International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

Formal Semantic Approach to Detect Smart Contract Vulnerabilities Using KEVM

Mrs. Rohini Pise[1], Dr. Sonali Patil[2]
[1]Department of Information Technology,
Pimpri Chinchwad College of Engineering
Pune, India
Email: rohini.pise@pccoepune.org
[2]Department of Information Technology,
Pimpri Chinchwad College of Engineering
Pune, India
Email: sonali.patil@pccoepune.org

Abstract— Smart contracts are self-executing programs that run on blockchain platforms. While smart contracts offer a range of benefits, such as immutability and transparency, they are not immune to vulnerabilities. Malicious actors can exploit smart contract vulnerabilities to execute unintended actions or access sensitive data[1]. One approach to mitigating smart contract vulnerabilities is formal verification. Formal verification is a method of verifying the correctness of software using mathematical techniques. It involves mathematically proving that a program conforms to a set of specifications. Formal verification can help detect and eliminate vulnerabilities in smart contracts before they are deployed on the blockchain. KEVM (K Framework-based EVM) is a framework that allows for formal verification of smart contracts on the Ethereum Virtual Machine (EVM). KEVM uses the K Framework, a formal semantics framework, to specify the behavior of the EVM. With KEVM, smart contract developers can verify the correctness of their contracts before deployment, reducing the risk of vulnerabilities. In this paper, we have studied smart contract vulnerabilities such as Over usage of Gas, Signature Replay attack, and misuse of fallback function. We have also written the formal specification for these vulnerabilities and executed it using KEVM.

Keywords- Smart contracts, Vulnerabilities, Formal verification, K Framework, EVM.

I. INTRODUCTION

A smart contract is a self-executing program that runs on a blockchain platform. It is a computer program that automatically executes the terms of a contract when specific conditions are met. Smart contracts use blockchain technology to provide a transparent, secure, and immutable way to execute transactions without the need for intermediaries[1][2].

1.1. Smart Contract:

Smart contracts are written in programming languages such as Solidity, Vyper, and Go, and they typically contain a set of rules and conditions that define the terms of an agreement between two or more parties. These rules can be as simple or complex as required and include time constraints, payment requirements, and performance obligations.

Once a smart contract is deployed on a blockchain, it is publicly accessible and cannot be modified. When the predetermined conditions are met, the smart contract automatically executes the agreed-upon terms, triggering the transfer of assets or payment to the appropriate party. Smart contracts can be used in various industries, including finance, real estate, and supply chain management, to automate and streamline processes.

Overall, smart contracts have the potential to revolutionize the way we conduct transactions and execute agreements[3]. By providing a transparent, secure, and automated way to execute contracts, they can reduce transaction costs, eliminate the need for intermediaries, and increase efficiency and trust in business processes.

1.2 Formal Verification Method:

Formal verification involves using mathematical methods to prove that a software system or component meets certain specified requirements or properties. Formal verification typically consists of the creation of a formal specification of the system's functionality, as well as formal proof that the implementation of the system satisfies the specification. Formal verification can be used to guarantee that a system is correct and secure under all possible inputs and configurations.

Writing test cases can help identify system defects, but it cannot guarantee that the system is correct and secure under all possible inputs and configurations. On the other hand,

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

formal verification can provide a rigorous proof that a system meets its specified requirements and is secure under all possible inputs and configurations[4].

Formal verification is typically highly automated and can be used to verify large and complex systems with minimal manual effort. Formal verification can provide rigorous proof of correctness and security[5].

1.3 KEVM

The KEVM (K Ethereum Virtual Machine) framework is a tool for verifying and testing smart contracts written in the Solidity programming language on the Ethereum blockchain. The framework is based on the K framework, a robust language semantics framework that allows for formal specification, verification, and execution of programming languages[6].

The KEVM framework provides formal semantics of the Ethereum Virtual Machine (EVM), which is the runtime environment for executing smart contracts on the Ethereum blockchain. This formal semantics allows for the rigorous analysis and verification of smart contracts, which is critical for ensuring their correctness and security.

The KEVM framework works by providing a way to automatically generate test cases for smart contracts based on their formal specification [7]. This helps developers to identify potential bugs or vulnerabilities in their contracts before deploying them on the Ethereum blockchain.

KEVM framework is a powerful tool for verifying and testing smart contracts on the Ethereum blockchain. By providing formal semantics of the EVM and generating test cases based on the formal specification of smart contracts, the framework helps ensure these contracts' correctness and security.

II. RELATED WORK

2.1 Working of KEVM:

The KEVM (K Ethereum Virtual Machine) is an implementation of the Ethereum Virtual Machine (EVM) based on the K framework. The KEVM provides a formal semantics of the EVM and allows for the rigorous analysis and verification of smart contracts written in the Solidity programming language.

To execute the KEVM, a developer writes a Solidity smart contract and then compiles it into EVM bytecode. The developer then uses the KEVM toolchain to execute the bytecode in the KEVM runtime environment.

The KEVM runtime environment provides an implementation of the EVM instructions and a gas metering mechanism for tracking the computational resources used by the contract. The KEVM also provides tools for debugging and analyzing the contract's behavior during execution.

During execution, the KEVM verifies that the bytecode conforms to the formal semantics of the EVM and checks for any potential security vulnerabilities or other issues. The KEVM also provides a way to simulate the execution of the contract under different inputs and configurations, which can help developers identify and fix any issues.

Overall, the KEVM is a powerful tool for verifying and testing smart contracts on the Ethereum blockchain[8][9]. By providing formal semantics of the EVM and a runtime environment for executing and analyzing smart contracts, the KEVM helps ensure these contracts' correctness and security.

2.2 Execution and verification of smart contract using KEVM:

Consider the following simple Solidity smart contract:

```
Pragma   solidity0.8.0;
Contract Adder{
      Function add ( uint256 a, uint256 b) public pure
returns (uint256) {
Return a+b;
      }
  }
```

This contract defines a simple function add that takes two integers a and b, add them together, and returns the result.

To use the KEVM framework to verify and test this contract, we first need to compile it into EVM bytecode. We can use the solc Solidity compiler to do this:

```
$ solc --bin adder.sol
```

This will generate a binary bytecode file called adder_sol_Adder.bin

Next, we can use the KEVM toolchain to execute the bytecode in the KEVM runtime environment:

```
$ evm exec –codefile adder_sol_Adder.bin
--input
0000000000000000000000000000000000000000000000000000000000000005
--input
0000000000000000000000000000000000000000000000000000000000000003
   --method add
```

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

This command will execute the add function of the Adder contract with inputs a=5 and b=3. The -- codefile option specifies the bytecode file generated by the Solidity compiler, and the --input options specify the two input values encoded as 256-bit integers. The --method option specifies the name of the function to execute.

The output of the command should be: 0000000000000000000000000000000000000000000000000

00000000000000000008

We can also use the KEVM framework to generate test cases for the Adder contract. To do this, we can use the kevm-testgen tool to automatically generate test cases based on the formal semantics of the contract:

Overall, this example demonstrates how to use the KEVM framework to verify and test a simple Solidity smart contract. By providing formal semantics of the EVM and a runtime environment for executing and analyzing smart contracts, the KEVM helps ensure these contracts' correctness and security[10].

2.2 KEVM specification example

Sure, here's an example KEVM specification for a simple Solidity smart contract:

Consider the following Solidity smart contract:

```
pragma solidity 0.8.0;

contract Adder {

function add(uint256 a, uint256 b) public pure returns (uint256) {

        return a + b;

    }

}
```

To specify the behavior of this contract in the KEVM framework, we first need to define the syntax and semantics of the EVM instructions that implement the contract's bytecode. We can do this using the K framework[11][12].

Here's an example KEVM specification for the Adder contract:

```
module ADDER
  imports INT

    syntax Value ::= Int | Address | NilValue syntax
  Stack ::= List{Value}

    syntax Memory ::= Map{Int,Int}
  syntax Storage ::= Map{Int,Int} syntax
  Globals ::= Map{Int,Int}

  configuration <T color="red">

<k> $PGM:Program</k>

<s> .Stack</s>

<m> .Memory</m>

<a> .Address</a>

<g> .Globals</g>

<o> .Storage</o>

</T>

  syntax Program ::=KInstruction*
syntax KInstruction ::= "STOP"
                      | "ADD"
                      | "MSTORE"
                      | "MLOAD"
                      | "SSTORE"
                      | "SLOAD"
                      | "PUSH" Int
                      | "DUP1"


  rule <k> PUSH A:Int => .Stack </k>

  <m> M </m>

 requires Int2ByteList(A) => X ensures
.Stack => A : .Stack

        /\ .Memory => M[0..IntSize-1] := X

        /\ .Memory => M[IntSize..] :=
.Memory[IntSize..]


 rule <k> ADD => .Stack => X : Y : S ...</k>
ensures .Stack => (X + Y) : S

        /\ .Memory => M

        /\ .Address => A

        /\ .Globals => G

        /\ .Storage => O


  rule <k> MSTORE => .Stack => V : P : S ...

  <m> M
 </m>
```

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

```
        requires Int2ByteList(V)
                => X

        /\ Int2ByteList(P) => Y

    /\  P  +  IntSize<MapSize
ensures .Stack => S

            /\ .Memory => M[0..P-1] := .Memory[0..P-1]

            /\ .Memory => M[P..P+IntSize-1] := X

            /\ .Memory  =>  M[P+IntSize..]  :=
  .Memory[P+IntSize..]


    rule <k> MLOAD => .Stack => P : S ...

  <m> M[P..P+IntSize-1] => X </m>

  ensures .Stack => IntList2Int(X) : S

            /\ .Memory => M

            /\ .Address => A

            /\ .Globals => G

            /\ .Storage => O


    rule <k> SSTORE => .Stack => V : P : S ...

  <o> O </o>

 requires Int2ByteList(P)  =>  X
ensures .Stack => S

            /\ .Memory => M

            /\ .Address => A

            /\ .Globals => G

    /\ .Storage  =>  O[X]  :=  V
rule <k> SLOAD => .
```

Formal specification is a precise and unambiguous description of the expected behavior of a system or component, written in a formal language that both humans and computers can understand. Formal specifications are often used in software engineering to ensure that software systems meet their requirements and behave correctly.

The syntax of a formal specification is often based on a mathematical notation and defines the set of symbols and rules for constructing valid expressions in the language. In other words, it specifies the grammar of the language [13]. The syntax typically includes definitions of data types, functions, and operators and rules for constructing expressions from these components. Notation is typically more precise and rigorous than natural language and allows for constructing complex expressions with unambiguous semantics.

Here's an example of the syntax of a simple formal specification language:

```
syntax Exp ::= Var | Int  | Bool | Exp + Exp | Exp *
Exp | If Exp then Exp else Exp

syntax Var ::= String syntax Int
::= Integer syntax Bool ::= True |
False
```

This specification defines a syntax for expressions containing variables, integers, booleans, and arithmetic and conditional expressions. The syntax keyword is used to define new syntax constructs, while the Var, Int, and Bool keywords represent data types. The +, *, if, and then/else keywords define operators and control structures. To check the execution of the above specification, we use the K Framework. The K Framework provides formal semantics for Ethereum smart contracts, which can be used to automatically verify the properties of the contract. Once the specification has been written, it can be loaded into the K Framework and executed using the K Run tool. The K Run tool simulates the execution of the contract according to the rules defined in the specification and can be used to check that the contract behaves correctly under different conditions[15][16].

III. MATHEMATICAL MODEL FOR PROPOSED METHOD

In this research, we have used KEVM framework for verification of smart contracts. As the first step towards the implementation, here we are proposing the mathematical model as shown in fig.3.1.

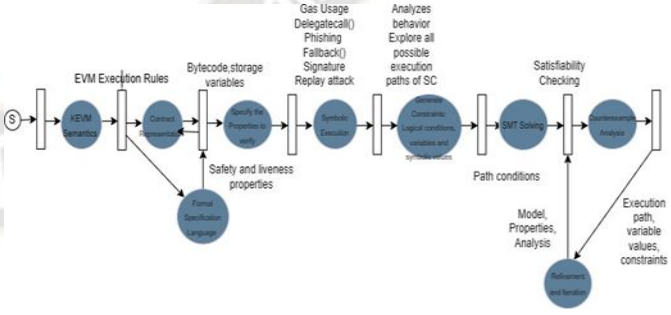

Fig. 3.1 Mathematical model

For mathematical modelling , we have used PetriNet Modelling technique. The complete process is represented by two types of elements, one is place and second is transition. In this directed graph, places represented as circles and transitions represented as rectangles.

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

IV. FORMAL SPECIFICATION TO IDENTIFYGAS USAGE IN A SMART CONTRACT

An example of formal specification in the K framework for identifying gas usage in executing a smart contract is discussed here:

```
module GAS
  imports INT
  syntax Value ::= Int | Address | NilValuesyntax Stack ::= List{Value}
  syntax Memory ::= Map{Int,Int} syntax Storage ::= Map{Int,Int} syntax Globals ::= Map{Int,Int} configuration <T color="red">
  <k> $PGM:Program</k>
  <s> .Stack</s>
  <m> .Memory</m>
  <a> .Address</a>
  <g> .Globals</g>
  <o> .Storage</o>
  <gas> Int </gas>
  </T>

  syntax Program ::=KInstruction* syntax KInstruction ::= "STOP"
                | "ADD"
                | "MSTORE"
                | "MLOAD"
                | "SSTORE"
                | "SLOAD"
                | "PUSH" Int
                | "DUP1"

  syntax GasCost ::= Int

  rule <k> PUSH A:Int => .Stack </k>
  <m> M </m>
  <gas> G </gas>
  requires Int2ByteList(A) => X
ensures .Stack => A : .Stack
        /\ .Memory => M[0..IntSize-1] := X
        /\ .Memory => M[IntSize..] := .Memory[IntSize..]
        /\ .gas => G - 3

  rule <k> ADD => .Stack => X : Y : S ...
  <gas> G </gas>
  ensures .Stack => (X + Y) : S
      /\ .Memory => M
      /\ .Address => A
      /\ .Globals => G
      /\ .Storage => O
      /\ .gas => G - 3

  rule <k> MSTORE => .Stack => V : P : S ...
  <m> M </m>
  <gas> G </gas>
  requires Int2ByteList(V) => X
    /\ Int2ByteList(P) => Y
   /\ P + IntSize<MapSize
ensures .Stack => S
        /\ .Memory => M[0..P-1] := .Memory[0..P-1]
        /\ .Memory => M[P..P+IntSize-1] := X
        /\ .Memory => M[P+IntSize..] := .Memory[P+IntSiz
                e..]
    /\ .gas => G - 3

  rule <k> MLOAD => .Stack => P : S ...
  <m> M[P..P+IntSize-1] => X </m>
  <gas> G </gas>
  ensures .Stack => IntList2Int(X) : S
        /\ .Memory => M
```

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

```
            /\ .Address => A
            /\ .Globals => G
            /\ .Storage => O
            /\ .gas => G - 2

      rule <k> SSTORE => .Stack => V : P : S ...
    <o> O </o>
    <gas> G </gas>
  requires Int2ByteList(P) => X ensures
.Stack => S
            /\ .Memory => M
            /\ .Address => A
            /\ .Globals => G
            /\ .Storage => O[X] := V
            /\ .gas => G - 3
      rule <k> SLOAD => .Stack => P : S ...
    <o> O[P] => V </o>
    <gas> G </gas> ensures
.Stack => V : S
```

The above formal specification is written in the K framework and describes a simple gas metering mechanism for a smart contract execution. The smart contract is assumed to be represented as a sequence of instructions, with each instruction taking a certain amount of gas to execute. The specification defines the syntax of the state of the system during the execution of the smart contract. The state consists of a stack of values, a memory map, a storage map, and a global map. The syntax also includes a gas counter, which keeps track of the remaining amount of gas available for execution. Each rule inputs the current state of the system and the gas counter, producing a new state of the system and a new gas counter. Each rule also specifies the amount of gas consumed by the instruction.

For example, the rule for the PUSH instruction takes the current stack, memory map, and gas counter as input and produces a new stack and memory map with the value of the PUSH instruction pushed onto the stack. The rule also subtracts three from the gas counter since the PUSH instruction consumes three gas. Similarly, the rule for the ADD instruction takes as input the current stack and gas counter and produces a new stack with the sum of the top two values on the stack. By defining a set of rules for each instruction in the smart contract, the specification provides a precise and unambiguous description of the expected behavior of the smart contract execution, including the amount of gas consumed by each instruction[17]. The amount of gas assigned to each instruction is specified in the gasmap function. For example, the PUSH instruction is assigned a gas cost of 3, which means that executing a PUSH instruction consumes three units of gas. During the execution of the smart contract, the gas counter keeps track of the remaining amount of gas available for execution. The initial value of the gas counter is set to N units of gas, where N is a parameter of the smart contract. The gas consumed by each instruction is subtracted from the gas counter. If the gas counter reaches zero or becomes negative during the execution of the smart contract, then the execution is aborted, and any changes to the system's state are reverted. The amount of gas used by the smart contract execution is equal to the difference between the gas counter's initial value and the gas counter's final value. In the above example, if the smart contract execution completes successfully, then the amount of gas used would be equal to the difference between the initial value of 100 units and the final value of 91 units, which is nine units of gas[18].

V. FORMAL SPECIFICATION FOR FALLBACK FUNCTION IN SMART CONTRACT

The formal specification for one of the important vulnerabilities of improper defining fallback function is given here.

```
syntax Int ::= "int" "(" Int ")"  // integers
syntax Address ::= "address" "(" Int ")" // Ethereumaddresses
syntax Bool ::= "true" | "false" // boolean values
syntax State ::= state(gascount: Int, sender: Address, value: Int, data: List{Byte}, balance: Map{Address, Int})
syntax Byte ::= byte(Int) // individual bytes
syntax List{T} ::= nil | cons(T, List{T}) // lists
syntax Instruction ::= CALLDATALOAD
        | CALLVALUE | CALLER | BALANCE | JUMPDEST
| PUSH(Int) | JUMP | JUMPI | DUP(Int) | SWAP(Int) |POP | MSTORE | RETURN
syntax Gasmap ::=gasmap(Instruction -> Int) syntax Contract ::= contract(State, Gasmap)
```

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

rule <k>EStack => I ...</k>

<state gascount: Gas, sender: Addr, value: Val, data:D, balance: B>

<gasmapGasmap>

requires Gas >= getGas(I, Gasmap)
ensures Gas == Gas - getGas(I, Gasmap)

when I != JUMPDEST

rule <k>EStack => I ...</k>

<state gascount: Gas, sender: Addr, value: Val, data:D, balance: B>

<gasmapGasmap>

requires Gas >= getGas(I, Gasmap)
ensures Gas == Gas - getGas(I, Gasmap)

when I == JUMPDEST

syntax Fallback ::= fallback(State, Gasmap)rule <k>EStack => Fallback </k>

<state gascount: Gas, sender: Addr, value: Val, data:D, balance: B>

<gasmapGasmap>

requires Gas >= 2300 // fallback function alwayshas a minimum gas of 2300

ensures Gas == Gas - 2300

This specification defines a Fallback syntax that represents the fallback function of a smart contract. The Fallback function takes the same input state, gasmap, and output stack as a regular instruction. The Fallback rule specifies that when the fallback function is executed, it requires a minimum of 2300 gas, which is the minimum amount of gas needed by the Ethereum protocol for executing a fallback function. The rule also ensures that 2300 units decrement the gas counter after the execution of the fallback function[19]. This is a simplified specification and does not include all possible instructions and behaviors that may occur during the execution of a fallback function.

VI. FORMAL SPECIFICATION FOR SIGNATURE REPLAY ATTACK IN SMART CONTRACT

A signature replay attack in blockchain is a type of attack where a valid digital signature is reused without the owner's consent to perform unauthorized transactions on the blockchain network. In a signature replay attack, an attacker intercepts a valid signature from a legitimate transaction and then uses it to create a new transaction without the owner's consent. Since the signature is valid, the transaction is accepted by the blockchain network and added to the blockchain[20][21].

To prevent this type of attack, we have written formal specifications to identify it. The given formal specification describes how to prevent a signature replay attack in a smart contract.

syntax Int ::= "int" "(" Int ")" // integers

syntax Address ::= "address" "(" Int ")" // Ethereum addresses

syntax Bytes32 ::= "bytes32" "(" Int ")" // 32-byte values

syntax Bool ::= "true" | "false" // boolean values

syntax State ::= state(signer: Address, nonce: Int, balance: Map{Address, Int})

syntax Transaction ::=tx(from: Address, to: Address, value: Int, data: List{Byte}, nonce: Int, gasprice: Int, gaslimit: Int, v: Int, r: Bytes32, s: Bytes32)

syntax SignatureReplayAttack ::=

sigAttack(from: Address, to: Address, value: Int, data: List{Byte}, nonce: Int, gasprice: Int, gaslimit: Int, v: Int, r: Bytes32, s: Bytes32, originalTx: Transaction) syntax Byte ::= byte(Int) // individual bytes

syntax List{T} ::= nil | cons(T, List{T}) // lists syntax Contract ::= contract(State)

syntax VerifySignature ::=verifySignature(data: List{Byte}, v: Int, r: Bytes32, s: Bytes32, pubkey:

Bytes32) -> Bool

rule <k>EStack => F ...</k>

<state signer: SignerAddr, nonce: Nonce, balance: Balance>

<verifySignature(data: Data, v: V, r: R, s: S, pubkey:PubKey) |G: Gas>

<tx from: FromAddr, to: ToAddr, value: Value, nonce: TxNonce>

requires G >= 20000 // gas cost for calling verifySignature function

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

ensures G == G - 20000

when FromAddr == SignerAddr and TxNonce == Nonce and VerifySignature(Data, V, R, S, PubKey)

== true

// if the signature is valid and transaction is sent by the signer

rule <k>EStack => F ...</k>

<state signer: SignerAddr, nonce: Nonce, balance: Balance>

<verifySignature(data: Data, v: V, r: R, s: S, pubkey:PubKey) |G: Gas>

<tx from: FromAddr, to: ToAddr, value: Value, nonce: TxNonce>

requires G >= 20000 // gas cost for calling verifySignature function

ensures G == G - 20000

when FromAddr == SignerAddr and TxNonce< Nonce and VerifySignature(Data, V, R, S, PubKey)

== true

// if signature is valid and transaction is sent by the signer, but with an old nonce

syntax TransactionPool ::= pool(List{Transaction})rule <k>EStack => F ...</k>

<state signer: SignerAddr, nonce: Nonce, balance:

Balance>

<verifySignature(data: Data, v: V, r: R, s: S, pubkey:PubKey) |G: Gas>

<sigAttack(from: FromAddr, to: ToAddr, value: Value, data: Data, nonce: TxNonce, gasprice: GasPrice, gaslimit: GasLimit, v: V, r: R, s: S, originalTx: OriginalTx)>

requires G >= 20000 // gas cost for calling verifySignature function

ensures G == G - 20000

when FromAddr == SignerAddr and TxNonce == Nonce and VerifySignature(Data, V, R, S, PubKey)

== true

The specification defines several syntax definitions for various data types, such as integers, addresses, bytes, and booleans. It then defines a State syntax to represent the state of the contract, which includes the signer's address, nonce, and balance. The specification also defines a Transaction syntax to represent Ethereum transactions, as well as a Verify Signature function to verify the authenticity of a given signature[22].

The rules of the specification define the behavior of the contract in certain situations. In particular, there are two rules that define how the contract should behave when a valid transaction is received: one for when the transaction has the correct nonce, and one for when the transaction has an old nonce. Both rules require that the transaction's signature be verified using the Verify Signature function, and that the transaction be sent by the signer. Additionally, the specification defines Signature Replay Attack syntax to represent a potential attack, as well as a Transaction Pool syntax to represent a pool of unprocessed transactions. The last rule specifies how the contract should behave when a signature replay attack is attempted[23]. It requires that the transaction's signature be verified, and that the transaction have the correct nonce and be sent by the signer. However, this rule only applies if the transaction is sent as part of an attack and is already present in the transaction pool. This is intended to prevent an attacker from replaying a valid transaction that was originally sent by the signer. Overall, the specification is designed to prevent a signature replay attack by ensuring that only valid transactions are processed and that transactions with old nonces are rejected. By verifying the authenticity of each transaction's signature using the VerifySignature function, the contract can ensure that the transaction is indeed being sent by the intended signer.

VII. DISCUSSION

Here is a general overview of the steps involved in executing a formal specification:

Define the formal specification: This involves writing the specification in a language that can be executed by a formal verification tool. For example, the specification could be written in the K Framework's syntax or another formal specification language such as Coq or Alloy. Compile the specification: Once the specification has been defined, it needs to be compiled into an executable format. This is typically done using a compiler or interpreter that is designed for the specific language and tool used. Run the specification: Once the specification has been compiled, it can be run using a tool that can execute the specification. For example, the K Framework provides a tool called KRun that can be used to execute K specifications[24]. Test the specification: After the specification has been executed, it should be tested to ensure

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 9s
DOI: https://doi.org/10.17762/ijritcc.v11i9s.7432
***Article Received**: 09 May 2023 **Revised:** 01 July 2023 **Accepted:** 28 July 2023*

that it behaves correctly. This can involve running test cases that cover different scenarios and checking that the specification produces the expected output.

Overall, the process of executing a formal specification can be complex and may require specialized knowledge and tools. However, the benefits of formal verification can be significant, as it can help to ensure the correctness and security of software systems.

VIII. CONCLUSION

In conclusion, smart contract vulnerabilities can have serious consequences for blockchain systems, and formal verification using frameworks such as KEVM can help mitigate these risks. The K Framework provides a set of tools for defining and executing formal specifications, including a parser, compiler, and execution engine. Overall, the K Framework provides a powerful way to specify and verify the behavior of smart contracts and can be used to ensure that contracts behave correctly under a wide range of conditions. By ensuring the correctness of smart contracts, we can increase the security and reliability of blockchain systems.

REFERENCES

[1] Atzei, Nicola, Bartoletti, Massimo, “A survey of attacks on Ethereum smart contracts”,2018.
[2] Liu J, Liu Z, “A Survey on Security Verification of Blockchain Smart Contracts”, ACCESS.2019.2921624 , IEEE Access 2019
[3] Atzei et al. "Smart Contract Vulnerabilities: Vulnerable DoesNot Imply Exploited" , 2018
[4] Ivica Nikolic, Aashish Kolluri et al. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale, 2018
[5] Sun, Tianyu Yu, Wensheng, “A formal verification framework for security issues of blockchain smart contracts”, Electronics (Switzerland), 10.3390/electronics9020255, 2020
[6] Bhargava et al. "Towards Safer Smart Contracts: A Survey of Languages and Verification Methods" by Bhargava et al.,2018
[7] Bhattacharya P, Singh A , “A Systematic Review on Evolution of Blockchain Generations ITEE Journal A Systematic Review on Evolution of Blockchain Generations”, International Journal of Information Technology and Electrical Engineering, December 2018.
[8] Praitheeshan, Purathani, Pan, Lei, “Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey”, ArxIvID 1908.08605, 2019
[9] Kshirsagar, P. R., Yadav, R. K., Patil, N. N., & Makarand L, M. (2022). Intrusion Detection System Attack Detection and Classification Model with Feed-Forward LSTM Gate in Conventional Dataset. Machine Learning Applications in Engineering Education and Management, 2(1), 20–29. Retrieved from http://yashikajournals.com/index.php/mlaeem/article/view/21
[10] Everett Hildenbrandt , Manasvi Saxena, Xiaoran Zhu et al. "KEVM: A Complete Semantics of the Ethereum Virtual Machine",2018
[11] Atzei et al. "A Survey of Attacks on Ethereum Smart Contracts (SoK)" by Atzei et al., 2017
[12] Ilya Grishchenko, “A Semantic Framework for the Security Analysis of Ethereum smart contracts”, 10.1007/978-3-319-89722- 6_10, 2018
[13] Nikolić et al. "Smart Contract Security: An Imperative for Blockchain Adoption" , 2018
[14] Conti et al, "A Classification of Blockchain-based Attacks and Vulnerabilities" , 2018
[15] Bhargava et al. "Smart Contract Security: Challenges and Future Directions" ,2019
[16] Gabriel Santos, Natural Language Processing for Text Classification in Legal Documents , Machine Learning Applications Conference Proceedings, Vol 2 2022.
[17] Bhargavan et al. ,"A Formal Verification Framework for Ethereum Virtual Machine Bytecode", 2016
[18] Tsankov et al., "Formal Verification of Smart Contracts: Short Paper", 2018
[19] Delmolino et al. ,"Formally Verified Smart Contracts in Ethereum", 2016
[20] Kavaliro et al. , "Automated Verification of Smart Contracts Using K Framework", 2019
[21] Braghin et al. , "Formal Verification of Smart Contracts withthe K Framework" ,2019
[22] Bansal et al., "Smart Contract Verification with KEVM: A Comprehensive Study", 2019
[23] Kalaivani, A. ., Karpagavalli, S. ., & Gulati, K. . (2023). Expert Automated System for Prediction of Multi-Type Dermatology Sicknesses Using Deep Neural Network Feature Extraction Approach. International Journal of Intelligent Systems and Applications in Engineering, 11(3s), 170–178. Retrieved from https://ijisae.org/index.php/IJISAE/article/view/2557
[24] Levy et al., "Formal Verification of Solidity Smart Contracts with K", 2019
[25] Arroyo et al. ,"On the Use of K for the Formal Verification of Smart Contracts" , 2020
[26] Aurrecoechea et al. ,"KEVM-IDE: A Development Environment for Smart Contracts Based on K Semantics" ,2019 [24]. Chen et al. ,"KEVM-IELE: Toward Translation Validation from EVM to IELE", 2020.