A Model-Checking Framework for the Verification of Move Smart Contracts

Eric Keilty, Keerthi Nelaturu, Bowen Wu and Andreas Veneris
Department of Electrical & Computer Engineering
University of Toronto, Toronto, Canada
{eric.keilty, keerthi.nelaturu, bw.wu}@mail.utoronto.ca, veneris@eecg.toronto.edu

Abstract—**As the popularity of distributed ledger technology and smart contracts continues to grow, so does the number of decentralized applications and their potential exposure to expensive exploits. The need for strong vulnerability detection tools is critical. Move is a recently developed smart contract language with safety and security at the core of its design containing formal verification tools embedded into the language. Currently, these tools can only verify local properties within a single Move function. They cannot verify global properties that result from multiple function executions. In this paper, we introduce VeriMove, an extension of the VeriSolid correct-by-design model checking framework that supports the Move language. We show that model checking is a feasible method to formally verify global properties in Move smart contracts.**

Index Terms—**Smart Contract; Verification; Solidity; Move**

I. Introduction

Blockchain technology is at the heart of decentralization and trustless transactions, enabling the development of cryptocurrencies such as Bitcoin [1] and Ethereum [2]. Smart contracts are self-executing programs that operate on the blockchain to carry out pre-defined, publicly supervised procedures. It is possible to use smart contracts for both basic payment transactions and more involved business logic. Inevitably, complexity introduces vulnerabilities; the DAO assault, which exploited the re-entrancy vulnerability, resulted in a $747 million financial loss and harmed the blockchain's confidence [3].

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 [4] accessible in traditional programming languages. Current smart contract coding practices include the following: *(i)* carefully understanding the semantics of the language specification [5], *(ii)* using safe coding practices highlighted by teams like Open Zeppelin [6], and *(iii)* mandatory source code auditing by qualified service providers [7]. 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 [8], [9]. Theorem proving tools encode the model and requirements in formal mathematical semantics and use a theorem prover to determine whether or not the requirements hold in the model (i.e., [10]–[14]). Symbolic execution represents program variables as symbols to compute all feasible execution paths, which are checked for the vulnerabilities or desired program properties (i.e. [15]–[22]). Finally, model checking represents the smart contract as a finite-state transition model and represents properties as propositions about the model. A combination of state-space search and satisfiability solvers are used to verify the requirements (i.e., [23]–[26]).

Solidity [27] is a popular programming language for writing smart contracts. Its success is mostly due to the thriving Ethereum ecosystem and its familiar JavaScript-based syntax, which made it approachable for a large number of developers. However, the language also includes new constructs, such as `delegatecall` and `fallback` functions, to support blockchain transactions. Due to the lack of developer awareness of the semantics behind these constructs, numerous exploits have emerged [28]. These vulnerabilities pose a critical problem as they can result in significant financial loss. Even with existing verification tools, smart contracts still often contain known and preventable vulnerabilities. As a result, some research effort is focused on tackling this problem at its root by creating new smart contract languages that prevent these common vulnerabilities by design.

Move [29] is a smart contract language developed by Diem (formerly known as Libra) [30], [31] designed to ensure security and verifiability while retaining code 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* [32] for static analysis to catch common vulnerabilities, as well as a theorem proving formal verification tool called the *move-prover* [33] for more complex vulnerabilities. The verification specification is expressed in the same style as the coding logic, allowing for state checks before and after each function is executed. Diem, 0L [34], Starcoin [35], Sui [36], and Aptos [37] blockchains are now officially supported by Move.

Currently, the bytecode verifier and move-prover 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 [23], [38] for verifying global properties in Solidity.

In this paper, we present a comparative analysis of the Solidity and Move smart contract languages and introduce *VeriMove*, a model checking framework for Move built on top of VeriSolid. The contributions of this research are as follows:

- We summarize the features of the Move language.
- We do 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.
- We introduce VeriMove, a model checking framework that leverages VeriSolid to verify and generate Move smart contracts automatically based on the specifications provided.
- As part of the model checking framework, we define the operational semantics for the new Move constructs.
- We discuss the workflow of the VeriMove prototype implementation in-depth.
- We compare VeriMove with VeriSolid by presenting the outcomes of our experiments with three types of smart contracts, showing that VeriMove can verify global properties in Move with reasonable performance.

The remainder of the paper is structured as follows. Section II provides context for Move and VeriSolid. The works that are relevant to this research are covered in Section III. Section IV compares both Solidity and Move in detail, examining their features, vulnerabilities, and trade-offs. In Section V, we examine the VeriMove model checking framework's workflow and introduce its components. In Section VI, we describe the operational semantics introduced for the Move language. In Section VII, we describe the prototype's implementation in detail and examine the experimental results. Finally, Section VIII concludes this work.

II. Background

A. The Move Language

Move is an executable bytecode language for writing smart contracts and custom transaction logic [39]. Smart contracts in Move are written as *modules*, which contain custom types called *structs* and module functions called *procedures*. Move programs are published under an *account address* and executed on a blockchain. To interact with the blockchain program, a user writes a Move *transaction script*, which can import modules and call their corresponding procedures.

Move was designed to easily and securely manage digital assets [29]. This is accomplished through two properties:

- *access control*: the owner of an asset should have full control over who can access the asset.
- *scarcity*: duplicating the asset should be prohibited (to prevent double spending) and the creation of a new asset should be a privileged operation.

These properties are implemented in the Move language through the Rust-like ownership system and the *resource* type.

To manage memory, Move implements a Rust-like system of ownership where each variable "owns" their 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. 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.

The *resource type* in Move is implemented as a struct that cannot be created nor destroyed by code outside its declaring module and can never be copied nor dropped. When a resource is initialized, it must 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 cannot be accidentally (or intentionally) copied nor destroyed by code outside the declaring module.

A unique feature of Move is its built-in verification tools. Before any Move program can be published, it must pass the *bytecode verifier*, which statically verifies basic, light-weight safety properties. These checks fall into three categories: structural checks to make sure statements are well-formed; semantic checks such as incorrect procedure arguments, dangling references, and duplicating a resource; and authorization checks [29]. Additionally, Move has an offline verifier called the *move-prover* [33], 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.

B. VeriSolid

VeriSolid [23] is an open-source, web-based, model checking, formal verification framework built on top of WebGME [40] and FSolidM [41], [42]. 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 [43], which is then translated into an NuSMV model [44]. The templated properties are used to generate Computational Tree Logic (CTL) specifications [45]. State space exploration in BIP can verify deadlock freedom properties and the NuSMV model can verify safety and liveness properties using the nuXmv model checker [46]. Once the developer is satisfied with the model and properties, VeriSolid generates the equivalent Solidity source code.

III. 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. Most verification tools can be classified as either theorem proving, symbolic execution, or model checking [8], [9].

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 Ethereum Virtual Machine (EVM) bytecode include the $\mathbb{K}$ framework, Lem, and F*. $\mathbb{K}$ [47] 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 created a deductive verifier called KEVM. This uses Reachability Logic reasoning to evaluate program specifications expressed as reachability claims. Lem [48] is a general language of types, higher-order functions, and inductive relation definitions design to be scaleable for a wide variety of applications. Lem definitions can be translated to interactive-proof tools such as Coq, HOL4, and Isabelle/HOL. Finally, Bhargavan et al. [49] 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 currently implemented for the Move language is the move-prover discussed in Section II-A.

Symbolic execution is a form of static analysis that 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 [17], Gasper [18], and Osiris [19] construct a control flow graph to represent the state-space and use an SMT solver to detect vulnerabilities. Other tools such as Slither [15] and SmartCheck [20] utilize a semantic tree along with an intermediate representation 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 or satisfiability solvers. Well-known model checking tools include Zeus and Verisolid. Zeus [14] 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) and leverages SeaHorn [50] to preform the symbolic model checking. VeriSolid was discussed in detail in Section II-B. 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 paper, is the first model checking tool implemented for the Move language.

IV. 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 designed for the Diem blockchain. The Move language continues to gain support by different blockchain networks due to its unique safety features. In this section we compare of the main differences between Move and Solidity and the trade-offs associated with each.

A. 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. Since state variables are stored on the blockchain, their values are persistent after execution. Similarly, each contract in Move is given its own account address on the blockchain. However, the source code (modules) and global variables (resources) are stored separately. Thus, a declared resource is not necessarily stored under the same account address as its defining module; instead, it is stored under the account address of its owner. 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 [51]. While Solidity claims this may change in the future, as it stands Solidity is incredibly susceptible to memory leaks. By contrast as discussed in Section II-A, Move implements a Rust-like memory management system where each value has exactly one owner. This makes all Move variables (both local and global) memory safe and guarantees no memory leaks.

B. Transfers

In order to preform a transfer from one contract address to another in Solidity, the sender contract must use either `send()`, `transfer()`, or `call()`, each of which behave differently and are intended for different applications. When one of these functions is called, 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 re-entrancy, mishandled exceptions, unchecked call return value, delegate call to untrusted callee, and DoS from unexpected revert [28].

In Move, resources were developed to be implemented as digital assets. Recall from Section II-A 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. This immediately mitigates all aforementioned vulnerabilities, including re-entrancy, from the Move language.

C. Trade-Offs

Move is a more restricted language compared to Solidity. Based on the results from Section VII, Move requires 80 additional lines of code on average for the same smart contract. 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.

However, these restrictions 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 re-entrancy. Moreover, the Move language and the resource type were designed for safe and secure transactions. While future vulnerabilities may be discovered that are unique to Move, currently no vulnerability has been discovered that is present in Move but not present in Solidity. Hence, Move is a strictly safer language.

V. VeriMove: Design and Verification Workflow

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

A. 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 VeriSolid 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. This feature makes VeriMove much more flexible than VeriSolid, as only the grammar definition needs to be changed in order to support other languages.

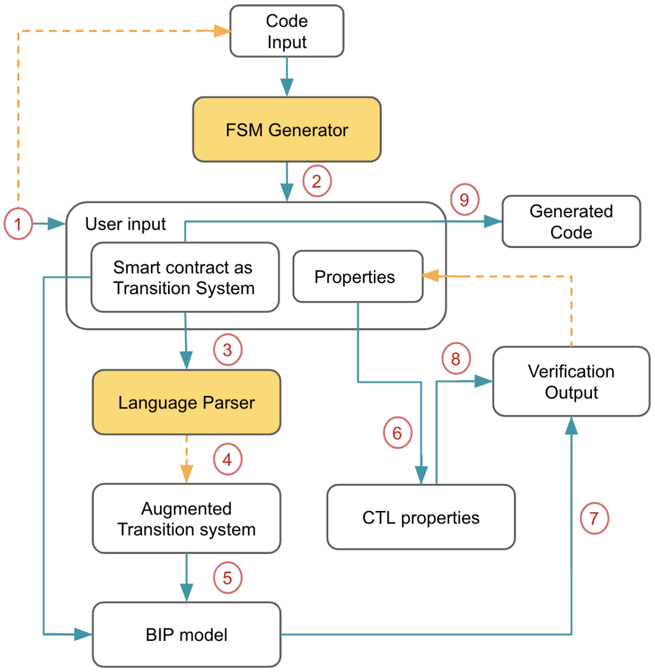

Fig. 1. Design and verification workflow

B. Finite State Machine Generator

VeriSolid requires the developer to write a smart contract in a very structured way. This can be a tedious process that takes a lot of time. Thus, VeriMove implements the ability to automatically create the transition system from Move source code. To accomplish this, the FSM generator creates an *initial state* and a *core state*. A transition from the initial state to the core state initializes the smart contract. All functions in the smart contract are represented as self-looping transitions in the core state. This generates a preliminary transition model that can be easily modified by the developer for their application. Due to the language parser, this functionality is easily extendable to other languages.

C. VeriMove Workflow

Figure 1 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. Like VeriSolid, VeriMove utilizes a graphical user interface (GUI), which allows the user to create and edit the transition system representation of the smart contract. 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 transition system. Step 3) begins the verification loop. 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 [23]. Here, the transition system is converted to a BIP model. The model properties are given by the user in natural language templates, which 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.

VI. Operational Semantics for Move

This section outlines the operational semantics necessary for the Move language. We define the subset of the Move language that VeriMove supports. The following are the custom types that are supported. Note that there is no $\langle event \rangle$ type in Move. Events are specified using resources.

$$\begin{aligned}\langle resource \rangle ::= \ & \texttt{resource struct}\ @identifier\ \texttt{\{} \\ & (@identifier\texttt{:}\ @type\texttt{,}\ )*\ \texttt{\}}\end{aligned}$$

VeriMove supports the following types of Move statements.

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

$$\begin{aligned}\langle declaration \rangle ::= \ & \texttt{let}\ @identifier\ (\texttt{:}\ @type)? \\ & (\texttt{=}\ @expression)?\end{aligned}$$

The operational semantics of the transition system for VeriMove are identical to that of VeriSolid [23]. 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.

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

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

VII. Empirical Evaluation

A. Implementation

Similar to VeriSolid, VeriMove was implemented as a web-based application utilizing WebGME [40] and FSolidM [41], [42] 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 [23] and our implementation of VeriMove: 1) VeriSolid is a `NodeJs` application whereas VeriMove is a `React` application, 2) VeriMove separated the language parser as a modular component, and 3) VeriMove adapted the FSM generator algorithm to be compatible with the Move language.

B. Experimental Setup

We will compare the performance of VeriSolid and VeriMove on the same set of smart contracts: `ERC20`, `ERC721`, and `BlindAuction`. The contracts `ERC20` and `ERC721` are implementations of the ERC20 Token Standard and ERC721 Non-Fungible Token Standard, respectively [52], [53]. The contract `BlindAuction` is an implementation of the blind auction example from the Solidity documentation [54]. In a blind auction, participants submit bids without knowing the bid amount of the other participants. To accomplish this one a blockchain platform where all transactions are public, participants submit the hash of their bids along with a deposit. After the bidding period is over, each bid is revealed. A bid is valid if the deposit is larger than the bid; the winner is the highest valid bid. 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 on the GitHub repository [1] along with the implementation.

C. Results

Both VeriSolid and VeriMove were able to successfully verify the contract properties. Tables I and II show the preformance 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 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.

D. Discussion and Limitations

As discussed in Section I, the current verification tools for Move, the bytecode verifier and the move-prover, can only verify properties within a single function. Model checking

[1] https://github.com/ekeilty17/move-smart-contracts

TABLE I
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}

TABLE II
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}

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 I and II 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 generally requires more statements to preform the same functionality compared to Solidity; thus, it requires more states. This is shown by the length of each contract in Move and Solidity. On average, Move required 80 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 II-A, 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.

VIII. CONCLUSION

In this work we modified and extended VeriSolid to support the formal verification of Move smart contracts. First, we gave a detailed comparison of the Move and Solidity, discussing the main differences between the design of the languages and their trade-offs. Next, we outlined the design and workflow of VeriMove. 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 make the model checking process more efficient. Additionally, we list the operational semantics introduced for the verification of a Move smart contract. Finally, we implement standard, widely used smart contracts in both VeriSolid and VeriMove, and compare their performance. The results showed that model checking is a feasible approach for verifying global properties in Move.

The experiments presented in this paper are limited due to the recency of Move's development. We expect that the Move language will soon gain popularity due to its unique safety features. With more examples of deployed Move smart contracts, a more in-depth comparison of Move/VeriMove and Solidity/VeriSolid can be done. Future work includes probing Move for vulnerabilities inherent to the language, comparing Move and Solidity against known vulnerabilities in more detail, determining what vulnerabilities are common to deployed Move contracts, and which of these vulnerabilities can be discovered and prevented by VeriMove.

REFERENCES

[1] F. Velde *et al.*, "Bitcoin: A primer," Federal Reserve Bank of Chicago, Tech. Rep., Dec. 2013.

[2] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, pp. 22–23, 2013.

[3] K. Finley, "A $50 million hack just showed that the DAO was all too human," Wired https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/, June 2016.

[4] 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.

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

[6] Open Zepellin, https://openzeppelin.com/, 2022, accessed on 04/13/2022.

[7] 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.

[8] D. Harz and W. Knottenbelt, "Towards safer smart contracts: A survey of languages and verification methods," 09 2018.

[9] 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. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3437378.3437879

[10] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 91–96. [Online]. Available: https://doi.org/10.1145/2993600.2993611

[11] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VeriSmart: A highly precise safety verifier for Ethereum smart contracts," *arXiv preprint arXiv:1908.11227*, 2019.

[12] Á. Hajdu, D. Jovanović, and G. Ciocarlie, "Formal specification and verification of solidity contracts with events," 05 2020.

[13] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu, "A formal verification tool for ethereum vm bytecode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 912–915. [Online]. Available: https://doi.org/10.1145/3236024.3264591

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

[15] 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.

[16] C. Diligence, "Mythx," 2020.

[17] 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.

[18] 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 (SANER)*, 2017, pp. 442–446.

[19] 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. New York, NY, USA: Association for Computing Machinery, 2018, p. 664–676. [Online]. Available: https://doi.org/10.1145/3274694.3274737

[20] 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.

[21] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," 2018. [Online]. Available: https://arxiv.org/abs/1809.03981

[22] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A framework for high-level analysis of ethereum bytecode," in *Automated Technology for Verification and Analysis*. Springer International Publishing, 2018, pp. 513–520.

[23] 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)*, February 2019.

[24] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[25] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A framework for high-level analysis of ethereum bytecode," in *Automated Technology for Verification and Analysis*. Springer International Publishing, 2018, pp. 513–520.

[26] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," 2018. [Online]. Available: https://arxiv.org/abs/1802.06038

[27] Solidity, https://docs.soliditylang.org/en/v0.8.15/, 2022, accessed on 06/21/2022.

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

[29] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. R. Rain, S. Sezer *et al.*, "Move: A language with programmable resources," *Libra Assoc.*, 2019.

[30] C. Catalini, O. Gratry, J. M. Hou, S. Parasuraman, and N. Wernerfelt, "The libra reserve," *Libra White Paper*, 2019.

[31] S. Bano, M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman *et al.*, "State machine replication in the libra blockchain," *Avalaible at: https://developers. libra. org/docs/state-machine-replication-paper (Consulted on December 19, 2020)*, 2020.

[32] L. Abraham and D. Guegan, "The other side of the coin: Risks of the libra blockchain," *University Ca'Foscari of Venice, Dept. of Economics Research Paper Series No*, vol. 30, 2020.

[33] J. E. Zhong, K. Cheang, S. Qadeer, W. Grieskamp, S. Blackshear, J. Park, Y. Zohar, C. Barrett, and D. L. Dill, "The move prover," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 137–150.

[34] 0L Network, https://0l.network/, 2022, accessed on 04/13/2022.

[35] Starcoin, https://starcoin.org/en/, 2022, accessed on 04/13/2022.

[36] Sui, https://sui.io/, 2022, accessed on 04/13/2022.

[37] Aptos Labs, https://aptoslabs.com/, 2022, accessed on 04/13/2022.

[38] 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.

[39] Diem, https://move-book.com/index.html, 2022, accessed on 06/21/2022.

[40] M. Maróti, T. Kecskés, R. Kereskényi, B. Broll, P. Völgyesi, L. Jurácz, T. Levendovszky, and Á. Lédeczi, "Next generation (meta)modeling: Web- and cloud-based collaborative tool infrastructure," in *MPM@MoDELS*, 2014.

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

[42] ——, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," 2018. [Online]. Available: https://arxiv.org/abs/1802.09949

[43] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[44] "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

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

[46] "Formal verification of infinite-state bip models," in *Automated Technology for Verification and Analysis*, B. Finkbeiner, G. Pu, and L. Zhang, Eds. Springer International Publishing, 2015, pp. 326–343.

[47] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 204–217.

[48] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: Reusable engineering of real-world semantics," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 175–188. [Online]. Available: https://doi.org/10.1145/2628136.2628143

[49] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 91–96. [Online]. Available: https://doi.org/10.1145/2993600.2993611

[50] 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 International Publishing, 2015, pp. 343–361.

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

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

[53] ——, "EIP-721: Non-Fungible Token Standard," https://eips.ethereum.org/EIPS/eip-721, 2022, accessed on 06/21/2022.

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