UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Fabricio Ismael Leyes Ontivero

Um estudo comparativo de ferramentas de auditoria em contratos inteligentes

Uberlândia, Brasil

2023

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Fabricio Ismael Leyes Ontivero

Um estudo comparativo de ferramentas de auditoria em contratos inteligentes

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Ivan da Silva Sendin

Universidade Federal de Uberlândia – UFU

Faculdade de Ciência da Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2023

Fabricio Ismael Leyes Ontivero

Um estudo comparativo de ferramentas de auditoria em contratos inteligentes

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 03 de Fevereiro de 2023:

Ivan da Silva Sendin
Orientador

Paulo Henrique Ribeiro Gabriel
Professor convidado

Rodrigo Sanches Miani
Professor convidado

Uberlândia, Brasil
2023

Dedico este trabalho aos meus pais que me incentivaram no processo de terminar os meus estudos, também gostaria de dedicar o trabalho ao professor Ivan Sendin que me inspirou a estudar a área de contratos inteligentes, assim como me orientar neste trabalho que foi complexo porém muito enriquecedor para meu conhecimento.

"O trabalho que nunca é começado e o que mais demora para terminar."(TOLKIEN, 1954)

Resumo

O autor Nick Szabo(SZABO, 1994) introduziu o conceito de contratos inteligentes como protocolos de transações computadorizadas que executam os termos de um contrato definido no código do contrato. Neste trabalho verificamos as vulnerabilidades registradas em contratos inteligentes, exploramos e colocamos em funcionamento ferramentas de análise de vulnerabilidades em contratos com o intuito de comparar as ferramentas assim como verificar os resultados apresentados pelas mesmas. As ferramentas Echidna, Mythril, Slither e Scribble foram testadas em diferentes contratos inteligentes. Contratos utilizados para fins didáticos como Ethernaut e SWC assim como contratos da rede `Ethereum` utilizados em NFTS e moedas virtuais obtidos pela página Etherscan. A conclusão do trabalho foi determinada após os testes das ferramentas, cada ferramenta possui suas vantagens e formas de uso, porém uma combinação de ditas ferramentas poderá ajudar o auditor de segurança dos contratos a obter resultados melhores em sua busca por falhas e vulnerabilidades.

Palavras-chave: Blockchain, vulnerabilidades, contratos inteligentes.

Lista de tabelas

Lista de abreviaturas e siglas

MVE	Máquina virtual Ethereum
ABI	Interface de aplicação binária
DAO	Organização descentralizada autônoma
P2P	Peer-to-peer (ponto a ponto)

Sumário

1 Introdução

Em 1994 o pesquisador Nick Szabo(SZABO, 1994) introduziu o conceito de contratos inteligentes, nas palavras do autor um contrato inteligente é um protocolo de transação computadorizada que executa os termos de um contrato. O objetivo geral dos contratos inteligentes é satisfazer cláusulas contratuais, minimizar exceções maliciosas e acidentais, e diminuir a necessidade de intermediários.

Em 2009 Satoshi Nakamoto(NAKAMOTO, 2009) apresentou um modelo de transação online do tipo peer-to-peer(P2P) que anula a necessidade de uma instituição intermediária. O modelo foi proposto como uma solução ao problema de duplo gasto. O duplo gasto é o que acontece quando uma moeda é utilizada 2 ou múltiplas vezes, o que requer um sistema de verificação de cada transação para lidar com este problema. O sistema funciona utilizando o método prova de trabalho (*proof of work*) eliminando a necessidade de utilizar uma entidade central para validar todas as transações.

A blockchain `Ethereum` foi uma das pioneiras no uso de contratos inteligentes, ela foi criada pelo programador Vitalik Buterin em 2015, a blockchain é caracterizada por ter seu código aberto e facilitar a seus usuários a criação de contratos inteligentes. No ano 2016 o grupo de desenvolvedores da SlockIt criou um ambiente de desenvolvimento utilizando um contrato inteligente que foi nomeado DAO(Organização autônoma descentralizada) pela comunidade da `Ethereum` , durante o período de criação qualquer usuário poderia fazer uma contribuição a um endereço de uma carteira virtual em retorno o usuário recebia 100 tokens por cada ether enviado(PRATAP, 2022). Em 17 de junho de 2016 um hacker achou uma vulnerabilidade no contrato, utilizando uma chamada recursiva na função *withdraw* o hacker retirava os fundos presentes no contrato antes de realizar a atualização da transação. Pela lógica do contrato o ether era enviado antes de realizar uma atualização do balanço, esta falha resultou em 3.6 milhões de ether (150 milhões de dólares) serem roubados em poucas horas. Este tipo de ataque é conhecido como ataque *Reentracy* e é um dos ataques mais comuns realizados a contratos inteligentes. Este evento foi um marco na história dos contratos inteligentes ressaltando a importância de realizar auditorias de segurança em contratos inteligentes que administram grandes quantias de dinheiro com o objetivo de minimizar as possíveis vulnerabilidades e erros presentes nos mesmos.

2 Revisão Bibliográfica

2.1 Contratos Inteligentes

O objetivo geral de um contrato inteligente é satisfazer condições contratuais comuns, minimizar exceções maliciosas ou acidentais e minimizar a necessidade de intermediários. Contratos inteligentes utilizam protocolos e interfaces para facilitar o processo de contratação, esta forma de processo digital é muito mais eficiente que seu antecessor de papel (SZABO, 1997). Um exemplo prático da vida real que pode ser considerado como o antecessor dos contratos inteligentes é o da máquina de vendas, a máquina recebe moedas em troca de seus produtos. Na blockchain da `Ethereum` os contratos inteligentes funcionam como contas `Ethereum` afinal eles possuem balanço e podem ser alvos de transações, no entanto eles agem de maneira autônoma independente de usuários, os usuários podem interagir com os contratos em funcionamento na rede da `Ethereum` . Dentro dos contratos é preciso declarar sua versão do pragma, esta é a versão da linguagem solidity, assim como declarar uma função do tipo `payable` que permite ao contrato receber ether.

2.2 Ethereum

Criada pelo programador Vitalik Buterin em 2015, o `Ethereum` é uma blockchain descentralizada com código aberto que permite aos usuários a criar seus próprios contratos inteligentes com o uso da linguagem Solidity. Sua proposta é facilitar e descentralizar as transações feitas por seus usuários através de contratos inteligentes que podem ser criados pelos próprios usuários(BUTERIN, 2014). A máquina virtual de `Ethereum` é uma máquina de estado baseada em transações. O estado pode incluir informações como balanço de conta, reputação, arranjos de confiança e dados do mundo real. As transações podem ser 'válidas' ou 'inválidas'. Um estado de transação inválida pode ser o caso de uma redução de valor em uma conta sem o aumento equivalente em outra conta. As transações são agrupadas em blocos utilizando um `hash` criptográfico como referência. Blocos funcionam como um `log`(registro), guardando uma série de transações com os blocos antigos e um identificador do estado final. A moeda utilizada dentro da máquina virtual `Ethereum` é o Ether, a mínima denominação do Ether é o Wei, a lista completa é mostrada na Tabela 1.

Contas no `Ethereum` são compostas por quatro items(WOOD, 2015):

nó é um valor igual ao número de transações enviados por este endereço ou no caso de contas com código associado, número de criação de contratos feitos por esta conta;

Multiplicador	Nome
10^{10}	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

Tabela 1 – Sub Denominações do Ether

balanço valor equivalente ao número de Wei presentes em este endereço;

raiz de armazenamento um `hash` de 256 bits do nodo raiz da árvore Merkle que codifica o conteúdo do armazenamento da conta;

código hash é o código `hash` da máquina virtual `Ethereum` de esta conta, este é o código executado quando este endereço recebe uma mensagem de chamada.

2.3 Gás

Na máquina virtual de `Ethereum` todas as transações estão sujeitas a taxa paga em gás, o custo do gás é decidido de maneira universal. Toda transação possui um custo limite de gás `gasLimit`. Esta quantia é implicitamente comprada pela conta do usuário que envia a transação, o valor do gás (`gasPrice`) é especificado antes da transação. O gás restante da transação é retornado para a conta do comprador. O gás não existe fora de uma transação. Usuários podem escolher o valor do gás em suas transações, porém os mineiros podem ignorar estas transações caso o valor do gás não seja o desejado.

2.4 Solidity

Solidity (SOLIDITY-DOCS, 2020) é uma linguagem orientada a objetos, projetada para criar contratos inteligentes para a máquina virtual `Ethereum` (EVM), é uma linguagem de tipagem estática, possui suporte para herança, bibliotecas, assim como tipos complexos definidos pelos usuários. A linguagem foi influenciada por outras linguagens famosas como C++, Javascript e Python. Porém a influência da linguagem C++ é percebida na sintaxe para declaração de variáveis, conceitos como sobreposição de funções, conversões implícitas e explícitas de tipos de variáveis.

2.4.1 Tipos de funções

Funções na linguagem Solidity são divididas em dois tipos, externas e internas. As funções externas podem ser vistas e acessadas por contratos fora do contrato atual, enquanto as funções internas podem ser utilizadas exclusivamente pelo próprio contrato.

Por definição padrão as funções são declaradas como internas. As funções ainda podem ter os seguintes atributos:

Payable a função ou construtor marcado com a variável `payable` pode receber Ether, caso uma função não seja marcada como `payable` o ether enviado a função será retornado. É possível enviar a quantia de 0 ether para uma função `payable`, afinal elas também são `non-payable`, porém é possível converter a função em `non-payable`.

Non-payable funções deste tipo não podem receber ether e não podem ser convertidas em `payable`.

2.4.2 Aspectos gerais

Algumas características relevantes encontradas na linguagem Solidity:

Inteiros os valores podem ser marcados como int a int256 em "saltos"de 8 bits, e os valores sem sinal são marcados como `unsigned` também até o `uint256`;

address armazena um valor de 20 bits, tamanho do endereço em `Ethereum` ;

address payable cumpre a mesma função que o endereço comum com o adicional de possibilitar uma transação para o endereço. As funções `send` e transfer podem enviar ether para endereços marcados como `payable`;

transfer a função `transfer` requer um endereço do tipo payable para poder enviar ether, durante a execução em qualquer tipo de falha ou falta de gás a transação é revertida;

send a função `send` é a versão de baixo nível da função transfer, em caso de falha na transferência a função `send` retorna o valor falso;

call, delegatecall, staticcall é possível interagir com contratos que não aderem a ABI utilizando as funções `call, delegatecall e staticcall`. Todas utilizam bytes de memória como parâmetro e retorna um valor boolean (bytes de memória) e dados (bytes de memória);

fallback a função `fallback` é ativada quando não existe outra função com a assinatura requisitada pelo remetente da mensagem. A função `fallback` deve ser marcada como `external`, é possível receber ether nesta função utilizando o modificador `payable`. Esta função possui um limite de gás de 2300 para completar sua função;

receive esta função não pode ter argumentos e não retorna qualquer valor, deve possuir os modificadores `external` e '`payable`'. A função `receive` é chamada quando uma mensagem com dados vazios é enviada para o contrato, também é utilizada quando

as funções `.send()` e `.transfer()` são chamadas para transferir ether para o contrato no qual a função `receive()` está presente. Em caso do contrato inteligente não possuir as funções `fallback` com modificador `payable`ou `receive` e não possuir alguma função com o modificador `payable`uma exceção será lançada e a transferência de ether será revertida devido que o contrato não pode receber ether neste caso.

2.4.3 Tratamento de erros em Solidity

A linguagem solidity utiliza exceções de reversão de estado para tratar erros. A exceção remove as mudanças realizadas no estado na chamada atual e marca um erro para o remetente. As estruturas utilizadas no mecanismo de tratamento de erro são:

assert a função `assert` verifica uma condição caso seja falsa cria um erro do tipo `Panic(uint256)`. Esta função deve ser implementada para lidar com erros internos.

require assim como a função `assert` verifica uma condição, caso seja falsa a função cria um erro sem nenhum dado ou um erro do tipo *Error(string)*, retornando uma mensagem de erro. Deve ser utilizada para validar condições que não podem ser detectadas até o tempo de execução.

revert uma reversão pode ser ativada utilizando o estado `revert` ou chamando a função `revert()`. O estado `revert` toma um erro específico ex: `revert` errotest() como argumento. É possível adicionar uma mensagem na função `revert()`, caso não seja provido uma mensagem a função não retorna dados.

2.5 Chamadas de mensagem

De acordo com (BUTERIN, 2014) contratos possuem a habilidade de enviar mensagens para outros contratos. Mensagem são objetos virtuais que nunca são serializados e existem apenas no ambiente de execução da Ethereum. Uma mensagem contém:

- O remetente da mensagem.
- O recipiente da mensagem.
- A quantidade de ether transferida com a mensagem.
- Um campo com dados opcionais.
- O valor inicial do gás.

Uma mensagem é como uma transação, porém é produzida por um contrato inteligente e não por um ator externo. Contratos podem interagir com outros contratos da mesma forma que atores externos interagem entre eles.

2.6 Transações

Transações em `Ethereum` são instruções assinadas criptograficamente construídas por atores externos ao escopo da blockchain. O remetente de uma transação não pode ser um contrato. As transações são validadas por mineradores que serão recompensados com o recebimento de Ether, as transações são agrupadas em blocos que compõem a blockchain. As transações possuem os seguintes campos em comum:

- Tipo da transação.
- Valor representando o número de transações enviadas pelo remetente.
- Preço do gás.
- Limite do gás.
- Endereço de 160 bits do remetente da chamada de mensagem ou para transação de criação de um contrato.
- Valor equivalente em Wei a ser transferido para o destinatário da chamada de mensagem ou no caso de criação de contrato como pagamento para a nova conta criada.
- Assinatura da transação utilizada para determinar o remetente da transação.

Sua proposta é facilitar e descentralizar as transações feitas por seus usuários através de contratos inteligentes que podem ser criados pelos próprios usuários(BUTERIN, 2014). A máquina virtual de `Ethereum` é uma máquina de estado baseada em transações. O estado pode incluir informações como balanço de conta, reputação, arranjos de confiança e dados do mundo real. As transações podem ser ’

2.7 Tokens

Na EVM tokens podem ser virtualmente qualquer coisa: bilhetes de loteria, pontos de habilidade de personagens ou uma moeda corrente como o dólar. O token ERC-20 foi proposto pelos autores (VOGELSTELLER; BUTERIN, 2015), o objetivo do token ERC-20 é permitir a reutilização do token em outras aplicações. O padrão do Ethereum é o token ERC-20 que permite alterações e interações com seus produtos e serviços.

O token ERC-721 foi proposto pelos autores (ENTRIKEN et al., 2018) o token erc-721 é utilizado para rastrear e transferir NFTs. NFTs podem representar propriedade sobre patrimônio digital ou físico. Os NFTs são identificados por um número de id único(uint 256), dentro dos contratos ERC-721. Este número não pode ser alterado.

2.8 Vulnerabilidades

Os autores (KUSHWAHA et al., 2022) da revisão sistemática realizada em 2022 reconheceram 17 tipos de vulnerabilidades recorrentes nos códigos solidity, são as seguintes:

1. Negação de serviço com limite de gás do bloco: cada bloco na EVM possui um limite de 10,000,000 gás que pode ser utilizado para realizar as operações. No caso de uma transação exceder o limite, a transação falha por negação de serviço, pode acontecer com arrays que aumentam de tamanho com o tempo.

2. Negação de serviço com falhas na chamada: chamadas externas podem falhar, por erro do programador ou por ação maliciosa de um *hacker*. É possível evitar que outros contratos interajam com o contrato que está sofrendo o ataque, com várias chamadas dentro de uma transação é possível gerar um loop que deixará o contrato parado

3. Geração de número randômico utilizando o bloco `hash`: é perigoso utilizar a função hash como número randômico, afinal é possível decifrar qual será o número gerado.

4. Uso da variável `tx.origin`: Não é recomendado utilizar a variável `tx.origin` para verificar acesso ao contrato dado que é possível passar por cima desta verificação usando um contrato intermediário.

5. Estouro da variável inteiro: este tipo de erro acontece quando é atingido o limite superior ou inferior da variável.

6. Reentrância: também conhecido como ataque de chamada recursiva. O contrato malicioso pode realizar diversas chamadas antes do processamento da primeira, possibilitando executar repetidas chamadas de retirada de ether como no famoso ataque DAO.

7. Exceção sem tratamento: quando um contrato executa uma chamada com outro contrato, caso a chamada retorne um erro ou exceção a falta de tratamento do retorno pode ser aproveitada como uma vulnerabilidade.

8. Transferência de fundos com a função `send` sem gás: quando a função `send` é utilizada para transferir fundos para outro contrato, o contrato que recebe o ether pode

possuir um limite de gás igual a 2300 na sua função `fallback`. Neste caso o contrato pode retornar uma exceção. Caso o retorno não for devidamente tratado é possível aproveitar esta vulnerabilidade para ficar com o ether que deveria ser enviado porém foi retornado.

9. Padrão de gás custoso: Todas as instruções nos contratos da Ethereum requerem gás como imposto. Códigos não otimizados podem conter códigos irrelevantes ou padrões caros que podem gerar um grande custo de execução.

10. Chamadas para contratos desconhecidos: A execução da função `fallback` sobre certas condições. A função chamada não existe. Isto pode acontecer devido a invocação de certas funções do contrato remetente ou a transferência de ether para outro contrato.

11. Colisão de hash com múltiplos argumentos de tamanhos variáveis: a função `abi.encodedPacked()` é um modo não padrão que é utilizado para indexar parâmetros empacotados. Esta função pode levar a colisão de `hash` pelo tamanho variável dos múltiplos argumentos. O *hacker* pode explorar esta vulnerabilidade em verificações de assinaturas mudando a posição dos elementos.

12. Aproveitamento de falta de gás: Contratos inteligentes que aceitam dados de fontes externas e utilizam os mesmos em uma sub chamada para outro contrato são vulneráveis a este tipo de ataque. No caso de uma falha na sub chamada toda a transação pode ser revertida. O *hacker* pode censurar a transação utilizando uma quantia limitada de gás suficiente para executar a chamada sem executar a sub chamada.

13. Retirada de ether sem proteção: Falta de proteção ou medidas de acesso inadequadas podem levar a retiradas de ether por contratos alheios.

14. Pragma flutuante: O contrato deve ser compilado pelo mesmo compilador que foi utilizado na fase de testes. Antes de carregar o contrato na blockchain, é recomendado que a versão pragma seja declarada como fixa.

15. Visibilidade da função padrão: O uso incorreto dos especificadores de visibilidade pode gerar vulnerabilidades nos contratos.

16. Chamada delegada: A chamada delegada pode ser executada quando não conhecemos a ABI(interface de aplicação binária). A chamada delegada pode ser utilizada por um *hacker* para gerar execuções inesperadas como autodestruição do contrato e perda de balanço.

17. Comando `‘self-destruct’` sem proteção: Após a execução da instrução `‘self-destruct’` é feita a transferência do balanço presente no contrato para outro

contrato predefinido. Caso existam falhas de controle de acesso, o *hacker* pode destruir o contrato para retirar o ether presente no mesmo.

2.9 Ferramentas de análise

Ferramentas de análise podem ser utilizadas por auditores de contratos inteligentes para auxiliar na procura por falhas e vulnerabilidades nos contratos analisados em uma análise primária, ajudando a destacar as falhas mais claras presentes nos mesmos. As ferramentas funcionam utilizando detectores que verificam falhas tipificadas nas mesmas, evitando erros previamente identificados por outros programadores da área de contratos inteligentes. O uso das ferramentas facilita a análise de contratos em grande escala que pode se tornar uma tarefa árdua para os auditores dos contratos. A seguir descrevemos as principais ferramentas disponíveis.

2.9.1 Slither

Slither (FEIST; GRIECO; GROCE, 2019) é uma ferramenta de análise estática, ferramentas de análises estáticas fazem análise do código sem executá-lo. A análise é feita em múltiplos estágios, primeiro Slither recebe como dado inicial a árvore de sintaxe abstrata de Solidity (AST) gerada pelo compilador Solidity do código fonte do contrato. No primeiro estágio, Slither recupera informação importante como o gráfico de herança do contrato e a lista de expressões. Slither transforma o código inteiro do contrato para SlitherIR, sua linguagem de representação interna. SlitherIR utiliza uma análise estática singular para facilitar a computação de uma variedade de análise de código. No terceiro estágio, a análise de código, Slither computa um conjunto de análises predefinidas, que fornecem informação para os outros módulos.

- Detecção de vulnerabilidades automáticas: uma grande variedade de bugs presentes em contratos inteligentes podem ser identificados sem intervenção do usuário.
- Optimização automática do código: Slither consegue detectar a optimização de código que o compilador não detectou.
- Entendimento de código: slither possui impressoras que demonstram as informações do contrato ao usuário facilitando o entendimento do mesmo.
- Revisão de código assistida: o usuário pode interagir com Slither por meio da API.

2.9.2 Echidna

Echidna é uma ferramenta que utiliza a técnica denominada *fuzzing*, a técnica consiste em inserir um conjunto de dados aleatórios de vários tipos diferentes para encontrar

bugs ou falhas inesperadas. Echidna foi desenvolvida pelo time da Crytic, o artigo (GRIECO et al., 2020) descreve a ferramenta. Ela é utilizada para inserir uma variedade de dados diferentes em contratos inteligentes para realizar testes de vulnerabilidades, falhas e comportamentos inesperados. O funcionamento da ferramenta é dividido em 2 etapas, a etapa inicial utiliza a ferramenta Slither para compilar os contratos e analisar eles com o objetivo de identificar constantes e funções que administram Ether. Na segunda etapa é feito um processo interativo gerando transações aleatórias por meio da interface binária fornecida pelo contrato, constantes definidas no contrato e qualquer conjunto de transações do corpus. Uma das características do echidna é a possibilidade de mudar as configurações de maneira manual, echidna possui 30 configurações com especificações padrão para diminuir a complexidade da ferramenta, porém elas podem ser alteradas utilizando um arquivo YAML. É possível fazer uso da ferramenta mediante a imagem do docker chamada *security-toolbox*.

2.9.3 Mythril

Mythril é uma ferramenta de análise simbólica utilizada para descobrir vulnerabilidades em contratos inteligentes no ambiente `Ethereum` (MUELLER, 2018). A ferramenta utiliza o interpretador de bytecode `Ethereum` chamado LASER, com ele é obtido o estado dá conta do contrato `Ethereum` . É produzido um caminho simbólico que será resolvido pelo programa Z3, o Z3 solver é um programa da Microsoft Research que resolve teoremas verificando a veracidade das fórmulas lógicas em um ou mais teoremas. Caso algum dos detectores da ferramenta Mythril for accionado em algum estado indesejado o Z3 é acionado para verificar o caminho possível até o estado.

2.9.4 Scribble

Scribble é uma linguagem de verificação, esta ferramenta verifica o código em execução e traduz especificações de alto nível para código Solidity. A ferramenta permite comentar contratos inteligentes com propriedades ao invés de escrever em um arquivo diferente. Scribble transforma anotações em asserções que verificam especificações. Após escrever as propiedades do contrato, os desenvolvedores podem utilizar diversas ferramentas para verificar os contratos, ferramentas como Mythx, Mythril e Diligence Fuzzing.

2.10 Detectores das ferramentas

Detectores são testes realizados pelas ferramentas, caso os testes executem de maneira esperada eles geram um alerta associado ao detector, as ferramentas produzem um arquivo de saída informando os detectores que foram acionados durante a análise do contrato com o intuito de remediar a falha ou vulnerabilidade descoberta pela ferramenta.

Lista de detectores da ferramenta Mythril pode ser verificada na página da ferramenta Mythril (CONSENSYS, 2022) e na Tabela 2.

Tipo de vulnerabilidade	SWC-ID
Integer Overflow e Underflow	101
Valor de retorno não verificado	104
Retirada de ether sem proteção	105
Instrução SELFDESTRUCT sem proteção	106
Reentrância	107
Violação de asserção	110
Uso de funções solidity depreciadas	111
Uso da chamada delegada	112
DoS com chamada falha	113
Valores de bloque utilizados como proxy para medir o tempo	116
Fonte não confiável para número randômico	120
Escrita para local de armazenagem arbitrário	124
Pulo arbitrário com variável do tipo função	127

Tabela 2 – Lista de detectores Mythril

A lista de detectores Slither foi realizada utilizando como referência os detectores da página da ferramenta (SLITHER, 2022) e comparando os resultados obtidos das análises dos contratos do registro SWC. Lista de detectores da ferramenta Slither na Tabela 3.

Tipo de vulnerabilidade	SWC-ID
Versão de compilador antiga	102
Pragma flutuante	103
Valor de retorno não verificado	104
Instrução SELFDESTRUCT sem proteção	106
Reentrância	107
Ponteiro de armazenamento não instanciado	109
Violação de asserção	110
Uso de funções solidity depreciadas	111
Chamada delegada	112
Autorização com tx.origin	115
Valores de bloque utilizados como proxy para medir o tempo	116
Fonte não confiável para número randômico	120
Erro tipográfico	129
Presença de variáveis sem uso	131
Código sem uso	135

Tabela 3 – Lista de detectores Slither

2.10.1 Resumo comparativo das ferramentas

Nome da ferramenta	Usabilidade	Funcionamento	Desvantagens
Slither	Código aberto, uso automático, produz dados de saida de interpretação simples.	Slither realiza análise estática do código do contrato, utilizando detectores para revelar as vulnerabilidades.	Os resultados obtidos pelos testes revelam que a ferramenta deixou de identificar várias falhas.
Mythril	Código aberto, uso automático, dados de saida detalhados com os tipos de vulnerabilidades e falhas.	A ferramenta utiliza análise simbólica para detectar vulnerabilidades. Utilizando transações para explorar os caminhos produzidos pelo LASER.	A ferramenta Mythril é uma versão limitada da ferramenta Mythx, possui menos detectores de vulnerabilidades.
Echidna	Código aberto, uso manual nos contratos, output de interpretação simples.	A ferramenta scrible em conjunto com outra ferramenta como Mythril ou Mythx realiza testes, para verificar a asserção criada pelo usuário.	A ferramenta é limitada pela ferramenta utilizada para realizar a verificação da asserção, exemplo: usar a ferramenta Mythril a busca será limitada pelos detectores da mesma.
Scribble	Código aberto, uso manual nos contratos, output de interpretação simples.	A ferramenta scrible em conjunto com outra ferramenta como Mythril ou Mythx realiza testes, para verificar a asserção criada pelo usuário.	A ferramenta é limitada pela ferramenta utilizada para realizar a verificação da asserção, exemplo: usar a ferramenta Mythril a busca será limitada pelos detectores da mesma.

Tabela 4 – Tabela das ferramentas de análise

3 Trabalhos Correlatos

No pesquisa feita pelos autores (KUSHWAHA et al., 2022), são apresentados os diferentes tipos de vulnerabilidades conhecidos de 2016 a 2021, assim como ferramentas de análises de contratos. Ataques reais foram documentados no artigo assim como métodos para remediar as vulnerabilidades exploradas nos ataques. Os autores elaboraram um quadro que apresenta as ferramentas populares de análise de contratos com características como disponibilidade, tipo de ferramenta, tipo de análise e linguagem de implementação.

No trabalho de mestrado de (LEID, 2020), inicialmente o autor descreve a blockchain `Ethereum` e suas características, contas, transações, mensagens e contratos. Continua detalhando a linguagem Solidity, funções, tipos, sintaxe e outras características. Descreve as vulnerabilidades presentes em contratos inteligentes assim como métodos que devem ser evitados em desenvolvimento de contratos inteligentes. Lista as ferramentas que são analisadas pelo autor do trabalho e faz uma descrição de cada uma delas, as ferramentas utilizadas pelo autor foram Mythril, Mythix, Manticore, Slither, Securify, Echidna. O foco do trabalho está na parte de desenvolvimento na qual o autor do trabalho faz diferentes testes com as ferramentas em vários tipos de conjuntos de contratos inteligentes de fontes como Ethernaut, Capture the Ether, que são contratos utilizados para aprendizagem na área de contratos inteligentes, assim como testes em contratos utilizados em moedas virtuais no mundo real, foram analisados os 20 tokens ERC-20 listados em ordem de acordo com sua límite de mercado.

4 Desenvolvimento

4.1 Instruções de uso da ferramenta Echidna

Para utilizar a ferramenta Echidna é preciso criar uma função com o nome echidna do tipo público, retorna um booleano. Dentro da função é colocada uma asserção, a ferramenta Echidna verifica métodos possíveis para negar a asserção ou "falhar o teste". O contrato token-with-backdoor.sol foi obtido no registro SWC, o contrato é do autor Josselin Feist, adaptado pelo autor Bernhard Mueller.

A ferramenta Echidna realiza uma série de inputs de diversos dados em um processo conhecido como 'fuzzing', o objetivo da ferramenta é verificar diversos resultados produzidos pelos dados inseridos no contrato analisado. É possível fazer testes através de funções chamadas de invariantes, estas funções devem ser declaradas com o nome 'echidna', devem retornar o tipo booleano, e não podem receber argumentos de entrada.

```
1  /*
2   * @source: TrailofBits workshop at TruffleCon 2018
3   * @author: Josselin Feist (adapted for SWC by Bernhard Mueller)
4   * Assert violation with 3 message calls:
5   * - airdrop()
6   * - backdoor()
7   * - test_invariants()
8   */
9  pragma solidity ^0.4.25;
10
11 contract Token {
12     mapping(address => uint256) public balances;
13
14     function airdrop() public {
15         balances[msg.sender] = 1000;
16     }
17
18     function consume() public {
19         require(balances[msg.sender] > 0);
20         balances[msg.sender] -= 1;
21     }
22
23     function backdoor() public {
24         balances[msg.sender] += 1;
25     }
26
27     function test_invariants() {
```

```
28          assert(balances[msg.sender] <= 1000);
29      }
30
31      function echidna_test() public returns (bool) {
32          return balances[msg.sender] <= 1000;
33      }
34  }
```

Para realizar a análise inserimos no terminal o código echidna-test token-with-backdoor.sol –contract Token. Na imagem 1 podemos conferir os resultados apresentados pela ferramenta, a ferramenta indica uma sequência de chamadas(*sequence call*) realizadas no teste.

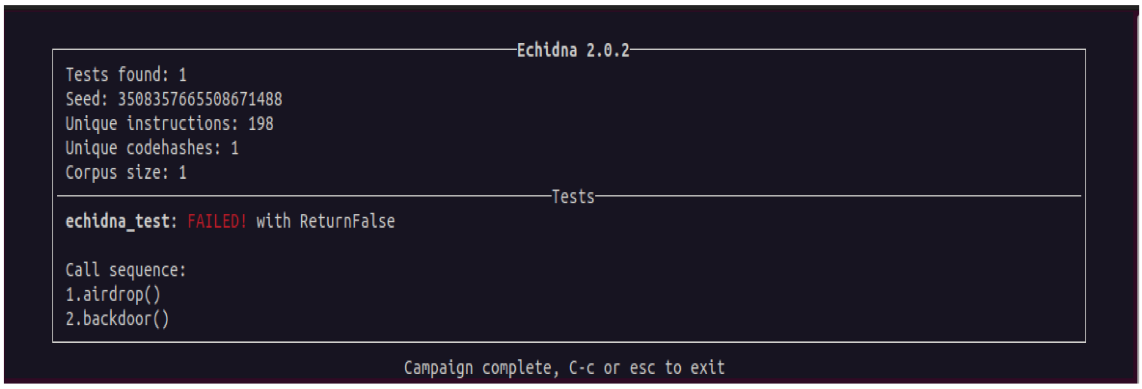

Figura 1 – Resultado do teste Echidna

4.2 Instruções de uso da ferramenta Scribble

A ferramenta Scribble permite ao usuário comentar o código transformando esse comentários em asserções. As asserções serão verificadas pela ferramenta complementar que pode ser a ferramenta Mythril ou Mythx. O contrato usado no exemplo é o contrato 'overflow-simple-add.sol' foi extraído do site SWC Registry.

```
1  pragma solidity 0.4.25;
2
3  contract Overflow_Add {
4      uint256 public balance = 1;
5
6      /// #if_succeeds {:msg 'A variavel balance nao sofre overflow'}
           balance < 0;
7      function add(uint256 deposit) public {
8          balance += deposit;
9      }
10 }
```

Em caso de falha da asserção será gerado a mensagem ‘ uma asserção provida pelo usuário falhou’. A interface do resultado é a da ferramenta utilizada em conjunto com Scribble, neste caso é a interface da ferramenta Mythril a qual é exibida no final da análise. A análise do contrato é feita com o comando ’scribble overflow-simple-add.sol –output-mode files –instrumentation-metadata-file metadata.json –arm’. O comando ‘scribble’ carrega o contrato ‘overflow-simple-add.sol’, com o comando –instrumentation-metadata-file metadata.json é gerado um arquivo json com os metadados do contrato analisado, o comando arm gera um arquivo do tipo .instrumented o qual será analisado pela ferramenta mythril. Na imagem 2 podemos analisar o resultado produzido pela ferramenta Scribble em conjunto com a ferramenta Mythril.

```
==== Exception State ====
SWC ID: 110
Severity: Medium
Contract: Overflow_Add
Function name: add(uint256)
PC address: 332
Estimated Gas Usage: 6996 - 27657
A user-provided assertion failed.
A user-provided assertion failed.
--------------------
In file: overflow-simple-add.sol:13

AssertionFailed("0: A variavel balance não sofre overflow")
```

Figura 2 – Resultado do teste Scribble

As ferramentas de análise de contratos serão colocadas para atuar em diferentes cenários analisando contratos didáticos assim como contratos do mundo real, os contratos obtidos de fontes diferentes como Etherscan, Openzeppelin, SWC Registry compoem o data set utilizado neste trabalho. Primeiro as ferramentas Mythril e Slither analisaram os contratos do SWC Registry para verificar os seus detectores contra as diferentes vulnerabilidades listadas no registro. Continuando a ferramenta Mythril analisou 10 tokens ERC20 e ERC721 listados como os mais utilizados no site Etherscan, a ferramenta foi testada em dois cenarios diferentes, utilizando uma e duas transações observando a quantia de erros detectada pela ferramenta assim como o tempo para concluir a análise dos tokens. Finalizando todas as ferramentas analisaram os 26 desafios do Ethernaut com o intuito de detectar a vulnerabilidade principal presente no desafio, listando como 1 caso a ferramenta tivesse sucesso e 0 em caso de falha.

4.3 Conjunto de dados

O SWC Registry(SWCREGISTRY, 2022) é uma página com os registros de diferentes tipos de vulnerabilidades comuns em contratos inteligentes, as vulnerabilidades

listadas possuem uma lista de exemplo de contratos inteligentes com as vulnerabilidades citadas para melhor entendimento das falhas, assim como descrições de possíveis métodos para corrigir as falhas descritas. O dataset utilizado neste trabalho é composto por contratos didáticos obtidos na página da openzeppelin e do registro de vulnerabilidades SWC Registry. Os 10 tokens ERC20 e ERC721 utilizados em contratos inteligentes presentes em moedas virtuais e NFTs foram obtidos na página da Etherscan (ETHERSCAN, 2022). Os contratos da página Etherscan foram listados como os 10 mais utilizados na data 10/10/2022.

SWC-ID	Tipos de vulnerabilidade	Contratos
100	Visibilidade padrão da função	1
101	Overflow e underflow	6
102	Versão de compilador não atualizada	1
103	Pragma flutuante	3
104	Retorno não verificado	1
105	Função de retirar Ether sem proteção	6
106	Instrução SELFDESTRUCT sem proteção	3
107	Re-entrancia	2
108	Variável de estado com visibilidade padrão	1
109	Ponteiro de armazenamento não instanciado	1
110	Violação da função assert	10
111	Uso de funções solidity depreciadas	1
112	Chamada delegada para usuário não confiável	1
113	DoS com falha de chamada	1
114	Dependência de ordem de transação	1
115	Autorização com função tx.origin	1
116	Uso de block.number, block.timestamp	2
117	Maleabilidade de assinatura	1
118	Nome de construtor incorreto	2
119	Variáveis de estado parecidas	2
120	Uso de valores randômicos com vulnerabilidade	3
123	Violação de requirement	1
124	Falta de gás	2
125	Ordem de herança incorreto	1
126	Aproveitamento de falta de gás	1
127	Pulo arbitrário com variável do tipo função	1
128	Dos com limite de gás do bloco	3
129	Erro tipográfico	3
130	Super fluxo de caractere de controle direita para esquerda	1
131	Presença de variáveis sem uso	2
132	Balanço de ether inesperado	1
133	Colisão de hash com argumentos de comprimento variável múltiplo	1
134	Chamada de mensagem com gás específico	1
135	Código sem efeito	2
136	Dados privados não criptografados na blockchain	1

Tabela 5 – Lista de vulnerabilidades comumente encontradas em Contratos Inteligentes. Fonte: (SWCREGISTRY, 2022)

4.4 Ferramentas

As ferramentas foram colocadas em cenários diferentes para observar o resultado produzido pelas mesmas após a análise dos contratos do conjunto de dados. Ferramentas como Mythril e Slither podem ser utilizadas em situações gerais por sua simplicidade de uso e configuração, enquanto ferramentas como Echidna e Scribble demandam do usuário

mais tempo para entender a lógica do contrato com o intuito de escrever asserções que consigam detectar falhas no contrato.

5 Resultados

A tabela 6 apresenta os resultados obtidos após as análises do registro SWC Registry que possui vulnerabilidades comumente encontradas em contratos inteligentes. Os detectores das ferramentas estão associados a estas vulnerabilidades, é possível verificar cada um dos detectores comparando com os contratos de testes encontrados no SWC Registry. Na coluna das ferramentas é colocado em quantos dos contratos analisados a vulnerabilidade do contrato foi encontrada pela ferramenta. Na coluna da direita está o número de contratos analisados em cada registro do SWC.

SWC-ID	Slither	Mythril	Contratos
100	1	0	1
101	0	5	6
102	1	0	1
103	3	0	2
104	1	1	1
105	1	3	6
106	3	2	3
107	2	2	2
108	0	0	1
109	1	0	1
110	1	7	10
111	1	0	1
112	1	1	1
113	1	1	1
114	0	0	1
115	1	1	1
116	1	2	2
117	0	0	1
118	0	0	2
119	0	0	2
120	1	1	3
123	0	0	1
124	0	2	2
125	0	0	1
126	0	0	1
127	0	1	1
128	0	0	3
129	3	0	3
130	0	0	1
131	1	1	2
132	1	0	1
133	0	0	1
134	0	0	1
135	0	0	2
136	0	0	1

Tabela 6 – Resultados das análises nos contratos do SWC

A Tabela 7 apresenta os erros registrados pela ferramenta Mythril após analisar os tokens ERC20, assim como o tempo para finalizar as análises de acordo com a quantia de transações utilizada no processo.

ERC20	ERROS		TEMPO	
	T1	T2	T1	T2
Tether	0	0	1	135
USD Coin	2	2	2	100
BNB	0	0	1	30
Binance USD	1	1	1	44
Matic Token	0	0	2	160
SHIBA	0	0	2	68
Dai Stablecoin	0	0	3	186
HEX	12	4	61	104
stETH	6	6	1	2
Theta Token	4	5	1	30

Tabela 7 – Tabela de análise dos tokens ERC20.

Podemos observar os resultados das análises dos tokens ERC721 na Tabela 8.

ERC721	ERROS		TEMPO	
	T1	T2	T1	T2
DeepObjects.Ai	0	0	83	1441
ETHEREUM NAME SERVICE	2	6	2	60
BoredRicksWubbaClub	2	3	4	185
TheFdaCharactersCollection	10	10	52	65
GardenLockDown	4	4	2	31
Fire	0	0	2	101
NekoNation	0	0	12	900
TheBabyChimpsons	0	0	4	502
FigmaLabGenesis	0	3	8	24
GENESIS HUSTLER	2	3	5	296

Tabela 8 – Tabela de análise dos tokens ERC721.

Na Tabela 9 é possível observar os erros descobertos pela ferramenta Mythril nos contratos ERC20 e ERC721, os erros foram associados ao registro SWC com seu id da vulnerabilidade presente no registro.

SWC-ID	Mythril ERC20	Mythril ERC721
101	2	1
104	0	0
105	0	0
106	0	0
107	4	11
110	3	0
111	0	0
112	4	0
113	1	1
115	0	5
116	8	11
120	4	0
124	0	0
127	0	0

Tabela 9 – Erros presentes nas análises dos tokens ERC20, ERC721

A página Ethernaut da Openzeppelin possui diversos desafios de contratos inteligentes com vulnerabilidades com um amplo rango de dificuldades. Estes desafios podem ser utilizados por iniciantes que desejam melhorar seu entendimento sobre vulnerabilidades em contratos inteligentes. Os contratos da página Ethernaut serão analisados pelas ferramentas Mythril, Slither, Echidna, Scribble em conjunto com a ferramenta Mythril.

Ethernaut	Slither	Mythril	Echidna	Scribble
Fallback	0	0	1	0
Fallout	0	0	1	1
CoinFlip	1	1	1	1
Telephone	0	1	0	1
Token	0	1	1	1
Delegation	1	1	0	1
Force	0	0	0	0
Vault	0	0	1	1
King	1	0	0	0
Re-entrancy	1	1	0	1
Elevator	0	1	0	1
Privacy	0	0	0	0
GatekeeperOne	0	0	0	0
GatekeeperTwo	0	0	0	0
NaughtCoin	0	0	0	0
Preservation	1	1	0	1
Recovery	0	1	0	1
MagicNumber	0	0	0	0
AlienCodex	0	0	0	0
Denial	1	1	0	1
Shop	0	0	0	0
Dex	0	0	0	0
Dex2	0	0	0	0
PuzzleWallet	0	0	0	0
MotorBike	0	0	0	0
DoubleEntryPoint	0	0	0	0

Tabela 10 – Resultados das análises dos exercícios Ethernaut. Fonte: (THE. . . , 2022)

Na Tabela 11 foram listadas as ferramentas de auditoria de segurança, assim como a quantidade de contratos analisados e o tipo de contrato analisado pela ferramenta. O 'X' denota que a ferramenta não analisou o contrato, porém o 'V' na tabela define que a ferramenta analisou o contrato marcado na tabela.

	SWC	Ethernaut	ERC-20 e ERC-721
Contratos	70	26	20
Mythril	V	V	V
Slither	V	V	X
Scribble	X	V	X
Echidna	X	V	X

Tabela 11 – Quadro comparativo com os contratos analisados por cada ferramenta

6 Conclusão

Neste trabalho listamos as vulnerabilidades registradas em contratos inteligentes, colocamos diversas ferramentas de análise de contratos inteligentes para analisar contratos obtidos em fontes diferentes, contratos de uso real assim como contratos de uso didático. As ferramentas assim como os contratos analisados são de livre distribuição e aberto ao público, as ferramentas podem ser obtidas no Github e os contratos na página Etherscan. É possível observar que cada uma das ferramentas apresentadas no trabalho possuem suas vantagens e pontos fracos. Com o intuito de obter melhores resultados em uma auditoria de segurança com os contratos inteligentes é possível utilizar um conjunto de ferramentas que atuem de forma complementar cobrindo mais opções identificando vulnerabilidades nos contratos analisados. Ferramentas automáticas como Mythril e Slither podem ser utilizadas em uma análise inicial seguida por testes manuais com as ferramentas de asserções Echidna e Scribble para uma análise personalizada. As ferramentas Echidna e Scribble precisam de testes de asserções escritos pelo auditor dos contratos, isto requer tempo para entender a lógica dos contratos com o objetivo de realizar testes manuais direcionados a possíveis falhas ou vulnerabilidades nos contratos. Enquanto as ferramentas Slither e Mythril são de uso automático sem necessidade de configuração ou testes manuais facilitando o uso das mesmas. Ferramentas como Mythril podem ser utilizadas em computadores mais potentes para obter melhores resultados no uso de análises mais complexas ou com mais transações, afinal estas análises requerem maior poder de processamento de dados. É recomendado no final do uso das ferramentas realizar uma análise manual dos contratos, afinal as ferramentas nem sempre reconhecem todas as vulnerabilidades ou falhas presentes nos contratos.

Referências

BUTERIN, V. Ethereum whitepaper. 2014. Disponível em: <https://ethereum.org/en/whitepaper/#ethereum>. Citado 3 vezes nas páginas 10, 13 e 14.

CONSENSYS. *Mythril: Security Analisis tool*. 2022. <https://mythril-classic.readthedocs.io/en/master/about.html>. Citado na página 19.

ENTRIKEN, W. et al. Eip-721: Non-fungible token standard. January. 2018. Disponível em: <https://eips.ethereum.org/EIPS/eip-721>. Citado na página 15.

ETHERSCAN. 2022. Acessado em: 2022-10-10. Disponível em: <https://etherscan.io/>. Citado na página 25.

FEIST, J.; GRIECO, G.; GROCE, A. Slither: A static analysis framework for smart contracts. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. [s.n.], 2019. p. 8–15. Disponível em: <https://doi.org/10.48550/arXiv.1908.09878>. Citado na página 17.

GRIECO, G. et al. Echidna: Effective, usable, and fast fuzzing for smart contracts. July. 2020. Disponível em: <https://agroce.github.io/issta20.pdf>. Citado na página 18.

KUSHWAHA, S. S. et al. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, v. 10, p. 6605–6621, 2022. Disponível em: <https://ieeexplore.ieee.org/document/9667515>. Citado 2 vezes nas páginas 15 e 21.

LEID, A. Testing smart contracts. 2020. Disponível em: <http://hdl.handle.net/10019.1/108088>. Citado na página 21.

MUELLER, B. Smashing ethereum smart contracts for fun and real profit. 2018. Disponível em: <https://github.com/muellerberndt/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf>. Citado na página 18.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009. Disponível em: <https://bitcoin.org/bitcoin.pdf>. Citado na página 9.

PRATAP, Z. *Reentrancy Attacks and The DAO Hack*. 2022. <https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/>. Acessado em: 2022-10-25. Citado na página 9.

SLITHER. 2022. <https://github.com/crytic/slither>. Acessado em: 2022-10-25. Citado na página 19.

SOLIDITY-DOCS. 2020. <https://docs.soliditylang.org/en/v0.8.17/>. Acessado em: 2022-10-20. Citado na página 11.

SWCREGISTRY. 2022. <https://swcregistry.io/>. Acessado em: 2022-10-25. Citado 3 vezes nas páginas 6, 24 e 26.

SZABO, N. Smart contracts. 1994. Disponível em: <https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html>. Citado 2 vezes nas páginas 5 e 9.

SZABO, N. Formalizing and securing relationships on public networks. *First Monday*, v. 2, n. 9, Sep. 1997. Disponível em: <https://firstmonday.org/ojs/index.php/fm/article/view/548>. Citado na página 10.

THE Ethernaut. 2022. Acessado em: 2022-10-10. Disponível em: <https://ethernaut.openzeppelin.com/>. Citado 2 vezes nas páginas 6 e 32.

TOLKIEN, J. R. R. *Fellow Ship of the ring*. [S.l.]: George Allen and Unwin, 1954. Citado na página 4.

VOGELSTELLER, F.; BUTERIN, V. Eip-20: Token standard. November. 2015. Disponível em: <https://eips.ethereum.org/EIPS/eip-20>. Citado na página 14.

WOOD, G. Ethereum: A secure decentralised generalised transaction ledger berlin version. 2015. Disponível em: <https://ethereum.github.io/yellowpaper/paper.pdf>. Citado na página 10.