which verifies the model with respect to the properties. Finally, once the developer is satisfied with the verified model, step 9) generates the equivalent Move source code.

3.6 Operational Semantics for Move

This section outlines the operational semantics necessary for the Move language. VeriMove supports a subset of the Move language. The following are the Move statements that are supported.

$$
\begin{aligned}
\langle \textit{statement} \rangle ::= & \\
& \mid \langle \textit{declaration} \rangle \ \texttt{;} \\
& \mid @\textit{expression} \ \texttt{;} \\
& \mid \texttt{return}\ (@\textit{pure})?\ \texttt{;} \\
& \mid \texttt{if}\ \texttt{(}@\textit{expression}\texttt{)}\langle \textit{statement} \rangle \\
& \quad (\texttt{else}\ \langle \textit{statement} \rangle)? \\
& \mid \texttt{while}\ \texttt{(}@\textit{expression}\texttt{)}\ \langle \textit{statement} \rangle\ \texttt{;} \\
& \mid \texttt{\{}\ (\langle \textit{statement} \rangle) * \texttt{\}}
\end{aligned}
$$

$$
\langle \textit{declaration} \rangle ::= \texttt{let}\ @\textit{identifier}\ (\texttt{:}\ @\textit{type})?\ (\texttt{=}\ @\textit{expression})?
$$

VeriMove supports the following custom types. Note that there is no $\langle event \rangle$ type in Move. Events are specified using resources.

$$
\langle \textit{resource} \rangle ::= \texttt{resource struct}\ @\textit{identifier}\ \texttt{\{}\ (@\textit{identifier}\texttt{:}\ @\textit{type}\texttt{,}\ ) * \texttt{\}}
$$

The operational semantics of the transition system for VeriMove are identical to that of VeriSolid [104]. Likewise, the operational semantics of the supported Move statements are identical to that of Solidity except for the FOR transition, which should be removed since Move does not support for-loops. The following are statement transitions that need to be modified.

$$
\text{VARIABLE} \quad \frac{\text{Decl}(\sigma, \text{Type}, \text{Name}) \rightarrow \langle (\sigma', x) \rangle}{\langle (\sigma, N), \text{let Name: Type;} \rangle \rightarrow \langle (\sigma', x), \cdot \rangle}
$$

$$
\text{VARIABLE-ASG} \quad \frac{\text{Eval}(\sigma, \text{Exp}) \rightarrow \langle (\sigma', x), v \rangle}{\langle (\sigma, N),\ \text{let Name: Type} = \text{Exp;} \rangle \rightarrow \langle (\sigma', x), \{\text{let Name: Type; Name} = v; \} \rangle}
$$

3.7 Empirical Evaluation

3.7.1 Implementation

Similar to VeriSolid, VeriMove was implemented as a web-based application utilizing WebGME [101] and FSolidM [102, 103] as its GUI for specifying the transition system with an NuSMV solver for model verification. The following are the differences between the implementation of VeriSolid [104] and the implementation of VeriMove: 1) VeriSolid is a `NodeJs` application whereas VeriMove is a `React` application, 2) VeriMove separates the language parser as a modular component, and 3) VeriMove implements an FSM generator algorithm for pre-written Move smart contract source code.

3.7.2 Experimental Setup

The performance of VeriSolid and VeriMove are compared on the same set of smart contracts: `ERC20`, `ERC721`, and `BlindAuction`. The contracts `ERC20` and `ERC721` are implementations of the ERC20 Token Standard [51] and ERC721 Non-Fungible Token Standard [52], respectively. The contract `BlindAuction` is an implementation of the blind auction example from the Solidity documentation [143], described in Section 3.4. These contracts were implemented in both VeriSolid and VeriMove. Each contract was given a series of verification properties and once verified the smart contract code was generated. The generated contracts and verification output can be found in the VeriMove GitHub repository[1] along with the implementation.

3.7.3 Results

Both VeriSolid and VeriMove were able to successfully verify the contract properties. Tables 3.1 and 3.2 show the performance results of the verification of VeriSolid and VeriMove on these smart contracts. The meaning of the columns in the tables are as follows. "Contract Length" refers to the number of lines of code (excluding whitespace) in the generated contract. Recall that in both VeriSolid and VeriMove the user-defined transition model is converted into a BIP model and then into an NuSMV model. "Total States" refers to the total number of states in the final NuSMV model. "Reachable States" refers to the number of states in the final NuSMV model that are reachable given the smart contract control flow logic. Finally, "System Diameter" is the depth of the state-space search during verification.

Table 3.1: Verification Performance of VeriSolid

Smart Contract	**Contract Length**	**System Diameter**	**Reachable States**	**Total States**
ERC20	132	7	23	2^{31}
ERC721	155	7	23	2^{32}
BlindAuction	149	11	41	2^{51}

[1] https://github.com/Veneris-Group/VeriMove

Table 3.2: Verification Performance of VeriMove

Smart Contract	Contract Length	System Diameter	Reachable States	Total States
ERC20	213	19	52	2^{63}
ERC721	249	18	59	2^{70}
BlindAuction	215	17	51	2^{62}

3.7.4 Discussion and Limitations

As discussed in Section 3.1, the current verification tools for Move (the bytecode verifier and the Move Prover) can only verify properties within a single function. Model checking allows for the verification of global properties that occur across functions. By successfully generating all contracts and verifying all contract properties, VeriMove has shown that model checking is a feasible approach for verifying global properties.

In terms of performance, Tables 3.1 and 3.2 show that in every contract VeriMove contains more total states, more reachable states, and requires a larger system diameter to verify the contracts compared to VeriSolid. This is due to the fact that Move requires more statements to perform the same functionality compared to Solidity. Thus, its augmented transition system contains more states, and the model checker is required to check more states. This is shown by the length of each contract in Move and Solidity. On average, Move required 35% more lines than Solidity for the same contract. This could be an issue in large contracts as model checking is susceptible to state-space explosion. As discussed in Section 2.4, Move was designed to be easily verified by the bytecode verifier and the Move Prover; a static analyzer and theorem prover, respectively. Therefore, it is not optimized for model checking verification, which is reflected in its results. However, these numbers are not large enough to render model checking an infeasible approach to formal verification in Move.

3.8 Conclusion

This chapter proposed VeriMove, which modified and extended VeriSolid to support the formal verification of Move smart contracts. First, a detailed comparison of the Move and Solidity was given, discussing the main differences between the design of the languages and their trade-offs. Next, the design and workflow of VeriMove was outlined. This included the introduction of the language parser component that allows the VeriSolid framework to be easily extended to other languages, and an FSM generator to alleviate the tedious nature of the model checking process. Additionally, the operational semantics introduced for the verification of a Move smart contract were listed. Finally, standard and widely used smart contracts were implemented in both VeriSolid and VeriMove, comparing their performance. The results showed that model checking is a feasible approach for verifying global properties in Move.

Chapter 4

Gas Optimization of Move Smart Contracts

4.1 Introduction

The *gas* of a smart contract is the cost of executing its logic on the blockchain. The *gas meter* refers to the mechanism that determines how much gas should be charged for a given smart contract. Gas is necessary to pay for the consumption of blockchain resources and to avoid excessive and malicious use of the network. However, it can also pose unexpected costs to developers. The purpose of *gas optimization* is to minimize the gas costs of a smart contract while maintaining equivalent functionality. Gas optimization is a highly researched area in Solidity, but as of the time of writing, there is no equivalent work done for the Move language. This thesis presents the first work in the field of gas optimization in Move.

In this thesis, Aptos [156] is chosen as the underlying platform for studying gas optimization of the Move language. It is currently the leading blockchain platform that utilizes the Move language, and most importantly it was the first blockchain platform to implement a gas meter for the Move language. Other Move-enabled blockchains such as OpenLibra [1] and StarCoin [147] have yet to implement gas meters. Sui [148] has modified core Move and integrated its own features. In the interest of establishing a baseline for gas optimization in the Move language, the analysis of gas optimization on Sui's version of Move should be left as future work.

The contribution of this chapter is summarized as follows:

- Detail the nuances of Aptos's gas meter for Move.
- Enumerate 11 gas optimization patterns for Move.
- Enumerate 5 patterns that reduce contract time complexity, but have no effect on the gas cost.
- Provide concrete examples which implement the proposed optimization patterns and evaluate their effectiveness in typical Move smart contracts. The experiments show that the proposed gas optimization patterns reduce gas consumption in a typical smart contract by 7 - 56%.

The rest of the chapter is organized as follows. Section 4.2 details the gas calculation of Move smart contracts in Aptos. Section 4.3 summarizes prior work on gas optimization in Solidity, and

discusses how it can be applied to Move. Section 4.4 provides gas optimization patterns in Move with concrete examples of these patterns. Section 4.5 identifies patterns that lower the time complexity of a smart contract, but have no effect on the gas cost. Section 4.6 gives empirical results of sample smart contracts to validate, evaluate, and quantify the optimization patterns proposed in Section 4.4. Finally, Section 4.7 concludes the work.

4.2 The Aptos Gas Meter

The *gas* of a smart contract is the cost of storing its items and executing its logic on the blockchain. It is necessary to pay for the use of blockchain resources and to avoid excessive consumption of resources. The *gas meter* refers to the mechanism that determines how much gas should be charged for a given smart contract deployment and invocation.

In Aptos, the native token is APT and the unit of gas is Octa. However, the Aptos gas meter operates using *internal gas units* where

$$100 \textit{ internal gas units } = 1 \text{ Octa } = 10^{-8} \text{ APT} \tag{4.1}$$

This gives a more fine-grain measurement of gas, which is then rounded after all calculations have been completed. When a transaction is submitted, the user must include, among others, the following fields:

- `max_gas_amount`: The maximum number of gas units that the transaction sender is willing to spend to execute the transaction. This determines the maximum computational resources that can be consumed by the transaction.
- `gas_price`: The gas price per unit the transaction sender is willing to pay, usually determined by the market.

When the transaction is executed by the MoveVM, it keeps a tally of the amount of gas used according to the gas meter. The total gas charged for the transaction is

$$\textit{total gas fee} = (\textit{gas used}) \times \texttt{gas\_price} \tag{4.2}$$

If the *gas used* surpasses `max_gas_amount`, then the transaction is aborted. Thus, the maximum amount a user can be charged is (`max_gas_amount`) × (`gas_price`).

The *gas used* by a transaction consists of summing the gas associated with the size of its payload, the virtual machine instructions it executes, and the global storage it accesses. This is explicitly expressed as follows.

$$\textit{gas used} = (\textit{payload gas}) + (\textit{instruction gas}) + (\textit{storage gas}) \tag{4.3}$$

Each gas consumption type is discussed next.

4.2.1 Payload Gas

The *payload gas* is the cost associated with publishing a transaction to the blockchain, *i.e.* the *transaction size*. When publishing a module, the bytecode is stored on the blockchain. Thus, the

transaction size depends on the length of the bytecode. When publishing a transaction script, module functions and their inputs need to be stored on the blockchain. Thus, the transaction size also depends on the size of the input parameters.

Equations 4.4 and 4.5 show the payload gas calculation. Every transaction is automatically charged $1,500,000$ internal gas units ($15,000$ Octa). This is sometimes called *intrinsic gas*. If the bytecode is greater than 600 bytes, the transaction is charged $2,000$ internal gas units for each of the excess bytes [9]. This is to prevent abuse of the network.

$$\textit{large tx penalty} = \max(0,\ (\textit{tx size} - 600 \text{ bytes}) \times 2,000) \tag{4.4}$$

$$\textit{payload gas} = 1,500,000 + \textit{large tx penalty} \tag{4.5}$$

4.2.2 Instruction Gas

The *instruction gas* is the gas associated with the execution of the virtual machine operations of a transaction. Each instruction of the MoveVM has been assigned a gas cost. Typically, each operation charges both a fixed *base gas* and a variable amount of gas proportional to the parameter sizes associated with the operation [91].

Since the MoveVM is a 64-bit stack-based virtual machine, instructions operate on *exactly* 64 bits. Thus, operations such as arithmetic, bitwise, boolean, and comparison charged per 64 bits. As a result, from the perspective of the MoveVM, there is no distinction between `u8` and `u64` integers. Operations on `u8` and `u64` integers will result in the same gas consumption. Conversely, operations on `u128` integers will generally require more gas, since they require at least one additional register.

Move modules are not executed when published to the blockchain. However, the instruction gas associated with a module function will be considered as the amount of gas it consumes when a transaction script executes it.

4.2.3 Storage Gas

The *storage gas* is the gas associated with accessing global storage on the blockchain. There are four types of interactions with resources in global storage. *(i)* A resource can be created or instantiated via `move_to`. *(ii)* The fields of a resource can be read via `borrow_global`. *(iii)* The fields of resources can be written to or updated via `borrow_global_mut`. *(iv)* A resource can be deleted or deallocated via `move_from`. Currently, creation, reading, and writing consume gas, whereas deletion does not consume any gas. Aptos has expressed that in the future it may refund gas for the deallocation of resources. At the time of writing, however, this has not yet been implemented.

As discussed in Section 2.4.1, an *item* is a generic term used for any key-value pair in global storage. The utilized gas consumed by each access type (except deletion) is calculated as follows [92].

$$\textit{utilized gas} = \textit{items} \times (\textit{per item gas}) + \textit{bytes} \times (\textit{per byte gas}) \tag{4.6}$$

The first term is the base cost for accessing an item. This amount is the constant up-front cost of the access type. The second term accounts for the size of the item, charging gas proportional to the size of the item accessed. Note that *bytes* in Equation 4.6 refers to the total number of bytes in all

fields of a resource, even if only once field was accessed. Table 4.1 gives the amount of gas charged for each access type [92].

Table 4.1: Storage Gas Fees

Operation	**Internal Gas Units**	**Octas**
per-item read	$300,000$	$3,000$
per-item write	$300,000$	$3,000$
per-item create	$300,000$	$3,000$
per-byte read	300	3
per-byte write	$5,000$	50
per-byte create	$5,000$	50

Original Storage Gas Calculation

Consider the calculation of the storage gas associated with the global read operation. Note that this description holds analogously for the other access types. If the smart contract does not call `borrow_global`, then the storage gas charged is 0. However, by the design of Equation 4.6, if any read operation is performed, then the minimum amount of gas charged for that access is the *per item read*, or $300,000$ internal gas units. This is denoted by g_{min}. Aptos sets a maximum allowable utilized gas, g_{max}, to $100 \times g_{min}$. In the case of a global read, this is $30,000,000$ internal gas units. If the gas utilized by the read operation surpasses g_{max}, then an out-of-gas exception is thrown and the smart contract is aborted. If the gas utilized by the read operation falls between g_{min} and g_{max}, then let g denote this amount and $u = g/g_{max}$ be the ratio of gas utilized. The following gives the formula, called the *utilization curve*, for the storage gas charged to a smart contract that utilizes g amount of gas.

$$\texttt{storage\_gas}(u) = g_{min} + \frac{b^u - 1}{b - 1}(g_{max} - g_{min}) \tag{4.7}$$

In particular, Aptos uses a base $b = 8192$. This function is graphed in Figure 4.1. The vertical axis is divided by g_{max} in order to make the units storage access agnostic. Note, that there is an independent utilization curve for each of the global access types: read, write, and create.

Current Storage Gas Calculation

On June 13, 2023, Aptos depreciated the utilization curve [79]. In the current version of the gas meter, the *utilized gas* is the amount of gas charged for the access operation, thus the new utilization curve is the identity function. Using the above notation, $\texttt{storage\_gas}(u) = u \times g_{max} = \textit{utilized gas}$.

One reason the utilization curve was depreciated is that it allowed for unorthodox gas optimization techniques. For example, consider a smart contract which requires a large number of read operations. Since the reading and writing utilization curves are independent, one could save on gas by splitting half of the reads into write operations. Under the original utilization curve, this could potentially save a huge amount of gas if the read operations were past 60% utilization. In the new utilization curve, this technique would result in no difference in storage gas costs.

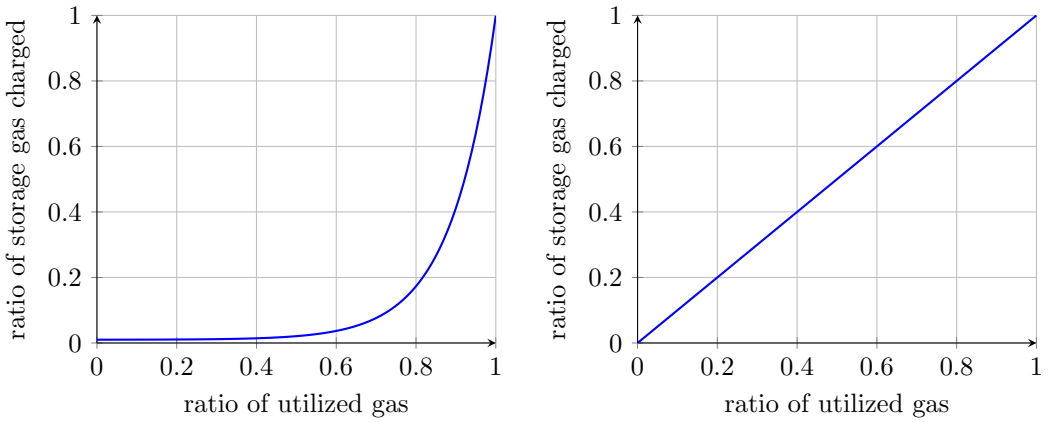

Figure 4.1: Aptos's Old (left) and New (right) Utilization Curves for Global Storage Accesses

4.3 Related Work

To date, this thesis is the first work that analyzes gas optimization for the Move language. The majority of the research on gas optimization is done for Solidity on the Ethereum blockchain [99, 139, 140].

Broadly, there are three ways to frame gas optimization. The first is in the field of vulnerability detection. An *out-of-gas* exception occurs when the smart contract uses more gas than the allowed gas limit. In some instances, this can be used as a denial-of-service attack on the smart contract. This is considered gas optimization since the parts of smart contracts susceptible to this attack inherently use a lot of gas. Fixing this vulnerability results in a more gas-efficient smart contract. The authors of [63] use symbolic execution to detect specific out-of-gas exception patterns. The authors of [98] use fuzzing techniques in order to find inputs that cause a high gas output.

A second way to frame gas optimization is to abstract the problem to code optimization [3, 26, 36, 54, 113, 115]. If the smart contract code is optimized, then the virtual machine will perform fewer operations, and thus gas consumption will be reduced. The authors of [36] developed the static analyzer tool, GASPER, which applies parallelized symbolic execution to Solidity bytecode to identify specific patterns such as dead code, opaque predicates, and expensive operations in loops. Lastly, the authors of [115] identify three loop patterns to reduce the number of virtual machine operations and global storage accesses.

Finally, gas optimization can be viewed as its own unique subject. The gas meter is not isomorphic to "the number of virtual machine operations". For example, addition and division are often given the same gas price per operation, even though division typically requires more virtual machine operations to compute. Some techniques will make the smart contract less efficient in the strict sense of virtual machine operations but nonetheless, reduce gas consumption. The authors of GASPER extended their work to develop GasReducer [37], which identifies 24 different optimization patterns in bytecode. The authors of [2] created a tool called GASOL, which targets optimizing gas consumption associated with the usage of storage operations by replacing multiple accesses to global memory with local variable operations.

The remaining sections consider the application of this large body of research to the Move language. Generally, code optimization techniques that operate on the source-code level can be

directly applied. However, code optimization techniques that operate on the bytecode level cannot be directly applied, since the EVM and MoveVM are fundamentally different. Work on the out-of-gas vulnerabilities in Ethereum generally detect malicious use of the `fallback` function. However, Move does not have dynamic dispatching, so these techniques do not apply. Finally, some techniques related to reducing the number of global storage accesses can be modified for Move.

4.4 Gas Optimization Patterns

Generally, gas optimization stands very close to time and space complexity optimization. The aim is to minimize the *number of virtual machine operations* and the *amount of global storage accessed* during a transaction. However, there is often a trade-off between time and space complexity with many ways to implement the same specification. Decreasing memory use may result in more virtual machine operations, and decreasing virtual machine operations may require an increase in memory use; in which case, it is not clear how to minimize gas consumption.

This section gives both general principles and concrete design patterns for optimizing the gas consumption of Move smart contracts on Aptos.

4.4.1 Payload Gas

The gas associated with the payload is typically much less than the gas associated with instructions and global storage. Thus, for most applications, its contribution is negligible. However, if the payload greatly exceeds 600 bytes, then it may cause a noticeable increase due to Aptos's large transaction penalty. The following general principles are given for payload gas optimization.

1) Minimize the Length of Modules: The code of a published module is stored on the blockchain, which consumes gas. Minimizing the length of the module, *i.e.* the number of bytes required to store its bytecode, reduces the total gas cost. Some instances include removing dead or unnecessary code, reducing redundant code, avoiding unnecessary additional variables, using standard libraries, and separating out the module into multiple smaller modules. Note that comments do not have an effect on this calculation, since the blockchain stores the bytecode and not the source code.

2) Minimize the Size of Parameters in Transaction Scripts: When executing a transaction script, the payload may contain the values of the parameters given by the user, which are stored on the blockchain and thus consumes gas. Minimizing the number and size of these parameters will reduce the total gas cost. For example, combining many small functions that require a lot of parameters into one larger function, and also avoiding passing resources as parameters into functions.

4.4.2 Instruction Gas

The gas associated with virtual machine instructions is akin to the time complexity of the smart contract. In general, less virtual machine operations results in less gas consumption. However, this is not always possible without sacrificing the necessary functionality. The following general principles are given for instruction gas optimization.

```
public entry fun short_circuit(addr: address) {
    if (cheap_condition() && expensive_condition()) {
        //
        // operations
        //
    };
}
```

```
public entry fun short_circuit(addr: address) {
    if (cheap_condition() || expensive_condition()) {
        //
        // operations
        //
    };
}
```

Figure 4.2: Short Circuit

1) Limit Function Calls: One of the most expensive instruction gas operations are function calls [91]. The gas saved from the lack of a function call is always larger than the gas gained from a larger module size. Therefore, abstracting smart contract functionality into helper functions should be avoided as much as possible. For instance, it is very common for programmers to write getter functions which are a single line or very few lines of code. Removing these is a small change, which saves a large percentage of gas (see Table 4.2). However, having large and complicated functions makes testing more difficult. It is up to the developer to find an acceptable balance.

2) Minimize Vector Element Operations: Vector operations charge gas on a per-element basis and are more expensive than operations on local variables. Thus, accessing vectors can be treated like accessing the global state, which means the principles 1) and 3) from Section 4.4.3 apply analogously for vectors. If one wishes to operate on an element from a vector more than once, then it should be copied to a local variable and then updated after all calculations are performed. Lastly, a vector element should be directly updated, rather than deleted and recreated.

3) Short Circuit: When using the logical connective `AND` (`&&`), if the first expression evaluates to `false`, then the second expression will not be evaluated. Likewise, when using the logical connective `OR` (`||`), if the first expression evaluates to `true`, then the second expression will not be evaluated. Thus, continued expressions in if-statements and while-loops should be *ordered by increasing gas cost*. If a cheap expression short-circuits the condition check, then gas is saved on evaluating the more expensive expressions. Figure 4.2 shows an example with the `AND` and `OR` connectives.

4) Write Values Explicitly: Since all virtual machine operations consume gas, any constant value should be written explicitly rather than implicitly computed via the smart contract.

5) Avoid Redundant Operations: Since all virtual machine operations consume gas, redundant operations should be avoided. For example, Move has a bytecode verifier that checks for common vulnerabilities such as integer overflow/underflow. Thus, checking for this in a smart contract is redundant and unnecessary.

4.4.3 Storage Gas

The gas associated with global storage is akin to the space complexity of the smart contract. In general, less accesses to global storage will result in less gas consumption. Moreover, storage gas will typically dominate both payload and instruction gas. Thus, it should be given the most attention when optimizing smart contract gas. The following general principles are given for storage gas optimization.

1) Operate on Local Variables: Operating directly on resources and resource fields consumes significantly more gas than operating on local variables. Whenever a smart contract is operating on the values of a resource, its ownership should be borrowed by a local variable. If necessary, those values can be transferred back to the resource at the end of the function.

```
public entry fun bad_resource_write(addr: address)
acquires MyResource{
    while (/* condition */) {
        let resource =
                    borrow_global_mut<MyResource>(addr);
        //
        // operate on resource.field
        //
    };
}
```

```
public entry fun good_resource_write(addr: address)
acquires MyResource {
    let resource = borrow_global_mut<MyResource>(addr);
    let intermediate = resource.field;
    while (/* condition */) {
        //
        // operate on intermediate
        //
    };
    resource.field = intermediate;
}
```

Figure 4.3: Loop Refactor - Operating on Local Variables

```
public entry fun bad_variable_storage(addr: address)
acquires MyResource {
    let resource = borrow_global<MyResource>(addr);

    let x8:  u8  = resource.x8;
    let x32: u64 = resource.x32;
    let x24: u64 = resource.x24;
}
```

```
public entry fun good_variable_storage(addr: address)
acquires MyResource {
    let resource = borrow_global<MyResource>(addr);
    let x:   u64 = resource.x;

    let x8:  u8  = (x & 0xF);
    let x32: u64 = ((x >> 8) & 0xFFFF);
    let x24: u64 = ((x >> 40) & 0xFFF);
}
```

Figure 4.4: Variable Packing

Figure 4.3 shows an example of implementing this principle. When using a resource field value in a loop, one should first store its field value in an intermediate local variable, and do all loop operations on this local variable. At the end of the function, the resource field is updated. This limits the number of accesses to the resource to a maximum of two, rather than the number of loop iterations.

2) Variable Packing: There are two facts about Move's gas meter that this pattern utilizes. First, global storage access consumes the most gas of any operation. Thus, one should aim to make as few as possible. Second, when accessing a resource, the per-byte charge consists of all fields in the resource, not just the ones that were accessed. *Variable packing* refers to storing many variables in a single resource field to optimize the efficiency of each storage access.

The example given in Figure 4.4 contains variables `x8`, `x32`, and `x24` that will only ever store 8, 32, and 24 bits of information, respectively. The naive way of storing these variables is to separate each into its own field. However, storage gas can be saved by packing these variables into a single `u64` integer. The per-byte gas of the resource access in the left figure is more expensive than the instruction gas cost of the operations in the right figure.

3) Resource Update: There is currently no incentive to deallocate global storage. Thus, in order to minimize gas consumption, unused resources should be overwritten rather than deallocating and creating new resources. Figure 4.5 shows an example of this optimization.

4) Read Instead of Write: Writing to a resource is more expensive per byte than reading (see Table 4.1). Thus, any resource access using `borrow_global_mut` that does not update the resource should be replaced with `borrow_global`.

4.5 Non-optimization

In addition to giving principles that will minimize gas consumption, it is equally useful to know what does not affect gas consumption.

```
public entry fun bad_resource_update(new_value: u128)
acquires MyResource {
    // transfer from global storage
    let resource = move_from<MyResource>(&signer);

    // deallocate
    MyResource {value: _} = resource;

    // create new resource
    move_to<MyResource>(&signer, MyResource {
        value: new_value
    });
}
```

```
public entry fun good_resource_update(new_value: u128)
acquires MyResource {
    let resource = borrow_global_mut<MyResource>
                                        (&signer);
    resource.value = new_value;
}
```

Figure 4.5: Resource Update

1) Equivalent Operations: Basic arithmetic operations {*add*, *sub*, *mul*, *div*, *mod*} cost the same amount of gas, even though division, for example, typically requires more computation than addition. Bit-wise operations {*and*, *or*, *xor*, *left shift*, *right shift*} cost the same amount of gas. Similarly, the comparison operations $\{<, >, \leq, \geq\}$ cost the same amount of gas, and the operations $\{=, \neq\}$ both cost slightly less.

2) Reads/Writes are Never Partial: Reading or writing only one field from a resource may save a little gas with respect to the instruction gas, but it does not save any gas with respect to storage gas. When `borrow_global_mut` is called and a resource field is updated, the per-byte cost of the update is for the entire resource, not just the updated field.

3) u8 Integers: There is no difference between doing operations with `u8` integers and `u64` integers, both locally and globally. This is because MoveVM has 64-bit registers. However, doing operations with `u128` integers will cause an increase in gas usage.

4) Ordering Fields in Resources: The order of the fields of the resource does not matter. All fields of a resource occupy their own space in storage. One can pack variables within a field, but not between fields.

5) Deallocation of Resources: Currently, Aptos is lacking any mechanism for rewarding the destruction of resources via `move_from`. Although, they have expressed interest in adding this in the future.

4.6 Experiments

This section presents the results of an experimental evaluation of the gas optimization patterns that were identified in Section 4.4. Through this, the gas savings of each gas optimization pattern are validated, evaluated, and quantified. While Move is rapidly gaining popularity, it has yet to become a standard in decentralized application development. As a result, there are fewer examples of smart contracts deployed to Aptos as well as a lack of developer tools and standardized benchmarks. Thus, a set of sample smart contracts have been created[1] to isolate each gas optimization pattern. Table 4.2 compares the gas consumption of the original and optimized smart contracts measured in Octa. The rightmost column is the percent decrease of gas in the optimized contract. The patterns "Minimize the Length of Modules", "Minimize the Size of Parameters in Transaction Scripts", and "Avoid Redundant Operations" were omitted as these are general principles rather than concrete design patterns.

[1] https://github.com/Veneris-Group/Move-Gas-Optimization-Patterns

Table 4.2: Gas savings comparison of optimization patterns

Gas Optimization Pattern	**Original Cost**	**Optimized Cost**	**Gas Savings Percent**
Limit Function Calls	47	26	44.7
Minimize Vector Element Operations	41	30	26.8
Short Circuit	2372	2	99.9
Write Values Explicitly	410	2	99.5
Operate on Local Variables	62	27	56.5
Variable Packing	746	630	15.5
Resource Update	130	120	7.7
Read Instead of Write	3663	56	98.5

The magnitude of the gas decrease cannot be used to compare the effectiveness of each optimization pattern as it is dependent on the number of virtual machine operations, the number of global memory accesses, and the size of local and global variables in the particular smart contract. The percentage decrease is more stable with respect to changes in the smart contract and gives a better measurement of the effectiveness of a pattern.

The gas optimization patterns "Minimize Vector Element Operations", "Short Circuit", "Write Values Explicitly", and "Read Instead of Write" heavily depend on the particular smart contract. Using "Short Circuit" in Figure 4.2 as an example, the gas savings depends on the cost of the statement `expensive_condition` relative to `cheap_condition`. The results validate that these gas patterns reduce gas consumption, but their effectiveness depends on the application.

The gas optimization patterns "Variable Packing" and "Resource Update" depends on the size of the global storage that is being accessed. The sample smart contracts use moderately sized resources, and the percent decrease is small. For applications with large global variables, these patterns are effective at reducing gas consumption. However, for applications accessing a small number of or modestly sized resources, the impact may be negligible.

Lastly, the gas optimization patterns "Limit Function Calls" and "Operate on Local Variables" are the most stable with respect to changes in the smart contract. Both patterns result in a substantial percent gas decrease.

4.7 Conclusion

Move is a new smart contract language that offers superior security and verifiability compared to existing smart contract languages. As it becomes more popular, gas optimization for Move smart contracts will become more important to developers. This thesis is the first to apply the vast research on gas optimization in Solidity to the Move language using Aptos as the underlying platform. This chapter detailed Aptos's gas meter, proposed 11 gas optimization patterns, and identified 5 patterns that decrease the time complexity of a smart contract but have no effect on gas consumption. Sample contracts were implemented for each proposed gas optimization pattern. The results showed that the proposed gas optimization patterns reduce gas consumption on a typical Move smart contract by $7 - 56\%$.

Chapter 5

Automated Auditing of TOD Vulnerabilities

5.1 Introduction

The *transaction order dependency* (TOD) vulnerability is present when a smart contract depends on the global state of the blockchain. This can be exploited as a result of two common features of blockchain platflorms. First, when a smart contract transaction is submitted to the network, it is first added to a public waiting area pending validation, called the *mempool*. Eventually, the transaction is selected by a validator, grouped into a block, and its instructions executed by all nodes in the blockchain network. Second, a *priority fee* is included in each transaction, which is a tip given to the validator of the transaction. This incentivizes validators to choose transactions in the mempool with the highest priority fee [11, 50, 142, 149]. In the *front running* attack, an attacker observes honest transactions in the public mempool which contain TOD vulnerabilities. Then, the attacker injects their transaction first by including a large priority fee, thus modifying the global blockchain state and consequently the final output of the honest transactions in their favor [12, 153].

This thesis investigates the automated detection and rectification of the TOD vulnerability. In detail, a static analysis approach is proposed to locate and rectify TOD vulnerabilities. In particular, an algorithm is proposed that extracts the data dependencies of a smart contract to determine how a change in its state affects the transaction outcome. This algorithm is implemented as a prototype tool for Solidity. It uses Slither [54], a static analyzer for Solidity, to extract control and data dependencies of smart contracts. This prototype tool is evaluated on a benchmark suite of 51 Solidity smart contracts. The results show that the proposed approach rectifies the vulnerabilities with only a few changes to the original smart contract.

In summary, this chapter makes the following contributions:

- The problem of automated detection and rectification of the TOD vulnerability is studied.
- A novel algorithm is proposed to automatically detect and rectify TOD vulnerabilities.
- A prototype implementation of the proposed approach is built.
- A smart contract benchmark suite of 51 smart contracts is developed to validate and evaluate

the proposed methodology. Experiments show this algorithm can successfully detect and rectify TOD vulnerabilities with few modifications to the original smart contract.

The rest of the chapter is organized as follows. Section 5.2 presents an overview of the TOD vulnerability with a motivating example. Section 5.3 describes the technical elements of the proposed approach to automatically locate and rectify this vulnerability. Section 5.4 presents a prototype implementation of the proposed approach for Solidity smart contracts and the empirical results of evaluating the prototype using a smart contract benchmark suite. Sections 5.5 discusses related work. Finally, Section 5.6 concludes the work.

5.2 Background and Motivating Example

This section analyzes the TOD vulnerability and provides a particular example of how it can be exploited by an attacker. Afterwards, a general mechanism is described to identify such a vulnerability and mitigate it.

5.2.1 The Cause of a TOD Vulnerability

A TOD vulnerability is present when a transaction is dependent on the global state of the blockchain. While an honest transaction is pending in the mempool, an attacker injects a transaction which modifies the global state. Then, after the honest transaction is validated, it is executed in a different environment than intended, resulting in a different outcome. If the attacker constructs their transaction correctly, it can result in financial gain at the expense of the honest transaction.

In detail, suppose σ_t denotes the global state of the blockchain after block t. Let Υ denote the transition function such that $\sigma_{t+1} = \Upsilon(\sigma_t, T)$ where T is any transaction. Suppose T_H is an honest transaction and T_A is the attacker transaction. In general, $\Upsilon(\Upsilon(\sigma_t, T_H), T_A) \neq \Upsilon(\Upsilon(\sigma_t, T_A), T_H)$. The honest transaction is expecting to execute under the state σ_t, but instead executes under a state modified by the attacker, *i.e.* $\Upsilon(\sigma_t, T_A)$. If the output of T_H depends on the global state of the blockchain, then T_A can maliciously change its state.

5.2.2 The Price Gouging TOD Vulnerability

Price gouging is an example of how an attacker can use the TOD vulnerability for financial gain. Figure 5.1 gives an example of such an attack in a marketplace smart contract written in Solidity [144] smart contract language. Clients call the function `buy` to purchase an amount of tokens that must be less than the contract inventory, stored in the variable `inventory`. The purchased amount of tokens is computed by dividing the value of `msg.value` by the value of the contract variable `cost`. However, utilizing a front-running attack, the value of `cost` can be increased by the contract owner, by maliciously calling the function `increasePrice` while an honest client transaction is pending approval. Therefore, this will result in a loss to the client where the obtained amount of tokens will be affected by the increase cost of a single token.

```
contract MMarketPlace {
  address owner;
  uint private cost = 100;
  uint private inventory = 30;

  event Purchase(address _buyer, uint _amt);

  function increasePrice(uint increaseCost) {
    require( msg.sender == owner );
    cost += increaseCost;
  }

  function buy() returns(uint) {

    uint amt = msg.value / cost;
    require( inventory > amt );
    inventory -= amt;
    emit Purchase(msg.sender, amt);
    return amt;
  }
}
```

```
contract MMarketPlace {
  address owner;
  uint private cost = 100;
  uint private inventory = 30;

  event Purchase(address _buyer, uint _amt);

  function increasePrice(uint increaseCost) {
    require( msg.sender == owner );
    cost += increaseCost;
  }

  function buy(uint costExpected) returns(uint) {
    require(cost == costExpected);
    uint amt = msg.value / cost;
    require( inventory > amt );
    inventory -= amt;
    emit Purchase(msg.sender, amt);
    return amt;
  }
}
```

Figure 5.1: Example of the Price Gouging TOD Vulnerability (left) and its Rectification (right)

5.2.3 Locating TOD Vulnerabilities

The first objective of this chapter is to locate TOD vulnerabilities in smart contracts. Since changing the order between the client transaction and attacker transaction affects the final output, this means that the client transaction outcome is dependent on one or more state variables that the attacker transaction modifies. Thus, to locate TOD vulnerabilities, state variables are found that affect the outcome of an honest transaction and that can be altered through setter functions that attackers can call to manipulate the smart contract state. For instance, in the smart contract on the left of Figure 5.1 the outcome of the transaction calling the function `buy` (the amount of inventory purchased, stored in the variable `amt`) is affected by the variable `cost` that can be increased by the setter function `increasePrice`.

5.2.4 Rectifying TOD Vulnerabilities

One method of rectifying a TOD vulnerability is to add a guard statement to check whether the state of a smart contract is has changed since the transaction was submitted. In particular, this will allow clients to pass values for the states variables that can be altered. Then, in the body of the called function, `require` statements are added to ensure that the current values of the state variables correspond to the expected values passed by the clients. For instance, the right of Figure 5.1 gives the rectified version of the smart contract on the left of the figure. Notice that in the final correct version an additional parameter is added to the function `buy`, *i.e.* `costExpected`, that has the same type as `cost`, *i.e.* `uint`. Then, in the body of `buy`, a `require` statement is added as a guard to check whether the current value of `cost` corresponds to the passed value of `costExpected`. Thus, if the global state of the blockchain was modified in such a way that affected the output of the `buy` function, then the smart contract will abort the transaction.

5.3 Analysis Approach

This section presents the proposed methodology to automatically locate and rectify TOD vulnerabilities in smart contracts.

5.3.1 Location Algorithm

Algorithm 1 A procedure for locating TOD vulnerabilities

```
1: procedure ListDependencies(F, G)
2:     Q ← {}
3:     for each f ∈ F
4:         for each p ∈ outputParams(f)
5:             G' = pointToAnalysis(f, p, G)
6:             for each x ∈ G'
7:                 if findSetter(x, F)
8:                     Q[f] ← x ⊎ Q[f]
9:     output Q
10: end procedure
```

The proposed approach aims to locate the vulnerability in a smart contract and transform the contract's code to rectify the vulnerability without changing the functionality of the contract. Alias and static code analysis are leveraged to compute relationships between the outcomes of `public` functions that can be called by users and *state variables* that can be manipulated through setter functions. Algorithm 1 presents the proposed procedure to locate TOD vulnerability in smart contracts. Given the lists of public functions $\mathcal{F}$ and state variables $\mathcal{G}$ extracted from the *abstract syntax tree* of a smart contract, the procedure `ListDependencies` computes, for each function `f` in $\mathcal{F}$, the set of state variables $\mathcal{Q}[\mathsf{f}] \subset \mathcal{G}$ that the outcome of `f` depends on and that can be modified by setter functions. In particular, `ListDependencies` computes for each output parameter of `f` (*i.e.* `outputParams(f)`) the state variables that it depends on, $\mathcal{G}'$, using the procedure `pointToAnalysis` that computes dependency relationships between variables in the context of a given function. For each variable g in $\mathcal{G}'$, the procedure uses `findSetter` to check whether there exist a public setter function that modifies the value of g. The proposed algorithm leverages the precision of the above procedures to find the optimal subset of state variables checks for each function, denoted the *dependency variables* of the function.

5.3.2 Rectification Algorithm

Once the dependency variables are identified for each public function, the proposed repair mechanism consists of inserting, for each dependency variable, an input parameter that has the same type in the corresponding function signature. Subsequently, a `require` statement is inserted in the function's body as a guard to check whether the current value of the dependency variable corresponds to the value passed as parameter by the client's transaction that is calling the function. This checks that the state of the dependency variables has not changed since the time when the client issued its transaction.

5.4 Empirical Evaluation

5.4.1 Implementation and Experimental Setup

Implementation

A prototype tool is developed, implementing the algorithm described in Section 5.3 that takes as input a Solidity smart contract. This tool utilizes the SLITHER [54] static analyzer framework for Solidity to construct Control Flow Graphs and dependency relationships in a given Solidity smart contract. Note that in this implementation, *public functions* refer to functions with signatures that contain either of the Solidity keywords `public` and `external`. The open-source code for the implementation is available on GitHub.[1]

Experimental Setup

The experiments are run on an Intel Core i3-4170 3.7GHz CPU, 8GB of DDR3 RAM, 256GB SSD machine running Linux Ubuntu 20.04.3LTS operating system in a local network environment.

5.4.2 DataSet Collection

The experiments comprise of a benchmark suite of 51 Solidity smart contracts constituted of three datasets. The first dataset is constituted of 11 contracts obtained from open-source GitHub repositories. It includes the reference smart contract used in [57] to evaluate static analysis tools for locating TOD vulnerabilities. It also includes two smart contracts extracted from Etherscan [53] without TOD vulnerabilities to test that the implementation does not flag non-existing TOD vulnerabilities. The second dataset is constituted of 20 contracts obtained from the benchmark contracts used in [100]. The third dataset is constituted of 20 contracts obtained from the benchmark contracts [124]. The complete dataset can be found on the Github repository with the implementation.

5.4.3 Results

The prototype tool runs with the benchmark suite of 51 Solidity smart contracts. Table 5.1 reports the experimental results. The first three columns in Table 5.1 list some characteristics of the benchmark suite, *i.e.* the contract name, the number of lines of code (`loc`), and the number of functions (`nof`). The last three columns in Table 5.1 list data concerning the application of the proposed tool. The column `nTOD` lists the number of TOD vulnerabilities the proposed tool locates in each contract. The column `loc'` lists the number of lines in contract's code in each rectified contract. Finally, the column `diff` lists the number of lines of code that were either added to or modified in the original contract. This shows that the code transformation to rectify the smart contracts is lightweight.

The smart contract `BitCash` is the reference contract that was used in [57] to test static analysis tools in locating TOD vulnerabilities. The proposed tool is able to report the TOD vulnerability in this contract and rectify it. The two smart contracts `Sale2` and `Crowdsale` do not have TOD vulnerabilities and they are used to test that the implementation does not give false negatives. The two smart contracts `Sale2-Vulnerable` and `Crowdsale-Vulnerable` are modified versions of `Sale2` and `Crowdsale` contracts, respectively, where a TOD vulnerability is inserted in each contract.

[1] https://github.com/Veneris-Group/TOD-Location-Rectification

Table 5.1: Empirical Results of TOD Rectification

Contract Name	loc	nof	nTOD	loc'	diff
BitCash	28	2	1	29	2
Sale	71	6	1	72	2
MMarketPlace	21	2	1	22	2
Purchase	31	3	1	32	2
YFT	79	7	2	81	4
TTC	78	7	2	80	4
PrivateSale	40	5	1	43	4
Sale2	125	10	0	125	0
Crowdsale	92	7	0	92	0
Sale2-Vulnerable	129	11	1	130	2
Crowdsale-Vulnerable	96	8	1	97	2
DSTContract	1268	39	9	1278	10
GenesMarket	1262	19	6	1265	3
F3DClick	1926	35	9	1935	9
KnowTokenCrowdSale	228	5	1	229	1
GrowToken	176	14	4	180	4
TrustZen	245	6	8	249	4
GetToken	81	5	2	82	1
Slotthereum	252	21	4	254	2
MyAdvancedToken7	125	12	6	128	3
Crowdsale2	69	10	2	70	1
SaleFix	692	63	2	693	1
Token	144	13	2	145	1
HQ	209	15	4	211	2
Oasis	290	14	7	297	7
SolidStamp	360	20	4	362	2
FairyFarmer	144	22	6	150	6
LISCTrade	399	34	2	400	1
InvestToken	936	92	4	940	4
FoMo3Dshort	1927	78	9	1936	9
DACMI	461	43	7	468	7
Lottery	45	6	1	46	1
kernelFun	118	6	3	121	3
Dickael	270	22	2	274	4
TetherToken	455	11	2	458	3
LinkToken	355	5	1	356	1
TokenSale	61	4	0	61	0
HuanCasino	151	8	1	153	2
MITxSubscriptionPayment	341	2	1	342	1
MultiPadLaunchApp	525	46	3	528	3
TokenUseV2	300	18	6	312	12
Sociol	92	12	1	93	1
Stableupgradeproxy	362	23	3	365	3
GravatarRegistry	67	4	1	68	1
NeoUsd	151	15	2	153	2
GAMCasino	188	13	2	190	2
FabricCrowdSale	105	9	1	106	1
PonziCoin	86	5	3	89	3
CliqStaking	358	28	4	362	4
Betting	225	15	1	226	1
LadaCoin	49	1	1	50	1

5.4.4 Limitations and Discussion

In the current setup, the proposed implementation rectifies all detected vulnerabilities, however, it might be the case that some vulnerabilities are not exploitable and repairing them may not be necessary. For instance, this can occur in the case where public users trust a smart contract's owner and they are assured that the contract's state will not be manipulated while their transactions are pending approval.

Another limitation in the proposed implementation is that the static analysis tool SLITHER does not consider inlined assembly statements within the smart contract code. Thus, the proposed implementation might miss dependencies between a transaction's outcome and state variables that can be manipulated.

5.5 Related Work

5.5.1 Analysis of Smart Contracts

A number of papers have investigated the problem of automated detection of common vulnerabilities in smart contracts. This prior research is either based on symbolic execution engines, *e.g.* [14, 73, 88, 118, 163], static analysis, *e.g.* [64, 83, 161, 164], or dynamic analysis, *e.g.* [66]. The past work based on symbolic execution and dynamic analysis can only establish correctness for *bounded* executions of smart contracts. On the other hand, the works based on static analysis are designed to expose certain coding patterns that are prone to critical vulnerabilities and do not establish full functional correctness. The most closely related work to this thesis is SECURIFY [164] and OYENTE [14], which investigate TOD among the patterns of vulnerabilities they detect. However, it was shown recently in [57], that those tools may produce false positives and/or false negatives, which is not the case in the proposed algorithm.

5.5.2 Automated Repairs of Smart Contracts

There is not much work on automated repairs of bugs in smart contracts. In [117], the authors propose an approach to automatically repair four different vulnerabilities in smart contracts, which are intra-function reentrancy, cross-function reentrancy, arithmetic, and `tx.origin` vulnerabilities. However, they do not handle the TOD vulnerabilities investigated in this thesis.

5.5.3 Functional Verification of Smart Contracts

Several previous works have developed frameworks for checking full functional correctness of smart contracts using proof assistants such as COQ, F* [21], and ISABELLE/HOL [5, 65, 75, 138]; automated theorem provers [69, 168]; or predicate abstraction [130]. These works rely on user-provided functional specifications while the work of this thesis focuses on the specific TOD vulnerability pattern, and makes it possible to locate and rectify this vulnerability in smart contracts for which functional specifications do not exist. On the other hand, the proposed work cannot establish the full functional correctness of smart contracts.

5.6 Conclusion

This chapter presents an automated technique for detecting and repairing TOD vulnerabilities in smart contracts. Using static analysis, dependency relations between public functions are derived that can be called by any user and state variables that can be manipulated by malicious users. The proposed technique is implemented in a prototype tool using an existing static analyzer for Solidity. The tool is used to detect and repair TOD vulnerabilities in 51 Solidity smart contracts demonstrating that its practical application. Furthermore, the rectified smart contracts contained only a small number of modifications compared to the length of the original smart contracts.

Chapter 6

Conclusion and Future Work

6.1 Contributions

As blockchain enabled smart contracts continues to infiltrate every aspect of industry, it is critical that they are grounded with a proper foundation. Numerous, significant vulnerabilities have been discovered on the Ethereum Virtual Machine, yet it remains to be the most popular smart contract language for creating decentralized applications. Move is a new smart contract language, which has security and verifiability as first-class features. As the adoption of Move increases, it necessitates robust developer tools and adherence to best practice principles, similar to the existing infrastructure present in Ethereum. This thesis has furthered the progress of this goal in three ways.

First, this thesis introduced VeriMove, the first model checking framework that supports the Move language. VeriMove expanded the capabilities of the VeriSolid model checking tool for Solidity, so that its correct-by-design model checking framework could be applied to the Move language. The experimental results showed that model checking is a feasible method to formally verify global properties in Move smart contracts.

Second, this thesis presented the first work on gas optimization in the Move language. The vast research on gas optimization in Solidity was analyzed and applied to the Move language. This thesis proposed 11 gas optimization patterns and principles for the Move language, presented 5 patterns that decrease the time complexity of the smart contract but have no effect on gas consumption, and implemented a sample smart contract for each proposed gas optimization pattern. The results showed that the proposed gas optimization patterns reduce gas consumption in a typical smart contract by $7 - 56\%$.

Third, this thesis investigated the transaction order dependency vulnerabilities in smart contracts. A static analysis based approach utilizing point-to analysis and guard statements was proposed to automatically locate and rectify such transaction order dependency vulnerabilities. The proposed algorithm was implemented as a prototype tool in Solidity, utilizing the Slither static analyzer. The empirical results on a benchmark suite containing 51 Solidity smart contracts showed that the proposed methodology can be used to detect and rectify such vulnerabilities, or to certify their absence.

6.2 Future Work

Blockchains such as Aptos, Sui, OpenLibra, and StarCoin which have Move as their smart contract language have yet to achieve the widespread adoption of Ethereum. Thus, at the time of writing, there lacks a sufficient amount of diversity in the deployed smart contracts to create standardized benchmarking suites. As a result, much of the testing in this thesis is limited. In Chapter 3, the proposed framework VeriMove was tested against three popular contracts transcribed from Solidity into Move. Likewise in Chapter 4, the tested smart contracts were designed specifically for the particular gas optimizations in question. While the results validate the correctness of the work, they do not provide insight into performance on deployed smart contracts and real-world applications. Furthermore, in Chapter 5, the proposed algorithm was implemented and tested on Solidity smart contracts rather than Move. Again, this is due to the lack of testing available to the Move language. With wider Move adoption anticipated, comprehensive testing of these Move tools is recommended for future research.

Bibliography

[1] 0L Network, https://0l.network/, Accessed on 04/13/2022., 2022.

[2] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, *Gasol: Gas analysis and optimization for ethereum smart contracts*, 2019. arXiv: 1912.11929 [cs.PL].

[3] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett, "Synthesis of super-optimized smart contracts using max-smt," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds., cham: Springer International Publishing, 2020, pp. 177–200, ISBN: 978-3-030-53288-8.

[4] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, pp. 1–19, ISBN: 9781450373869. DOI: 10.1145/800028.808479. [Online]. Available: https://doi.org/10.1145/800028.808479.

[5] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, J. Andronick and A. P. Felty, Eds., ACM, 2018, pp. 66–77. DOI: 10.1145/3167084. [Online]. Available: https://doi.org/10.1145/3167084.

[6] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: A model checker for concurrent software," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 484–487, ISBN: 978-3-540-27813-9.

[7] E. Androulaki *et al.*, "Hyperledger fabric," in *Proceedings of the Thirteenth EuroSys Conference*, ACM, Apr. 2018. DOI: 10.1145/3190508.3190538. [Online]. Available: https://doi.org/10.1145%2F3190508.3190538.

[8] F. A. Aponte-Novoa, A. L. S. Orozco, R. Villanueva-Polanco, and P. Wightman, "The 51% attack on blockchains: A mining behavior study," *IEEE Access*, vol. 9, pp. 140 549–140 564, 2021. DOI: 10.1109/ACCESS.2021.3119291.

[9] *Aptos labs*, GitHub repository, 2023. [Online]. Available: https://github.com/aptos-labs/aptos-core/blob/cdb1f27868890a49075356d626e91d73f8ee3170/aptos-move/aptos-gas-meter/src/meter.rs.

[10] Aptos Labs, https://aptoslabs.com/, Accessed on 04/13/2022., 2022.

[11] Aptos Labs, *Gas and storage fees*, https://aptos.dev/concepts/gas-txn-fee/, Accessed on 07/26/2023., 2023.

[12] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, M. Maffei and M. Ryan, Eds., ser. Lecture Notes in Computer Science, vol. 10204, Springer, 2017, pp. 164–186. DOI: 10.1007/978-3-662-54455-6_8. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6%5C_8.

[13] A. Back, "Hashcash - a denial of service counter-measure," 2002.

[14] S. Badruddoja, R. Dantu, Y. He, K. Upadhayay, and M. Thompson, "Making smart contracts smarter," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2021, pp. 1–3. DOI: 10.1109/ICBC51069.2021.9461148.

[15] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[16] C. Baier and J.-P. Katoen. "Introduction to model checking - lecture # 1: Motivation, background, and course organization." Lecture slides. (2018), [Online]. Available: https://pages.di.unipi.it/gadducci/SVV-22/slideA/svv_01.pdf.

[17] A. Basu *et al.*, "Rigorous component-based system design using the bip framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011. DOI: 10.1109/MS.2011.27.

[18] D. Bayer, S. Haber, and W. S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," in *Sequences II*, R. Capocelli, A. De Santis, and U. Vaccaro, Eds., New York, NY: Springer New York, 1993, pp. 329–334, ISBN: 978-1-4613-9323-8.

[19] *Beauty chain: Bectoken*, Accessed 08/02/2023. [Online]. Available: https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d%5C#code.

[20] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[21] K. Bhargavan *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, T. C. Murray and D. Stefan, Eds., ser. PLAS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 91–96, ISBN: 9781450345743. DOI: 10.1145/2993600.2993611. [Online]. Available: https://doi.org/10.1145/2993600.2993611.

[22] Binance, *What is a smart contract security audit?* https://academy.binance.com/en/articles/what-is-a-smart-contract-security-audit, Accessed on 3/16/2022.

[23] S. Blackshear *et al.*, "Move: A language with programmable resources," *Libra Assoc.*, 2019.

[24] "Blockchain: A fundamental shift for financial service institutions," Capgemini, Tech. Rep., 2017.

[25] M. Bowman, D. Das, A. Mandal, and H. Montgomery, "On elapsed time consensus protocols," in *Progress in Cryptology – INDOCRYPT 2021*, A. Adhikari, R. Küsters, and B. Preneel, Eds., Cham: Springer International Publishing, 2021, pp. 559–583, ISBN: 978-3-030-92518-5.

[26] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski, "Characterizing efficiency optimizations in solidity smart contracts," in *2020 IEEE International Conference on Blockchain (Blockchain)*, 2020, pp. 281–290. DOI: 10.1109/Blockchain50366.2020.00042.

[27] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[28] V. Buterin, *Hard fork completed*, https://blog.ethereum.org/2016/07/20/hard-fork-completed, Jul. 2016.

[29] V. Buterin *et al.*, *Combining ghost and casper*, 2020. arXiv: 2003.03052 [cs.CR].

[30] J. Carlton and D. Crocker, "Escher verification studio perfect developer and escher c verifier," *Industrial Use of Formal Methods: Formal Verification*, pp. 155–193, 2012.

[31] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99, New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186, ISBN: 1880446391.

[32] R. Cavada *et al.*, "The nuxmv symbolic model checker," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., Cham: Springer International Publishing, 2014, pp. 334–342, ISBN: 978-3-319-08867-9.

[33] D. Chaum, "Blind signatures for untraceable payments," in *Advances in Cryptology*, D. Chaum, R. L. Rivest, and A. T. Sherman, Eds., Boston, MA: Springer US, 1983, pp. 199–203, ISBN: 978-1-4757-0602-4.

[34] H. Chen, M. Pendleton, L. Njilla, and S. Xu, *A survey on ethereum systems security: Vulnerabilities, attacks and defenses*, 2019. DOI: 10.48550/ARXIV.1908.04507. [Online]. Available: https://arxiv.org/abs/1908.04507.

[35] J. Chen and E. Estevez, *Liberty reserve*, https://www.investopedia.com/terms/l/liberty-reserve.asp, 2020.

[36] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 442–446. DOI: 10.1109/SANER.2017.7884650.

[37] T. Chen *et al.*, "Towards saving money in using smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, 2018, pp. 81–84.

[38] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: A new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000. DOI: 10.1007/s100090050046. [Online]. Available: https://doi.org/10.1007/s100090050046.

[39] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on logic of programs*, Springer, 1981, pp. 52–71.

[40] B. Cook, D. Kroening, and N. Sharygina, "Symbolic model checking for asynchronous boolean programs," in *Model Checking Software*, P. Godefroid, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 75–90, ISBN: 978-3-540-31899-6.

[41] S. A. Cook, "Soundness and completeness of an axiom system for program verification," *SIAM Journal on Computing*, vol. 7, no. 1, pp. 70–90, 1978. DOI: 10.1137/0207005. eprint: https://doi.org/10.1137/0207005. [Online]. Available: https://doi.org/10.1137/0207005.

[42] Coq, https://coq.inria.fr/documentation, Accessed 07/10/2023.

[43] N. T. Courtois, *On the longest chain rule and programmed self-destruction of crypto currencies*, 2014. arXiv: 1405.0534 [cs.CR].

[44] E. Cubides and S. O'Brien, "2022 findings from the diary of consumer payment choice," The Federal Reserve, Tech. Rep., 2022.

[45] E. Cubides and S. O'Brien, "2023 findings from the diary of consumer payment choice," The Federal Reserve, Tech. Rep., 2023.

[46] W. Dai, *B-money*, http://www.weidai.com/bmoney.txt, 1998.

[47] E. Davis and G. Marcus, "The scope and limits of simulation in automated reasoning," *Artificial Intelligence*, vol. 233, pp. 60–72, 2016, ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2015.12.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370215001794.

[48] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Advances in Cryptology — CRYPTO' 92*, E. F. Brickell, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–147, ISBN: 978-3-540-48071-6.

[49] *E-gold*, https://cs.stanford.edu/people/eroberts/cs201/projects/2010-11/Bitcoins/e-gold.html, 2010.

[50] Ethereum, *Gas and fees*, https://ethereum.org/en/developers/docs/gas/, Accessed on 07/26/2023., 2023.

[51] Ethereum Improvement Proposals, *EIP-20: Token Standard*, https://eips.ethereum.org/EIPS/eip-20, Accessed on 06/21/2022., 2022.

[52] Ethereum Improvement Proposals, *EIP-721: Non-Fungible Token Standard*, https://eips.ethereum.org/EIPS/eip-721, Accessed on 06/21/2022., 2022.

[53] Etherscan, 2021. [Online]. Available: https://etherscan.io/.

[54] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, 2019, pp. 8–15.

[55] R. W. Floyd, "Assigning meanings to programs," *Proceedings of Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967. [Online]. Available: http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf.

[56] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, "A survey on formal verification for solidity smart contracts," in *2021 Australasian Computer Science Week Multiconference*, ser. ACSW '21, Dunedin, New Zealand: Association for Computing Machinery, 2021, ISBN: 9781450389563. DOI: 10.1145/3437378.3437879. [Online]. Available: https://doi.org/10.1145/3437378.3437879.

[57] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds., ACM, 2020, pp. 415–427. DOI: 10.1145/3395363.3397385. [Online]. Available: https://doi.org/10.1145/3395363.3397385.

[58] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Shanghai, China: Association for Computing Machinery, 2017, pp. 51–68, ISBN: 9781450350853. DOI: 10.1145/3132747.3132757. [Online]. Available: https://doi.org/10.1145/3132747.3132757.

[59] P. Glazman, *Qq coin — tencent's early virtual currency*, https://medium.com/@cryptomango/qq-coin-tencents-early-virtual-currency-fdff1090d910, 2018.

[60] P. Godefroid, "Model checking for programming languages using verisoft," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97, Paris, France: Association for Computing Machinery, 1997, pp. 174–186, ISBN: 0897918533. DOI: 10.1145/263699.263717. [Online]. Available: https://doi.org/10.1145/263699.263717.

[61] G. Golan Gueta *et al.*, "Sbft: A scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 568–580. DOI: 10.1109/DSN.2019.00063.

[62] M. J. Gordon and T. F. Melham, *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[63] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. DOI: 10.1145/3276486. [Online]. Available: https://doi.org/10.1145/3276486.

[64] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 116:1–116:27, 2018. DOI: 10.1145/3276486. [Online]. Available: https://doi.org/10.1145/3276486.

[65] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, L. Bauer and R. Küsters, Eds., ser. Lecture Notes in Computer Science, vol. 10804, Springer, 2018, pp. 243–269. DOI: 10.1007/978-3-319-89722-6_10. [Online]. Available: https://doi.org/10.1007/978-3-319-89722-6%5C_10.

[66] S. Grossman *et al.*, "Online detection of effectively callback free objects with applications to smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, 48:1–48:28, 2018. DOI: 10.1145/3158136. [Online]. Available: https://doi.org/10.1145/3158136.

[67] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds., Springer, 2015, pp. 343–361, ISBN: 978-3-319-21690-4.

[68] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," *Journal of Cryptology*, vol. 3, no. 2, pp. 99–111, 1991. DOI: 10.1007/BF00196791. [Online]. Available: https://doi.org/10.1007/BF00196791.

[69] Á. Hajdu and D. Jovanovic, "Solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, S. Chakraborty and J. A. Navas, Eds., ser. Lecture Notes in Computer Science, vol. 12031, Springer, 2019, pp. 161–179. DOI: 10.1007/978-3-030-41600-3_11. [Online]. Available: https://doi.org/10.1007/978-3-030-41600-3%5C_11.

[70] D. Harz and W. Knottenbelt, *Towards safer smart contracts: A survey of languages and verification methods*, Sep. 2018.

[71] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 366–381, 2000.

[72] R. Hayashi, *What are the average credit card processing fees that merchants pay?* https://paymentdepot.com/blog/average-credit-card-processing-fees/, Apr. 2022.

[73] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. T. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds., ACM, 2019, pp. 531–548. DOI: 10.1145/3319535.3363230. [Online]. Available: https://doi.org/10.1145/3319535.3363230.

[74] E. Hildenbrandt *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium*, pp. 204–217. DOI: 10.1109/CSF.2018.00022.

[75] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, M. Brenner *et al.*, Eds., ser. Lecture Notes in Computer Science, vol. 10323, Springer, 2017, pp. 520–535. DOI: 10.1007/978-3-319-70278-0_33. [Online]. Available: https://doi.org/10.1007/978-3-319-70278-0%5C_33.

[76] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, ISSN: 0001-0782. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[77] G. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997. DOI: 10.1109/32.588521.

[78] G. J. Holzmann, "Software model checking with spin," in ser. Advances in Computers, vol. 65, Elsevier, 2005, pp. 77–108. DOI: https://doi.org/10.1016/S0065-2458(05)65002-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0065245805650024.

[79] A. Hu, *Deprecate the storage gas curves*, GitHub commit, 2023. [Online]. Available: https://github.com/aptos-labs/aptos-core/commit/38f60b316a74042f0b2e17c1e518ddd337f14d20.

[80] "Ieee standard for property specification language (psl)," *IEEE Std 1850-2005*, pp. 1–143, 2005. DOI: 10.1109/IEEESTD.2005.97780.

[81] A. Imeri, N. Agoulmine, and D. Khadraoui, "Smart contract modeling and verification techniques: A survey," in *8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020)*, 2020, pp. 1–8.

[82] Kaden. "Smart Contract Vulnerabilities." (2023), [Online]. Available: https://github.com/kadenzipfel/smart-contract-vulnerabilities (visited on 06/03/2023).

[83] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018%5C_09-1%5C_Kalra%5C_paper.pdf.

[84] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts.," in *Ndss*, 2018, pp. 1–12.

[85] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-aided reasoning: ACL2 case studies*. Springer Science & Business Media, 2013, vol. 4.

[86] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Test input generation with java pathfinder: Then and now (invited talk abstract)," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 1–2, ISBN: 9781450356992. DOI: 10.1145/3213846.3234687. [Online]. Available: https://doi.org/10.1145/3213846.3234687.

[87] S. King and S. Nadal, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," 2012.

[88] J. Krupp and C. Rossow, "Teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds., USENIX Association, 2018, pp. 1317–1333. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.

[89] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57 037–57 062, 2022. DOI: 10.1109/ACCESS.2022.3169902.

[90] J. Kuszmaul, "Verkle trees," *Verkle Trees*, vol. 1, p. 1, 2019.

[91] A. Labs, GitHub repository, 2023. [Online]. Available: https://github.com/aptos-labs/aptos-core/blob/3791dc07ec457496c96e5069c494d46c1ff49b41/aptos-move/aptos-gas-schedule/src/gas_schedule/instr.rs.

[92] A. Labs, *Computing transaction gas*, GitHub repository, 2023. [Online]. Available: https://github.com/aptos-labs/aptos-core/blob/3791dc07ec457496c96e5069c494d46c1ff49b41/developer-docs-site/docs/concepts/base-gas.md.

[93] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982, ISSN: 0164-0925. DOI: 10.1145/357172.357176. [Online]. Available: https://doi.org/10.1145/357172.357176.

[94] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[95] Y. Lewenberg, Y. Bachrach, Y. Sompolinsky, A. Zohar, and J. S. Rosenschein, "Bitcoin mining pools: A cooperative game theoretic analysis," in *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '15, Istanbul, Turkey: International Foundation for Autonomous Agents and Multiagent Systems, 2015, pp. 919–927, ISBN: 9781450334136.

[96] C. Li, F. Long, and G. Yang, *Ghast: Breaking confirmation delay barrier in nakamoto consensus via adaptive weighted blocks*, 2020. arXiv: 2006.01072 [cs.CR].

[97] C. Li *et al.*, "A decentralized blockchain with high throughput and fast confirmation," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 515–528, ISBN: 978-1-939133-14-4. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/li-chenxing.

[98] F. Ma *et al.*, *V-gas: Generating high gas consumption inputs to avoid out-of-gas vulnerability*, 2021. arXiv: 1910.02945 [cs.CR].

[99] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design patterns for gas optimization in ethereum," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2020, pp. 9–15. DOI: 10.1109/IWBOSE50093.2020.9050163.

[100] B. Mariano, Y. Chen, Y. Feng, S. K. Lahiri, and I. Dillig, "Demystifying loops in smart contracts," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, IEEE, 2020, pp. 262–274. DOI: 10.1145/3324884.3416626. [Online]. Available: https://doi.org/10.1145/3324884.3416626.

[101] M. Maróti *et al.*, "Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure," in *MPM@MoDELS*, 2014.

[102] A. Mavridou and A. Laszka, *Designing secure ethereum smart contracts: A finite state machine based approach*, 2017. DOI: 10.48550/ARXIV.1711.09327. [Online]. Available: https://arxiv.org/abs/1711.09327.

[103] A. Mavridou and A. Laszka, *Tool demonstration: Fsolidm for designing secure ethereum smart contracts*, 2018. DOI: 10.48550/ARXIV.1802.09949. [Online]. Available: https://arxiv.org/abs/1802.09949.

[104] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," in *Proceedings of the 23rd International Conference on Financial Cryptography and Data Security (FC)*, Feb. 2019.

[105] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, 1993.

[106] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378, ISBN: 978-3-540-48184-3.

[107] *Merkle patricia trie*, `https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/`, Accessed: July 21, 2023, Jul. 2023.

[108] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Vienna, Austria: Association for Computing Machinery, 2016, pp. 31–42, ISBN: 9781450341394. DOI: `10.1145/2976749.2978399`. [Online]. Available: `https://doi.org/10.1145/2976749.2978399`.

[109] ModulTrade, *New erc20 batchoverflow bug*, `https://blog.goodaudience.com/new-erc20-batchoverflow-bug-2cd191668f0d`, Apr. 2018.

[110] S. Motepalli and H.-A. Jacobsen, "Reward mechanism for blockchains using evolutionary game theory," in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2021, pp. 217–224. DOI: `10.1109/BRAINS52497.2021.9569791`.

[111] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[112] C. A. Munoz and R. A. Demasi, "Advanced theorem proving techniques in pvs and applications," *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pp. 96–132, 2012.

[113] J. Nagele and M. A. Schett, *Blockchain superoptimizer*, 2020. arXiv: `2005.05912 [cs.LO]`.

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

[115] K. Nelaturu, S. M. Beillahi, F. Long, and A. Veneris, "Smart contracts refinement for gas optimization," in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2021, pp. 229–236. DOI: `10.1109/BRAINS52497.2021.9569819`.

[116] K. Nelaturu, A. Mavridou, A. Veneris, and A. Laszka, *Open-source implementation of extended VeriSolid*, `https://github.com/smartcontractsfc/verifier`, Accessed on 12/19/2019.

[117] T. D. Nguyen, L. H. Pham, and J. Sun, "SGUARD: towards fixing vulnerable smart contracts automatically," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, IEEE, 2021, pp. 1215–1229. DOI: `10.1109/SP40001.2021.00057`. [Online]. Available: `https://doi.org/10.1109/SP40001.2021.00057`.

[118] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, ACM, 2018, pp. 653–663. DOI: 10.1145/3274694.3274743. [Online]. Available: https://doi.org/10.1145/3274694.3274743.

[119] U. Norell, "Dependently typed programming in agda," Jan. 2009, pp. 1–2, ISBN: 978-3-642-04651-3. DOI: 10.1007/978-3-642-04652-0_5.

[120] Numen Cyber Labs, *Analysis of the first critical vulnerability of aptos move vm*, Medium, Oct. 2022. [Online]. Available: https://medium.com/numen-cyber-labs/analysis-of-the-first-critical-0-day-vulnerability-of-aptos-move-vm-8c1fd6c2b98e.

[121] Numen Cyber Labs, *The story of a high-risk vulnerability in move reference safety verify module*, Numen, Accessed on 07/25/2023. [Online]. Available: https://www.numencyber.com/the-story-of-a-high-risk-vulnerability-in-move-reference-safety-verify-module/.

[122] OpenZepellin, https://openzeppelin.com/, Accessed on 06/14/2023.

[123] OpenZepellin, *Safemath*, https://docs.openzeppelin.com/contracts/2.x/api/math, Accessed on 07/31/2023.

[124] M. Ortner and S. Eskandari, "Smart contract sanctuary," [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary.

[125] p0n1, *A disastrous vulnerability found in smart contracts of beautychain (bec)*, https://medium.com/secbit-media/a-disastrous-vulnerability-found-in-smart-contracts-of-beautychain-bec-dbf24ddbc30e, Apr. 2018.

[126] S. Palladino, *The parity wallet hack explained*, https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7, Jul. 2017.

[127] *Parity bug: Trigger*, Accessed 08/02/2023. [Online]. Available: https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4%5C#code.

[128] L. C. Paulson, *Isabelle: A Generic Theorem Prover*. Springer Verlag, 1994.

[129] L. C. Paulson, *Natural deduction as higher-order resolution*, 2000. arXiv: cs/9301104 [cs.LO].

[130] A. Permenev, D. K. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. T. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, IEEE, 2020, pp. 1661–1677. DOI: 10.1109/SP40000.2020.00024. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00024.

[131] O. Porkka, "Attacks on smart contracts," Master's thesis, University of Helsinki, 2022.

[132] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: 10.1145/359340.359342. [Online]. Available: https://doi.org/10.1145/359340.359342.

[133] K. Y. Rozier, "Linear temporal logic symbolic model checking," *Computer Science Review*, vol. 5, no. 2, pp. 163–203, 2011, ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2010.06.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013710000407.

[134] J. Rushby, "Tutorial: Automated formal methods with pvs, sal, and yices," in *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, 2006, pp. 262–262. DOI: 10.1109/SEFM.2006.37.

[135] N. F. Samreen and M. H. Alalfi, "Reentrancy vulnerability identification in ethereum smart contracts," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, IEEE, Feb. 2020. DOI: 10.1109/iwbose50093.2020.9050260. [Online]. Available: https://doi.org/10.1109%2Fiwbose50093.2020.9050260.

[136] S. Schulz, "E - a brainiac theorem prover," *AI Commun.*, vol. 15, no. 2,3, pp. 111–126, Aug. 2002, ISSN: 0921-7126.

[137] K. Sekniqi, D. Laine, S. Buttolph, and E. G. Sirer, "Avalanche platform," vol. 1, Jun. 2020.

[138] I. Sergey, A. Kumar, and A. Hobor, "Temporal properties of smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, T. Margaria and B. Steffen, Eds., ser. Lecture Notes in Computer Science, vol. 11247, Springer, 2018, pp. 323–338. DOI: 10.1007/978-3-030-03427-6_25. [Online]. Available: https://doi.org/10.1007/978-3-030-03427-6%5C_25.

[139] B. Severin, M. Hesenius, F. Blum, M. Hettmer, and V. Gruhn, "Smart money wasting: Analyzing gas cost drivers of ethereum smart contracts," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 293–304. DOI: 10.1109/ICSME55016.2022.00034.

[140] C. Signer, "Gas cost analysis for ethereum smart contracts," M.S. thesis, ETH Zürich, Department of Computer Science, 2018.

[141] C. Smith, *Smart contract security*, https://ethereum.org/en/developers/docs/smart-contracts/security/, Accessed 07/31/2023., Jun. 2023.

[142] Solana, *Transaction fees*, https://docs.solana.com/transaction_fees, Accessed on 07/26/2023., 2023.

[143] Solidity, *Solidity by example: Blind auction*, https://solidity.readthedocs.io/en/develop/solidity-by-example.html#blind-auction/, Accessed on 06/21/2022., 2022.

[144] Solidity, https://docs.soliditylang.org/en/v0.8.15/, Accessed 05/21/2023.

[145] "Solidity documentation (release 0.8.16)," Ethereum, Tech. Rep., 2022.

[146] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," Jan. 2015, ISBN: 978-3-662-47853-0. DOI: 10.1007/978-3-662-47854-7_32.

[147] Starcoin, https://starcoin.org/en/, Accessed on 04/13/2022., 2022.

[148] Sui, https://sui.io/, Accessed on 04/13/2022., 2022.

[149] Sui, *Sui gas fees*, https://docs.sui.io/build/sui-gas-charges, Accessed on 07/26/2023., 2023.

[150] J. Sun, Y. Liu, and J. S. Dong, "Model checking csp revisited: Introducing a process analysis toolkit," in *Leveraging Applications of Formal Methods, Verification and Validation*, T. Margaria and B. Steffen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 307–322, ISBN: 978-3-540-88479-8.

[151] Y. Sun, B. Yan, Y. Yao, and J. Yu, "Dt-dpos: A delegated proof of stake consensus algorithm with dynamic trust," *Procedia Computer Science*, vol. 187, pp. 371–376, 2021, 2020 International Conference on Identification, Information and Knowledge in the Internet of Things, IIKI2020, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2021.04.113. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050921009236.

[152] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," in *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '11, Tokyo, Japan: Association for Computing Machinery, 2011, pp. 266–278, ISBN: 9781450308656. DOI: 10.1145/2034773.2034811. [Online]. Available: https://doi.org/10.1145/2034773.2034811.

[153] *Swc-114: Transaction order dependence.* 2021. [Online]. Available: https://swcregistry.io/docs/SWC-114.

[154] SWC-registry, https://swcregistry.io/, Accessed on 06/21/2022., 2022.

[155] N. Szabo, *Bit gold*, http://unenumerated.blogspot.com/2005/12/bit-gold.html, 2005.

[156] The Aptos Labs Team, "The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure," Aptos Labs, Tech. Rep., Aug. 2022.

[157] *The history of ethereum*, https://ethereum.org/en/history/, Accessed 07/31/2023., Jul. 2023.

[158] *The rust programming language - understanding ownership*, https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html, Accessed 05/21/2023., 2023.

[159] *Thedao token*, Accessed 08/02/2023. [Online]. Available: https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4%5C#code.

[160] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018, pp. 9–16.

[161] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*, ACM, 2018, pp. 9–16. [Online]. Available: https://ieeexplore.ieee.org/document/8445052.

[162] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18, San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 664–676, ISBN: 9781450365697. DOI: 10.1145/3274694.3274737. [Online]. Available: https://doi.org/10.1145/3274694.3274737.

[163] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, ACM, 2018, pp. 664–676. DOI: 10.1145/3274694.3274737. [Online]. Available: https://doi.org/10.1145/3274694.3274737.

[164] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds., ACM, 2018, pp. 67–82. DOI: 10.1145/3243734.3243780. [Online]. Available: https://doi.org/10.1145/3243734.3243780.

[165] U.S. Securities and Exchange Commission (SEC), "Report of investigation pursuant to section 21(a) of the securities exchange act of 1934: The DAO," Tech. Rep., 2017. [Online]. Available: https://www.sec.gov/files/litigation/investreport/34-81207.pdf.

[166] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *Open Problems in Network Security*, J. Camenisch and D. Kesdoğan, Eds., Cham: Springer International Publishing, 2016, pp. 112–125, ISBN: 978-3-319-39028-4.

[167] Q. Wang, R. Li, Q. Wang, S. Chen, and Y. Xiang, *Exploring unfairness on proof of authority: Order manipulation attacks and remedies*, 2022. arXiv: 2203.03008 [cs.CR].

[168] Y. Wang *et al.*, "Formal verification of workflow policies for smart contracts in azure blockchain," in *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, S. Chakraborty and J. A. Navas, Eds., ser. Lecture Notes in Computer Science, vol. 12031, Springer, 2019, pp. 87–106. DOI: 10.1007/978-3-030-41600-3_7. [Online]. Available: https://doi.org/10.1007/978-3-030-41600-3%5C_7.

[169] R. Wille, G. Fey, M. Messing, G. Angst, L. Linhard, and R. Drechsler, "Identifying a subset of system verilog assertions for efficient bounded model checking," in *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, 2008, pp. 542–549. DOI: 10.1109/DSD.2008.53.

[170] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[171] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19, Toronto ON, Canada: Association for Computing Machinery, 2019, pp. 347–356, ISBN: 9781450362177. DOI: 10.1145/3293611.3331591. [Online]. Available: https://doi.org/10.1145/3293611.3331591.

[172] G. Zhang and H.-A. Jacobsen, "Prosecutor: An efficient bft consensus algorithm with behavior-aware penalization against byzantine attacks," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21, Québec city, Canada: Association for Computing Machinery, 2021, pp. 52–63, ISBN: 9781450385343. DOI: 10.1145/3464298.3484503. [Online]. Available: https://doi.org/10.1145/3464298.3484503.

[173] G. Zhang and H.-A. Jacobsen, *Escape to precaution against leader failures*, 2022. arXiv: 2202.09434 [cs.DC].

[174] G. Zhang and C. Xu, "An efficient consensus protocol for real-time permissioned blockchains under non-byzantine conditions," in *Green, Pervasive, and Cloud Computing*, S. Li, Ed., Cham: Springer International Publishing, 2019, pp. 298–311, ISBN: 978-3-030-15093-8.

[175] J. E. Zhong *et al.*, "The move prover," in *International Conference on Computer Aided Verification*, Springer, 2020, pp. 137–150.