LEONARDO TEIXEIRA

SEGETH: UM AGREGADOR DE FERRAMENTAS PARA IDENTIFICAÇÃO DE VULNERABILIDADES EM CONTRATO INTELIGENTE

(*versão pré-defesa, compilada em 30 de maio de 2022*)

Trabalho apresentado como requisito parcial à conclusão do Curso de Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Luis C. E. de Bona.

CURITIBA PR

2018

RESUMO

Blockchain é uma tecnologia que permite o armazenamento de informações e a execução de código de forma descentralizada, ela vem sendo cada vez mais adotada no mercado financeiro e monetário. Um programa executado em uma Blockchain é chamado de contrato inteligente, por ser executado em uma Blockchain e não em um ambiente comum, os contratos inteligentes possuem diversas diferenças quando comparado com aplicações escritas para um computador pessoal, como possuir um tamanho máximo do código fonte significativamente menor além de poderem ser executadas por qualquer usuário, incluindo outros contratos. Com todas as novas soluções, também vem novos problemas, isso não foi diferente com o Blockchain e mais especificamente com contrato inteligente, que por funcionar em um ambiente descentralizado, ele apresenta um novo conjunto de possibilidades que pode por sua vez ser maliciosamente explorado para fora da intenção do seu respectivo programador. Vulnerabilidades, como é o caso da reentrada, podem causar grande danos financeiros, para ambos programadores e usuários. Por causa disso várias ferramentas para análise de contrato inteligente foram criadas, entretanto elas não são de fácil acesso para pessoas sem conhecimento técnico. Esse trabalho descreve o Segeth, uma ferramenta para realizar análise de contrato inteligente, tudo isso de forma rápida e prática, provendo uma interface Web para sua utilização de forma acessível entre usuário mais casuais. O Segeth serve como um tipo de agregador de diversas ferramentas unificando em uma resposta única, tirando proveito assim dos pontos fortes de cada uma de suas ferramentas. Além disso o Segeth pode facilmente ser atualizado para integralizar novas ferramentas, viabilizando então uma simples adaptação conforme o estado da arte progride. O Segeth foi capaz de melhorar a identificação de vulnerabilidades em 33% quando comparado com a melhor ferramenta que o compõem, em contrapartida houve um incremento de apenas 10.9% dos casos de falso positivo.

Palavras-chave: Vulnerabilidade, Ethereum, Contrato Inteligente, Análise Estática, Blockchain

ABSTRACT

Blockchain is a technology that allows the storage of information and the execution of code in a decentralized way, it has been increasingly adopted in the financial and monetary market. A program running on a Blockchain is called a smart contract, because it runs on a Blockchain and not in a common environment, smart contracts have several differences when compared to applications written for a personal computer, for example they have a size limit much greater, also they can be executed by any user, including other contracts. With new solutions, new problems also come, this was no different with Blockchain and more specifically with smart contracts, which by working in a decentralized environment, it presents a new set of possibilities that can, in turn, be maliciously exploited. Vulnerabilities, like reentrancy, can cause great financial damage, to both programmers and users. Because of this, several tools for smart contract analysis have been created, however, they are not easily accessible to people without technical knowledge. This paper describes Segeth, a tool to analyze smart contracts, all this in a fast and practical way, providing a Web interface for an accessible way to casual users to use the tool. Segeth works as an aggregator for many tools, unifying the answers to a simple one, taking advantage of the strong point of each one of the tools aggregated. Beyond that Segeth can easily be updated to integrate new tools, making then a viable way to adapt the tool to the new state of art as it progresses. Segeth was able to improve vulnerability identification by 33% when compared to the best tool that composes it, on the other hand there was an increase of only 10.9% of false-positive cases.

Keywords: Vulnerability, Ethereum, Smart Contract, Static Analysis, Blockchain

SUMÁRIO

1 INTRODUÇÃO 6
2 FUNDAMENTOS DE UM CONTRATO INTELIGENTE 8
2.1 MECANISMOS DE CONSENSO 9
2.1.1 Prova de trabalho (Proof of Work - PoW) 9
2.1.2 Prova de participação (Proof of Stack - PoS) 9
2.1.3 Outros 10
2.2 CONTRATOS INTELIGENTES 10
2.2.1 Ethereum Virtual Machine (EVM) 11
2.2.2 ERC-20 - Token Fungível 11
2.2.3 ERC-1967 - Proxy 12
2.2.4 Outros 12
3 VULNERABILIDADES EM CONTRATO INTELIGENTE 13
3.1 VULNERABILIDADES 13
3.1.1 Problema da reentrada 13
3.1.2 Contratos que dependem do tempo 13
3.1.3 Dependência da ordem das transações em contratos 14
3.1.4 Problema com exceções não tratadas 14
3.1.5 Execução sequencial de contratos inteligentes 15
3.1.6 Argumentos propositalmente não-válidos 15
3.1.7 Ataque de *overflow/underflow* em valores inteiros 15
3.1.8 Contrato guloso 15
3.1.9 Contrato suicida 16
3.2 FRAUDES 16
3.2.1 Rug pull 16
3.2.2 Honeypot 17
3.3 TIPOS DE FERRAMENTAS 18
3.3.1 Análise estática 18
3.3.2 Análise dinâmica 18
3.4 PRINCIPAIS FERRAMENTAS 19
3.4.1 Slither 19
3.4.2 Mythril 19
3.4.3 Oyente 20
3.4.4 Securify 20
3.4.5 SmartCheck 20

3.5 DESEMPENHO 20
3.6 POSSIBILIDADES 21
4 FERRAMENTA PROPOSTA 22
4.1 SEGETH: UM AGREGADOR PARA IDENTIFICAÇÃO DE VULNERABILIDADES 22
4.1.1 Um aprofundamento da ferramenta: Slither 22
4.1.2 Um aprofundamento da ferramenta: Mythril 23
4.1.3 Funcionamento do Segeth. 24
4.1.4 Resultados. 25
5 CONCLUSÃO 27
REFERÊNCIAS 28
APÊNDICE A – EXEMPLOS 32
A.1 SLITHER 32
A.2 MYTHRIL 32

1 INTRODUÇÃO

Uma Blockchain ou tecnologia de registro distribuído (*distributed ledger technologies* - DLT) é um sistema que permite o registro de informações de forma descentralizada, que pode ser utilizado de forma publica ou para um grupo privado (Di Pierro, 2017). Uma das suas principais características é a imutabilidade, ou seja, impossibilidade de apagar qualquer dado já escrito nela (Hofmann et al., 2017).

Atualmente um grande número de novas soluções em redes Blockchain tem surgido, impulsionadas tanto pelos interesses financeiros como tecnológico (Treleaven et al., 2017). Uma preocupação cada vez maior é o crescimento das vulnerabilidades dessas tecnologias (Guggenberger et al., 2021).

Contrato inteligente é um programa que é executado e guardado por uma Blockchain, por causa disso esse programa pode ser utilizado de forma descentralizada por seus usuários (Zheng et al., 2020). Um contrato inteligente é executado em uma máquina virtual, de forma semelhante a aplicações Java que rodam em uma máquina virtual Java (Java virtual machine - JVM). A máquina virtual Ethereum (Ethereum virtual machine - EVM) é uma das principais ferramentas utilizada para a execução de contrato inteligente (Hildenbrandt et al., 2018). A linguagem mais utilizada para desenvolvimento de contrato inteligente para uma EVM é chamada de Solidity. Apesar de existir outros ambientes (Ellul e Pace, 2018) (Yang e Lei, 2018) e linguagens (Sergey et al., 2018) para contrato inteligente, a EVM é significativamente mais utilizada. Por causa disso existe uma grande diversidade de contratos, além de padrões de design muito mais robustos e bem definidos que contratos não compatíveis a uma EVM (Bartoletti e Pompianu, 2017).

Uma vulnerabilidade em um contrato inteligente pode ser definido como a capacidade de interagir com o contrato gerando um comportamento não esperado pelo seu desenvolvedor (Destefanis et al., 2018a). Um contrato inteligente possui vulnerabilidades diferente das então exploradas em aplicações normais (Singh et al., 2020).

Além de vulnerabilidades também existem as fraudes, essas podem ser definidas como um contrato que é feito de tal forma a enganar os seus respectivos usuários para ganhos financeiros. Os principais ataques dessa categoria são conhecidos como *rug pull* (Xia et al., 2021), que é uma remoção abrupta de uma grande quantia monetária de um contrato, e *honeypot* (Torres et al., 2019), que é quando uma função de um contrato finge ter uma funcionalidade mas tem outra.

A principal diferença entre vulnerabilidade e fraude é o fato de que a vulnerabilidade é um erro por parte do desenvolvedor do contrato, abrindo assim a possibilidade de um uso de certas funcionalidades providas pelo contrato que não foram previstas no momento de desenvolvimento precedente à publicação do contrato, enquanto a fraude não é um erro do desenvolvedor, mas um contrato malicioso.

Tanto fraudes como vulnerabilidades podem causar um grande custo financeiro para as vítimas. Em relação às principais vulnerabilidades criadas por erro no desenvolvimento do contrato, se observa que algumas delas são semelhantes à vulnerabilidade em programas tradicionais, como é o caso do *overflow/underflow* em variáveis inteiras, ou até mesmo o problema de exceções que não são devidamente tratadas (Durieux et al., 2020). Porém existem outras vulnerabilidades que são específicas ao ambiente da Solidity, linguagem responsável por gerar o código binário que consequentemente será executado por uma EVM.

As soluções para o problema de analisar vulnerabilidade apresentada por algum contrato podem ser agrupadas em três categorias diferentes, análise estática, análise dinâmica e verificação formal. Em ambas análises estática e dinâmica é feito uma verificação do código fonte do contrato, com a diferença de que a dinâmica também leva em consideração o comportamento do contrato já publicado. A verificação formal expressa o contrato de forma matemática e tenta encontrar vulnerabilidades (Bhargavan et al., 2016).

Uma das primeiras ferramentas para análise estática foi a Oyente (Loi Luu, 2016), após isso foram criadas diversas outras ferramentas, como é o caso da Slither (Feist et al., 2019), Mythril (Mueller, 2018b) e diversas outras (Durieux et al., 2020).

Foi avaliado qual dessas ferramentas tem a melhor capacidade de identificar as vulnerabilidades discutidas, levando em consideração o tempo gasto nas suas respectivas execuções, ou seja, não pode ter uma discrepância muito grande no tempo entre elas para analisar o mesmo contrato. No trabalho (Durieux et al., 2020) posteriormente foi concluído quais delas foram mais bem avaliadas considerando os requisitos anteriores, além disso algumas delas são capazes de identificar certas vulnerabilidades que as outras não são.

Este trabalho propõe a ferramenta Segeth que serve como um agregador das melhores ferramentas, Slither e Mythril, considerando as métricas analisadas nesse trabalho. Isso foi feito obtendo o resultado da execução de cada uma delas e calculando o tempo gasto, após isso esses resultados são unificados em um único formato padronizado. Com isso é removido as repetições entre os resultados e é então exposto ao usuário final. Também, visando a acessibilidade da ferramenta, foi fornecido uma interface Web.

O restante deste trabalho é organizado da seguinte maneira. O Capítulo 2 apresenta os fundamento de um contrato inteligente. Já o Capítulo 3 fala sobre diversos tipos de vulnerabilidades, além de apresentar brevemente algumas ferramentas utilizadas para identificar vulnerabilidades. O Capítulo 4 propõe a ferramenta Segeth junto com sua motivação e os resultados obtidos. Finalmente o Capítulo 5 apresenta as conclusões e trabalhos futuros.

2 FUNDAMENTOS DE UM CONTRATO INTELIGENTE

A Blockchain é uma estrutura de dados que guarda os seus dados de forma descentralizada. Na Blockchain os dados são linearmente correlacionados entre si de forma criptograficamente segura, impossibilitando assim a adição de dados que não estejam em concordância com as informações já armazenadas, formando então uma corrente de dados. A ideia de blocos conectados entre si para garantir a imutabilidade de seus respectivos parentes já existia desde muito antes, com o conceito de árvores de Merkle idealizada por Ralph Merkle (Merkle, 1988).

A forma como o consenso na adição de novos blocos é o principal diferencial entre as diversas implementações de uma Blockchain. Os ataques conhecidos (Li et al., 2020) que os mecanismos de consenso tentam solucionar são:

- **Negação de serviço:** as formas mais usuais de se realizar um ataque de negação de serviço em uma rede Blockchain seria criando diversos nós gerando uma grande quantidade blocos consumindo assim grande quantidade dos recursos dispostos pela rede sobrecarregando-a.
- **Gasto duplo:** Esse por sua vez tem a intenção de manipular a rede de tal forma que certos dados são modificados para beneficiar o agente malicioso. Isso pode ser obtido por três maneiras em geral, sendo elas:
  - **Ataque dos 51%:** o agente malicioso controla a maioria da rede podendo assim remover blocos.
  - **Ataque de corrida:** duas transações que se contradizem são feitas ao mesmo tempo, uma para o agente não malicioso e outra para o próprio agente malicioso que fez essas duas transações, então antes de confirmar uma das transações, o agente malicioso obtém os benefícios da transação para com o agente não malicioso, esperando então que somente a transação que foi enviada a si mesmo seja confirmada e a outra seja negada pois elas se contradizem.
  - **Ataques de Finney:** esse ataque é restringido a Blockchains que tem funcionalidade monetária e se assemelha ao ataque de corrida porém se utiliza da capacidade de Blockchains que podem gerar novas moedas.

A seção 2.1 apresenta diferentes mecanismos para obter consenso em um sistema Blockchain. Na seção 2.2 é introduzido o conceito de contrato inteligente e alguns padrões de contratos.

2.1 MECANISMOS DE CONSENSO

Um mecanismo de consenso é utilizado pelo sistema Blockchain, ou qualquer outro sistema distribuído, para alcançar um acordo necessário sobre um único estado da rede entre processos distribuídos ou sistemas de múltiplos agentes, é uma ferramenta essencial para guardar dados em um sistema descentralizado (Wang et al., 2019).

2.1.1 Prova de trabalho (Proof of Work - PoW)

Para obter o consenso na rede, o PoW, termo esse que foi cunhado por Markus Jakobsson e Ari Juels (Jakobsson e Juels, 1999), depende de uma quantia não insignificante porém fazível de trabalho para que uma operação seja confirmada. Tudo isso deve ser regulado de acordo com as demandas da rede para que se possa manter uma boa interação entre os seus usuários. Inicialmente esse consenso foi utilizado para conter spams em caixas de e-mails, além de conter ataques de negação de serviço.

A primeira rede Blockchain a implementar o sistema de PoW como mecanismo de consenso foi o Bitcoin (Nakamoto, 2008). O algoritmo proposto procura por um valor que quando gerado o *hash* do bloco, tal *hash* comece com uma sequência de zeros, a função *hash* pode ser o SHA-256. O nome desse valor que se modifica até gerar o *hash* requisitado se chama *nouce*, e ele é um dos campos da estrutura do bloco no Bitcoin, sendo assim é capaz de modificar o *hash* gerado do bloco.

Tal implementação impede que o bloco seja modificado sem ter que refazer o trabalho de procurar o *nouce* ideal para gerar o *hash* que comece com uma certa sequência de zeros. Além disso, essa implementação permite que a dificuldade de encontrar um bloco que esteja de acordo com as demandas da rede possa mudar de forma dinâmica, controlando apenas o tamanho da sequência de zeros que precisam ser gerados na função *hash*.

Entretanto o design desse mecanismo também apresenta falhas, principalmente no quesito de escalabilidade, já que conforme mais transações são feitas maior é o interesse de burlar tais transações dado que o sistema está suportando mais agentes interessados no mesmo, sendo assim será necessário aumentar a dificuldade de inserir novos blocos, o que por sua vez entra em direta contradição com a demanda de escalabilidade do próprio sistema.

2.1.2 Prova de participação (Proof of Stack - PoS)

Esse consenso é exclusivamente designado a Blockchain que possuem um valor monetário associado às suas informações. Tal valor será então utilizado para definir quem tem o maior interesse no funcionamento adequado da rede. Dessa forma será designado a essas entidades a confirmação dos blocos que devem então ser acoplados na rede.

O PoS pode também ser utilizado para definir o poder de mineração entre os seus membros, fazendo com que aqueles que possuem mais moedas consigam receber melhores resultados ao minerar novos blocos. O ato de minerar novas moedas é comumente chamado de

stacking em sistemas PoS, pois diferente de sistemas PoW não é necessário realizar um trabalho (minerar) para confirmar novos blocos.

A validação é feita por entidades denominadas de Validadores. Para se tornar um validador é necessário cumprir uma série de requisitos, isto é feito para garantir que somente entidades comprometidas com o sistema possam se tornar um validador, além de também garantir que seja capaz de realizar as tarefas necessárias de um validador. Esses requisitos variam de acordo com cada sistema e normalmente poucos conseguem cumpri-los, por isso foram feitas *pools* para viabilizar a participação desse processo de validação para qualquer membro da rede.

O sistema escolherá aleatoriamente um Validador para validar um bloco, se este erroneamente fizer de forma a conflitar com outros Validadores isto poderá gerar um gasto duplo, entretanto isso pode ser mitigado penalizando Validadores (Xiao et al., 2020) ou modificando o design do sistema de tal forma a remover qualquer incentivo de criar tais conflitos (Saleh, 2020).

2.1.3 Outros

Existem também vários outros mecanismos de consenso como é o caso do prova de autoridade (Proof of Authority - PoA) (Angelis et al., 2018), prova de história (Proof of History - PoH) (Yakovenko, 2019) ou os grafos acíclicos dirigidos (Directed Acyclic Graphs - DAGs) (He et al., 2021). Entretanto, eles ainda estão em fase experimental e não possuindo o mesmo tempo de exposição do que o PoW e o PoS.

2.2 CONTRATOS INTELIGENTES

Apesar de Blockchain ser mais conhecidas por sua capacidade de transferência de valores monetários, tal operação é a mera execução de uma determinada função, sendo assim, é possível obter uma Blockchain que execute outros tipos de função, de tal forma que tais funções possam ser definidas pelos próprios usuários do sistema. Permitindo assim que usuários possam interagir não somente com os parâmetros de uma função pré-definida, que no caso é a função de transferir valores de uma conta para outra, mas criar a sua própria função que deve ser executada na Blockchain.

Um contrato inteligente pode ser visto como um conjunto de funções que compartilham um espaço de memória e que estão armazenados dentro da Blockchain, permitindo assim que interajam entre si e entre outros usuários da rede. As principais características de contrato inteligente é o fato de permitir a interação entre agentes de forma a não depender da confiança entre ambos (Zheng et al., 2020).

Para executar um contrato inteligente é necessário uma máquina virtual, assim como funciona aplicações Java. A principal máquina virtual para contrato inteligente é a máquina virtual Ethereum (Ethereum Virtual Machine - EVM). Apesar de existir várias linguagens para escrever um contrato inteligente (Sergey et al., 2018), a principal linguagem utilizada para escrever um contrato inteligente é a Solidity (Wöhrer e Zdun, 2018).

2.2.1 Ethereum Virtual Machine (EVM)

A execução de contrato inteligente é feita pela EVM, que funciona de forma semelhante a de aplicações Java. O custo da execução de uma função é chamada de taxa de gás, isso é computado dentro da EVM. Os outros componentes são semelhantes a de um computador tradicional como pode ser visto na Figura 2.1.

Existem certas restrições que devem ser respeitas por qualquer programa sendo rodado na EVM, são elas:

- O código de um contrato deve ter uma tamanho máximo de 24 KB.
- Após o contrato ter sido publicado na rede ele se torna imutável.
- Pode impossibilitar a execução de uma função se for muito custosa, pois em cada execução é configurado um limite para o gasto de gás, se esse limite for ultrapassado a função não é executada.

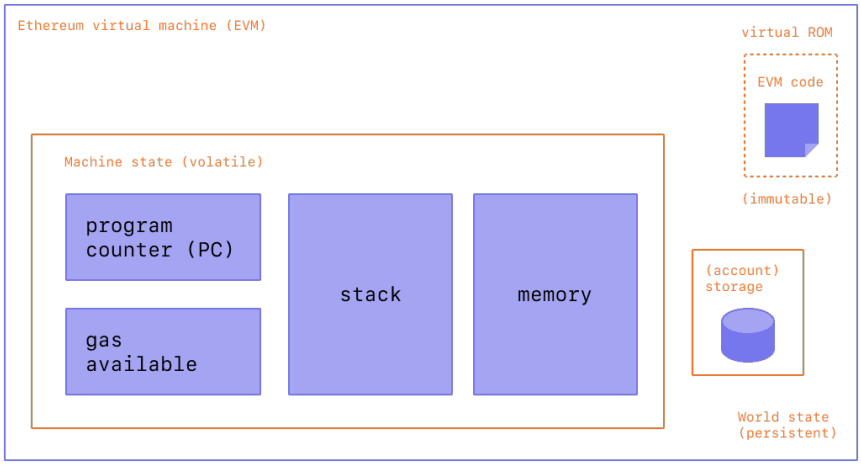

Figura 2.1: Arquitetura de uma EVM.

Essas restrições fazem com que exista uma diferença significativa no design de um programa que rodará em uma EVM em contraponto com um programa que rode em um ambiente como o Java Virtual Machine (JVM). Por causa desses e outros motivos, como a capacidade de interação entre contratos e manipulação de ativos digitais, é necessário definir uma série de padrões para manter a compatibilidade entre contratos.

2.2.2 ERC-20 - Token Fungível

Essa implementação foi requisitada em 2015 e é o padrão mais importante na rede Ethereum. Ele é responsável pela definição de um *token* fungível, ou seja, um *token* que tem a habilidade de ser trocado com outras unidades de valores equivalentes e não únicos.

As principais funcionalidades providas por esse padrão são a capacidade de transferir o *token* e de aprovar esse *token* para ser transferido por uma terceira parte.

2.2.3 ERC-1967 - Proxy

Um dos principais problemas da imutabilidade de um contrato está no fato de que após ele ser publicado na rede, se torna impossível atualizar para adicionar uma nova funcionalidade. Isso compromete a competitividade de um contrato que não pode se atualizar de acordo com as novas tecnologias de um mercado tão dinâmico. Uma possível solução para esse problema seria simplesmente publicar um outro contrato e então tentar redirecionar de forma manual os usuários para utilizar o novo contrato, essa solução não é tão problemática se esse processo poder ser escondido dentro do próprio *front-end* da aplicação. Contanto, para certos projetos isso não é o suficiente, dado que pode existir interação entre contratos, sendo possível que o endereço do contrato esteja escrito dentro de algum outro contrato, comprometendo assim a interação entre esses contratos, afinal não será possível apontar para o novo contrato.

Sendo assim foi criado uma forma de permitir que contratos tenham a possibilidade de serem atualizados, isto é feito a partir de um design por *proxy*, onde o contrato principal é um *proxy* que apontará para o contrato funcional, podendo este ser modificado por alguma autoridade definida no contrato *proxy*, pois para fazer isso basta atualizar os dados que apontam para o contrato funcional, apontando assim para um novo contrato que contenha as atualizações requisitadas.

Ainda não existe um consenso entre qual é o modelo mais adequado para o contrato *proxy*. Essa falta de consenso se dá por causa da forma de como funciona o comando *delegatecall* que é responsável por repassar a execução de uma função para um outro contrato, ou seja, é um comando essencial para o contrato *proxy*. Além também do mecanismo responsável por encontrar funções dentro de um contrato.

É importante observar que nem sempre é interessante aplicar este modelo, pois apesar da imutabilidade comprometer a adição de novas funcionalidades, ela garante que o contrato não irá mudar e isso é interessante para manter o princípio da interação entre agentes que não dependem de confiarem entre si, dado que o agente autorizado a atualizar o contrato pode fazer isso de forma maliciosa.

2.2.4 Outros

Existem diversos outros padrões já publicados, cada um com a sua respectiva finalidade, como é o caso do ERC-721 que apresenta um padrão para *tokens* infungíveis, ou o ERC-2535 que apresenta uma nova forma de fazer contratos *proxies* removendo também a limitação de 24 KB para o tamanho do contrato.

3 VULNERABILIDADES EM CONTRATO INTELIGENTE

Será discutido as principais vulnerabilidades que aparecem em contrato inteligente. Como é o caso da vulnerabilidade de reentrada que foi responsável por causar danos milionários em 2016 (Kolber, 2018). Também será considerado as técnicas utilizadas para implementar uma ferramenta de identificação de vulnerabilidade, apresentando brevemente algumas ferramentas mais populares. No Subcapítulo 3.1 será discutido as principais classes de vulnerabilidades. O Subcapítulo 3.2 apresentará os principais tipos de fraudes em contrato inteligente. Já no Subcapítulo 3.3 será mostrado brevemente algumas ferramentas para identificar vulnerabilidades. Por fim no Subcapítulo 3.4 será apresentado os resultados dessas ferramentas e o Subcapítulo 3.5 introduz de forma breve uma nova ferramenta.

3.1 VULNERABILIDADES

Uma vulnerabilidade em um contrato inteligente pode ser definida como a capacidade de interagir com o contrato gerando de tal forma um comportamento não esperado pelo seu desenvolvedor. Alguns termos importantes para entender os tipos de vulnerabilidades (Moubarak et al., 2018) são:

- *Fallback* em um contrato no Solidity é uma função que não possui um nome em sua assinatura, ou argumentos, nem retorna nenhum valor. Ela é responsável por executar um conjunto de operações quando uma quantia de Ether for enviado para o contrato.
- O método Call é usado para executar uma função de um contrato externo, ela não levanta nenhuma exceção se algum erro acontecer, porém retorna um valor booleano falso.

3.1.1 Problema da reentrada

A reentrada acontece quando um laço infinito é formado com a chamada de um contrato externo dentro uma função, permitindo assim que esse contrato externo chame novamente a mesma função que foi utilizada para chamá-lo, formando assim um laço infinito, que dependendo do que é feito antes da chamada externa pode causar um prejuízo significativo ao contrato, como foi o caso no DAO ataque (Moubarak et al., 2018).

3.1.2 Contratos que dependem do tempo

Cada bloco contém o momento no tempo que ele foi minerado. Esse valor é normalmente configurado pelo o tempo do sistema do computador do minerador. O momento no tempo de um bloco pode variar por aproximadamente 900 segundos em comparação com o bloco anterior.

Por causa disso existe uma certa flexibilidade na escolha do momento do tempo por parte do minerador, de tal forma na qual ele pode manipular esse valor de forma maliciosa para obter vantagens caso exista uma parte crítica do código do contrato que dependa desse valor, como por exemplo, a geração de um número aleatório para a escolha de um ganhador de um contrato de loteria (Nikolic et al., 2018), pois se a semente do gerador for feita a partir do tempo, essa pode então ser manipulada.

3.1.3 Dependência da ordem das transações em contratos

Por causa do fato de que em uma Blockchain existem várias transações a todo o momento e de forma descentralizada é impossível prever qual é o estado do contrato quando uma determinada função é executada. Por causa disso certas operações devem levar em consideração a volatilidade das informações confinadas no estado da qual elas estão executando.

Afinal a ordem delas não pode ser garantida (Nakamoto, 2008), mesmo que uma transação seja requisitada antes, isso não garante que ela será executada antes, dado que quem determina a ordem de uma transação são os mineradores dos blocos, que tomam suas respectivas escolhas por diversos fatores, como por exemplo a prioridade de uma transação determinada pela quantidade a mais que o usuário está disposto a pagar.

Essa vulnerabilidade é extremamente importante para contratos que lidam com valores de ações de mercado, pois quando um agente compra um determinada ação o seu preço subirá, logo com diversos agentes interagindo com tal contrato não será possível garantir que o preço de uma ação se manterá até a confirmação da transação, essa diferença é chamada de *slippage*. É possível ocorrer inanição na transação se o *slippage* for limitado com um valor muito pequeno.

3.1.4 Problema com exceções não tratadas

Normalmente um contrato requer dentro de sua implementação uma chamada para outro contrato externo. No contrato chamado pode ocorrer o levantamento de exceções, o que culminará no término da execução e na reversão do seu estado. Tais exceções podem ser causadas por diversos motivos, como por exemplo, a pilha de chamadas excedeu seu limite, não se tem gás o suficiente para concluir a operação, ou algum outro erro inesperado do sistema (Praitheeshan et al., 2020).

Um usuário malicioso pode executar um contrato e fazer com que a função responsável pela chamada de uma execução externa falhe propositalmente. Como a pilha de chamada tem uma profundidade máxima de 1024 unidades em uma EVM (Hildenbrandt et al., 2018), se esse valor for excedido então isso resultará em uma exceção, que por consequência irá parar a execução do programa e reverterá o estado do contrato. Sendo assim, se o agente malicioso conseguir com que o contrato externo seja executado 1023 vezes ele conseguirá interromper a execução do contrato principal.

Por causa disso o valor de retorno de uma chamada de algum contrato externo deve ser explicitamente testado independente da situação.

3.1.5 Execução sequencial de contratos inteligentes

Na rede Ethereum a execução de contratos inteligentes funciona de forma sequencial. Inicialmente um conjunto de contratos selecionados pelo seu respectivo bloco é ordenado pelo mecanismo de consenso. Então esses contratos são executados na mesma ordem por diversos outros nós da rede (Mueller, 2018b).

Esse método tem diversos problemas de desempenho, pois um usuário malicioso poderia publicar um contrato muito custoso, causando assim um gargalo para os contratos subsequentes.

E com o aumento de contratos na rede isso pode vir a ser um problema significativo para uma fluida execução. Isso poderia ser resolvido paralisando a execução de contrato, de tal forma que não permita que contratos custosos comprometam o tempo de execução da rede.

3.1.6 Argumentos propositalmente não-válidos

É importante observar que qualquer pessoa pode executar uma função em qualquer contrato, cabe ao contrato limitar a execução para as pessoas pertinentes. Sendo assim é de extrema importância nunca reter a responsabilidade dos parâmetros de uma chamada de alguma função em um contrato para o usuário. Os argumentos dados em uma função devem sempre ser checados e validados dentro da execução do contrato, para impedir assim que o contrato tenha um comportamento anormal.

3.1.7 Ataque de *overflow/underflow* em valores inteiros

O valor de um inteiro em Solidity tem o tamanho de 256 bits, sendo assim se uma variável de um inteiro chegar ao seu valor máximo $2^{256} - 1$, então será automaticamente redefinido para zero caso seja incrementado por um. O mesmo acontece de forma inversamente proporcional para o caso de valor mínimo equivalente a zero.

Um ator malicioso tentará usar essas variáveis incrementando ou decrementando para abusar da funcionalidade provida pelo contrato. Em 2018 o projeto Proof-of-Week-Hands sofreu esse ataque, causando um dano de 2000 Ethers (Luu et al., 2016).

3.1.8 Contrato guloso

Quando um contrato se utiliza de um contrato externo para prover funcionalidades para si, esse contrato está em risco de se tornar um contrato guloso, principalmente se as funções que lidam com Ethers tem relação com contratos externos. Pois se em alguma circunstância, seja intencional ou não, algum desses contratos externos ter sua existência terminada ou destruída por algum usuário, então isso resultará na impossibilidade de remover os fundos direcionados ao contrato principal. Dado que essas respectivas funções sempre levantaram uma exceção,

impossibilitando assim sobre certa circunstância, que tais fundos sejam removidos do contrato, supondo que tal funcionalidade não esteja já disponível no próprio contrato.

O ataque à carteira de múltiplas assinaturas do Parity explorou essa vulnerabilidade, tornando este o segundo maior ataque na rede Ethereum em termos de quantidade de Ethers roubados (Destefanis et al., 2018b). Neste caso, o atacante conseguiu ganhar o direito de posse de um contrato intermediário usado como biblioteca pelos principais contratos da carteira de múltiplas assinaturas. Com tal autoridade dentro dessa biblioteca ele conseguiu ter o direito de destruir esse contrato, congelando por consequência os valores guardados nas carteiras providas pela Parity. No total foram em torno de 151 carteiras congeladas totalizando 15.153.037 Ethers.

3.1.9 Contrato suicida

Existe uma operação primitiva chamada *selfdestruct* em Solidity responsável por destruir o contrato e enviar os seus fundos restantes ao endereço que executa a função que possui esse comando. Nem todos os contratos decidem adicionar essa função dentro da sua implementação, entretanto, alguns desenvolvedores acreditam que essa função tem como intuito de em uma eventualidade em que o contrato seja atacado essa função conteria os danos causados (Nikolic et al., 2018)

Porém quando uma operação como está exposta para qualquer usuário, ou sobre alguma circunstância um usuário consegue acesso ao *selfdestruct*, então nesse caso pode ser considerado uma vulnerabilidade, pois um agente malicioso pode tirar proveito dessa capacidade e remover todos os fundos armazenados dentro de um contrato, até mesmo utilizar essa vulnerabilidade para gerar vulnerabilidades em outros contratos, podendo então criar contratos gulosos, por exemplo.

3.2 FRAUDES

As fraudes podem ser definidas como contratos que são escritos com a intenção de tirar algum tipo de proveito de seus usuários. Os casos mais comuns são o *rug pull* e o *honeypot* (Torres et al., 2019).

3.2.1 Rug pull

Com a evolução do sistema financeiro proposto de forma descentralizada por diversas entidades dentro da rede Ethereum, foram formadas as Corretoras Descentralizadas (Decentralize exchanges - DEX), que em contra partida com as Corretoras Centralizadas (Centralize exchanges - CEX) não são compostas de uma entidade física (Schär, 2021). Nas DEXs a troca de valores monetários são conciliadas por meio de contratos inteligentes, não precisando assim de uma empresa física. Isso possibilitou um maior dinamismo na possibilidade de venda de novas moedas, pois ao criar uma nova moeda já é possível criar uma pool de liquidez dentro de alguma DEX, sem precisar assim ser aceito por uma CEX que tem seu procedimento de inserção de novas moedas muito mais rígido.

Tal rigidez se dá por dois motivos, o primeiro é a dificuldade e gastos de adicionar uma nova moeda no sistema, podendo não compensar o lucro obtido. O segundo motivo seria para proteger os seus respectivos usuários de moedas que podem ter como intuito enganá-los, danificando assim a experiência de uso dos seus usuários.

Para uma DEX conseguir adequar o preço dos seus ativos de acordo com a oferta e demanda do mercado é utilizado o Automated Market Makers (AMM) que é um algoritmo que define o preço de um ativo a partir da liquidez disposta em uma pool (Othman et al., 2013).

O AMM funciona a partir da seguinte expressão matemática $x * y = k$, onde **x** e **y** representam respectivamente a quantia das moedas que estão sendo trocadas na pool e **k** é um valor constante criado na inicialização do par **xy**. Agora suponha uma troca na qual um novo valor **z** do mesmo tipo que a moeda **y** esteja sendo trocada nessa pool, então para manter a corretude para a constante **k** será removido a seguinte quantia da moeda **x**, $a = x - k/(y + z)$, mantendo assim a corretude da expressão $(x - a) * (y + z) = k$, ou seja, por **z** moedas do tipo da **y** seria possível comprar **a** moedas do tipo da **x**.

Com isso diversas novas moedas conseguiram ser trocadas em DEX como a Uniswap por todo o mundo de forma simples e rápida, entretanto tal facilidade abriu oportunidades para novos golpes e fraudes. Sendo uma delas a *rug pull* que ocorre quando um agente com uma quantia significativa de uma moeda, podendo ser o seu próprio desenvolvedor, vende uma quantia grande o suficiente de moedas para remover grande parte da liquidez gerada na sua determinada pool, não só diminuindo significativamente o valor dessa moeda, mas também em alguns casos até mesmo impedindo outros usuários de vender (Xia et al., 2021).

Existe uma infinidade de técnicas usadas por golpistas para aplicar esse golpe, ademais a grande maioria delas se baseiam na tentativa de aumentar a credibilidade de sua moeda, seja distribuindo entre contas fictícias para dar a entender que é uma moeda usada por diversos usuários, ou manipulando o seus preço, entre diversas outras.

3.2.2 Honeypot

Existem dois tipos de *honeypot* que se definem de acordo com as suas vítimas. O primeiro tipo tem como alvo vítimas que não possuem um grande conhecimento em contratos inteligentes e simplesmente são enganadas para investir dinheiro em diversas promessas feitas pelo desenvolvedor do contrato, porém quando elas tentam remover o dinheiro delas do contrato, elas se veem incapacitadas, pois a função de venda está disponível somente para um seleto grupo de usuários.

Em alguns casos a própria proposta da moeda é justamente essa, como foi o caso do King of the Hill (Xia et al., 2021), que era um jogo na qual cada investimento deveria ser maior que o anterior, ao chegar no final do tempo limite o último investidor levaria o investimento dos outros usuários, contando para ser um *honeypot* é necessário ter uma vulnerabilidade que somente o ator malicioso conheça, para então no momento oportuno ele conseguir tirar proveito do contrato.

O segundo caso tem como vítimas usuários que tenham um bom conhecimento de contratos inteligentes, nesse caso o agente malicioso escreve um contrato com uma vulnerabilidade que se pareça óbvia, porém para que essa suposta vulnerabilidade possa ser aproveitada é necessário um gasto inicial. Após a vítima ter tomado todos esses passos então ao tentar tirar proveito da vulnerabilidade ela se encontra incapacitada, pois o contrato foi escrito de tal forma a prever tal atitude por parte da vítima, impossibilitando então a vítima de retirar o seu investimento inicial (Torres et al., 2019).

Tudo isso é possível pois quando um contrato é publicado na rede Ethereum, somente o código binário, que é o resultado da compilação, fica disponível para o público, existem ferramentas que tentam descompilar para código fonte, entretanto a legibilidade é significativamente afetada. Sendo assim, é utilizado certas plataformas para disponibilizar o código fonte do contrato, como é o caso da Etherscan para a rede Ethereum. Sabendo disso o ator malicioso pública o código que não dispõe de informações precisas o suficiente para que a vítima perceba a impossibilidade de abusar das supostas vulnerabilidades do contrato.

3.3 TIPOS DE FERRAMENTAS

A análise de vulnerabilidade tem como principal objetivo encontrar erros intencionais ou não em um contrato, isso pode ser eventualmente usado para corrigi-los antes de publicar o contrato, ou no caso do contrato já ter sido publicado, consertá-lo utilizando o padrão de *proxy* se suportado pelo contrato, ou evitá-lo caso seja um contrato com intuito fraudulento.

3.3.1 Análise estática

Na análise estática é feita uma verificação do código gerado de um contrato inteligente antes de ser rodado, ou seja, utilizando somente o código binário. Procurando assim por padrões de vulnerabilidade ou erros no código.

Certas ferramentas fazem a análise estática utilizando em conjunto o estado do contrato por meio de algum explorador de Blockchain, como é o caso do Oyente (Luu et al., 2016). Essa ferramenta em específico consegue identificar as seguintes vulnerabilidades: dependência do tempo, dependência da ordem das transações em contratos, vulnerabilidades relacionadas a reentrada e exceções que não são tratadas devidamente.

3.3.2 Análise dinâmica

Diferente da análise estática, a análise dinâmica é executada enquanto o contrato está rodando, ou seja, acaba agindo de forma semelhante a um agente malicioso que fica tentando constantemente encontrar vulnerabilidades no contrato que possam ser eventualmente exploradas. Enquanto alguns resultados utilizando a análise estática pode retornar um valor falso-negativo, esses em uma análise dinâmica podem ser devidamente encontrados.

Uma implementação da análise dinâmica em contratos inteligentes é a Maian (Nikolic et al., 2018), que tenta encontrar contratos gulosos, suicidas, e prodigal. Para a execução dinâmica do contrato é utilizado uma bifurcação privada da rede Ethereum. Além disso, nada impede a análise dinâmica de se utilizar parcialmente da análise estática para encontrar alguma vulnerabilidade e então confirmar tal vulnerabilidade utilizando a análise dinâmica, resultando assim em uma diminuição de falso-negativo.

3.4 PRINCIPAIS FERRAMENTAS

Constantemente são criadas novas ferramentas para análise de contrato inteligente, contanto será citado as que possuem os resultados com maior consistência e qualidade. Além disso é importante ressaltar que foram analisadas somente as ferramentas que possuem o seu código fonte publicada em um repositório publico. Outra restrição foi a capacidade de realizar a análise somente utilizando o código fonte do contrato inteligente, que no caso, estaria escrito em *Solidity*. Também tal ferramente deve ser capaz de identificar um conjunto de vulnerabilidades e não somente construir artefatos que auxiliariam na identificação dessas vulnerabilidades.

3.4.1 Slither

É uma ferramenta de análise estática (Feist et al., 2019) com uma rápida capacidade de detecção de um grande conjunto de vulnerabilidades. Exclusiva para contratos baseados em sistemas EVM, a Slither se utiliza de uma linguagem intermediária chamada SlithIR, que faz uso de Static Single Assignment (SSA) e um conjunto reduzido de instruções capazes de facilitar a implementação da análise, que é feita pelo Slither, sem remover valor semântico na sua execução.

O Slither também é capaz de encontrar oportunidades de otimização de contrato inteligente, diminuindo assim o custo de gás para a publicação do contrato e sua respectiva execução. Além de também fornecer um conjunto de dados que facilitam a compreensão de um contrato inteligente, auxiliando assim em um possível revisão de tal contrato.

3.4.2 Mythril

Essa ferramenta foi desenvolvida pela ConsenSys (Mueller, 2018b), ela utiliza de uma análise concreta e simbólica além de uma análise que se baseada na possibilidade de injeção de qualquer código malicioso no contrato, que caso tal padrão seja identificado então o contrato e dado como vulnerável, essa análise foi inspirada em vulnerabilidades de injeção de SQL. É também realizado uma checagem do fluxo de controle feito pelo código binário para a EVM, isso é feito com o intuito de diminuir o espaço de busca, facilitando e otimizando assim a identificação de vulnerabilidades.

3.4.3 Oyente

Essa foi uma das primeiras ferramentas para análise de contrato inteligente (Loi Luu, 2016). Para identificar de vulnerabilidades o Oyente utiliza execução simbólica no código binário gerado para o EVM.

3.4.4 Securify

Realiza uma análise estática no código binário produzido para inferir informação semântica importante e precisa sobre o contrato inteligente (Tsankov et al., 2018), é utilizado um resolvedor chamado Souffle Datalog. O Securify consegue até mesmo verificar se um determinado código é seguro ou não para uma respectiva propriedade dada.

3.4.5 SmartCheck

Inicialmente SmartCheck transforma o código de Solidity para uma representação intermediária baseada na formatação em XML, após isso é checado contra padrões do tipo XPath (Tikhomirov et al., 2018). Entretanto SmartCheck possui algumas limitações em contratos mais sofisticados, necessitando assim outros tipos de análises, como é o caso de uma análise manual do conteúdo disposto no contrato.

3.5 DESEMPENHO

As métricas de avaliação para determinar o desempenho foi a quantia de vulnerabilidades encontradas em um tempo razoável, ou seja, não pode haver uma discrepância muito grande entre o tempo de execução. As vulnerabilidades foram separadas entre quatro categorias, são elas:

Tabela 3.1: Vulnerabilidades identificadas por essas ferramentas dividido por categoria. Vulnerabilidades encontradas por total de vulnerabilidades, porcentagem de acerto.

Categorias	Slither	Mythril	Oyente	Securify	SmartCheck
Aritmética	0/22 0%	15/22 68%	12/22 55%	0/22 0%	1/22 5%
Negação de serviço	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%
Re-entrada	7/8 88%	5/8 62%	5/8 62%	5/8 62%	5/8 62%
Manipulação de tempo	2/5 40%	0/5 0%	0/5 0%	0/5 0%	1/5 20%

1. **Aritmética**: vulnerabilidades que estão relacionadas com *overflow* de inteiros.
2. **Negação de serviço**: é possível sobrecarregar o contrato com operações custosas temporalmente.
3. **Re-entrada**: existe alguma função que permite entrar novamente de forma recursiva na função inicial, fazendo o contrato se comportar de forma inesperada.

4. **Manipulação de tempo**: a manipulação do tempo, feita pelo minerador, afeta o comportamento do contrato.

3.6 POSSIBILIDADES

Com isso é possível observar na Tabela 3.1 que a melhor combinação entre as ferramentas será o par Slither e Mythril, considerando que isso incluiria todas as vulnerabilidades encontradas por todas as ferramentas utilizadas. Isso abre a possibilidade de combinar varias ferramentas para obter um resultado melhor do que a obtida individualmente por cada uma delas.

4 FERRAMENTA PROPOSTA

No trabalho será analisado meios e formas de solucionar os problemas aqui relatados. Atualmente existem quatro técnicas principais de identificar e solucionar vulnerabilidades (Praitheeshan et al., 2020) em contratos inteligentes escritos para uma EVM. Três delas serão discutidas, enquanto uma delas que seria auditoria manual do código de um contrato por uma certificadora confiável não será apresentado, entretanto grande parte do conhecimento necessário para realizar a auditoria do contrato já foi apresentado, afinal conhecer funcionalidades essenciais de um contrato em conjunto com as suas possíveis vulnerabilidades, já contêm grande parte do conhecimento requisitado para realizar uma auditoria de um contrato inteligente, e justamente por causa disso não será discutido.

4.1 SEGETH: UM AGREGADOR PARA IDENTIFICAÇÃO DE VULNERABILIDADES

Utilizando um conjunto de ferramentas que realizam análise estática em contrato inteligente, foi gerado o Segeth que junta o resultado de diversas ferramentas com o intuito de maximizar os ganhos de identificação de vulnerabilidades. Após os estudos feitos, foi chegado a conclusão de que a melhor combinação entre as ferramentas para possuir um maior ganho, mantendo entretanto um tempo razoável de execução, será a combinação entre a ferramenta Slither e a Mythril.

Nossa ferramente também disponibilizou uma interface gráfica via web para a utilização da mesma. Não só isso mas todos os componentes utilizados para a publicação dessa interface está sendo provida em *containers* via Docker, o que por sua vez facilita significativamente a sua utilização.

4.1.1 Um aprofundamento da ferramenta: Slither

A análise é feita utilizando um procedimento de múltiplo estágios, como pode ser visto na figura 4.1. Inicialmente é executado o compilador do Solidity no código fonte, que por sua vez gerará o Árvore sintática abstrata (Abstract Syntax Tree - AST), com isso o Slither conseguirá gerar diversos artefatos importantes para a sua execução.

Como é o caso do grafo de herança do contrato, na qual cada uma das arrestas são utilizadas para simbolizar a relação de herança de uma parte do código para com a outra. Também é feito o Grafo de fluxo de controle (Control Flow Graph - CFG) do código, e uma lista das expressões utilizadas no código do contrato.

Com esses artefatos o Slither traduz o código do contrato de Solidity para uma linguagem interna representativa chamada SlithIR, um exemplo é dado no Apêndice A.1. Para facilitar a computação da análise do código o SlithIR faz uso do SSA (Static Single Assignment), que nada

mais é do que um grafo que gera um novo nó para cada modificação de escrita em cada uma das variáveis do contrato, mantendo assim uma versão diferente da variável para cada período de sua respectiva mudança.

Por fim é feito a análise propriamente dita, utilizando todos os artefatos obtidos até então é executado um conjunto de regras, chamadas de *policies*, que recebendo como entrada os artefatos gerados determina a existência ou não de uma potencial vulnerabilidade. Cada uma dessas *policies* tem como objetivo identificar um determinado tipo de vulnerabilidade.

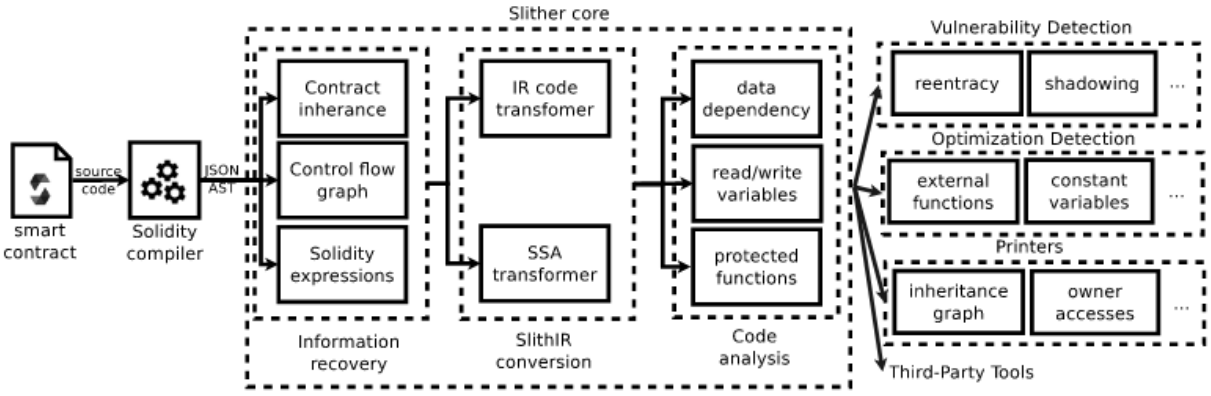

Figura 4.1: Visão geral do Slither.

Apesar de existirem um conjunto já proposto de *policies* para a identificação das vulnerabilidades, nada impede a criação de novas abordagens, que talvez possam ainda obter resultados ainda mais expressivos, como é o caso de alguns trabalhos feitos com tal intuito (Ma et al., 2020) (Bang et al., 2020) (Zhang et al., 2019).

A *policy* referente ao ataque de re-entrada verifica para cada uma das funções definidas no contrato se uma variável interna é modificada com a operação de escrita após uma chamada externa ao contrato. Tipificando assim um potencial ataque de re-entrada. No total existem mais de 20 *policies* para a identificação das mais diversas vulnerabilidades.

4.1.2 Um aprofundamento da ferramenta: Mythril

Semelhante ao Slither, o Mythril também faz uso do CFG gerado a partir do código fonte, ademais é realizado uma transformação, por meio de uma ferramenta chamada LASER (Mueller, 2018a), do código em uma serie de declarações formais, ou seja teoremas, que utilizam logica proposicional. Esses teoremas expressão o estado e os caminhos que um contrato pode possuir, um exemplo é fornecido no Apêndice A.2.

LASER é um interpretador simbólico para o código binário gerado para um EVM. Dado o código fonte de um ou mais contratos como entrada, retornará um conjunto abstrato de estados de programa. Tal *estado* consiste de um conjunto de valores das variáveis de uma máquina virtual possui em determinado tempo. Essas variáveis podem ser o Contador de programa (Program Counter - PC), o *stack* da máquina e o saldo das contas referente a um contrato ERC-20.

No total existem três conjuntos de variáveis, ilustrados na Figura 4.2, são eles:

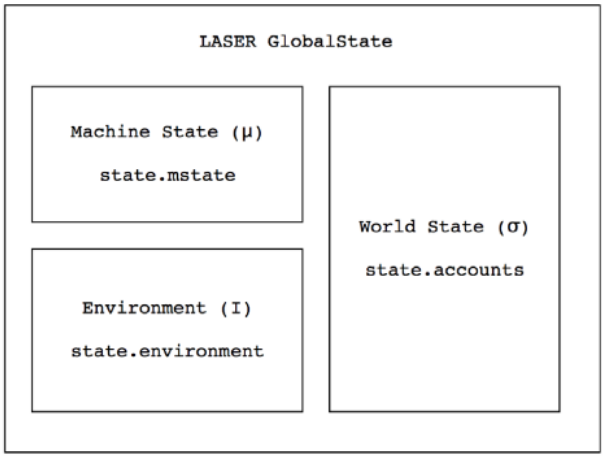

Figura 4.2: O estado global do LASER.

1. **Estado do mundo:** Um mapa dos endereços Ethereum para contas, que por sua vez inclui os saldos.

2. **Estado da maquina:** Possui o PC, memória e o *stack* da máquina virtual.

3. **Ambiente de execução:** As variáveis relevantes para a atual transação que está sendo executada, como é o caso do endereço da conta que executou a função, o valor da transação e assim por diante.

Com isso é utilizado um solucionador Z3 (Moura e Bjørner, 2008), esse por sua vez foi desenvolvido pela Microsoft com o intuito de solucionar problemas de verificação de software e análise de programa. A principal utilidade do Z3 para o Mythril é diminuir o espaço de busca e obter valores concretos que expressam potenciais vulnerabilidades.

A estratégica para identificar a vulnerabilidade de re-entrada é definido por dois passos:

1. Detecta todas as chamadas para endereços fornecido pelo usuário que também não limita o uso do gás.

2. Se uma chamada externa para um endereço não confiável é detectada, faz então uma análise no CFG para possíveis mudanças de estado que ocorrem depois do retorno da chamada. Gerando assim um alerta se uma mudança de estado é detectada.

4.1.3 Funcionamento do Segeth

Segeth (Teixeira, 2022) consegue funcionar somente utilizando o código fonte em Solidity, ou o endereço caso o contrato já tenha sido publicado na rede Ethereum. É também importante ressaltar que apesar de que cada contrato possa rodar em uma versão diferente do compilador da Solidity, o Segeth automaticamente já baixa e utiliza a versão pertinente ao contrato sem precisar se preocupar com essa limitação para cada contrato, automatizando assim o seu funcionamento.

Com isso é então executado cada um dos analisadores estáticos, utilizando o compilador adequado, e coletado os seus respectivos resultados, que por sua vez são então usados para a determinação de uma vulnerabilidade ou não. Dado a forma do problema, é sempre considerado o pior caso, visando assim minimizar os casos de verdadeiro-positivo na custa de aumentar potencialmente os casos de falso-positivo. Por fim é apresentado ao usuário final o resultado obtido por meio da interface gráfica. Uma ilustração é apresentada na Figura 4.3.

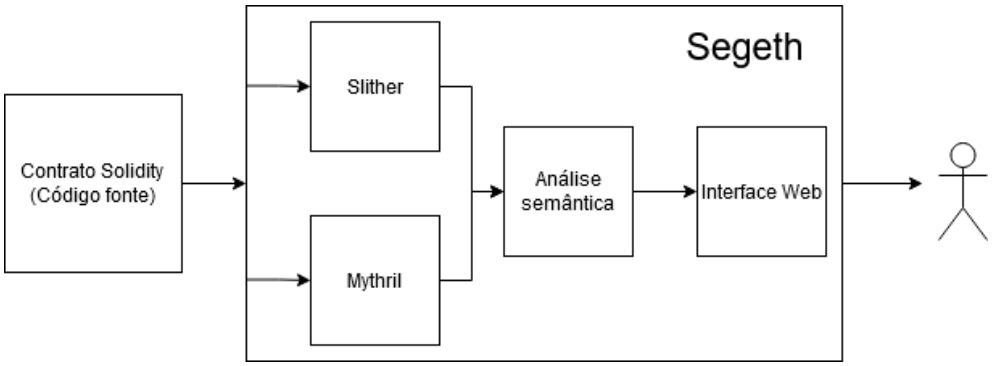

Figura 4.3: Diagrama do funcionamento do Segeth.

4.1.4 Resultados

Foi utilizado um banco de dados de contratos já rotulados, como vulnerável ou não, de 115 itens. O resultado obtido é apresentado na Tabela 4.1.

Tabela 4.1: Vulnerabilidades identificadas por cada ferramenta.

Ferramenta	Vul. identificadas	Total	Porcentagem
`Slither`	20	115	17.3%
`Mythril`	31	115	26.9%
`Segeth`	42	115	36.5%

Entretanto muitas dessas vulnerabilidades são identificadas por ambas as ferramentas, sendo assim para saber os ganhos de identificação de um agregador é necessário verificar as identificações distintas. A Figura 4.4 mostra proporcionalmente os casos identificados por ambas e individualmente.

Por fim foi analisa o tempo de execução médio pelas ferramentas e pelo Segeth.

Tabela 4.2: Tempo de execução médio.

Ferramenta	Tempo de execução	Tempo médio por contrato
`Slither`	11.85 minutos	6.18 segundos
`Mythril`	3.17 horas	99.23 segundos
`Segeth`	3.64 horas	114 segundos

O Segeth possui um pequeno sobrecarga de 16.35 minutos quando comparado com o tempo das ferramentas utilizadas, isso ocorre pois além de realizar a análise semântica dos

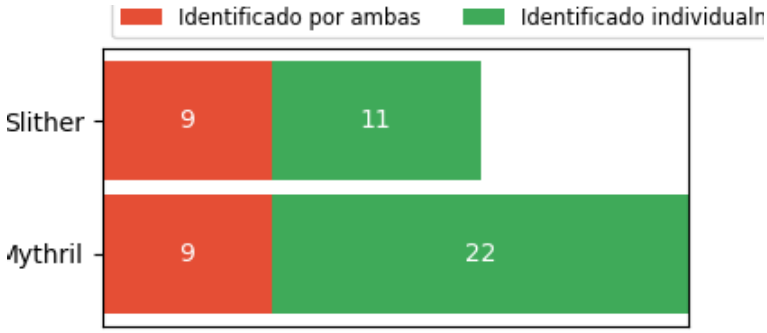

Figura 4.4: Identificação mutua e distinta.

resultados providos é também gasto tempo implementando o protocolo HTTP para conseguir prover os seus resultados para a sua interface Web.

Gerando o conjunto da união das vulnerabilidades identificáveis pelas ferramentas usadas, foi obtido um resultado total de **42/115 36%** de acerto. O que resultou em um incremento de **33%** da ferramenta que individualmente melhor se performou.

Houve um incremento de **10.9%** dos casos de falso positivo, que apesar de não ser maior que os ganhos obtidos, ainda pode ser considerado um problema. Entretanto o Segeth também pode ser utilizado para mitigar os casos de falso positivo gerados pelas ferramentas individualmente. Isso pode ser feito considerando uma vulnerabilidade somente se ambas as ferramentas, Slither e Mythril, considerarem como uma vulnerabilidade, resultando em uma diminuição dos casos de falso positivo as custas dos casos de verdadeiro positivo. A principal limitação do Segeth acaba sendo a qualidade das ferramentas que utiliza.

Segeth seria altamente impactante para usuários mais casuais, que não possuem muito conhecimento técnico na programação de contrato inteligente. Eles podem consultar o Segeth de forma prática, por meio da interface Web, se um contrato é seguro ou não, como já apresentado o Segeth não é capaz de garantir se um contrato é plenamente seguro ou não, entretanto ele consegue fazer uma análise muito mais profunda que um usuário casual. O Segeth também pode ser utilizado por desenvolvedores para auxiliar na construção de um contrato inteligente mais robusto e seguro.

As vulnerabilidades não identificadas e os casos de falso positivo são os maiores problemas que o Segeth enfrenta, apesar de que os casos de falso positivo não serem tão danosos para o usuário final, ainda assim é um problema. Já no caso das vulnerabilidades que não foram capazes de serem identificadas, é extremamente problemático, pois dará uma noção falsa de segurança ao usuário final. Sendo assim é importante utilizar essa ferramenta com certas precauções.

5 CONCLUSÃO

Apesar de diversos avanços na área de tecnologia de registro distribuído, ainda existe diversos problemas que não foram solucionados, como é o caso do mecanismo de consenso. Já em relação a contrato inteligente, foram estipulados alguns padrões a serem seguidos com o intuito de promover segurança e compatibilidade entre aplicações descentralizadas.

Contrato inteligente possui vulnerabilidades encontradas em aplicações tradicionais, como é o caso de ataques que abusam do *overflow/undeflow* de algumas variáveis chaves para o funcionamento do contrato. Contanto as principais vulnerabilidade são mais especificas. Como é o caso do problema de reentrada, que acontece quando um uma função é chamada recursivamente dentro de uma operação de transação de valores.

Para identificar essas vulnerabilidades existem diversas ferramentas, algumas delas foram apresentadas nesse trabalho. Além disso existe três técnicas principais para o desenvolvimento de uma dessas ferramentas, a análise estática, dinâmica e a verificação formal. Cada ferramenta tem uma abordagem diferente para o problema de identificação de vulnerabilidade.

O Segeth foi uma ferramenta apresentada nesse trabalho com o objetivo de servir como um agregador para diversas ferramentas. Além de simplificar o uso disponibilizando uma interface Web. Das ferramentas apresentadas nesse trabalho, foi escolhido duas para serem agregadas pelo Segeth, a Slither e a Mythril. Essas escolha foi feita a partir do critério da quantia de vulnerabilidade identificadas em um tempo minimamente razoável.

Os resultados obtidos pela Segeth foi um incremento na identificação de vulnerabilidades de 33% quando comparado com a Mythril que teve o melhor desempenho individualmente. Isso ocorreu pois a Slither é capaz de identificar certas vulnerabilidades que a Mythril não foi, melhorando assim a identificação do Segeth. Tudo isso ocorreu em um tempo muito semelhante a da execução da Mythril, que é a ferramenta mais demorada.

Futuramente seria interessante integrar analisadores dinâmicos, sendo capaz de integrar os resultados de forma a unificar a resposta de acordo com os analisadores mais pertinentes, para então realizar a identificação de vulnerabilidades de forma a monitorar constantemente contratos publicados, possuindo assim uma compreensão maior do comportamento do mesmo.

Segeth poderia ser melhorado criando uma nova capacidade de integrar de forma automática potenciais novas ferramentas, requisitando apenas que seja fornecido o comando necessário para a execução da ferramenta, após isso deveria ser fornecido de forma padronizada uma forma de obter certos resultados para a identificação de um conjunto de vulnerabilidades pré definidas. Por fim o Segeth automaticamente verificaria se a ferramenta fornecida incrementaria ou não a capacidade de identificação, preservando uma resposta rápida.

REFERÊNCIAS

Angelis, S. D., Aniello, L., Baldoni, R., Lombardi, F., Margheri, A. e Sassone, V. (2018). Pbft vs proof-of-authority: Applying the cap theorem to permissioned blockchain.

Bang, T., Nguyen, H. H., Nguyen, D., Trieu, T. e Quan, T. (2020). Verification of ethereum smart contracts: A model checking approach. *International Journal of Machine Learning and Computing*, 10(4).

Bartoletti, M. e Pompianu, L. (2017). An empirical analysis of smart contracts: platforms, applications, and design patterns. Em *International conference on financial cryptography and data security*, páginas 494–509. Springer.

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N. e Zanella-Béguelin, S. (2016). Formal verification of smart contracts: Short paper. Em *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, página 91–96, New York, NY, USA. Association for Computing Machinery.

Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A. e Hierons, R. (2018a). Smart contracts vulnerabilities: a call for blockchain software engineering? Em *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, páginas 19–25. IEEE.

Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A. e Hierons, R. (2018b). Smart contracts vulnerabilities: a call for blockchain software engineering? Em *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, páginas 19–25.

Di Pierro, M. (2017). What is the blockchain? *Computing in Science & Engineering*, 19(5):92–95.

Durieux, T., Ferreira, J. a. F., Abreu, R. e Cruz, P. (2020). Empirical review of automated analysis tools on 47,587 ethereum smart contracts. Em *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, página 530–541, New York, NY, USA. Association for Computing Machinery.

Ellul, J. e Pace, G. J. (2018). Alkylvm: A virtual machine for smart contract blockchain connected internet of things. Em *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, páginas 1–4. IEEE.

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

Guggenberger, T., Schlatt, V., Schmid, J. e Urbach, N. (2021). A structured overview of attacks on blockchain systems. Em *Proceedings of the Pacific Asia Conference on Information Systems (PACIS)*.

He, J., Wang, G., Zhang, G. e Zhang, J. (2021). Consensus mechanism design based on structured directed acyclic graphs. *Blockchain: Research and Applications*, 2(1):100011.

Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A. et al. (2018). Kevm: A complete formal semantics of the ethereum virtual machine. Em *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, páginas 204–217. IEEE.

Hofmann, F., Wurster, S., Ron, E. e Böhmecke-Schwafert, M. (2017). The immutability concept of blockchains and benefits of early standardization. Em *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*, páginas 1–8. IEEE.

Jakobsson, M. e Juels, A. (1999). *Proofs of Work and Bread Pudding Protocols(Extended Abstract)*, páginas 258–272. Springer US, Boston, MA.

Kolber, A. J. (2018). Not-so-smart blockchain contracts and artificial responsibility. *Stan. Tech. L. Rev.*, 21:198.

Li, X., Jiang, P., Chen, T., Luo, X. e Wen, Q. (2020). A survey on the security of blockchain systems. *Future Generation Computer Systems*, 107:841–853.

Loi Luu, Duc-Hiep Chu, H. O. (2016). Making smart contracts smarter. Em *In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, página 254–269, New York, NY, USA.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P. e Hobor, A. (2016). Making smart contracts smarter. Em *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, página 254–269, New York, NY, USA. Association for Computing Machinery.

Ma, R., Jian, Z., Chen, G., Ma, K. e Chen, Y. (2020). *ReJection: A AST-Based Reentrancy Vulnerability Detection Method*, páginas 58–71.

Merkle, R. C. (1988). A digital signature based on a conventional encryption function. Em Pomerance, C., editor, *Advances in Cryptology — CRYPTO '87*, páginas 369–378, Berlin, Heidelberg. Springer Berlin Heidelberg.

Moubarak, J., Filiol, E. e Chamoun, M. (2018). On blockchain security and relevant attacks. Em *2018 IEEE Middle East and North Africa Communications Conference (MENACOMM)*, páginas 1–6.

Moura, L. d. e Bjørner, N. (2008). Z3: An efficient smt solver. Em *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, páginas 337–340. Springer.

Mueller, B. (2018a). Laser-ethereum. `https://github.com/b-mueller/laser-ethereum`.

Mueller, B. (2018b). Smashing ethereum smart contracts for fun and real profit. Em *In 9th Annual HITB Security Conference (HITBSecConf)*, Amsterdam, Netherlands.

Nakamoto, S. (2008). Bitcoin whitepaper. *URL: https://bitcoin. org/bitcoin. pdf-(: 17.07. 2019)*.

Nikolic, I., Kolluri, A., Sergey, I., Saxena, P. e Hobor, A. (2018). Finding the greedy, prodigal, and suicidal contracts at scale.

Othman, A., Pennock, D. M., Reeves, D. M. e Sandholm, T. (2013). A practical liquidity-sensitive automated market maker. *ACM Transactions on Economics and Computation (TEAC)*, 1(3):1–25.

Praitheeshan, P., Pan, L., Yu, J., Liu, J. e Doss, R. (2020). Security analysis methods on ethereum smart contract vulnerabilities: A survey.

Saleh, F. (2020). Blockchain without Waste: Proof-of-Stake. *The Review of Financial Studies*, 34(3):1156–1190.

Schär, F. (2021). Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review*.

Sergey, I., Kumar, A. e Hobor, A. (2018). Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687*.

Singh, A., Parizi, R. M., Zhang, Q., Choo, K.-K. R. e Dehghantanha, A. (2020). Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security*, 88:101654.

Teixeira, L. A. (2022). Segeth: um agregador de ferramentas para identificação de vulnerabilidades em contrato inteligente. `https://github.com/lelaut/segeth`.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E. e Alexandrov, Y. (2018). Smartcheck: Static analysis of ethereum smart contracts. Em *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '18, página 9–16, New York, NY, USA. Association for Computing Machinery.

Torres, C. F., Steichen, M. e State, R. (2019). The art of the scam: Demystifying honeypots in ethereum smart contracts. Em *28th USENIX Security Symposium (USENIX Security 19)*, páginas 1591–1607, Santa Clara, CA. USENIX Association.

Treleaven, P., Gendal Brown, R. e Yang, D. (2017). Blockchain technology in finance. *Computer*, 50(9):14–17.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F. e Vechev, M. (2018). Securify: Practical security analysis of smart contracts. Em *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, página 67–82, New York, NY, USA. Association for Computing Machinery.

Wang, W., Hoang, D. T., Hu, P., Xiong, Z., Niyato, D., Wang, P., Wen, Y. e Kim, D. I. (2019). A survey on consensus mechanisms and mining strategy management in blockchain networks. *Ieee Access*, 7:22328–22370.

Wöhrer, M. e Zdun, U. (2018). Design patterns for smart contracts in the ethereum ecosystem. Em *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, páginas 1513–1520. IEEE.

Xia, P., wang, H., Gao, B., Su, W., Yu, Z., Luo, X., Zhang, C., Xiao, X. e Xu, G. (2021). Trade or trick? detecting and characterizing scam tokens on uniswap decentralized exchange.

Xiao, Y., Zhang, N., Lou, W. e Hou, Y. T. (2020). A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys Tutorials*, 22(2):1432–1465.

Yakovenko, A. (2019). Solana: A new architecture for a high performance blockchain.

Yang, Z. e Lei, H. (2018). Formal process virtual machine for smart contracts verification. *arXiv preprint arXiv:1805.00808*.

Zhang, W., Banescu, S., Pasos, L., Stewart, S. e Ganesh, V. (2019). Mpro: Combining static and symbolic analysis for scalable testing of smart contract. Em *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, páginas 456–462.

Zheng, Z., Xie, S., Dai, H.-N., Chen, W., Chen, X., Weng, J. e Imran, M. (2020). An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491.

APÊNDICE A – EXEMPLOS

A.1 SLITHER

A conversão feita pelo SlithIR do seguinte código:

```
using SafeMath for uint;
mapping(address -> uint) balances;

function transfer(address to, uint val) public {
   balances[msg.sender] = balances[msg.sender].min(val);
   balances[to] = balances[to].add(val);
}
```

Seria convertido para:

```
Function transfer(address uint256)
   REF_0(uint256) -> balances[msg.sender]
   REF_1(uint256) -> balances[msg.sender]
   TMP_1(uint256) = LIB_CALL SafeMath.sub(REF_1, val)
   REF_0 := TMP_1(uint256)

   REF_3(uint256) -> balances[to]
   REF_4(uint256) -> balances[to]
   TMP_3(uint256) = LIB_CALL. dest:SafeMath.add(REF_4, val)
   REF_3 := TMP_3(uint256)
```

A.2 MYTHRIL

Um exemplo da conversão de código em Solidity para uma representação de lógica proposicional seria, dado o código abaixo:

```
contract Assertions {
   function assertion2(uint256 input) {
      if (input > 256) {
         throw;
      }

      assert(input * 4 <= 1024);
   }
}
```

Seria representado pela seguinte lógica:

$$I_d[0:4] = 0xe166a663 \wedge I_v = 0 \wedge UINT256(I_d[4:36]) < 0x100 \wedge 4 * UINT256(I_d[4:36]) > 0x400$$

Aonde I_d é o vetor de bytes contendo a entrada dos dados e I_v é o valor da chamada.

Com isso é possível afirmar que utilizando as duas últimas expressões da formula, conseguimos afirmar que a seguinte expressão é sempre satisfeita:

$$(input < 0x100) \wedge \neg(input < 0x100)$$

Sendo assim a exceção na função **assertion2()** nunca será executada.