TÉCNICO LISBOA

Análise Estática de Smart Contracts

Nuno Manuel Olival Veloso

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientador: Prof. Pedro Miguel dos Santos Alves Madeira Adão

Júri

Presidente: Prof. António Manuel Ferreira Rito da Silva
Orientador: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Vogal: Prof. João Fernando Peixoto Ferreira

Janeiro 2021

Dedicado aos meus pais, família e à minha namorada.

Agradecimentos

Quero agradecer ao meu orientador Professor Pedro Adão pela proposta deste tema para esta dissertação e por todo o apoio e recursos dados ao longo do último ano. Quero agradecer também a disponibilidade do Professor João Ferreira em ajudar em qualquer problema que houvesse na integração da Conkas na framework SmartBugs.

Foi uma honra fazer parte da equipa de segurança do técnico, a STT, onde pude aprender vários temas e técnicas usadas na exploração de vulnerabilidades. Nada disto seria possível sem a ajuda dos membros da equipa nem do capitão da equipa, Professor Pedro Adão.

Agradecer também aos meus pais e irmãos por todo o apoio ao longo de todo o meu percurso académico assim como aos meus familiares e amigos.

Por último, agradecer à minha namorada Beatriz por todo o apoio, paciência e amor. Agradecer também aos seus pais e família pelo apoio.

Resumo

Com o avanço da tecnologia surgiu o Bitcoin e mais tarde o Ethereum. Com esta evolução, o conceito de moeda virtual tornou-se cada vez mais popular. Com o advento do Ethereum existe a possibilidade da criação de Smart Contracts que podem codificar quaisquer regras. Estes contratos podem armazenar milhões de euros em moedas virtuais. Uma vez que um contrato seja *deployed* é difícil de o corrigir, desta forma, se for descoberta uma vulnerabilidade num contrato *deployed*, um atacante vê uma grande oportunidade para atacar estes contratos de modo a extrair todas as suas moedas. Surge então a necessidade de poder realizar uma auditoria aos contratos para perceber se contêm alguma vulnerabilidade.

Neste relatório de dissertação é apresentado o estado da arte da investigação nesta área assim como é feita a introdução da Conkas, uma ferramenta de análise estática modular, que analisa o *bytecode* do contrato executando-o simbolicamente. É estendida a ferramenta Rattle que dará suporte à representação intermédia usada pela Conkas. É também realizada uma comparação da Conkas com 10 ferramentas consideradas estado da arte. Esta comparação mostra que a Conkas tem uma performance melhor que a ferramenta Mythril e tem uma taxa de verdadeiros positivos melhor do que todas as ferramentas usadas na comparação.

Palavras-chave: Ethereum Virtual Machine, Blockchain, Smart Contracts, Análise Estática, Vulnerabilidades, Execução Simbólica

Abstract

With the advance of technology, Bitcoin came up and later Ethereum. With this evolution, the concept of the virtual coin became even more popular. With the advent of Ethereum, there is the possibility to create Smart Contracts that can code any rules. These contracts can handle millions of euros in virtual coins. Once a contract is deployed it is hard to fix it, this way, if a vulnerability is found in a deployed contract, an attacker sees a great opportunity to exploit these contracts to steal all the coins. It comes up with the need to audit these contracts to understand if they have any vulnerability.

In this dissertation, is shown the state of the art of research in this area and is introduced Conkas, a modular tool of static analysis that analyses the contract's bytecode executing it symbolically. The Rattle tool is extended to give support to the intermediate representation of Conkas. A comparison is also made against 10 tools which are considered state of the art. This comparison shows us that Conkas has a better performance compared to Mythril and has a better true positive rate than all other tools used in the comparison.

Keywords: Ethereum Virtual Machine, Blockchain, Smart Contracts, Static Analysis, Vulnerabilities, Symbolic Execution

Conteúdo

Lista de Tabelas

Lista de Figuras

Listings

Lista de Algoritmos

Capítulo 1

Introdução

1.1 Motivação

Hoje em dia é constante a evolução tecnológica de tal modo que é possível obter soluções impensáveis há poucos anos atrás. Desde há alguns anos que existe a Blockchain, uma tecnologia inovadora. Mais tarde, em 2009, Satoshi Nakamoto introduziu o Bitcoin [1], uma moeda virtual / cripto moeda que não necessita de confiar em terceiros. Graças à tecnologia Blockchain, o Bitcoin ganhou bastante popularidade pois garante consenso por todos os participantes na rede. O consenso distribuído é garantido através de uma *Proof-of-Work* que também torna impraticável um *sybil-attack*[1] [2]. Com o passar dos anos percebeu-se que seria interessante criar algo que desse a possibilidade ao utilizador de programar o seu próprio negócio de forma distribuída sobre a camada Blockchain. Surgiu então o Ethereum, uma plataforma distribuída que tem uma criptomoeda nativa chamada Ether (ETH) mas que também é capaz de ser programada para criar novas aplicações. Estas aplicações são distribuídas (DApps) escritas em contratos (Smart Contracts) e uma vez que sejam "deployed" não podem ser atualizados. Assim é importante garantir que os contratos antes de serem transferidos para a rede Ethereum não contenham vulnerabilidades, e é nesse sentido que este relatório de tese tenta dar resposta, fazendo uma análise estática desses contratos.

Smart Contracts são programas que são executados por cada utilizador ou nó na rede (blockchain). Nick Szabo descreve um Smart Contract como uma máquina de venda direta em que partilha exatamente as mesmas propriedades de um Smart Contract na blockchain [3]. A máquina de venda direta tem *hard coded* as suas regras que define o que deve acontecer em certas condições e executa certas ações quando essas condições são satisfeitas. Por exemplo, se a Alice inserir uma moeda de 1 € e o produto que ela selecionou custar 1 € a ação deve ser só entregar esse produto à Alice sem dar nenhum troco. Numa outra situação em que a Alice insira mais dinheiro do que o valor do produto selecionado, a Alice deverá receber o seu troco. Assim um Smart Contract pode implementar uma série de aplicações como por exemplo, financeiras, seguradoras, etc. Ethereum foi a primeira plataforma

[1] Um *sybil-attack* é um ataque em que um participante da rede opera várias identidades simultaneamente, tendo como consequência a rede poder ter um número maior de identidades falsas do que identidades verdadeiras, o que permitiria ao atacante controlar a rede.

na blockchain a implementar uma máquina virtual *Turing-Complete* [4]. Estes contratos são escritos numa linguagem como a Solidity [5], que no momento é a mais utilizada, e são compilados para uma linguagem (Ethereum *bytecode*) que depois é executada na Ethereum Virtual Machine (EVM).

Os contratos inteligentes não estão livres de terem vulnerabilidades, e já foram explorados no passado levando a grandes perdas de cripto moedas. Em Ferreira et al. [6] foi criado um *dataset* com 69 contratos vulneráveis e outro *dataset* com todos os contratos existentes na blockchain Ethereum que tenham o código fonte Solidity disponível no Etherscan[2]. Os autores escolheram 9 ferramentas de análise estática consideradas estado da arte para os analisar. Do *dataset* com 69 contratos vulneráveis, todas as ferramentas em conjunto detetaram apenas 42% dos contratos vulneráveis. Em relação ao outro *dataset*, 97% dos contratos foram considerados vulneráveis pelas mesmas ferramentas, indicando um número alto de falsos positivos. Dificilmente os 97% dos contratos não serão vulneráveis pelo que é necessário arranjar ferramentas que apresentem um menor número de falsos positivos. Este estudo indica que existe espaço para melhorar a eficácia, taxa de verdadeiros positivos e o número de falsos positivos das ferramentas de análise estática e é neste sentido que esta tese se propõe a apresentar uma ferramenta de análise estática com uma melhor eficiência, taxa de verdadeiros positivos e um menor número de falsos positivos.

1.2 Visão Geral do Tópico

A estrutura de um Smart Contract pode ser observada na Figura 1.1. Um Smart Contract é identificado por um endereço (identificador de 160-bit) e o código correspondente está presente na blockchain. Os utilizadores para invocarem o código de um Smart Contract enviam transações para o endereço do contrato. Existem dois tipos de contas no Ethereum, *Externally Owned Accounts* (EOA) que são controladas por um par de chaves pública e privada e Contract Accounts (CA) que são controladas pelo código armazenado nessa conta. As contas podem ser "deployed" com código (CA) ou sem código (EOA). Para invocar um contrato (CA) é necessário haver uma transação feita por uma EOA em que é necessário indicar a função a ser chamada como os seus argumentos no parâmetro "data" da transação. Dependendo da função presente no Smart Contract, o estado deste pode alterar dependendo da lógica implementada na função. As funções presentes no Smart Contract podem invocar outros Smart Contracts através de transações ou despoletar eventos. Todos os participantes na rede de mineração executam as transações, pelo que o valor de retorno e o próximo estado da blockchain precisam de ser acordados entre todos através de um protocolo de consenso. O Ethereum suporta contratos com estado, ou seja, contratos em que os valores podem ser persistentes na blockchain permitindo serem usados em diferentes execuções.

Os Smart Contracts podem lidar com um grande número de moedas virtuais valendo centenas de euros cada uma, atraindo facilmente adversários que tentam transferir essas moedas virtuais para as suas carteiras. Existem dois tipos de rede em que as plataformas podem incluir-se sendo abertas (*permissionless*) ou fechadas (*permissioned*). As blockchains abertas são aquelas em que cada utilizador é

[2] `https://etherscan.io/`

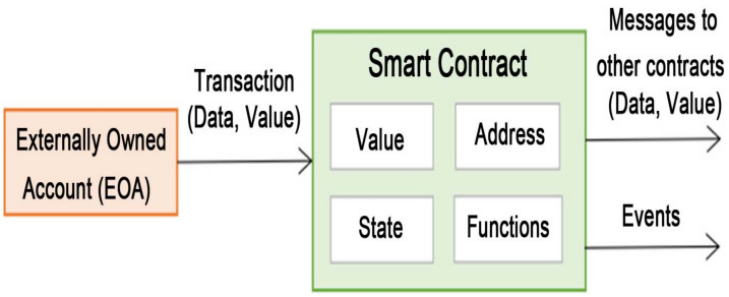


Figura 1.1: Visão global da estrutura de um Smart Contract [7]

livre de participar na rede, ou seja, são públicas. As blockchains fechadas são aquelas em que os utilizadores precisam de ter autorização para fazer parte da rede, ou seja, privadas. O Ethereum insere-se na rede aberta (*permissionless*), permitindo a adversários arbitrários tentarem manipular a sua execução, o que a torna potencialmente vulnerável. Contudo, os utilizadores têm de seguir protocolos predefinidos quando participam na rede, mas mesmo assim podem manipular a execução de Smart Contracts. Exemplos disso, são os participantes decidirem quais as transações que querem aceitar, a ordem pela qual são processadas as transações e poderem alterar o *timestamp* de um bloco. Assim é necessário que os programadores tenham em atenção estes exemplos, mas sobretudo a semântica da linguagem em que escrevem os contratos de forma a minimizar vulnerabilidades nesses contratos.

Uma limitação da análise do *bytecode* EVM é que a EVM quando foi desenhada não tinha como requisito ser fácil auditar o *bytecode*, sendo, portanto, um desafio maior realizar essa análise.

1.3 Objetivos

Apesar de as implementações de Smart Contracts sobre a blockchain serem relativamente recentes, como é o caso da Ethereum que foi lançada em 2015, já existem algumas ferramentas de análise estática como de análise dinâmica para verificarem a existência de vulnerabilidades nos Smart Contracts. O objetivo principal desta tese será estudar as ferramentas de análise estática de Smart Contracts consideradas estado da arte e construir uma nova ferramenta de análise estática de Smart Contracts que apresente melhores resultados em termos de eficácia, taxa de verdadeiros positivos e um menor número de falsos positivos comparados com os resultados apresentados pelas ferramentas estudadas, contribuindo para a comunidade de forma positiva. A ferramenta analisará apenas o *bytecode* sendo portanto agnóstica à linguagem de programação usada. No entanto, quando for necessário ter em conta alguma linguagem de programação esta será a Solidity (por exemplo, na criação de heurísticas para diminuir os falsos positivos e dar como opção ao utilizador da nova ferramenta a possibilidade de fornecer contratos escritos nesta linguagem).

De forma a atingir os objetivos definidos ir-se-á inicialmente estudar a plataforma Ethereum e explorar a tecnologia Blockchain. De seguida serão estudados os Smart Contracts e as suas especificidades tais como a linguagem *bytecode* que é o código que será executado na Ethereum Virtual Machine (EVM) e a sua semântica. Serão apresentadas as TOP 10 vulnerabilidades mais comuns segundo o

Decentralized Application Security Project (DASP). Estas vulnerabilidades estão suscetíveis de serem abusadas por diferentes tipos de utilizadores, incluindo utilizadores normais e *miners*. Serão também discutidas algumas ferramentas de análise estática consideradas estado da arte. O passo seguinte será a construção da nova ferramenta de análise estática de Smart Contracts. Primeiro será necessário estender a representação intermédia Rattle [8] de forma a poder ser usada na nova ferramenta. Depois será a implementação do motor de execução simbólica seguido de alguns módulos em que cada um será responsável por detetar um tipo de vulnerabilidade.

Existem dois requisitos para a construção da nova ferramenta que são por um lado ser fácil adicionar novos módulos para detetar outros tipos de vulnerabilidades e por outro ser igualmente fácil adicionar novas instruções ao motor de execução simbólica que possam surgir com as novas atualizações à EVM. Por fim, será realizada uma comparação com a maior parte das ferramentas estudadas - ver Secção 2.6 - usando um *dataset* igual, de forma a obter resultados comparáveis.

Os objetivos principais que se pretendem atingir com a ferramenta desenvolvida são os seguintes:

- Elevar o *bytecode* para uma representação mais alto nível;
- Ser compatível com as versões do compilador de Solidity até à data (v0.6.7);
- Detetar mais vulnerabilidades do que as ferramentas Oyente e Mythril que são as que usam o mesmo modelo da ferramenta que será construída e apresentam melhor taxa de verdadeiros positivos de acordo com o estudo realizado em [6];
- Ser fácil de adicionar novos módulos para detetar novas vulnerabilidades;
- Ser fácil de adicionar novas instruções ao motor de execução simbólica.

1.4 Estrutura da Tese

Este relatório de tese está organizado da seguinte forma:

- No Capítulo 2 é apresentada de forma sucinta a tecnologia blockchain, a plataforma Ethereum assim como os Smart Contracts e a Ethereum Virtual Machine. Será também apresentado o TOP 10 das vulnerabilidades mais comuns segundo o DASP10 e algumas das ferramentas de análise estática, apresentando as ferramentas já existentes que tentam resolver o mesmo problema, quais as estratégias que foram usadas e uma comparação com outras ferramentas;
- No Capítulo 3 são apresentadas, a ferramenta Rattle bem como a extensão realizada da representação intermédia da ferramenta Rattle e a ferramenta de análise estática construída ao longo do semestre chamada Conkas;
- No Capítulo 4 são apresentados os resultados obtidos assim como uma análise usando duas frameworks para o efeito;
- No Capítulo 5 são apresentadas as conclusões relativamente ao trabalho realizado, o trabalho futuro assim como outras contribuições.

Capítulo 2

Background

Neste capítulo será relatado o trabalho relacionado com o tema desta tese. Primeiramente o leitor terá a possibilidade de perceber como funciona a tecnologia Blockchain, Ethereum, Smart Contracts e a Ethereum Virtual Machine (EVM), como atuam e como se relacionam nas Subsecções 2.1, 2.2, 2.3 e 2.4. Na Subsecção 2.5 serão apresentadas as vulnerabilidades mais comuns. A Secção 2.6 será dedicada a apresentar e descrever as ferramentas já existentes, as estratégias usadas e a sua comparação com outras ferramentas.

2.1 Blockchain

A blockchain é replicada por todos os utilizadores. É organizada como uma cadeia de *hash* de blocos ordenados pelo tempo, onde cada bloco tem uma série de campos. Esses campos podem ser observados na Figura 2.1. Simplificando, pode ser vista como uma base de dados ou linked-list pertencente a uma rede em que cada bloco apenas tem um ponteiro para o bloco anterior. Uma vez um bloco adicionado não é possível atualizar nem remover esse bloco, sendo imutável[1]. Existem redes que são redes públicas (permissionless), ou seja, não requerem nenhuma permissão por parte de um novo participante para aceder a essa mesma rede. Em oposição, existem as redes privadas (permissioned) que requerem permissão por parte de novos participantes para se juntarem a essa rede.

Cada bloco precisa de ser minado por um *miner* e para o bloco ser aceite este precisa de satisfazer uma determinada restrição imposta pela rede, quer seja a rede Bitcoin, Ethereum ou outra. Em resumo, os *miners* usam hardware para executar algoritmos que por sua vez servem para verificar e adicionar blocos que contêm transações, à Blockchain. No Ethereum, este processo chama-se de Proof of Work (PoW) em que os miners têm de encontrar uma solução que satisfaça uma restrição. Esta restrição varia ao longo do tempo mas para exemplificar assume-se que o *hash* do bloco necessita de começar com 5 zeros. O miner irá concatenar um valor ao bloco e realizar o hash do bloco com o valor concatenado e verificar se começa com 5 zeros. Se for verdade então encontrou a solução para a restrição, caso contrário terá de concatenar outro valor até encontrar uma solução que satisfaça a restrição.

[1] Assumindo que um atacante não tenha mais de 50% de poder de processamento da rede.

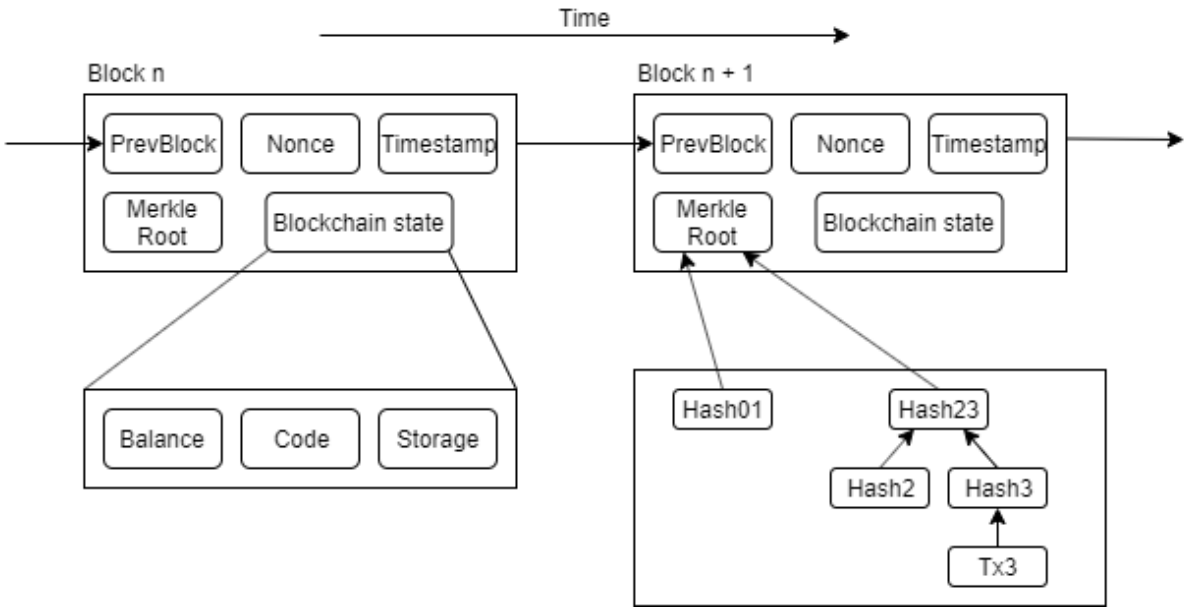


Figura 2.1: Estrutura da blockchain Ethereum (Adaptado de [7])

Em cada *epoch*, cada *miner* propõe um bloco para adicionar à blockchain e nesse bloco está presente a lista de transações que esse *miner* incluiu no bloco, bem como todos os outros elementos presentes na Figura 2.1. Para o *miner* conseguir adicionar o seu bloco à blockchain este necessita de realizar uma *Proof-of-Work*, no caso da rede Ethereum. Se o *miner* encontrar uma solução que satisfaça a *Proof-of-Work*, então este faz *broadcast* do bloco para todos os *miners*. Estes últimos irão verificar se a solução está correta e caso se confirme então o bloco é adicionado à blockchain de todos os *miners* e o *miner* que encontrou a solução é recompensado com Ether, considerando a rede Ethereum. Este protocolo chama-se *Nakamoto consensus*. Para mais detalhes o leitor pode ler o artigo do Bitcoin [1] ou do Ethereum [2].

2.2 Ethereum

O Ethereum foi criado por Vitalik Buterin em 2014 e formalmente escrito por Gavin Wood [4]. É uma plataforma descentralizada que assenta sobre a tecnologia Blockchain. Esta rede tem a sua própria cripto moeda chamada Ether (ETH). Outra característica desta rede é que é uma rede pública (permissionless). O que a torna diferente do Bitcoin é que suporta Smart Contracts e estes são executados sobre a Ethereum Virtual Machine (EVM).

O estado da blockchain Ethereum corresponde ao último bloco adicionado à Blockchain e é o mapeamento de endereços para contas, ou seja, um dicionário em que a chave é um endereço (de 160 bits) e o valor é uma conta. Sendo σ o último estado da blockchain e γ o endereço de uma conta, então $\sigma[\gamma]$ dá o último estado dessa conta no endereço γ.

O Ethereum suporta *Contract Accounts* (CA) que contêm moedas, código para ser executado e memória persistente. Também suporta *Externally Owned Accounts* (EOA) que só contêm moedas, não tendo código associado, e são controladas por um par de chaves pública e privada. Estas últimas são as

tradicionais *wallets* dos utilizadores. O foco desta tese irá incidir nas CA uma vez que são estas contas que contêm código que será executado pela EVM. Um utilizador para transferir moedas ou interagir com um contrato que esteja presente no Ethereum terá de inicializar uma transação.

2.3 Smart Contracts

Smart Contract - ou apenas contract (contrato) - é um agente autónomo armazenado na blockchain, em que a sua criação é feita através de uma transação em que o endereço do recetor/destinatário é vazio e o campo *data* é o código compilado (*bytecode*).

Uma vez criado com sucesso um contrato, este é identificado por um endereço de 160-bit. O código que representa o contrato pode manipular variáveis tal como acontece nas linguagens de programação tradicionais. Ao contrário das aplicações desenvolvidas fora do contexto da Blockchain, os Smart Contracts não podem sofrer atualizações, sendo de extrema importância não conterem bugs que possam ser explorados por terceiros. Por isso, é extremamente importante auditar o código para assegurar que não têm qualquer bug antes de serem *deployed*.

A linguagem de programação mais comum para escrever um Smart Contract é a Solidity. É uma linguagem orientada a objetos, influenciada pelas linguagens C++, Python e JavaScript e é compilada para ser executada pela EVM. A Solidity é uma linguagem fortemente tipificada, suporta herança, bibliotecas, tipos complexos definidos pelos utilizadores entre outras funcionalidades. A Listagem 1 é um exemplo de um contrato escrito usando a linguagem Solidity.

Na Listagem 1 é possível observar o uso de três tipos de memória, a *stack*, a *memory* e o *storage*. As variáveis declaradas dentro das funções do utilizador estão presentes na *stack*, um tipo de memória volátil. O segundo tipo de memória, a memória chamada *memory*, é para armazenar dados temporários, como por exemplo, os argumentos das funções e o valor de retorno de uma função. Este tipo de memória também é volátil. O outro tipo de memória é a memória persistente chamada de *storage*. Cada contrato tem o seu *storage* e um contrato não pode ler nem escrever noutro *storage*. Esta memória é declarada fora das funções do utilizador, mas dentro do contexto do contrato. Na Listagem 1, como exemplo, existe um contrato que contém duas variáveis na memória *storage*, linhas 2 e 3. Nas linhas 6 e 7 são declaradas duas variáveis voláteis, presentes na *stack*. Os argumentos *a* e *b* na linha 5 e o valor de retorno na linha 8 estão presentes na *memory*.

```
1  contract Example {
2      mapping (address => uint) userBalances;
3      address owner;
4
5      function add(uint a, uint b) public returns(uint) {
6          uint x = a;
7          uint y = b;
8          return x + y;
9      }
10 }
```

Listagem 1: Exemplo de memória persistente e memória volátil num contrato

2.4 Ethereum Virtual Machine

Dado que a EVM executa código presente nos Smart Contracts através de uma linguagem *Turing-Complete*, existe o problema da não terminação. Tendo em conta este problema, um possível cenário de forma a causar um ataque de *Denial-of-Service* (DoS), seria um utilizador criar um contrato que não termina, o que levaria a todos os *miners* ficarem indefinidamente a executar o código desse contrato. Para mitigar este problema o criador do Ethereum introduziu o conceito de *gas*. A execução de um contrato está sob um ambiente restrito podendo aceder e modificar o seu próprio estado e despoletar a execução de outro contrato. Outra característica é que a execução de um contrato é completamente determinística e produz o mesmo resultado partindo do mesmo estado.

Cada instrução que a EVM suporta tem um preço pré-definido, ou seja, qual o valor de *gas* necessário para executar essa instrução. Quando um utilizador envia uma transação para invocar um contrato, este necessita de especificar a quantidade de *gas* que está disposto a fornecer, chamado de *gasLimit*. Também necessita fornecer qual o preço pela unidade de *gas*, chamado *gasPrice*. Assim quando um *miner* introduzir a transação no bloco proposto e este for aceite, o *miner* irá receber uma taxa correspondente à quantidade de *gas* que foi usado para a execução, multiplicado pelo preço da unidade de *gas* (*gasPrice*). Se a execução ficar sem *gas* e necessitar mais do que o *gasLimit* então a execução é terminada com uma exceção e o estado é revertido para o estado inicial, ficando como se nunca tivesse sido executado. Neste caso, o utilizador que invocou o contrato não recebe o *gas* (Ether) de volta, e este é dado ao *miner* que publicar a transação. Caso o utilizador especifique o valor do *gas* mais alto do que o necessário para executar as instruções, então o excesso de *gas* é devolvido ao utilizador. Para saber quanto *gas* é necessário para cada instrução, o leitor pode ver em [4].

A Ethereum Virtual Machine (EVM) opera em pseudo-registos de 256-bit, o que significa que não opera sobre registos, mas sobre uma *stack* expansível que é usada para passar parâmetros a funções e a outros *opcodes*. A *stack* da EVM usa valores de 256-bit e tem um máximo de 1024 elementos sendo esta memória volátil. A *memory* é um *byte-array* e pode ser acedida ou escrita a partir de qualquer índice e com tamanho arbitrário. Esta é inicializada com tamanho zero e só pode ser estendida. O *storage* é um dicionário de palavras de 256-bit para palavras de 256-bit. Existem mais de 100 *opcodes* estando divididos em categorias como aritmética, booleana, cripto, contexto do contrato, contexto da blockchain, storage e execução, logging, entre outras. Na Figura 2.2 é possível observar um exemplo de uma execução na EVM para o seguinte *bytecode*: 6001600201. Este *bytecode* começa por inserir o valor 1 na *stack* seguido do valor 2 e por fim realiza a soma desses dois valores retirando-os da *stack* e coloca o resultado da soma na mesma.

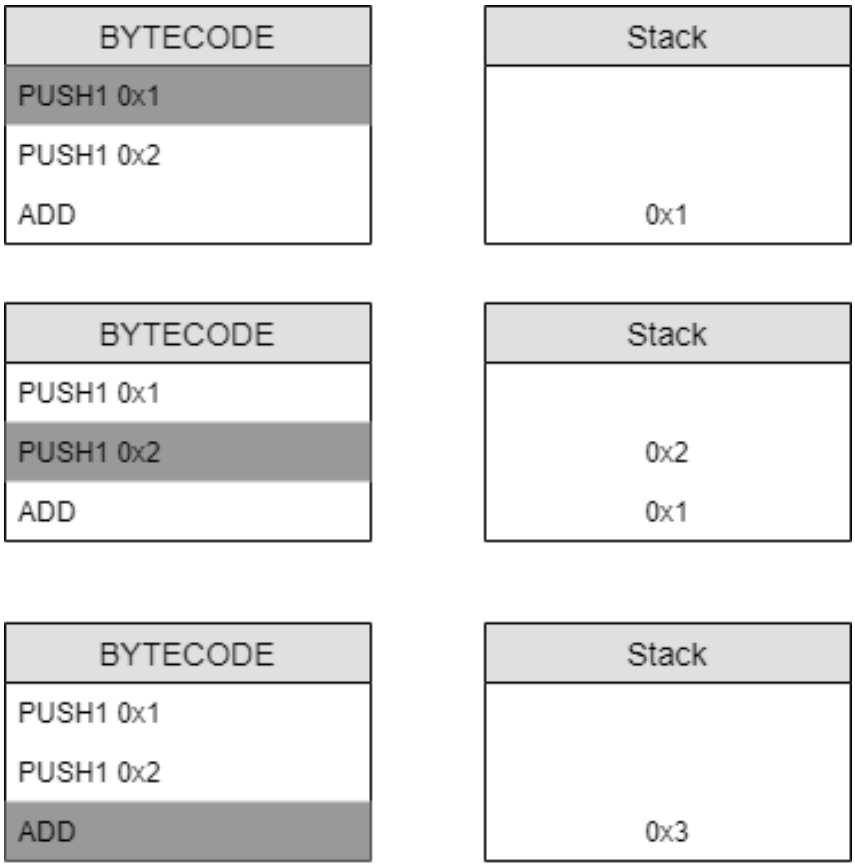


Figura 2.2: Exemplo de uma execução na EVM. Primeiro insere o valor 1 na *stack* e posteriormente o valor 2. Soma estes dois valores retirando-os da *stack*. Por último coloca o resultado da soma na mesma

2.5 Vulnerabilidades Conhecidas

Nesta Secção serão apresentadas as vulnerabilidades conhecidas segundo a classificação do DASP Top 10 [9] bem como alguns exemplos de possíveis ataques para as explorar. Estas vulnerabilidades estão ativas, quer isto dizer que não existe nenhuma mitigação automática pelo que o programador tem de as ter em atenção ao escrever um Smart Contract. Serão também apresentadas possíveis formas de mitigar cada uma das vulnerabilidades apresentadas.

2.5.1 Vulnerabilidade de Reentrância

Esta vulnerabilidade é uma das mais conhecidas por causa do TheDAO attack [10]. No Ethereum, quando um contrato invoca outro, a execução do primeiro contrato espera que a invocação do segundo contrato termine para prosseguir. Desta forma, um atacante pode tirar proveito deste estado intermédio.

Na Listagem 2 existem 2 contratos, o contrato “Example” e o contrato “Exploit”. O contrato “Example” tem uma variável que mantém informação sobre o balanço de cada utilizador (endereço). A função *withdrawBalance()* permite levantar dinheiro correspondente ao utilizador que a invoca. No final da função, se a transferência de Ether for completada com sucesso, a variável que guarda o balanço de cada utilizador é colocada a 0 indicando que esse utilizador já não tem dinheiro para levantar. No

contrato “Exploit” existe uma função de *fallback*[2] que verifica se o balanço do contrato “Example” é maior ou igual ao valor de Ether que está a ser enviado para o contrato “Exploit” e no caso desta condição ser verdadeira irá invocar a função *withdrawBalance()* presente no contrato “Example”.

No contrato “Example” presente na Listagem 2, a função *withdrawBalance()* que permite levantar dinheiro, contém uma vulnerabilidade de reentrância. Considere um utilizador malicioso usar o contrato “Exploit”, presente também na Listagem 2, em que a função de *fallback* desse contrato invoca a função *withdrawBalance()* (linha 20). Quando a execução é transferida para a função presente na linha 4, na linha 5 será enviado Ether para o contrato “Exploit”, que corresponde à variável *msg.sender*, com o valor armazenado em *userBalances[msg.sender]*. Isto irá invocar a função de fallback (linha 18) do contrato chamador. Este pode voltar a invocar a função *withdrawBalance()* que irá voltar a invocar a função de fallback (linha 18) do contrato “Exploit”. Isto é possível porque a função *withdrawBalance()* não é capaz de detetar se está na presença de uma chamada reentrante, ou seja, se está a ser chamada pela segunda vez sem a primeira vez ter terminado. O utilizador malicioso irá levantar todo o Ether do contrato “Example” porque a variável que armazena o Ether, *userBalances[msg.sender]*, tem sempre o mesmo valor uma vez que a linha 8 ainda não foi executada.

```
1  contract Example {
2      mapping (address => uint) userBalances;
3
4      function withdrawBalance() public {
5          require(msg.sender.call.value(userBalances[msg.sender])());
6          userBalances[msg.sender] = 0;
7      }
8  }
9
10 contract Exploit {
11
12     Example public ex;
13
14     (...)
15
16     function () payable external { // Fallback function
17         if (address(ex).balance >= msg.value) { // Withdraw all the Ether in Example
               contract
18             ex.withdrawBalance();
19         }
20     }
21 }
```

Listagem 2: Exemplo de um contrato com uma vulnerabilidade do tipo reentrância

Uma das primeiras mitigações para este tipo de vulnerabilidade era evitar o uso da função *call* e passar a usar ou a função *send* ou a função *transfer*. Isto porque estas funções limitam o *gas*, usando 2300 como valor predefinido. Com este valor definido, qualquer chamada a outro contrato iria resultar

[2] A função de *fallback* é executada quando o identificador da função a ser chamada não existe ou quando só é fornecido Ether sem fornecer mais dados. Esta função não tem nome, não pode ter argumentos nem retornar nada.

numa exceção de *RunOutOfGas*, uma vez que não havia *gas* suficiente para realizar a invocação a outro contrato, revertendo a transação. No entanto o valor do *gas* associado a cada instrução pode variar e por isso o valor 2300 usado como valor predefinido pode ser suficiente para um contrato invocar outro sem haver o lançamento de exceção, havendo possibilidade de haver chamadas reentrantes [11]. Outras mitigações passam por usar o padrão *Checks-Effects-Interactions*, em que todas as chamadas externas devem ser feitas no final da função, ou usar uma guarda, equivalente aos *mutex* usados em programas *multi-threading*.

Na Listagem 3 é possível observar o contrato "Example" presente na Listagem 2 usando o padrão *Checks-Effects-Interactions* que previne que o dinheiro do contrato seja todo levantado por um utilizador malicioso.

```
1  contract Example {
2      mapping (address => uint) userBalances;
3
4      function withdrawBalance() public {
5          uint amount = userBalances[msg.sender];
6          userBalances[msg.sender] = 0;
7          require(msg.sender.call.value(amount)());
8      }
9  }
```

Listagem 3: Exemplo do padrão *Checks-Effects-Interactions*

Na Listagem 4 é possível observar outra possível mitigação onde é realizada uma verificação para detetar uma chamada reentrante e cancelá-la. É necessário haver alguma cautela para certificar que todos os caminhos possíveis da função terminam com a variável *locked* igual a *false* para evitar situações de *deadlock*.

```
1  contract Guarded {
2      ...
3
4      bool locked = false;
5
6      function withdraw() external {
7          require(!locked, "Reentrancy not allowed!");
8          locked = true;
9          ...
10         locked = false;
11     }
12 }
```

Listagem 4: Exemplo de usar uma guarda

2.5.2 Vulnerabilidade de Controlo de Acesso

Vulnerabilidades de controlo de acesso são transversais à maioria das plataformas ou tecnologias usadas. Estas vulnerabilidades ocorrem quando um utilizador tem a liberdade de tomar uma ação que não

era suposto tomar. Outro problema é a visibilidade de uma função ou variável. Uma variável ter visibilidade privada não significa, no contexto da Blockchain, que ninguém irá saber o valor dessa variável. Isto porque a Blockchain é pública e em algum momento houve uma transação que atribuiu um valor a essa variável e essa transação também é pública. Esta vulnerabilidade pode ocorrer também quando é usada a variável *tx.origin* para validar um utilizador, ou exista uma lógica grande numa cláusula *require*, ou não faz o uso correto da funcionalidade *delegatecall*. A variável *tx.origin* indica o endereço do contrato chamador inicial, quer isto dizer que se houver três contratos C_1, C_2 e C_3 e C_1 invocar C_2 e C_2 invocar C_3, se C_3 usar a variável *tx.origin* esta terá o valor do endereço do contrato C_1 ao invés da variável *msg.sender* que terá o valor do endereço do contrato C_2.

Na Listagem 5 é possível observar uma função que tem uma vulnerabilidade de controlo de acesso. Esta função atribui a variável *owner* ao argumento passado à função. A vulnerabilidade reside em qualquer utilizador poder invocar esta função e o esperado seria apenas o dono do contrato poder alterar para um novo dono.

```
1 function setNewOwner(address new_owner) public {
2   owner = new_owner;
3 }
```

Listagem 5: Função com vulnerabilidade de controlo de acesso

Uma forma de mitigar esta vulnerabilidade é adicionar modificadores[3] às funções. Estes modificadores servem para adicionar políticas de controlo de acesso às funções que usam esses modificadores. Na Listagem 6 é apresentado um exemplo de um modificador que restringe o acesso apenas ao dono do contrato (linhas 1 - 4). O *underscore* serve para indicar onde deverá ser colocado o código da função que estiver anotada com este modificador. Para eliminar a vulnerabilidade presente na Listagem 5, o modificador deve ser adicionado à função *setNewOwner()*, como mostra a Listagem 6.

```
1 modifier onlyOwner {
2     require(owner == msg.sender);
3     _;
4 }
5
6 function setNewOwner(address new_owner) public onlyOwner {
7     owner = new_owner;
8 }
```

Listagem 6: Exemplo de um modificador

2.5.3 Vulnerabilidade de Aritmética

A vulnerabilidade de *integer overflow/underflow* acontece quando uma variável não consegue armazenar um determinado valor e armazena outro. Se for considerada uma variável que só armazena valores

[3]Modificador é um pedaço de código que é executado antes da função que usa esse modificador ser chamada e pode ou não ter código para ser executado depois da função que usa o modificador.

de 0 a 255 e se esta tiver o valor 255 e for adicionado o valor 1, ocorre um *overflow* e o valor passa a ser 0. O *underflow* é idêntico mas acontece quando se subtrai algo.

Na Listagem 7 é apresentada uma função que contém uma vulnerabilidade de *integer underflow*, pois se o parâmetro *_amount* for maior do que a variável *balances[msg.sender]* irá ocorrer um *underflow*. Os programadores devem ter isto em atenção ao escreverem os seus Smart Contracts e adicionar verificações para eliminar este tipo de vulnerabilidades. Na Listagem 8 a vulnerabilidade foi corrigida adicionando a verificação de que o valor da variável *_amount* tem de ser menor ou igual à variável *balances[msg.sender]*.

```
1 function withdraw(uint amount) {
2     userBalance[msg.sender] -= amount;
3 }
```

Listagem 7: Vulnerabilidade de *integer underflow*

```
1 function withdraw(uint amount) {
2     require(amount <= userBalance[msg.sender]);
3     userBalance[msg.sender] -= amount;
4 }
```

Listagem 8: Corrigida a vulnerabilidade de *integer underflow*

2.5.4 Vulnerabilidade de Não Verificação do Valor de Retorno de Chamadas de Baixo Nível

No Ethereum existem várias maneiras para um contrato invocar outro, através da função *send*, *call*, *callcode*, *delegatecall* e *staticcall*, consideradas funções de baixo nível. Se houver uma exceção (ex. ficou sem *gas*) no contrato que foi chamado, este termina a sua execução, revertendo o estado e retorna *false*. Dependendo de como a chamada foi feita, a exceção pode ou não ser propagada para o chamador. Por exemplo, se a chamada for feita através das funções de baixo nível apresentadas anteriormente, o chamador tem de verificar o valor de retorno dessa função para certificar se houve ou não exceção. No caso de a chamada ser feita através da função *transfer* fornecida pela Solidity, esta função propaga a exceção, não havendo necessidade de verificar o valor de retorno da função.

```
1  contract Example {
2      address payable owner;
3      uint money;
4
5      function changeOwner() public {
6          owner.send(money);
7          owner = msg.sender;
8      }
9      (...)
10 }
```

Listagem 9: Exemplo de um contrato com uma vulnerabilidade de não verificação do valor de retorno de chamadas de baixo nível

Se existir um contrato, como o apresentado na Listagem 9, que contém uma função que altera o dono de um contrato mas antes de alterar envia o dinheiro presente para o dono original através da função *send*, então o dono original pode ficar sem o dinheiro que lhe pertence. Isto acontece se o contrato chamador não verificar o valor de retorno da função *send*, pois esta invocação ao contrato chamado pode ter lançado uma exceção (ex. divisão por zero, indexar um *array* fora dos seus limites, etc) e consequentemente terminar a sua execução, revertendo o estado, não enviando assim o dinheiro para o dono original. O contrato chamador uma vez que não verifica o valor de retorno irá prosseguir a sua execução como se a função *send* tivesse sido bem sucedida e altera o dono do contrato.

```
1  contract Example {
2      address payable owner;
3      uint money;
4
5      function changeOwner() public {
6          require(owner.send(money), "Cannot send the money to the owner.");
7          owner = msg.sender;
8      }
9      (...)
10 }
```

Listagem 10: Versão sem vulnerabilidade do contrato na Listagem 9

Na Listagem 10 foi adicionada a clausula *require*. Desta forma, se a função *send* retornar *false*, a execução é revertida, eliminando assim a vulnerabilidade presente na Listagem 9.

2.5.5 Vulnerabilidade de Negação de Serviço

Esta vulnerabilidade inclui várias variantes tais como, atingir o limite de *gas* definido, o lançamento inesperado de exceção, tornar um contrato como morto[4] ou violar o controlo de acesso. Estas variantes podem levar a que um contrato fique *offline* para sempre e no caso de haver Ether, este nunca mais será possível movimentar.

Na Listagem 11 é apresentada uma função que é vulnerável a um ataque de negação de serviço. O problema reside na função *transfer()* que propaga uma exceção sempre que ela ocorre. Se o *president* anterior for um Smart Contract e lançar uma exceção sempre que receber um pagamento então nunca haverá outro presidente, pois a linha 4 nunca será executada. O *president* anterior também não irá receber o dinheiro.

```
1  function becomePresident() payable {
2      require(msg.value >= price); // must pay the price to become president
3      president.transfer(price);   // we pay the previous president
4      president = msg.sender;      // we crown the new president
5      price = price * 2;           // we double the price to become president
6  }
```

Listagem 11: Exemplo de uma função com uma vulnerabilidade do tipo negação de serviço [9]

[4]Um contrato morto é um contrato que já não tem código nem estado mas mantém o seu endereço. Sendo que o endereço continua válido podem haver transações que enviam Ether para um contrato morto sendo este Ether perdido para sempre.

Uma possível mitigação será armazenar a informação de que o presidente anterior deve receber um pagamento. Desta forma será retirada a função *transfer()* do contexto da função *becomePresident()*, e deverá ser criada uma nova função com o objetivo de transferir esse valor para o anterior presidente, mantendo de forma explícita que o anterior presidente terá de invocar essa nova função para receber o pagamento. Na Listagem 12 é apresentada uma possível mitigação da vulnerabilidade presente na Listagem 11.

```
function becomePresident() payable {
    require(msg.value >= price); // must pay the price to become president
    if (president != address(0)) // check if president's address is different from 0
        refunds[president] += price;

    president = msg.sender;      // we crown the new president
    price = price * 2;           // we double the price to become president
}

function withdraw() external {
    uint amount = refunds[msg.sender];
    refunds[msg.sender] = 0;
    require(msg.sender.call.value(amount)());
}
```

Listagem 12: Mitigação da vulnerabilidade apresentada na Listagem 11

2.5.6 Vulnerabilidade de Má Aleatoriedade

É difícil obter aleatoriedade no Ethereum, pois nada é privado. Se for usada uma ou mais das variáveis *block.coinbase*, *block.difficulty*, *block.gaslimit*, *block.number*, *block.timestamp* ou *block.blockhash* de forma a tentar obter um número aleatório, este número será controlado pelo *miner* uma vez que é este que atribui os valores a estas variáveis. Outro problema é os programadores usarem uma variável *seed* privada para tentarem obter um número aleatório. Dada a natureza da blockchain, nada é privado e qualquer pessoa pode consultar esse valor.

Na Listagem 13 é apresentada na função *play()* uma vulnerabilidade de má aleatoriedade. Esta função tenta gerar um número aleatório com base na variável privada *seed* e a variável *iteration* que é incrementada a cada invocação. Estas duas variáveis são somadas e é usada a função de *hash keccak256* para obter um número aleatório. No entanto, não é pelo facto da variável *seed* ser privada que o seu valor é privado. No momento de criação do contrato foi atribuído um valor a esta variável, e esse valor é público.

```
uint256 private seed;

function play() public payable {
    require(msg.value >= 1 ether);
    iteration++;
    uint randomNumber = uint(keccak256(seed + iteration));
    if (randomNumber % 2 == 0) {
```

```
8         msg.sender.transfer(this.balance);
9     }
10 }
```

Listagem 13: Vulnerabilidade de má aleatoriedade [9]

A Blockchain não fornece nenhuma fonte criptograficamente segura de aleatoriedade. Por isso é difícil de mitigar este tipo de vulnerabilidade. O problema reside no facto de a fonte de entropia ter de ser externa à Blockchain, portanto a recomendação é desenvolver contratos que não necessitem de aleatoriedade.

2.5.7 Vulnerabilidade de *Front Running*

Esta vulnerabilidade é conhecida também como Dependência da Ordem das Transações. Como mencionado na Secção 2.1, um bloco contém uma lista de transações. Se for assumido o estado da blockchain ser σ e este contiver duas transações (T_1 e T_2) e cada uma delas invocar o mesmo contrato, então a transação T_1 pode ocorrer no estado σ ou então no estado σ' se a transação T_2 executar primeiro e resultar nesse estado σ'. Desta forma, cabe ao *miner* que adiciona o bloco à blockchain decidir a ordem destas transações.

Na Listagem 14 é apresentada uma função com uma vulnerabilidade deste tipo. A função *submitSolution()* recebe um argumento *solution*. O parâmetro da função *submitSolution()* será usado como argumento à função *isCorrect*. Esta função irá determinar se a solução submetida está correta ou não. Caso esteja correta será enviado Ether correspondente à variável *reward* para o utilizador que submeteu a solução.

```
1 function submitSolution(solution) {
2     if (isCorrect(solution))
3         require(msg.sender.call.value(reward)());
4 }
```

Listagem 14: Vulnerabilidade de *front running*

Os *miners* são recompensados com Ether por executarem o código do contrato se o bloco for adicionado à Blockchain. Então, um utilizador pode fornecer uma taxa do *gas* mais alta de forma a que a sua transação seja minada mais rapidamente. Assim, se um utilizador estiver a fornecer a solução para um puzzle ou outro segredo, um outro utilizador malicioso pode roubar a solução e copiar essa transação fornecendo uma taxa do *gas* mais elevada para ser minada primeiro.

Não existem mitigações possíveis para este tipo de vulnerabilidade, pois a ordem das transações depende do *miner* que as minar. Sendo assim os programadores devem estar cientes desta situação e não devem criar contratos em que um utilizador malicioso possa tirar proveito da ordem das transações.

2.5.8 Vulnerabilidade de Manipulação do Tempo

Os contratos podem necessitar de ter informação do tempo atual. Uma forma de obter esta informação é através da variável *block.timestamp* que irá ler a informação presente no bloco em que a transação

se encontra. No entanto esta informação é controlada pelo *miner* e não deveria ser usada, pois o *miner* pode manipular a informação do tempo para beneficiar com isso. A especificação do Ethereum (yellow paper) não especifica nenhuma restrição em relação ao valor que deve ser usado, apenas indica que deve ser maior que o do bloco anterior. No entanto, nas implementações do Ethereum Geth[5] e Parity[6], ambos rejeitam tempos que se desviem mais 15 segundos do tempo atual. Para um bloco ser aceite nestas duas implementações do Ethereum a Equação 2.1 tem de ser satisfeita.

$$block.timestamp \leq now + 15 \wedge block.timestamp > parent.timestamp \tag{2.1}$$

Na Listagem 15 pode ser observada uma função que contém uma vulnerabilidade de manipulação do tempo. Esta função só aceita chamadas que sejam efetuadas depois de uma data específica e, a partir do momento em que a transferência de 1500 Ethers tenha sucesso, esta função não poderá transferir Ether novamente. Como os *miners* podem influenciar o tempo do bloco, eles podem minar um bloco com um tempo definido para o futuro, dentro do intervalo definido na Equação 2.1, não dando oportunidade a outros utilizadores de tentarem jogar, pois a função só permite uma jogada, e assumindo que os outros utilizadores só começam a jogar a partir da data definida.

```
1 function play() public {
2   require(now > 1521763200 && neverPlayed == true);
3   neverPlayed = false;
4   msg.sender.transfer(1500 ether);
5 }
```

Listagem 15: Função vulnerável a ataques de manipulação do tempo [9]

Não existe nenhuma maneira de obter o tempo atual de uma forma segura, portanto os programadores devem estar cientes dos problemas que possam surgir da utilização da variável *block.timestamp*. De forma a mitigar esta vulnerabilidade deve ser desaconselhado o seu uso.

2.5.9 Vulnerabilidade de Endereços Curtos

Ataques de endereços curtos ocorrem porque a EVM aceita argumentos que não estejam devidamente alinhados. Os utilizadores maliciosos podem explorar esta vulnerabilidade passando endereços desalinhados em que a aplicação cliente irá codificar erradamente, pois os endereços estão desalinhados, antes de os introduzir na transação.

Na Listagem 16 existe uma função que recebe um endereço mas não faz nenhuma verificação do seu tamanho. Uma possível invocação normal é a seguinte:

0000000000000000000000001234567890abcdef1234567890abcdef12345678 (endereço *to*, alinhado a 32 bytes)

0000000000000000000000000000000000000000000000000000000000000002 (valor do parâmetro *amount*, alinhado a 32 bytes)

[5] https://github.com/ethereum/go-ethereum/blob/b71334ac3de38338e618aaf8ea6b4a884d2d80f5/consensus/ethash/consensus.go#L46

[6] https://github.com/openethereum/openethereum/blob/ee2b16dfe4895c19c052322a588878e447a13706/ethcore/src/verification/verification.rs#L392

```
1 event Transfer(address indexed _from, address indexed _to, uint256 _value);
2
3 function sendCoin(address to, uint amount) returns(bool sufficient) {
4     if (balances[msg.sender] < amount) return false;
5     balances[msg.sender] -= amount;
6     balances[to] += amount;
7     Transfer(msg.sender, to, amount);
8     return true;
9 }
```

Listagem 16: Vulnerabilidade de endereços curtos [12]

Uma possível invocação para explorar a vulnerabilidade de endereços curtos seria gerar um endereço em que o último byte fosse 0, por exemplo, *0x1234567890abcdef1234567890abcdef12345600* seria um endereço válido para conseguir explorar a função presente na Listagem 16. Desta forma o atacante pode reduzir o seu endereço cortando o último byte. Assim a invocação seria a seguinte:

0000000000000000000000001234567890abcdef1234567890abcdef123456 (endereço *to*, não alinhado a 32 bytes)

0000000000000000000000000000000000000000000000000000000000000002 (valor do parâmetro *amount*, alinhado a 32 bytes)

Nesta invocação, o endereço foi truncado removendo o último byte que era zero, ficando o primeiro argumento com 31 bytes em vez do suposto que são 32 bytes. A EVM irá consumir um byte do segundo argumento de forma ao primeiro argumento estar alinhado a 32 bytes, reduzindo o tamanho do segundo argumento para 31 bytes, ficando desalinhado. A EVM quando consumir o segundo argumento irá adicionar zeros para alinhar o argumento a 32 bytes, aumentando assim o valor do segundo argumento de 2 para 512.

Uma possível mitigação é adicionar uma verificação para determinar se o tamanho do conteúdo da mensagem é igual ao número de parâmetros a multiplicar por 32. Na Listagem 17 é apresentada uma solução para eliminar a vulnerabilidade de endereços curtos. Na linha 4 é somado o número 4 porque o identificador da função a ser chamada ocupa 4 bytes.

```
1  event Transfer(address indexed _from, address indexed _to, uint256 _value);
2
3  function sendCoin(address to, uint amount) returns(bool sufficient) {
4      require(msg.data.length == 2 * 32 + 4);
5      if (balances[msg.sender] < amount) return false;
6      balances[msg.sender] -= amount;
7      balances[to] += amount;
8      Transfer(msg.sender, to, amount);
9      return true;
10 }
```

Listagem 17: Mitigação da vulnerabilidade apresentada na Listagem 16

2.6 Análise das Ferramentas Existentes

Nesta Secção irão ser estudadas e descritas as ferramentas, Oyente, Maian, Gasper, Osiris, teEther, Mythril e Manticore, que usam execução simbólica e três ferramentas, Zeus, Securify e Slither, que usam representação intermédia para realizarem a sua análise e que não usam necessariamente o modelo de execução simbólica na sua análise. Será importante estudar as ferramentas já existentes para perceber o que já está feito e o que pode ser melhorado, como essas ferramentas funcionam e que passos é que estas seguem para realizar a análise. Serão descritas as estratégias usadas e a sua comparação com outras ferramentas. A maioria das ferramentas descritas usam execução simbólica dado que a ferramenta que será construída usará o mesmo método de análise. A ferramenta a ser construída usará uma representação intermédia, por isso foi escolhida uma ferramenta que usa a linguagem LLVM como representação intermédia. Foram também escolhidas outras duas que criaram a própria representação intermédia, sendo uma mais antiga e outra mais recente. Na Tabela 2.1 é apresentado um sumário do motivo pela qual foi feita a escolha das ferramentas apresentadas, o nome da representação intermédia que é usada por algumas quando usadas, qual o tipo de *input* que cada uma recebe e em que Subsecções é que podem ser encontradas. A ferramenta Securify não atribui nenhum nome para a representação intermédia criada.

Ferramentas	Motivo da Escolha	Nome da Representação Intermédia	Tipo de Input	Apresentada na Subsecção
Oyente	Usa execução simbólica	-	Bytecode	2.6.1
Zeus	Usa uma representação intermédia	LLVM	Solidity	2.6.2
Maian	Usa execução simbólica	-	Bytecode	2.6.3.1
Gasper	Usa execução simbólica	-	Bytecode	2.6.3.2
Osiris	Usa execução simbólica	-	Bytecode	2.6.3.3
teEther	Usa execução simbólica	-	Bytecode	2.6.3.4
Mythril	Usa execução simbólica	-	Bytecode	2.6.4
Securify	Cria a própria representação intermédia	(Sem nome)	Bytecode	2.6.5
Slither	Cria a própria representação intermédia	SlithIR	Solidity	2.6.6
Manticore	Usa execução simbólica	-	Bytecode	2.6.7

Tabela 2.1: Sumário das ferramentas escolhidas

2.6.1 Oyente

Luu et al. [13] começam por apresentar algumas das vulnerabilidades presentes nos contratos que permitem a um *miner* malicioso ou a um utilizador malicioso explorar e tirar partido disso. Introduz então quatro vulnerabilidades que já foram apresentadas anteriormente:

- Vulnerabilidade de reentrância (ver Subsecção 2.5.1);
- Exceções não tratadas (ver Subsecção 2.5.4);
- Dependência da ordem das transações (ver Subsecção 2.5.7);
- Dependência do *timestamp* (ver Subsecção 2.5.8).

Depois de apresentadas as quatro vulnerabilidades, é apresentada a ferramenta Oyente, o seu desenho e a sua implementação. Na Figura 2.3 é mostrada a arquitetura da ferramenta.

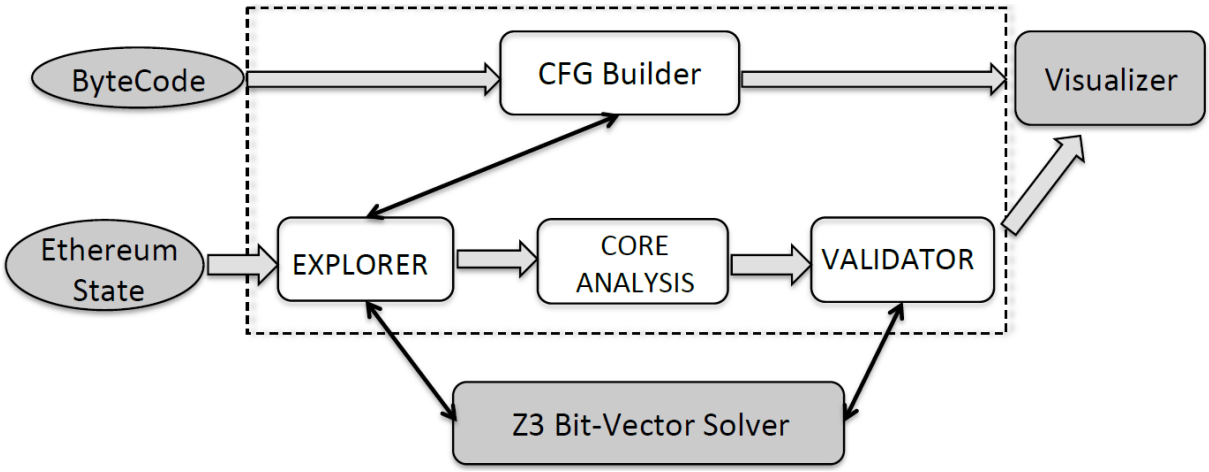


Figura 2.3: Arquitetura da ferramenta Oyente. Componentes principais estão dentro da área a tracejado. Caixas cinzentas estão disponíveis publicamente. [13] (Republished with permission of ACM, from Making Smart Contracts Smarter, L. Luu et al., Proceedings of the 2016 ACM SIGSAC Conferenceon Computer and Communications Security, 2016; permission conveyed through Copyright Clearance Center, Inc.)

A Oyente analisa os Smart Contracts diretamente através do seu *bytecode*. Esta escolha deve-se ao facto de serem raros os Smart Contracts que têm associado o seu código fonte. Isto significa que esta ferramenta, tal como todas as que analisam o *bytecode* ao invés do código fonte, são compatíveis com as diferentes linguagens de programação que existam para a EVM. A análise feita pela Oyente é uma análise simbólica.

Para analisar um contrato é necessário fornecer como entrada o *bytecode* correspondente ao contrato e o estado global atual da rede Ethereum. A ferramenta começa por construir um *Control Flow Graph* (*CFG Builder* na Figura 2.3). Este CFG é uma árvore com os caminhos possíveis que a execução pode tomar. Quando é encontrada uma expressão condicional, a execução pode seguir por dois caminhos, o caminho em que a condição seja verdadeira ou o caminho em que a condição seja falsa. Para cada um dos casos é adicionado um nó a essa árvore. O resultado final é passado ao componente *explorer*.

O *explorer* itera por todos os estados, sem nenhuma heurística, fazendo uma execução simbólica, parando quando não houver mais estados ou se houver *timeout*. Para cada expressão booleana, o *explorer* questiona o Z3 para este determinar se a expressão pode ser verdadeira ou falsa. No final desta fase é criado um conjunto de traços simbólicos em que cada traço está associado a um caminho. O módulo Z3 é usado nesta fase para eliminar caminhos impraticáveis. O resultado final deste módulo é então passado ao módulo *Core Analysis* que contém sub módulos.

Existem quatro sub módulos, um para cada vulnerabilidade. Cada sub módulo é responsável por verificar se deteta a vulnerabilidade que lhe compete.

Por fim, o último módulo *Validator* tenta remover falsos positivos usando novamente o Z3 para verificar se os traços reportados pelo módulo *Core Analysis* realmente refletem alguma vulnerabilidade.

Este módulo ainda está longe de ser completo uma vez que os autores não simulam totalmente o ambiente de execução do Ethereum.

Algumas das vulnerabilidades presentes nos Smart Contracts devem-se ao facto de os programadores assumirem uma semântica da EVM que não é a real. Em Luu et al. [13] é também proposta uma solução para esta falha onde os autores formalizam a semântica dos Ethereum Smart Contracts e propõem recomendações como soluções para alguns *bugs*. Grishchenko et al. [14] tentam resolver o mesmo problema onde introduzem e formalizam a primeira semântica completa de *small-steps*.

Esta ferramenta serviu como base a quase todas as descritas de seguida e foi a primeira - do nosso conhecimento - a realizar execução simbólica sobre o *bytecode* EVM. A Oyente não usa nenhuma representação intermédia e consequentemente não usa as instruções EVM na forma atribuição única estática (SSA), o que se considera ser um ponto negativo. Por outro lado, é difícil acrescentar novos sub módulos para ser possível detetar novas vulnerabilidades.

2.6.2 Zeus

No artigo [15] é apresentada uma framework para verificar a correção e a justiça dos Smart Contracts. A correção é considerada como a prática de programação segura. A justiça é considerada como a concordância da lógica de negócio no nível mais elevado. Os autores começam por apresentar algumas formas em que os Smart Contracts podem ser considerados incorretos ou injustos.

Os desafios de implementação que levam ao comportamento incorreto são: a reentrância, enviar dinheiro sem verificar se a operação sucedeu ou não, envios com falhas, *integer overflow/underflow* bem como a dependência do estado da transação.

Os contratos também podem ser injustos devido a erros da lógica de negócio. Esses erros podem ser a ausência da lógica necessária, lógica incorreta ou lógica correta mas injusta. Um exemplo da ausência de lógica é um contrato poder ser destruído por qualquer utilizador, quando o que faria sentido seria ser destruído apenas pelo dono do contrato, considerando ser esta última a funcionalidade pretendida. Exemplos de lógica incorreta ou lógica correta mas injusta, são fáceis de imaginar.

Outra fonte de vulnerabilidade são os *miners*. Os autores indicam dois tipos de vulnerabilidades nesta categoria que são a dependência do estado do bloco e a dependência da ordem das transações.

A ferramenta proposta pelos autores chama-se Zeus. Esta recebe como entrada o código fonte de um contrato e uma política, escrita numa linguagem específica, com a qual o contrato irá ser verificado. É realizada uma análise estática do código do contrato e são inseridas em certos pontos do contrato cláusulas *assert*, com base nos predicados da política. Depois é convertido o código fonte do contrato com as cláusulas *assert* para *bitcode* LLVM. Finalmente a ferramenta invoca um verificador para determinar se os *assert* são violados de forma a indicar se as políticas são violadas.

Os contratos são modelados usando uma linguagem abstrata. O código fonte Solidity é traduzido para essa linguagem abstrata que irá incluir *asserts* para definir propriedades atingíveis. A linguagem das políticas permite ao utilizador especificar regras de justiça para um contrato. Com isto, a Zeus interpreta a linguagem das políticas para extrair os predicados a serem usados nos *asserts* e os locais

de controlo corretos para inserir as cláusulas *assert* na linguagem abstrata.

Dado o programa aprimorado pelas políticas, a Zeus converte para *bitcode* LLVM. Finalmente, a Zeus fornece a representação do *bitcode* LLVM para um mecanismo de verificação já existente para rapidamente verificar a segurança do contrato.

No final os autores comparam os resultados obtidos com a ferramenta Oyente. A ferramenta Zeus procura mais vulnerabilidades do que a ferramenta Oyente. Para as vulnerabilidades procuradas por ambas as ferramentas, a ferramenta Zeus tem muito menos falsos positivos do que a ferramenta Oyente. A ferramenta Zeus encontra mais contratos como sendo *safe* e mais como sendo *unsafe* do que a ferramenta Oyente, para cada uma das vulnerabilidades que ambas procuram. Para cada vulnerabilidade que a ferramenta Zeus procura, esta não teve nenhum falso negativo. Quando comparado em termos de tempo que demora a analisar, a ferramenta Zeus concluiu a sua análise, em 97% dos contratos, em menos de um minuto. A ferramenta Oyente apenas concluiu a sua análise nesse mesmo tempo em 43% dos contratos.

A Zeus é uma ferramenta desenvolvida pela *IBM Research India* e não é *open source*. Um dos problemas é que esta ferramenta precisa que o utilizador tenha algum conhecimento da linguagem das políticas para poder especificar as suas regras.

2.6.3 Ferramentas Baseadas no Oyente

2.6.3.1 Maian

Ao contrário das ferramentas Oyente e Zeus, Nikolić et al. [16] não analisam as vulnerabilidades que podem ser causadas por uma única invocação de um contrato, mas analisam o que pode acontecer quando mais do que uma invocação acontece num dado período de tempo. Os autores definem um traço de vulnerabilidade como uma série de invocações de um contrato.

Os autores descrevem três tipos de traços de vulnerabilidade que são contratos pródigos (*prodigal*), suicidas (*suicidal*) e gananciosos (*greedy*).

A definição de contratos *pródigos* são os que enviam Ether para endereços arbitrários que não são proprietários do contrato, ou que nunca depositaram Ether nesse contrato e que só forneceram dados que são facilmente produzidos por terceiros. Um exemplo de um contrato pródigo é um contrato que tem uma função que recebe uma lista de endereços e uma lista de valores e para cada endereço envia Ether. Esta função não tem qualquer tipo de restrição em quem envia dinheiro, o que permite a qualquer utilizador invocar esta função passando como argumentos o que bem entender.

Contratos *suicidas* são contratos que podem ser eliminados por qualquer utilizador através da invocação de uma função suicida. Uma função suicida é uma função que executa a instrução *SUICIDE*. O exemplo que os autores dão é o ataque *Parity Wallet* [17] em que o atacante primeiro invoca uma função que permite-lhe tornar-se dono do contrato e desta forma poder invocar qualquer outra função uma vez que tem os privilégios necessários. O atacante depois invoca a função que executa a instrução *SUICIDE*, tornando o contrato morto.

Contratos *gananciosos* são os contratos que permanecem vivos e que bloqueiam Ether indefinida-

mente e não permitem que o Ether seja libertado em nenhuma circunstância. O ataque *Parity Wallet* serve novamente de exemplo onde após a remoção da biblioteca *Parity*, os contratos que usavam funções desta biblioteca não podem libertar mais Ether.

Os autores definem ainda uma subclasse dos contratos gananciosos que são os contratos póstumos (*posthumous*). Estes são contratos que foram mortos, mas que o seu saldo é maior que zero.

Com base nestes três traços de vulnerabilidade, a ferramenta Maian procura contratos que violem as seguintes propriedades dos traços definidos: propriedades *safety*, afirmando que existe um traço a partir de um estado específico da blockchain que causa a violação de um contrato em certas condições; propriedades *liveness*, afirmando que algumas ações não podem ser tomadas em nenhuma execução partindo de um estado específico da blockchain. De uma forma simplificada, isto significa que para a propriedade *safety*, no final da execução nada de mal acontece e quando isto estiver confirmado, a propriedade *liveness* verifica os contratos à procura dos resultados desejados.

Para analisar os contratos, a ferramenta utiliza análise simbólica e validação concreta. Em relação à análise simbólica, esta recebe como entrada o *bytecode* do contrato e alguns parâmetros. Os parâmetros são a categoria de vulnerabilidade, conforme definido anteriormente, e a profundidade de chamadas, isto é o limite na profundidade da pesquisa. Para facilitar a execução simbólica foi implementada uma Ethereum Virtual Machine personalizada. Para cada contrato, são executados os possíveis traços de caminhos simbolicamente até encontrar um traço que satisfaça um conjunto de propriedades predeterminadas.

Quando um contrato viola as propriedades, são retornados valores concretos para as variáveis simbólicas. O passo final é executar o contrato concretamente e validar os resultados como verdadeiros positivos. Se a violação não for validada então é marcado como falso positivo. Esta execução é realizada num *fork* privado da Ethereum, não causando alteração na blockchain oficial do Ethereum.

Os candidatos a contratos pródigos são testados para verificar se libertam Ether através de uma conta de testes que sequencialmente envia transações e espera receber Ether de volta. Os contratos suicidas são testados de forma igual. O *bytecode* de um contrato morto é *0x*, que é usado para verificar se o contrato viola essa propriedade tornando-se num verdadeiro positivo. Para os contratos gananciosos é verificado na análise simbólica se o contrato aceita Ether e nesse caso a validação só necessita de verificar se não existem as instruções *CALL*, *DELEGATECALL* e *SUICIDE* para ser considerado como verdadeiro positivo.

Os autores avaliaram a ferramenta com base em 970.898 contratos presentes no Ethereum. Tiveram um total de 89% de verdadeiros positivos. Os contratos gananciosos foram os mais encontrados com 91,2% e os suicidas e os pródigos tiveram a percentagem de 4,4%. Os contratos suicidas tiveram 99% de verdadeiros positivos. Os pródigos tiveram 97% de verdadeiros positivos enquanto que os gananciosos tiveram 69% de verdadeiros positivos. Esta baixa percentagem deve-se ao facto de a definição destes contratos e a estratégia que usaram não foi a melhor. Por fim, foram encontrados 853 contratos póstumos e o mais interessante que a ferramenta descobriu é que 294 contratos receberam Ether depois de estarem mortos. Estas vulnerabilidades que são procuradas nos contratos são vulnerabilidades muito específicas. Os autores também não referem a possibilidade de criar novos módulos

para detetar outros tipos de vulnerabilidades.

2.6.3.2 Gasper

Em Chen et al. [18] é apresentada a ferramenta Gasper, também esta baseada na Oyente (apresentada na Subsecção 2.6.1). A Gasper procura por padrões no custo do *gas*. Para isso, usa os módulos *CFG Builder* e *Explorer* da ferramenta Oyente. Os padrões que procura são código morto (*dead code*), ciclos que contenham instruções com um custo de *gas* elevado - só tem em consideração três instruções - e predicados opacos, ou seja, contratos em que um dos caminhos do CFG construído não possa ser executado.

2.6.3.3 Osiris

Em Torres et al. [19] é apresentada outra ferramenta baseada na Oyente, chamada Osiris. Esta ferramenta destina-se a encontrar problemas relacionados com operações aritméticas. Recebe como entrada o *bytecode* do Smart Contract e constrói um CFG. Sobre este é realizada a execução simbólica que questiona o SMT Solver Z3 para determinar os caminhos possíveis, tal como acontece com a Oyente. A ferramenta usa também *taint analysis*, mantendo registo da propagação dos dados no CFG. Desta forma é possível distinguir *overflows* ou *underflows* benignos dos maliciosos.

2.6.3.4 teEther

Em Krupp and Rossow [20] é apresentada a ferramenta teEther. Esta ferramenta funciona de maneira idêntica em algumas partes à ferramenta Oyente. A teEther é capaz de detetar vulnerabilidades, criar *exploits* e verificar se é possível explorar a vulnerabilidade com o *exploit* criado. Enquanto a Oyente procura por vulnerabilidades que possam ser exploradas tanto por *miners* como por utilizadores normais, esta ferramenta só procura por vulnerabilidades que possam ser exploradas por utilizadores normais. O tipo de vulnerabilidade que a teEther procura são vulnerabilidades que enviem dinheiro para endereços arbitrários.

Esta ferramenta constrói um CFG com base no *bytecode*. Depois é realizada a execução simbólica para determinar os caminhos possíveis e usa o SMT Solver Z3 para eliminar os caminhos impossíveis para reduzir o espaço de procura. São então criadas as restrições para chegar a um certo ponto do CFG. O *exploit* é criado tentando satisfazer todas as restrições. De forma a reduzir falsos positivos, os *exploits* são testados numa Blockchain privada.

2.6.4 Mythril

No artigo [21] é apresentada a ferramenta Mythril que não se limita a identificar vulnerabilidades com interações só de um contrato mas sim com outros contratos tal como a ferramenta Maian. Esta ferramenta usa análise simbólica para detetar vulnerabilidades. Tal como a ferramenta apresentada anteriormente, esta também usa execução concreta. Isto quer dizer que na sua análise, a ferramenta usa execução concreta combinada com execução simbólica, também designada por *concolic execution*.

O funcionamento desta ferramenta é idêntico às ferramentas já apresentadas. Recebe como entrada o *bytecode* do contrato e extrai um *CFG*. A partir deste, é realizada então a análise simbólica. O modelo concólico incorpora variáveis concretas e simbólicas. A Mythril usa valores concretos quando estes são atribuídos explicitamente durante a execução, mas todas as variáveis de estado são inicialmente simbólicas. No artigo é referido que uma possível melhoria seria inicializar o estado global com valores concretos, quer seja executando o construtor do contrato - e neste caso o código fonte teria de ser fornecido como entrada em vez do *bytecode* - ou lendo as variáveis de estado referenciadas na blockchain.

O autor não realizou nenhuma comparação da sua ferramenta com as já existentes, o que torna difícil tirar conclusões sobre a mesma. No entanto, a sua avaliação experimental consistiu em executar alguns contratos de competições *Capture The Flag* (CTF) e verificar se detetava as vulnerabilidades. O autor refere que para trabalho futuro quer reduzir a taxa de falsos positivos.

2.6.5 Securify

Em Tsankov et al. [22] identificam um conjunto de padrões nos grafos de fluxo de dados dos contratos, expressos como regras num *Datalog*, que podem verificar a conformidade com uma propriedade de segurança desejada ou a não conformidade. Além de um catálogo de padrões prontos para usar, os utilizadores da ferramenta Securify também podem expressar as suas próprias regras específicas. Este catálogo inclui padrões para detetar a dependência da ordem das transações, exceções não tratadas, entre outras.

O modo de operação da Securify começa com o *bytecode* EVM - ou o código fonte do contrato em Solidity que será depois compilado para *bytecode*. O *bytecode* EVM é primeiro transformado no formato de atribuição única estática (SSA), sendo este um padrão na análise de código. Muitas instruções (como *push*, *pop*, etc) são eliminadas durante esta tradução para uma representação sem pilha (*stackless*). Ter uma representação na forma SSA requer que a cada variável apenas lhe seja atribuído um valor uma única vez. Por exemplo, a sequência $x = 3; y = x + +; x = y + x$, usando a forma SSA fica $x_1 = 3; y_1 = x_1; x_2 = x_1 + 1; x_3 = y_1 + x_2$. Uma das principais vantagens de usar a forma SSA é facilitar a computação dos caminhos *def-use*, permitindo uma análise da dependência dos dados mais direta. Depois de convertido para a forma SSA, é realizada uma análise estática que analisa todos os caminhos e depois é convertido num conjunto de factos do *Datalog*. O conjunto de padrões expressos como regras de um *Datalog* são então avaliados e as correspondências são compiladas num relatório. Na figura 2.4 é ilustrado o fluxo de análise. Começa com o *bytecode*, onde são derivados os factos semânticos inferidos pela análise do gráfico de dependência do contrato. De seguida, usa esses factos para verificar um conjunto de padrões de conformidade e violação. Por fim, classifica-os e produz o relatório final.

Os padrões são divididos em padrões de conformidade, que indicam que uma propriedade é considerada satisfeita, e padrões de violação que indicam que uma propriedade não é válida. Quando uma propriedade não puder ser categorizada definitivamente como um padrão de conformidade ou violação,

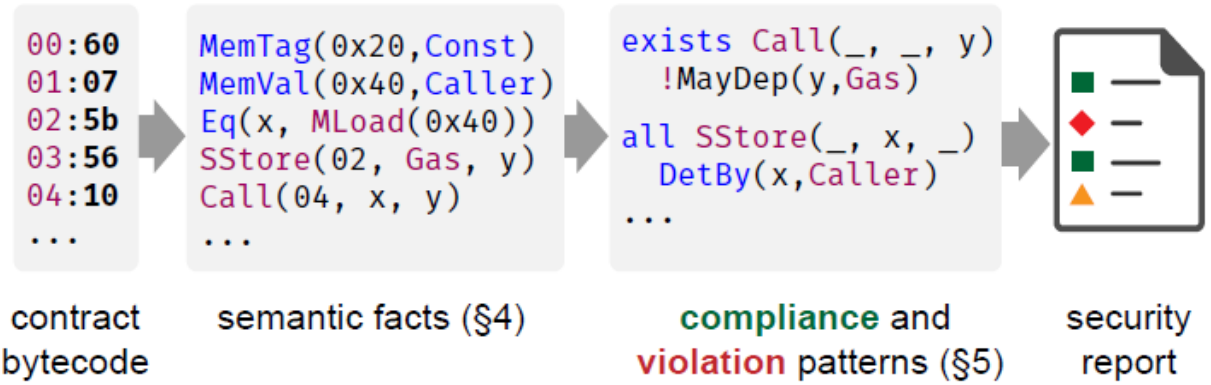


Figura 2.4: Abordagem do Securify baseada na inferência automática de factos da semântica do programa seguido da verificação de conformidade e violação de padrões de segurança sobre estes factos [22] (Republished with permission of ACM, from Securify: Practical Security Analysis of Smart Contracts, P. Tsankov et al., Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 2018; permission conveyed through Copyright Clearance Center, Inc.)

é sinalizado um aviso para o utilizador e será necessária uma avaliação manual.

Para as instruções que não são removidas na tradução para a forma SSA, o Securify infere um conjunto de factos base na forma $instr(L, Y, X_1, ..., X_n)$, onde $instr$ é o nome da instrução, L é a etiqueta da instrução, Y é a instrução que guarda o resultado da instrução (se houver) e X_i são os argumentos. Para além disto, o Securify adiciona um facto *follow*, $Follow(L_1, L_2)$ para indicar que a instrução da etiqueta L_2 acontece imediatamente a seguir à instrução L_1. Por exemplo, o facto base *assign(l,x,caller)* significa que *assign* é o nome da instrução, *l* é a etiqueta da instrução, *x* é a variável que guarda o endereço do chamador e *caller* é o endereço do chamador. Com este facto base é possível construir um facto semântico *Eq(x,caller)* que significa que *x* é igual a *caller*.

A Securify permite aos utilizadores escreverem predicados para capturar os padrões que os utilizadores pretendem. Os utilizadores expressam os seus padrões numa linguagem de domínio específica (DSL). Existe uma gramática das regras que os utilizadores terão de usar para expressar os seus padrões. No artigo [22] é apresentada essa gramática como exemplo de alguns padrões para identificar a conformidade ou violação de propriedades de segurança.

Os autores comparam a Securify com a ferramenta Oyente e com a ferramenta Mythril. Quando comparado a essas ferramentas, a Securify reporta mais violações. No entanto, a Securify apresenta alguns falsos avisos, mas sempre menos do que as outras duas ferramentas. Em relação à vulnerabilidade de reentrância, todas as ferramentas reportam o mesmo número de vulnerabilidades, sendo que a Securify não apresentou nenhum falso aviso.

Uma limitação desta ferramenta é que não é capaz de raciocinar sobre propriedades numéricas como *overflows*. Outra é que os seus utilizadores precisam de conhecer a gramática da DSL para poderem expressar os seus padrões.

2.6.6 Slither

Feist et al. [23] apresentam a Slither, uma ferramenta de análise estática que também é usada para

realçar o entendimento do utilizador sobre um Smart Contract, assistir na revisão de código e detetar otimizações que não foram realizadas.

A análise feita pela Slither é feita em várias fases. Começa por receber como entrada a árvore de sintaxe abstrata (AST) da Solidity gerada pelo compilador da Solidity a partir do código fonte do contrato. Na primeira fase, a Slither recolhe informações importantes, como o grafo de herança do contrato, o grafo de controlo de fluxo (CFG) e a lista de expressões. De seguida, é traduzido o código inteiro do contrato para a linguagem de representação intermédia SlithIR [24]. Esta linguagem interna usa a atribuição única estática (SSA) para facilitar a computação da análise de código. Na terceira fase é realizada a análise de código. A Slither calcula um conjunto de análises predefinidas que fornecem informações para os outros módulos. Na Figura 2.5 são indicadas todas as fases.

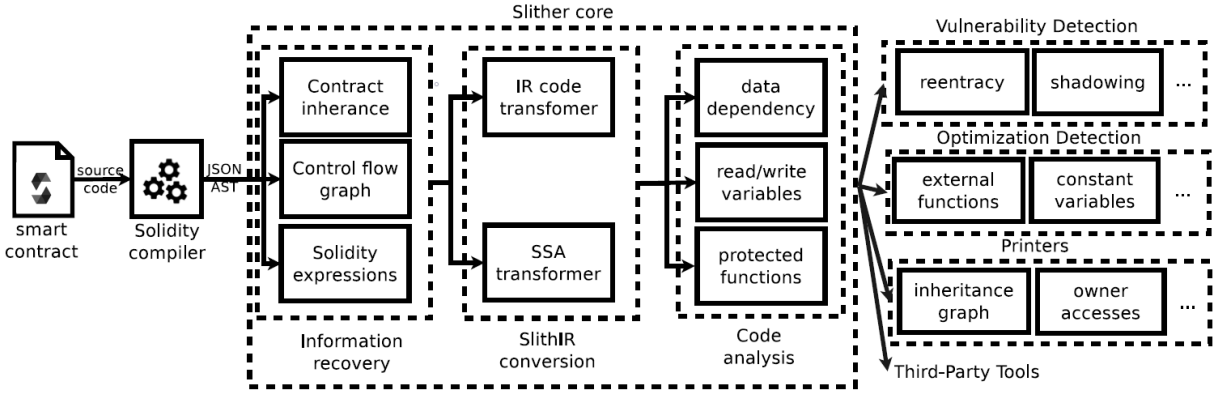


Figura 2.5: Visão global da Slither [23] © 2019 IEEE

Os utilizadores podem construir as suas ferramentas usando a API em Python fornecida pela Slither. Essas ferramentas podem usar módulos internos da Slither para construir ferramentas de análise mais avançadas como execução simbólica em cima da SlithIR ou a conversão de SlithIR para outra representação intermédia como a LLVM. Uma limitação desta ferramenta é que necessita da AST da Solidity e não funciona para contratos que não tenham o código fonte associado.

2.6.7 Manticore

Mossberg et al. [25] apresentam outra ferramenta de análise dinâmica que usa execução simbólica chamada Manticore, que se destaca não só por fazer análises a Smart Contracts mas também a binários (x86/64, ARM).

A execução simbólica nos Smart Contracts envolve transações simbólicas, onde o valor e os dados são simbólicos. As transações simbólicas podem ser executadas várias vezes para explorar o espaço de estados de um contrato. A Manticore suporta um número arbitrário de interações entre contratos tal como a ferramenta Maian.

2.6.8 Representações Intermédias

Usar uma representação intermédia ajuda a implementar analisadores de código mais facilmente, sem perder informações de semântica críticas presentes no código fonte Solidity. SlithIR é uma representa-

ção intermédia híbrida usada pela Slither para representar código Solidity. Cada nó do grafo do controlo de fluxo pode conter no máximo uma expressão Solidity que depois é convertida num conjunto de instruções SlithIR. Este conjunto de instruções é relativamente pequeno, com menos de 40 instruções. Esta representação intermédia não produz nenhuma representação do controlo de fluxo interno pois está associada a cada nó do grafo. Esta representação usa a forma atribuição única estática (SSA).

Para além da SlithIR há a representação intermédia da ferramenta Rattle [8] que recebe como entrada o *bytecode* EVM, reutiliza o *disassembler* da ferramenta Manticore, constrói um CFG e converte para uma representação intermédia na forma SSA, transformando as instruções da forma *stack* para a forma de registos. Existem outras representações intermédias, mas nenhuma delas permite uma análise mais facilitada por ter uma representação mais alto nível em comparação com outras representações intermédias que são mais baixo nível, o que torna a análise mais complicada. Os autores da Solidity criaram também uma linguagem intermédia chamada YUL [26] mas esta não fornece o mesmo nível de informação que a SlithIR ou que a representação intermédia usada na ferramenta Rattle. Albert et al. [27] apresenta outra representação intermédia, a EthIR, que é uma extensão da ferramenta Oyente. Esta representação intermédia usa a Oyente para construir o CFG e a partir deste produz uma *rule-based representation* (RBR). Esta representação mais alto nível permite inferir propriedades do *bytecode* que será executado na EVM. No entanto, nem a YUL nem a EthIR fornecem uma representação intermédia na forma SSA. Por outro lado, a SlithIR e a representação intermédia do Rattle não são perfeitas e carecem de uma semântica formal, o que permitiria uma análise mais rigorosa. Uma outra limitação é que as representações são demasiado alto nível para fornecerem informações rigorosas de mais baixo nível, como é o caso do *gas*.

2.6.9 Resumo das Ferramentas Analisadas

Depois de estudadas as ferramentas que usam o modelo de execução simbólica, todas têm em comum alguns dos passos utilizados na construção dessas. Começam pela construção de um CFG que servirá para mais tarde iterar sobre os vários caminhos. Mais tarde, é elaborado um motor de execução simbólica que tem como propósito executar simbolicamente as instruções EVM. Neste passo são criadas restrições simbólicas que servirão para mais tarde verificar se produzem algum tipo de vulnerabilidade. O último passo em comum é o desenvolvimento de alguns detetores de vulnerabilidades.

As desvantagens destas ferramentas é que não usam uma representação intermédia na sua análise nem as instruções na forma SSA. Considera-se desvantagens pois os resultados obtidos pelas ferramentas estudadas que usam representação intermédia mostraram resultados melhores. Exemplos são a ferramenta Zeus e a Securify que apresentam melhores resultados comparativamente à ferramenta Oyente, e no caso da ferramenta Securify apresenta também melhores resultados que a ferramenta Mythril. Apesar de ambas as ferramentas, Zeus e Securify, usarem um modelo diferente das ferramentas Oyente e Mythril, pensa-se que ao usar uma representação intermédia, esta ajude bastante na análise feita por qualquer ferramenta uma vez que permite raciocinar melhor, ter um nível de abstração maior e ter mais informações do que ao não usar uma representação intermédia.

As ideias que serão aproveitas para a elaboração da ferramenta de análise estática de Smart Contracts são os passos em comum que ambas as ferramentas que usam o modelo de execução simbólica usam. Deste modo, será construído um CFG, um motor de execução simbólica e detetores para detetar alguns tipos de vulnerabilidades. Será ainda usada uma representação intermédia na elaboração da ferramenta, o que se pensa que será uma vantagem em relação às outras ferramentas. Do conhecimento adquirido, pensa-se que será a primeira ferramenta que usa o modelo de execução simbólica a adotar o uso de uma representação intermédia na sua análise.

Na Tabela 2.2 são apresentadas as ferramentas estudadas, qual o tipo de *input* que recebem e quais os tipos de vulnerabilidades, segundo o DASP 10, que detetam. As ferramentas que recebem como *input* o *bytecode* do contrato a analisar podem também receber o código fonte Solidity ou então este pode ser compilado antes de ser fornecido à ferramenta que irá analisá-lo. A categoria "Others" indica que a ferramenta deteta outro tipo de categoria que não está presente na taxonomia usada, DASP 10. A ferramenta teEther não deteta nenhum tipo de vulnerabilidade em específico, apenas gera *exploits*[7].

Ferramenta	Tipo de Input	Reentrancy	Access Control	Arithmetic	Unchecked Low Level Calls	Denial of Service	Bad Randomness	Front Running	Time Manipulation	Short Addresses	Others
Gasper	Bytecode										✓
Maian	Bytecode		✓								✓
Manticore	Bytecode	✓	✓	✓	✓				✓		✓
Mythril	Bytecode	✓	✓	✓	✓			✓			✓
Osiris	Bytecode	✓		✓		✓			✓		
Oyente	Bytecode	✓	✓	✓		✓		✓	✓		
Securify	Bytecode	✓	✓		✓			✓			✓
Slither	Código Fonte Solidity	✓	✓		✓	✓			✓		✓
teEther	Bytecode										
Zeus	Código Fonte Solidity	✓		✓	✓	✓		✓			✓

Tabela 2.2: Resumo das ferramentas analisadas, qual o tipo de input e as categorias que cada ferramenta deteta segundo o DASP Top 10

Na Tabela 2.3 pode ser observado, para cada ferramenta, o nome das vulnerabilidades que foram consideradas "Others" na Tabela 2.2. A vulnerabilidade *Gas-Costly Patterns* procura por padrões que usem *gas* em excesso e que possa ser reduzido. *Locked Ether* refere-se a contratos que podem bloquear Ether indefinidamente, não dando a possibilidade de movimentar Ether. *INVALID Instruction Used* procura por caminhos que executem a instrução *INVALID*. A vulnerabilidade *Block/Transaction State Dependence* procura por instruções que extraiam informações presentes no bloco ou transação, por exemplo *BLOCKHASH*, *NUMBER* ou *TIMESTAMP*. *Uninitialized Variable* procura por variáveis da *stack*, *memory* ou *storage* que não sejam inicializadas. Outra vulnerabilidade é *Exception State* que procura por *asserts* que possam gerar uma exceção. Por fim, a vulnerabilidade *Incorrect-Equality* procura por restrições impostas numa igualdade que possam ser manipuladas por um atacante.

[7]Um *exploit* é um pedaço de código capaz de explorar e tirar proveito de uma vulnerabilidade.

Ferramenta	Nome(s) da(s) Vulnerabilidade(s) Considerada(s) “Others”
Gasper	Gas-Costly Patterns
Maian	Locked Ether
Manticore	INVALID Instruction Used; Block/Transaction State Dependence; Uninitialized Variable
Mythril	Exception State; Block/Transaction State Dependence
Securify	Locked Ether
Slither	Incorret-Equality; Locked Ether; Uninitialized Variable
Zeus	Block/Transaction State Dependence

Tabela 2.3: Nome das vulnerabilidades consideradas “Others” na Tabela 2.2 para cada ferramenta

Capítulo 3

Conkas: Ferramenta Modular

Neste Capítulo é apresentada a Conkas, uma ferramenta de análise estática que usa o modelo de execução simbólica. É também apresentada uma versão modificada da representação intermédia da ferramenta Rattle bem como as razões para essas modificações. São ainda discutidos os obstáculos encontrados durante o desenvolvimento da Conkas e como foram ultrapassados.

3.1 Rattle

A Rattle é desenvolvida por Ryan Stortz e é uma ferramenta desenhada para funcionar em Smart Contracts que já estejam presentes na Blockchain, funcionando através do *bytecode*. Nesta ferramenta é construída uma representação intermédia na forma SSA e são convertidas as instruções da forma *stack* para a forma de registos. Para além disto é realizada uma otimização, removendo várias instruções tais como *DUPs*, *SWAPs*, *PUSHs* e *POPs*. Esta transformação remove mais de 60% das instruções EVM. Esta remoção acontece devido ao facto das instruções EVM serem convertidas da forma *stack* para a forma de registos. Não é possível afirmar que esta remoção garanta a 100% o mesmo comportamento que o *bytecode* original, para tal seria necessária uma semântica formal.

Esta ferramenta recupera as funções assim como os seus argumentos, a localização da memória e do *storage*. Na figura 3.1 é mostrado um exemplo do output da ferramenta Rattle[1]. Neste exemplo é indicado o *offset* de onde começa a função no *bytecode* assim como o índice do *storage* que é usado. As funções na EVM são identificadas pelos 4 primeiros bytes do *hash* produzido pela função *SHA3*. A esta função de *hash* apenas é passada a assinatura da função. É possível observar pela Figura 3.1 que a Rattle apresenta o identificador da função (4 primeiros bytes do *hash*) assim como a assinatura da função. Para a ferramenta recuperar as assinaturas das funções, esta acede a uma base de dados local que tem o mapeamento de algumas *hashes* para a assinatura da função correspondente.

A função apresentada na Figura 3.1 começa por verificar se esta é invocada com Ether presente na transação, caso seja verdade irá reverter a transação. No caso de não ser verdade irá carregar para a *stack* o valor presente no índice 6 do *storage*. Este valor será copiado para o índice de memória

[1] `https://github.com/crytic/rattle/blob/a3fa9c7c773bc8c805317df597da1f216f1b9293/example.png`

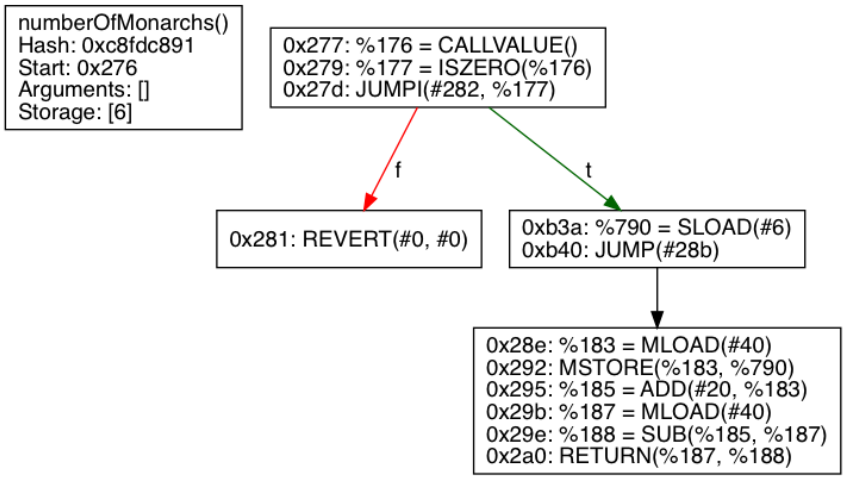


Figura 3.1: Exemplo do *output* da ferramenta Rattle [28]

presente em M[0x40] (valor da memória no índice 0x40). Será retornado o valor presente no índice de memória em M[0x40:0x40 + 0x20]. Simplificando, o valor no índice 0x6 do *storage* armazena o *length* de um *array*. Esta função está a retornar o *length* de um determinado *array*. No entanto só é possível retornar conteúdo presente na memória e este conteúdo está a ser colocado na memória para ser retornado. A Listagem 18 mostra o código Solidity correspondente à função apresentada na Figura 3.1.

```
1 function numberOfMonarchs() constant returns (uint n) {
2     return pastMonarchs.length;
3 }
```

Listagem 18: Código Solidity correspondente ao exemplo do *output* da ferramenta Rattle na Figura 3.1 [28]

3.2 Modificações à Ferramenta Rattle

Inicialmente a ferramenta Rattle gerava alguns erros, principalmente para contratos escritos em Solidity com a versão igual ou posterior à 0.5.0. Outros erros eram gerados por falta de algumas restrições. Outra razão é que havia uma otimização realizada que deixava o estado interno desatualizado. Foi então necessário realizar algumas modificações para a tornar mais robusta. De seguida serão apresentadas as modificações à Rattle.

Um dos primeiros passos foi atualizar a base de dados local com Auxílio à API fornecida pela *Ethereum Function Signature Database*[2]. O passo seguinte foi remover o *disassembler* usado, que foi aproveitado da ferramenta Manticore, e adicionar a dependência *pyevmasm*[3] fazendo as ações necessárias para integrar a nova dependência. Desta forma, sempre que existam atualizações à EVM e sejam adicionadas novas instruções, é esperado que o *disassembler* esteja sempre atualizado.

Quando um Smart Contract é compilado usando o compilador da Solidity é concatenado o *Swarm*

[2] https://www.4byte.directory/
[3] https://github.com/crytic/pyevmasm

Hash[4] no final do *bytecode*. A ferramenta Rattle gerava erros em alguns contratos porque esta procurava por uma assinatura do *Swarm Hash*. No entanto esta assinatura podia vir a sofrer alterações como é mencionado na documentação da Solidity, e, portanto, os contratos que não usavam a assinatura esperada pela ferramenta geravam erros. Assim, a ferramenta não era compatível com todas as versões da Solidity. Para tornar a ferramenta compatível com todas as versões da Solidity e de forma a poder ser usada na Conkas, foi atualizada a função responsável por detetar o *Swarm Hash* e removê-lo. No entanto a ferramenta abortava com algumas exceções, pelo que foi necessário investigar as causas e corrigir. Chegou-se à conclusão que haviam restrições que não estavam impostas e que deveriam de estar pelo que foram adicionadas. Exemplos disso são a não verificação do número de argumentos de uma instrução ou a não verificação se uma instrução pertence à categoria *is_branch* que inclui as instruções *JUMP* e *JUMPI*.

Quando é construído um CFG correspondente a uma função e usando essas instruções na forma SSA irá haver um problema. Na figura 3.2 é apresentado um exemplo de um CFG na forma SSA.

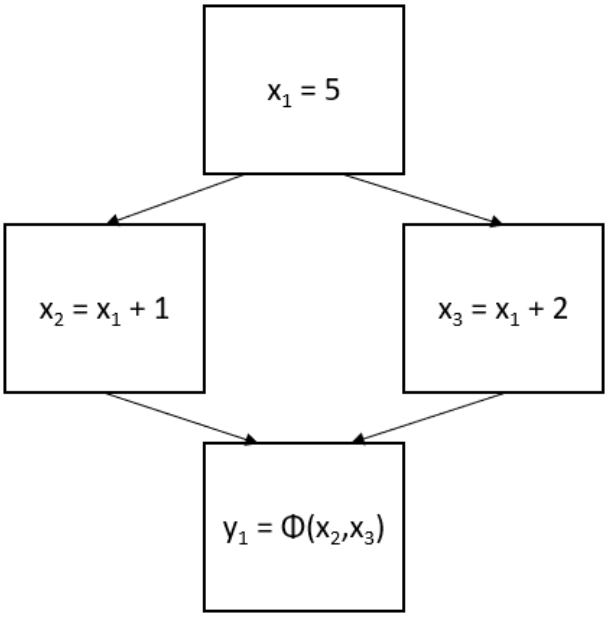


Figura 3.2: Exemplo de um CFG na forma SSA

O último bloco presente na Figura 3.2 contém uma instrução *PHI*, quer isto dizer, que o valor para a variável y tanto pode ser x_2 ou x_3 dependendo do caminho que a execução percorreu. Se x_2 e x_3 forem duas instruções de *PUSH* então a instrução *PHI* terá dois valores concretos como argumentos. No momento em que a execução chega à instrução *PHI* existem dois valores possíveis mas, durante a execução, só um dos valores é que será possível. Para não aumentar o espaço de procura foi decidido que as instruções *PUSH* que estão relacionadas com a instrução *PHI* não são removidas.

Outro problema que a ferramenta Rattle apresentava é que havia uma variável global e quando era invocada esta ferramenta mais do que uma vez na mesma execução, o resultado da segunda invocação não era o desejado. Foi necessário assim tornar esta variável local. Outro problema era a localização dos *JUMPs* que nem sempre era atualizada quando havia otimizações. Se houvesse três blocos se-

[4] `https://eth.wiki/en/concepts/swarm-hash`

guidos em que o do meio só continha uma instrução de *JUMP*, esse bloco era removido. No entanto a localização do *JUMP* não era atualizada. Outro ponto importante foi adicionar à classe *SSAInstruction*, que representa uma instrução na forma SSA, a variável *instruction_offset* que representa o *offset* em *bytes* desde o início das instruções. Isto será útil para fazer o mapeamento da instrução *assembly* para a linha correspondente no código fonte Solidity.

Uma limitação da Rattle é que quando um contrato depende de uma biblioteca, quer esta seja declarada no mesmo ficheiro que o contrato que depende dela ou não, a Rattle não é capaz de fazer a ligação da biblioteca ao contrato.

Algum do trabalho realizado nesta ferramenta foi proposto e aceite pelo autor da ferramenta e está presente publicamente no GitHub desta[5]. O trabalho restante que não foi proposto deve-se ao facto de se achar que iria contra a ideologia da ferramenta, nomeadamente o facto de serem mantidas algumas instruções *PUSH* quando um dos objetivos desta ferramenta é removê-las. Todas as alterações estão públicas num *fork* da ferramenta Rattle no GitHub[6].

3.3 Conkas

Conkas é uma ferramenta de análise estática que usa o modelo de execução simbólica. É uma ferramenta modular, fácil de estender, tanto em adicionar novos módulos para detetar novas vulnerabilidades como em adicionar novas instruções EVM que possam surgir e é compatível com todas as versões da Solidity até à data de escrita deste relatório (v0.6.7). Esta ferramenta estará disponível publicamente no GitHub [7].

3.3.1 Arquitetura da Conkas

Conkas usa uma arquitetura modular e esta pode ser observada na Figura 3.3.

Conkas disponibiliza uma *Command-Line Interface* (CLI) onde os utilizadores podem interagir com a ferramenta. Pode ser fornecido o *bytecode* associado ao contrato que se pretende analisar ou o código fonte Solidity. Quando é fornecido o código fonte, este será compilado usando a versão do compilador que mais se adeque. De momento só é possível fornecer o código fonte de contratos escritos em Solidity mas pode ser estendido para outras linguagens que tenham um compilador para a EVM, uma vez que a Conkas é agnóstica à linguagem de programação. O *bytecode* será posteriormente passado ao módulo *Rattle+*.

No módulo *Rattle+* é realizada a elevação do *bytecode* para a representação intermédia usada na ferramenta Rattle. As instruções são passadas para a forma SSA e são transformadas da forma *stack* para a forma de registos. Este resultado é passado ao módulo seguinte, *Motor de Execução Simbólica*.

O módulo *Motor de Execução Simbólica* é responsável por percorrer o CFG e gerar traços de execução. Um traço representa um caminho possível de ser executado e contém informação sobre o

[5] https://github.com/crytic/rattle
[6] https://github.com/nveloso/rattle
[7] https://github.com/nveloso/conkas

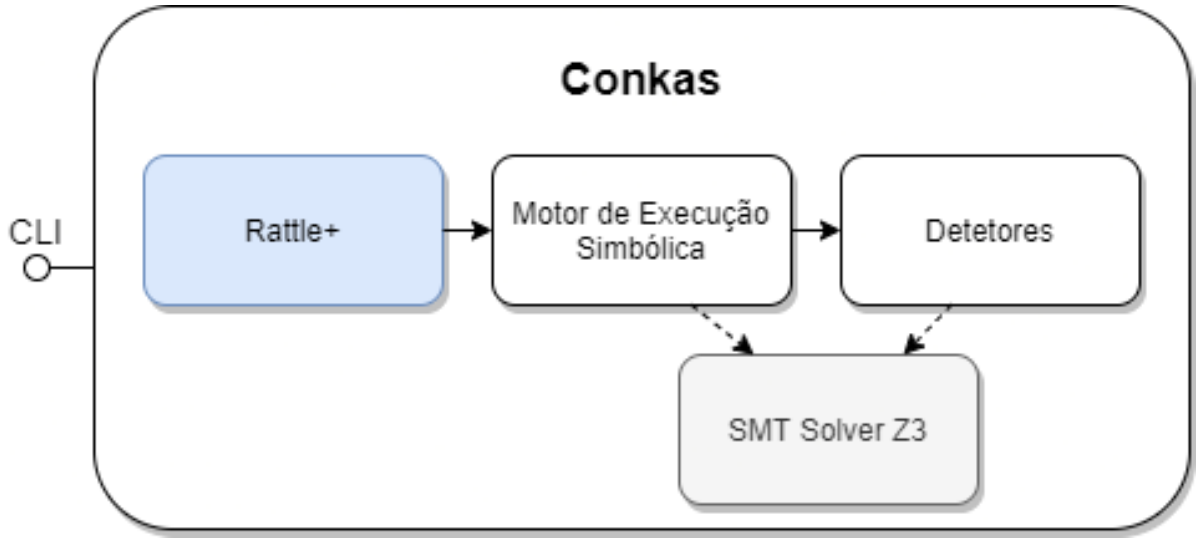


Figura 3.3: Arquitetura da ferramenta Conkas. O módulo Rattle+ (a azul) é a modificação ao módulo original Rattle. A cinzento é o módulo Z3 (já existente).

estado do registo, da *memory*, *storage*, valor de retorno, a profundidade atingida, as restrições para percorrer este caminho, o ambiente em que este caminho foi percorrido e informação sobre se a transação foi revertida, se foi parada, se o contrato foi destruído ou se houve uma instrução inválida.

Quando um bloco que termina num *JUMP* condicional dá origem a dois caminhos diferentes. O *Motor de Execução Simbólica* irá percorrer sempre primeiro o caminho em que a condição do *JUMP* seja falsa. Quando existem ciclos em que a condição de paragem é simbólica, por exemplo, $x < 5$ e x é uma variável sem sinal, e em todas as iterações é somado 1 a essa variável, a condição de paragem será atualizada a cada ciclo até que um dos caminhos seja impossível. No caso de a condição de paragem conter duas variáveis simbólicas, por exemplo $x < y$, e em todas as iterações é somado 1 à variável x, a condição de paragem será atualizada a cada ciclo. No entanto se as restrições impostas a essas variáveis forem nulas antes de se iniciar o ciclo, a execução deste módulo poderá não terminar pois irá estar a aumentar o espaço de procura sem um limite definido. Para resolver este problema, existe um parâmetro, *profundidade*, para limitar o espaço de procura podendo este ser alterado através da CLI.

Os blocos resultantes da construção do CFG podem conter uma ou mais instruções. Um bloco é terminado quando existe uma instrução de *JUMP*, *JUMPI* ou uma instrução terminal como *RETURN*, *STOP*, etc. O conceito de profundidade corresponde ao número de blocos que foram analisados.

Os traços resultantes do módulo *Motor de Execução Simbólica* são depois passados ao último módulo, *Detetores*, e é neste que será realizada a deteção ou não de possíveis vulnerabilidades. Este módulo irá iterar sobre todos os sub módulos que estiverem registados, em que cada sub módulo é responsável por detetar um tipo de vulnerabilidade. O módulo *Detetores* produz uma lista de vulnerabilidades que serão apresentadas ao utilizador. Neste módulo é feita uma tentativa de mapear o identificador da função Solidity (4 primeiros bytes do *hash*) para a assinatura da função Solidity. Quando é fornecido o código fonte Solidity é feito o mapeamento da instrução *assembly* para o número da linha no código fonte Solidity onde essa vulnerabilidade ocorre. No Algoritmo 1 é apresentado, de forma abstrata, o

funcionamento da Conkas e a separação de cada um dos módulos apresentados anteriormente.

Algoritmo 1: Algoritmo abstrato da Conkas

Result: Vulnerabilidades encontradas impressas ao utilizador

```
1 bytecodes = process_user_input() ;                // one file can have several contracts
2 for bytecode in bytecodes do
3 |   ssa = Rattle+.process(bytecode);
4 |   traces = SymExec(ssa).execute();
5 |   vulns = Detetors(traces).analyse();
6 |   print(vulns);
7 end
```

Os módulos *Motor de Execução Simbólica* e *Detetores* recorrem a um outro módulo que é o *SMT Solver Z3*. Estes módulos criam restrições simbólicas com auxílio do *Z3* e questionam este para resolver essas restrições e, no caso do módulo *Motor de Execução Simbólica*, são introduzidas variáveis simbólicas com auxílio do *Z3*.

3.3.2 Modelação das Instruções e dos Blocos

O *bytecode* na primeira fase é traduzido para instruções EVM. Estas instruções EVM são modeladas por uma dependência que contém várias informações tais como o número de bytes que a instrução ocupa, se a instrução interage com a *stack*, entre outras informações. Mais tarde, a Rattle+ converte as instruções da forma *stack* para a forma de registos e traduz para a forma SSA. Neste momento, estas instruções na forma SSA são modeladas usando a classe *SSAInstruction*. Nesta existem informações tais como uma referência para uma instrução EVM, em que bloco é que esta instrução se encontra, quantos argumentos e quais esta instrução contém, uma outra referência para o valor de retorno, entre outras. Ainda na Rattle+, os blocos são modelados através da classe *SSABasicBlock*. Nesta existem informações como uma lista de *SSAInstructions* que indica as instruções que pertencem a um bloco, um *set* de *SSABasicBlock* que indica quais os blocos que fazem parte do caminho de um bloco e que deram origem a esse bloco, um *set* de *SSABasicBlock* que indica para quais os blocos que um bloco possa saltar, uma variável com uma referência para um *SSABasicBlock* que indica o bloco para o qual a execução prosseguirá no caso de um bloco terminar com um salto condicional e a condição for falsa, entre outras.

3.3.3 Motor de Execução Simbólica

O Motor de Execução Simbólica começa por criar um ambiente que tenta simular o ambiente de execução da EVM. No entanto, muita informação não está disponível sem haver uma consulta à blockchain. Este Motor de Execução Simbólica não faz nenhuma consulta mas se no futuro houver essa necessidade não será algo difícil de acrescentar.

Ao receber o *bytecode* modelado pela Rattle+, começa por criar um traço de execução. Cada traço corresponde a uma possível transação. Este traço é modelado com a classe *Trace*. Nesta classe está

presente um objeto do tipo *State* que indica o estado da execução desse traço. Na classe *State* estão os objetos que modelam os registos, a *memory* e o *storage*. É possível também saber em que estado terminou este traço, por exemplo se reverteu ou se foi retornado algo. Na classe *Trace* para além do objeto *State*, estão os blocos que já foram analisados, qual o bloco que está para ser analisado, o ambiente em que é executado, a profundidade máxima que pode atingir e uma lista de restrições simbólicas impostas pela execução simbólica das instruções presentes na EVM. Na Figura 3.4 é apresentado o diagrama da classe *Trace* bem como o da classe *State*.

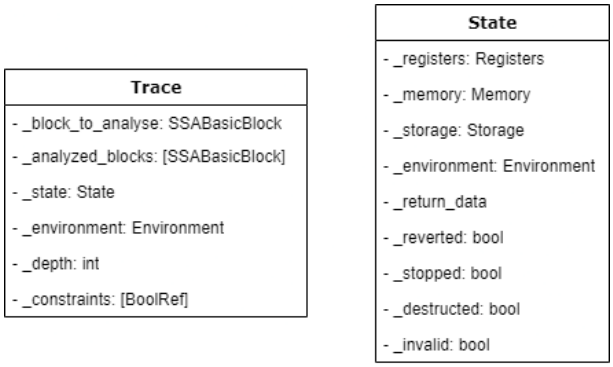


Figura 3.4: Diagrama da classe *Trace* e da classe *State*

O objetivo final do Motor de Execução Simbólica é criar uma lista de traços de execução que sejam possíveis de percorrer. Para atingir este objetivo começa por analisar o primeiro traço que contém o primeiro bloco a ser analisado. Para analisar o bloco, este irá executar simbolicamente todas as instruções presente nesse bloco. Depois de as analisar, é retornada uma lista com os possíveis blocos para onde a execução pode saltar. O primeiro bloco dessa lista continua a ser analisado no traço que está a ser analisado. No caso de haver mais do que um bloco para onde a execução possa saltar, para cada um dos restantes blocos serão criadas réplicas de traços iguais ao traço a ser analisado e serão analisados quando o traço a analisar não contiver mais blocos ou a profundidade máxima seja atingida. É garantido que quando existe mais do que um bloco nessa lista, o primeiro bloco dessa lista corresponde ao caminho em que a condição do salto seja falsa. No final de cada análise de um bloco é somado 1 à profundidade e caso esta seja maior ou igual à profundidade máxima definida, o traço a ser analisado chegará ao fim da sua análise e outro traço cuja análise esteja pendente será iniciada. No final haverá um conjunto de traços em que cada traço tem um conjunto de restrições impostas pela execução simbólica das instruções presentes na EVM. No Algoritmo 2 são apresentados os passos necessários para obter novos traços de execução simbólica que resultam da análise de um conjunto de traços. Estes novos traços serão executados quando o traço em causa estiver totalmente analisado.

3.3.3.1 Execução Simbólica das Instruções EVM

Cada instrução presente na EVM é executada simbolicamente pelo Motor de Execução Simbólica. Existem instruções de comparação, por exemplo a instrução *LT* que verifica se o primeiro operando é menor do que o segundo operando, instruções aritméticas, por exemplo *ADD* que soma dois operandos, instruções do contexto da blockchain, por exemplo *TIMESTAMP* que indica o *timestamp* presente no

Algoritmo 2: Algoritmo da geração de traços de execução simbólica

```
Result: Traços de execução simbólica
new_traces = [];
for trace in traces do
    while trace.block_to_analyse do
        new_blocks = sym_exec(trace.block_to_analyse, state);
        trace.add_analysed_block(trace.block_to_analyse, trace.constraints);
        trace.depth += 1;
        if trace.depth >= MAX_DEPTH then
            break;
        end
        for new_block in new_blocks[1:] do
            new_trace = copy(trace);
            new_trace.block_to_analyse = new_block;
            new_traces.append(new_trace);
        end
        trace.block_to_analyse = new_blocks[0];
    end
    return new_traces;
end
```

bloco, instruções de execução, por exemplo *JUMP* que muda a execução para a localização presente no primeiro operando, entre outras. No Apêndice A.1 são apresentados todos os nomes das instruções que a Conkas executa simbolicamente.

Para todas as instruções do contexto da blockchain, ou instruções que necessitem de informações do contrato, por exemplo *CALLVALUE* que indica o valor de Ether com que uma determinada função foi invocada, ou qualquer outra instrução em que o seu valor é desconhecido, é criada uma variável simbólica com auxílio do *Z3*. Essa variável terá um nome, que será o nome da instrução e um tamanho, que será 32 bytes (256 bits). Estas variáveis são consideradas *taint*, o que significa que são variáveis que podem ser controladas por um atacante. A execução de todas as instruções presentes num traço de execução faz propagar essas variáveis *taint* que mais tarde, nos sub módulos do módulo Detetores, são verificadas com auxílio ao *Z3*. Esta propagação acontece pelas expressões simbólicas criadas ao longo da execução simbólica das várias instruções. Estas expressões simbólicas propagam as variáveis *taint* porque são expressões em que têm como argumentos essas variáveis. Na Listagem 19 é apresentado um exemplo de uma função existente na Conkas que executa simbolicamente a instrução *TIMESTAMP*. Começa por garantir que esta instrução tem um valor de retorno e cria uma variável simbólica com auxílio do Z3. Coloca no registo a variável simbólica no índice dado pelo valor de retorno da instrução.

```
1  def inst_timestamp(instruction: SSAInstruction, state: State):
2      rv = instruction.return_value
3      if rv is None:
4          logger.error('TIMESTAMP instruction needs return value')
5          raise Exception
6
7      rv = rv.value
8      bv = BitVec('timestamp', WORD_SIZE)
9      state.registers.set(rv, bv)
```

Listagem 19: Exemplo de uma função que executa simbolicamente a instrução *TIMESTAMP*

Quando são executadas as instruções aritméticas, é verificado se todos os operandos são valores concretos e se forem a execução é idêntica à da EVM. No caso de haver um operando com um valor simbólico, a operação realizada pela execução dessa instrução será a criação de uma expressão simbólica com o auxílio do *Z3*. Nas instruções de comparação é verificado também se todos os operandos são valores concretos e se forem a execução é idêntica à da EVM. No caso de haver um operando com um valor simbólico é criada também uma expressão simbólica com auxílio do Z3. Essa expressão simbólica é uma condição simbólica, em que essa condição é a própria comparação. No caso da condição ser verdadeira o valor assumido será o valor 1, no caso contrário é assumido o valor 0. Nas instruções de execução, nomeadamente na instrução de salto condicional, a condição imposta é verificada se é simbólica e no caso de ser é retornada uma lista de tuplos. Cada tuplo contém o bloco para onde é possível saltar e é criada uma restrição, com auxílio do Z3, em que só é possível executar esse bloco se essa restrição for satisfeita.

3.3.4 Vulnerabilidades Detetadas

No módulo Detetores existem atualmente 5 sub módulos em que cada um é responsável por detetar uma vulnerabilidade presente no DASP Top 10. As vulnerabilidades detetadas são: DASP1 - Reentrância, DASP3 - Aritmética, DASP4 - Não Verificação do Valor de Retorno de Chamadas de Baixo Nível, DASP7 - Dependência da Ordem das Transações (*Front Running*) e DASP8 - Manipulação do Tempo. Foi estipulado inicialmente que a Conkas teria de detetar 5 categorias das vulnerabilidades presente no DASP Top 10. Estes tipos de vulnerabilidades foram escolhidos por serem as vulnerabilidades mais detetadas pelas ferramentas atuais [6].

Reentrância A política usada para detetar vulnerabilidades de reentrância passa por verificar traços que contenham a instrução *CALL*. Quando é detetada essa instrução é necessário verificar se existe uma guarda antes da instrução *CALL*. Considera-se uma guarda, uma variável booleana que num determinado valor permite a execução da função e no valor oposto proíbe a sua execução. Para verificar se existe uma guarda é necessário verificar se a instrução *CALL* pode ser invocada novamente e por isso é preciso verificar se a restrição imposta para a instrução *CALL* ser executada pode ser satisfeita novamente nas condições em que a instrução *CALL* está a ser executada. Se for detetada a guarda não existe vulnerabilidade. No caso de não haver é necessário verificar se o valor de Ether presente nessa instrução é igual a 0. Se for 0 então não haverá vulnerabilidade, no caso de ser diferente de 0 haverá uma vulnerabilidade. No entanto esta política não é suficiente. Considere o exemplo apresentado na Listagem 20, um contrato com uma função que não é reentrante mas, que no entanto, segundo a política apresentada anteriormente, seria considerado como reentrante.

```
1 contract NonReentrant {
2     mapping (address => uint256) balanceOf;
```

```
3       uint256 private counter = 1;
4
5       function nonReentrant() public payable {
6           counter += 1;
7           uint256 guard = counter;
8           uint256 amount = balanceOf[msg.sender];
9           bool success = msg.sender.call.value(amount)("");
10          require(success);
11          balanceOf[msg.sender] = 0;
12
13          require(guard == counter);
14      }
15  }
```

Listagem 20: Exemplo de um contrato não reentrante

A função *nonReentrant()* não tem nenhuma guarda e o valor de Ether presente na variável do *storage* (*balanceOf[msg.sender]*) no momento da instrução *CALL* é diferente de zero. A política anterior iria considerar uma vulnerabilidade no entanto não existe. Cada vez que a função é chamada é incrementada a variável *counter* presente no *storage* e cada execução da função guarda na sua *stack* o valor que leu do *storage*. No final o valor guardado na *stack* é comparado com o valor guardado no *storage* e estes precisam de ser iguais para a função não reverter. Se a função *nonReentrant()* for invocada e durante essa invocação a mesma função seja chamada outra vez, esta última irá passar a última condição, no entanto a execução da primeira invocação irá reverter porque a última condição é falsa devido ao valor do *storage* ter um valor mais alto pois foi incrementado pela segunda invocação.

A política apresentada anteriormente iria considerar a função do contrato na Listagem 20 como vulnerável à reentrância, sendo necessário então tornar esta política mais forte, no sentido de diminuir os falsos positivos. Para tentar perceber se a técnica apresentada na Listagem 20 é usada, é necessário verificar que exista uma restrição em que os operandos desta restrição sejam iguais, mas que um seja uma variável da *stack* e o outro seja uma variável do *storage*. Esta restrição quando não é satisfeita tem de originar a reversão da transação.

A política usada neste sub módulo responsável por detetar vulnerabilidades de reentrância não é exatamente igual à descrita no parágrafo anterior porque quando é obtida uma restrição e obtidas as variáveis dessa restrição, as variáveis não têm a informação se pertencem à *stack*, podendo uma variável da *stack* ser confundida com uma variável do *storage*. Não é possível também verificar que a restrição, quando não satisfeita, leve a transação a reverter devido ao facto de não ser possível partilhar estado da análise entre traços. Devido a estes dois problemas irão haver falsos positivos, no entanto os resultados são mais satisfatórios do que o das outras ferramentas, ver Capítulo 4.

Este módulo usa o módulo *Z3* para verificar se as restrições que existam até ao bloco que contém a instrução *CALL* possam ser satisfeitas e se as restrições criadas por este módulo também possam ser satisfeitas. Quando é encontrada a instrução *CALL* são verificadas todas as restrições impostas até esse bloco, com auxílio do *Z3*. Nessas restrições é verificado se existe alguma variável que seja uma variável do *storage*, dado que para haver uma guarda essa guarda terá de usar uma variável do

storage. É adicionada uma restrição ao *Z3* que se for satisfeita indicará que não existe uma guarda e portanto existirá uma vulnerabilidade. Essa restrição consiste em que o conteúdo do *storage* no momento da instrução *CALL* seja igual ao conteúdo do *storage* no momento da restrição. Por outro lado, é necessário verificar também a variável do *storage* que guarda o valor de Ether a transferir. É obtida essa variável e é verificado, com o *Z3*, se essa variável pode ser diferente de 0. Se o *Z3* conseguir satisfazer essa restrição então também existe uma vulnerabilidade.

Aritmética O detetor de vulnerabilidades aritméticas é capaz de detetar *integer overlfow* e *underflow*. Os *overflows* são detetados nas instruções *ADD* e *MUL*. A política para detetar na instrução de *ADD* é verificar se o resultado dessa operação é maior do que o primeiro operando. Para a instrução *MUL* é dividido o resultado desta instrução pelo primeiro operando e o resultado desta divisão tem de ser igual ao segundo operando. O primeiro operando tem de ser diferente de zero, caso contrário não existe vulnerabilidade. Os *underflows* são detetados na instrução *SUB*. A política para detetar é verificar se o segundo operando é maior do que o primeiro operando.

Quando existem variáveis do *storage* com visibilidade pública, a Solidity cria uma função para o valor dessa variável ser retornado ao utilizador. Em alguns dos tipos que essa variável pode tomar, o *bytecode* associado a esta função contém *underflows* propositados. Estes *underflows* ocorrem propositadamente para verificar se um determinado *bit* está ativo ou não [8]. Este bit serve para indicar se o conteúdo é menor ou igual a 31 bytes. No caso de ser maior que 31 bytes o conteúdo está num *slot* diferente de onde está a informação correspondente ao *length* do tipo usado. Se for menor ou igual a 31 bytes o conteúdo está no mesmo *slot* que a informação do *length* do tipo usado. Para não detetar estes *underflows* foi criada uma heurística que consiste em verificar se existe uma instrução de *MUL* ou *EXP*, em que o primeiro argumento é o valor 0x100 (256) seguido de uma instrução *SUB* em que o segundo argumento é o valor 1. No entanto esta heurística não é suficiente, ver Capítulo 4.

Para detetar os *overflows*, este módulo percorre todas as instruções de *ADD* e *MUL*. Para as instruções de *ADD* é criada uma restrição em que o resultado dessa instrução tem de ser menor do que o primeiro operando. É passada essa restrição juntamente com todas as restrições impostas pelo Motor de Execução Simbólica até essa instrução ao módulo *Z3* e este irá verificar se estas restrições são possíveis de satisfazer. Se for possível de satisfazer indica que existe um *overflow*. Para as instruções *MUL* é usada uma função do *Z3* que cria uma restrição, com base nos operandos da instrução *MUL*. Esta restrição em conjunto com todas as restrições impostas até esta instrução são passadas ao módulo *Z3* para este verificar se estas restrições são satisfazíveis. No caso de serem satisfazíveis então indica que existe um *overflow*.

Para detetar os *underflows*, este módulo percorre todas as instruções *SUB*. Para cada uma destas instruções é criada uma restrição em que o segundo operando tem de ser maior do que o primeiro operando. Esta restrição em conjunto com todas as instruções impostas pelo Motor de Execução Simbólica são passadas ao módulo *Z3*. Este irá verificar se as restrições são satisfazíveis e em caso afirmativo indica que existe um *underflow*.

[8] `https://solidity.readthedocs.io/en/v0.6.6/miscellaneous.html#bytes-and-string`

Não Verificação do Valor de Retorno de Chamadas de Baixo Nível Um outro tipo de vulnerabilidades que são detetadas são as de Não Verificação do Valor de Retorno de Chamadas de Baixo Nível. Esta é das políticas mais básicas uma vez que só é necessário verificar se o resultado da execução da instrução *CALL* é usado em alguma restrição a partir do momento em que essa instrução é executada.

Este módulo percorre todas as instruções que possam invocar outro contrato. Para cada uma dessas instruções é verificado se o valor de retorno de uma instrução está presente nas restrições que foram impostas a seguir à execução do bloco que contenha essa instrução.

Dependência da Ordem das Transações (*Front Running*) A deteção das vulnerabilidades de *Front Running* são difíceis de identificar pois existem vários sub tipos muito específicos. Um exemplo é quando um parâmetro de uma função é usado para gerar o *sha3* e realizada uma comparação desse valor com um valor definido no *storage*, e após suceder deve haver uma instrução *CALL*. Com este exemplo um utilizador pode ser recompensado quando descobre uma solução para um puzzle. Existem vários fatores que tornam difícil de detetar, por exemplo uma função ser chamada várias vezes e com isso um utilizador tirar benefício pela sua transação ser a segunda ou terceira a ser executada. É difícil de perceber quando é que um utilizador pode beneficiar com a ordem das transações. Assim, a política desenvolvida apenas considera um caso específico. Este caso é quando existem duas funções, em que numa função é usada uma variável do *storage* numa instrução *CALL* e posteriormente essa variável é alterada. A outra função apenas faz uso dessa variável do *storage* numa instrução *CALL*. Uma forma de melhorar esta política passaria por modular as transações e assim perceber se um utilizador poderia receber um valor de Ether diferente se uma transação fosse executada primeiro ou depois.

Este módulo percorre todas as instruções que possam enviar dinheiro (*CALL* e *CALLCODE*). Para cada instrução é verificado o argumento que indica o valor de Ether a transferir. São guardadas as variáveis que são do *storage*, separadas por cada traço de execução. Mais tarde, as variáveis de um traço são comparadas com outras variáveis de outro traço. Se for possível uma variável de um traço ser diferente de outra variável de outro traço, então indica que existe uma vulnerabilidade.

Manipulação do Tempo A política usada para detetar as vulnerabilidades de Manipulação do Tempo é verificar se existem restrições na qual exista um operando que contenha uma variável que seja baseada no tempo. As variáveis que são baseadas no tempo são variáveis introduzidas pela execução da instrução *TIMESTAMP*. Deve ser também verificado o valor de retorno da função, a afetação de variáveis do *storage* e se a instrução *SHA3* usa o tempo como argumento. Estes são os pontos críticos onde uma vulnerabilidade deste tipo pode acontecer. Quando a vulnerabilidade está no valor de retorno, a Conkas não diz a linha correta, ou seja, a linha onde acontece o retorno da função, mas sim a linha onde a função é declarada. Isto acontece porque o mapeamento fornecido pelo compilador mapeia a última instrução para a linha onde a função é declarada.

Este módulo percorre todas as restrições de todos os traços e procura por variáveis simbólicas que sejam baseadas no tempo. São também procuradas variáveis simbólicas que sejam baseadas no tempo no valor de retorno presente em cada traço se houver, no conteúdo que é escrito no *storage* ou

no conteúdo da *memory* que servirá como argumento à instrução *SHA3*.

3.3.5 Modelo de Memória

Na EVM existem 3 tipos de memória que são a *stack*, um tipo de memória volátil em que são guardadas variáveis das funções, a *memory*, outro tipo de memória volátil que guarda os argumentos das funções e o *storage*, sendo este um tipo de memória persistente que guarda variáveis declaradas fora das funções. Estes tipos de memória foram discutidos anteriormente na Secção 2.3.

Na Conkas não existe o conceito de *stack* uma vez que a Rattle+ converte as instruções da forma *stack* para a forma de registos. Sendo assim, a Conkas modela os registos com o auxílio de um dicionário. O índice deste é um valor único criado pela Rattle+, uma vez que as instruções estão na forma SSA e cada instrução tem um valor único associado. Desta forma, nunca haverá um índice que seja simbólico. No entanto o dicionário pode armazenar para um dado índice um valor concreto ou um valor simbólico.

Em relação ao tipo de memória *memory* e *storage*, estas são modeladas de forma idêntica à da EVM quando os seus índices e tamanhos são valores concretos. A modelação do *storage* é a mais simples uma vez que todos os acessos e escritas são feitos alinhados a 32 bytes. Assim sendo, é modelado com o auxílio de um dicionário em que a chave corresponde ao índice deste tipo de memória e o valor é um *array*. Cada posição do *array* terá o valor que se pretende escrever no *storage*. É usado um *array* para manter informação dos valores escritos anteriormente para o mesmo índice, tornando assim também o *storage* na forma SSA. O valor a ser escrito pode ser um valor concreto ou um valor simbólico. Em relação à *memory* o procedimento é idêntico. É modelado usando um dicionário em que cada índice corresponde ao *offset* desde o início da *memory*. Uma vez que a *memory* pode ser acedida num *offset* aleatório com um tamanho aleatório, o valor deste dicionário será um valor de apenas 1 *byte*. No entanto este valor de 1 *byte* estará presente num *array* pela mesma razão do *storage*, para manter informação dos valores armazenados anteriormente. Os valores a armazenar são divididos *byte* a *byte* quer estes sejam simbólicos ou concretos. Os modelos de memória *memory* e *storage* usam um *array* para armazenar o histórico de valores pois o módulo Detetores pode necessitar dessa informação para determinar se existe alguma vulnerabilidade ou não.

3.3.5.1 Acesso Simbólico à Memória

Um problema da execução simbólica é quando existe acesso a uma estrutura de dados em que os índices e/ou os tamanhos são simbólicos. Na EVM existem instruções que podem ler ou escrever na *memory* e recebem como argumento o índice e o tamanho onde será realizada a leitura ou escrita. Se o índice e/ou o tamanho forem simbólicos, estas leituras ou escritas tornam-se complicadas de serem realizadas. Em ambos os casos a ferramenta irá aproximar a solução. Se o tamanho é simbólico, a instrução retorna uma nova variável simbólica, tal como em [20]. Se o tamanho é um valor concreto, mas o índice é simbólico então o valor lido ou armazenado será dividido *byte* a *byte* e o índice usado será a variável simbólica somando 1 até satisfazer o tamanho.

Todos os acessos e escritas à *storage* têm tamanho 32 bytes. Assim sendo, o único caso necessário de tratar é quando o índice da leitura ou escrita no *storage* é simbólico. Quando este índice é uma variável simbólica, esta variável servirá como chave ao dicionário.

3.3.6 Adicionar Novas Vulnerabilidades

A Conkas foi desenhada para um programador com conhecimentos básicos de programação poder adicionar facilmente novos módulos para detetar novas vulnerabilidades. Assim, o programador apenas necessita de adicionar ao ficheiro __init__.py dentro da pasta vuln_finder uma entrada ao objeto *available_modules*, sendo a sua chave o nome do módulo, que será apresentado na mensagem de ajuda da CLI, e o valor será a função que será chamada para detetar essa vulnerabilidade. Este objeto pode ser observado na Listagem 21.

```
1 from vuln_finder.reentrancy import reentrancy_analyse
2 ...
3 from vuln_finder.vuln_name import vuln_x_analyse
4
5 available_modules = {
6     'reentrancy': reentrancy_analyse,
7     ...,
8     'vuln_name': vuln_x_analyse
9 }
```

Listagem 21: Objeto com as funções de análise

A função a ser adicionada tem de seguir a assinatura presente na Listagem 22: recebe no primeiro parâmetro uma lista de *Trace* e o segundo parâmetro é uma variável booleana que quando verdadeira indica para detetar todas as vulnerabilidades presentes, ou seja, não deve ser abortada assim que encontrar uma vulnerabilidade. Como retorno deve devolver uma lista de *Vulnerability*.

```
1 def vuln_x_analyse(traces: [Trace], find_all: bool) -> [Vulnerability]:
2     pass
```

Listagem 22: Assinatura da função para adicionar novos módulos

3.3.7 Adicionar Novas Instruções EVM

O desenho da Conkas permite de forma idêntica e fácil adicionar novas instruções EVM que surjam com as atualizações que possam existir no futuro. Assim, será relativamente fácil manter a ferramenta atualizada. O processo para adicionar novas instruções é semelhante ao apresentado na Subsecção 3.3.6. O objeto para adicionar a função está localizado no ficheiro __init__.py na diretoria *sym_exec/instruction/*.

A assinatura da função a adicionar que irá executar a instrução de forma simbólica deve receber dois parâmetros: o primeiro é uma instância de *SSAInstruction* e o segundo uma instância de *State* e pode retornar uma lista de instâncias do tipo *SSABasicBlock*. Na Listagem 23 é apresentada a assinatura desta função.

```
1 def inst_x(instruction: SSAInstruction, state: State) -> [SSABasicBlock]:
2     pass
```

Listagem 23: Assinatura da função para adicionar novas instruções

Na Listagem 19, apresentada anteriormente, está presente um exemplo de uma função existente na Conkas que executa simbolicamente uma instrução EVM.

3.3.8 Testes Unitários

Os testes unitários dão alguma garantia quando existem mudanças no código. Foram implementados vários testes unitários para todas as instruções EVM moduladas na Conkas. Para cada instrução EVM existem em média 2 testes, havendo no total 194 testes unitários.

Para além de testes unitários foram criados 14 contratos escritos em Solidity para testar a ferramenta Rattle+ com as modificações realizadas e descritas na Secção 3.2. Estes testes foram verificados manualmente, um a um, para perceber se a ferramenta Rattle+ não gerava erros e era capaz de apresentar o resultado esperado, tornando-a assim mais robusta.

3.3.9 *Command-Line Interface* (CLI)

A Conkas disponibiliza uma *Comand-Line Interface* (CLI) que permite aos utilizadores interagirem com a mesma. Os utilizadores devem fornecer um ficheiro com o *bytecode* do contrato a ser analisado ou com o código fonte escrito em Solidity. É possível também especificar quais as vulnerabilidades que se pretendem detetar assim como a opção de abortar a deteção após a identificação da primeira vulnerabilidade. Os utilizadores podem também definir o nível de *logging* assim como a profundidade máxima para limitar a procura. Por fim, podem ainda fornecer o valor para o *timeout* que será usado no Z3, sempre que este for invocado para verificar se as restrições são satisfazíveis ou não. Para a Conkas analisar apenas um tipo de vulnerabilidade, os utilizadores podem invocar a Conkas passando o nome da vulnerabilidade que pretendem que a Conkas analise. Esses nomes e todas as informações descritas anteriormente podem ser consultadas invocando o seguinte comando:

```
python3 conkas.py -h
```

Um exemplo de um comando para analisar o *bytecode* presente no ficheiro *test.bin* e analisar todas as vulnerabilidades, é o seguinte:

```
python3 conkas.py test.bin
```

Caso o ficheiro contenha o código fonte escrito em Solidity e o utilizador queira apenas analisar vulnerabilidades de reentrância, o comando deve ser o seguinte:

```
python3 conkas.py -vt reentrancy -s test.sol
```

Na tabela 3.1 é possível observar todos os argumentos opcionais que a CLI da Conkas fornece e para cada um a sua descrição.

Argumentos Opcionais	Descrição
–help, -h	Mostra uma mensagem com todas as opções disponíveis
–solidity-file, -s	Indica que o utilizador pretende analisar o ficheiro como código fonte Solidity
–verbosity, -v	Nível de *log*
–vuln-type, -vt	Procura apenas vulnerabilidades específicas podendo ser indicada uma ou mais vulnerabilidades
–max-depth, -md	Valor máximo para a profundidade
–find-all-vulnerabilities, -fav	Indica que o utilizador pretende procurar por todas as vulnerabilidades
–timeout, -t	Timeout para usar no módulo Z3 quando é necessário verificar restrições

Tabela 3.1: Descrição dos argumentos opcionais disponíveis na CLI da Conkas

3.3.10 Requisitos do Sistema

A Conkas está escrita em Python3 sendo, portanto necessário uma versão do Python3 com os seguintes módulos: *cbor2*, *py-solc-x*, *pycryptodome*, *pyevmasm*, *solidity_parser* e *z3-solver*. Existe também uma imagem disponível para ser executada no Docker. Esta imagem existe porque uma das frameworks usada na análise dos resultados da Conkas necessita que a Conkas tenha uma imagem Docker.

3.3.11 Limitação da Conkas

A Conkas apresenta uma limitação em que não é capaz de analisar contratos que dependam de bibliotecas mesmo que estas estejam declaradas no mesmo ficheiro que o contrato a ser analisado, abortando com erro. Esta limitação advém de uma limitação da Rattle que não é capaz de fazer esta ligação, ver Secção 3.2.

As razões para não detetar outros tipos de vulnerabilidades como por exemplo vulnerabilidades de controlo de acesso deve-se ao facto de a Conkas não ter informação da visibilidade das funções ou não ter acesso ao código do construtor do contrato. Desta forma poderia ter informação de quais as variáveis que só deveriam de ser acedidas por certos utilizadores. Uma outra razão para não detetar vulnerabilidades do tipo negação de serviço é que estas podem ocorrer de várias formas por exemplo, exceções inesperadas serem lançadas ou explorar outro tipo de vulnerabilidade como as de controlo de acesso de forma a aceder e/ou alterar variáveis que só deveriam ser acedidas por alguns utilizadores.

Capítulo 4

Resultados

Neste Capítulo são apresentados e discutidos os resultados obtidos da comparação da ferramenta apresentada no Capítulo 3 com outras ferramentas de análise estática. Muitas destas ferramentas de análise estática foram apresentadas na Secção 2.6.

Existem diversas formas de analisar a ferramenta desenvolvida e apresentada no Capítulo 3. Uma das formas é criar um *dataset* com vários contratos, podendo ser obtidos contratos reais, através do Etherscan[1] por exemplo, ou criando contratos propositadamente vulneráveis. Usando este método, os contratos escolhidos ou criados podem beneficiar a ferramenta desenvolvida, ou seja, podem ser criados contratos vulneráveis em que os autores sabem que a ferramenta deteta essas vulnerabilidades. Outro método é usar o mesmo *dataset* que foi usado na análise feita pelos autores das diferentes ferramentas que se queira comparar, no entanto estes *datasets* não estão públicos. Ainda há um terceiro método que consiste em usar um *dataset* público e em comparar todas as ferramentas usando esse mesmo *dataset*. Assim, em vez de criar um *dataset* que pudesse beneficiar a ferramenta desenvolvida (1ª opção) ou em vez de fazer como em Pérez and Livshits [29], em que para analisar as várias ferramentas existentes, os autores tiveram de contactar os autores das diferentes ferramentas para tentar obter os *datasets* que foram usados na análise de cada uma dessas ferramentas (2ª opção), o que será feito nesta dissertação é usar o terceiro método descrito, usar um *dataset* público.

Existem duas frameworks, do nosso conhecimento, que fornecem um *dataset* para as ferramentas poderem comparar os seus resultados com outras ferramentas de forma automática. Em Ghaleb and Pattabiraman [30] é apresentada a ferramenta SolidiFI, sendo baseada na injeção de bugs em todas as potenciais localizações que resultem em vulnerabilidades. Depois da injeção, todas as ferramentas incluídas nesta framework são executadas e são anotados os falsos positivos e os falsos negativos (bugs que não foram detetados ou que foram mal classificados). Outra framework é apresentada em Ferreira et al. [6], chamada SmartBugs, onde é referido que foi desenhada para facilitar e ajudar na execução e análise das ferramentas. Outra propriedade interessante desta framework é a de ser fácil adicionar novas ferramentas sendo também o *output* de cada ferramenta normalizado, tornando assim a sua análise mais facilitada.

[1] `https://etherscan.io/`

4.1 SmartBugs

A Conkas foi adicionada à SmartBugs, sendo este processo bastante simples. Foi apenas necessário criar uma imagem Docker e seguidamente publicar essa imagem no DockerHub. Depois foi preciso adicionar um ficheiro que realiza a normalização do *output* da Conkas, apresentado no Apêndice A.2. A framework contém 10 ferramentas já prontas a serem utilizadas, em que quase todas elas foram apresentadas na Secção 2.6.

A SmartBugs fornece dois *datasets*, um manualmente anotado com as vulnerabilidades presentes e outro com contratos reais extraídos através do Etherscan. Este último *dataset* é bastante grande e não foi usado para analisar a Conkas pois não fornece nenhuma informação de que as vulnerabilidades detetadas pela Conkas realmente existam. O *dataset* usado foi o manualmente anotado de modo a ter certezas sobre os verdadeiros positivos. Este *dataset* contém vulnerabilidades de todas as categorias apresentadas na Secção 2.5. Na Tabela 4.1 é apresentada a caracterização do *dataset* usado, são apresentados, o número de contratos e o número de ficheiros que existem para uma dada categoria. É também feita uma observação a uma dada categoria presente no *dataset* no caso de existir.

Categoria	Nº de Ficheiros	Nº de Contratos	Observações
Access Control	18	25	-
Arithmetic	15	22	4 vulnerabilidades presentes só no *source code* Solidity e não no *bytecode*
Bad Randomness	8	12	-
Denial of Service	6	6	-
Front Running	4	4	-
Other	3	3	-
Reentrancy	318	578	-
Short Address	1	1	-
Time Manipulation	5	6	-
Unchecked Low Calls	52	93	Existem contratos em que a instrução *CALL* só está presente no *source code* Solidity e não no *bytecode*

Tabela 4.1: Caracterização do *dataset* anotado manualmente presente na SmartBugs

Este *dataset* está dividido em sub *datasets*, sendo que cada *dataset* X apenas tem anotadas as vulnerabilidades da categoria X. No entanto, como discutido mais à frente, alguns destes *datasets* contêm também vulnerabilidades de outras categorias apesar de estas não se encontrarem anotadas. Foi recentemente adicionado à SmartBugs o *dataset* presente na framework SolidiFI mas uma vez que este não está anotado com as vulnerabilidades presentes não se considerou este *dataset* para a análise e comparação da ferramenta Conkas, essa análise encontra-se na Secção 4.2. Todos os ficheiros no *dataset* usado têm o código fonte Solidity associado.

No contexto desta tese foram criados dois contratos com vulnerabilidade de reentrância que tentam usar o padrão mostrado na Listagem 20 mas de forma errada provocando assim uma vulnerabilidade de reentrância. O sub *dataset* da reentrância não continha nenhum exemplo deste género e por isso surgiu a necessidade de criar estes dois contratos.

A SmartBugs não fornece nenhuma forma de comparar os resultados obtidos com as outras ferramentas presentes, no entanto no artigo onde esta é apresentada são mostrados os resultados de cada ferramenta e a comparação com as outras. O *script* usado está presente publicamente no GitHub[2] pelo que foi usado, sendo necessário realizar algumas modificações para adicionar a Conkas, em particular fazer o mapeamento do nome das vulnerabilidades da Conkas para o nome das vulnerabilidades consideradas no SmartBugs-results, modificar a condição para uma vulnerabilidade ser considerada verdadeiro positivo e adicionar código que irá percorrer todas as vulnerabilidades reportadas pela Conkas e verificar se de facto são verdadeiros positivos.

4.1.1 Análise da Conkas

Nesta Subsecção serão apresentados os resultados obtidos através da comparação da ferramenta Conkas com outras ferramentas consideradas estado da arte usando a framework SmartBugs. Será analisada a taxa de verdadeiros positivos desta ferramenta comparativamente às restantes, os falsos positivos detetados pelas ferramentas e reportados pela SmartBugs, a performance da Conkas comparada com as restantes e por fim é apresentada uma sugestão de ferramentas que podem ser combinadas para detetarem um maior número de vulnerabilidades.

4.1.1.1 Taxa de Verdadeiros Positivos

Em Ferreira et al. [6], o *script* usado para gerar a comparação dos resultados obtidos por cada ferramenta só conta uma vulnerabilidade como verdadeiro positivo quando a ferramenta indica a categoria da vulnerabilidade e a linha correta onde ela ocorre no código fonte. Na Tabela 4.2 são apresentados os resultados tendo em conta estes mesmos critérios.

Pelos resultados obtidos é visível que a maioria das ferramentas não deteta muitas vulnerabilidades. Tendo em conta todas as vulnerabilidades detetadas e consideradas verdadeiros positivos pelas ferramentas, a Conkas é a única que fica acima dos 50% e as que ficam mais perto são a Smartcheck e a Slither com 42% e 40% respetivamente. As restantes têm uma percentagem baixa. Após a análise destes resultados conjeturou-se que as ferramentas não devem acertar na linha exata, mas andar numa vizinhança próxima. Esta conjetura deve-se ao facto de uma das dificuldades do desenvolvimento da Conkas ter sido não mostrar a linha exata, mas ficar numa vizinhança próxima, com falha de 2 ou 3 linhas.

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/24 0%	0/24 0%	0/24 0%	0/24 0%	4/24 17%	0/24 0%	0/24 0%	0/24 0%	6/24 25%	2/24 8%	8/24 33%
Arithmetic[3]	19/23 83%	0/23 0%	0/23 0%	1/23 4%	7/23 30%	13/23 57%	16/23 70%	0/23 0%	0/23 0%	1/23 4%	22/23 96%
Denial Service	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%
Front Running	2/7 29%	0/7 0%	0/7 0%	0/7 0%	1/7 14%	0/7 0%	2/7 29%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	30/34 88%	0/34 0%	0/34 0%	2/34 6%	15/34 44%	21/34 62%	28/34 82%	14/34 41%	33/34 97%	30/34 88%	33/34 97%
Time Manipulation	5/7 71%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	1/7 14%	0/7 0%	0/7 0%	2/7 29%	1/7 14%	6/ 7 86%
Unchecked Low Calls	61/75 81%	0/75 0%	0/75 0%	0/75 0%	27/75 36%	0/75 0%	0/75 0%	49/75 65%	48/75 64%	60/75 80%	70/75 93%
Other	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/ 5 0%
Total	117/224 52%	0/224 0%	0/224 0%	5/224 2%	54/224 24%	35/224 16%	46/224 21%	65/224 29%	89/224 40%	94/224 42%	141/224 63%

Tabela 4.2: Resultados com o critério de acertar na categoria da vulnerabilidade e na linha exata

[2] `https://github.com/smartbugs/smartbugs-results`

[3] Para as ferramentas que analisam apenas o *bytecode* o máximo de vulnerabilidades são 19.

De forma a ter uma comparação um pouco mais justa e de não penalizar as ferramentas com uma taxa de sucesso baixa, o critério foi modificado ligeiramente e assim uma vulnerabilidade para ser considerada verdadeiro positivo por cada ferramenta tem na mesma de acertar na categoria dessa vulnerabilidade e indicar a linha onde a vulnerabilidade ocorre no código fonte do Smart Contract exceto que não necessita de ser a linha exata mas sim num intervalo de -5 a +5. Este critério será usado na restante Subsecção e os correspondentes resultados são apresentados na Tabela 4.3.

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/24 0%	0/24 0%	0/24 0%	5/24 21%	4/24 17%	0/24 0%	0/24 0%	1/24 4%	6/24 25%	2/24 8%	8/24 33%
Arithmetic	19/23 83%	0/23 0%	0/23 0%	13/23 57%	16/23 70%	13/23 57%	18/23 78%	0/23 0%	0/23 0%	1/23 4%	22/23 96%
Denial Service	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	1/14 7%	1/14 7%
Front Running	2/7 29%	0/7 0%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	2/7 29%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	30/34 88%	19/34 56%	0/34 0%	15/34 44%	25/34 74%	21/34 62%	28/34 82%	14/34 41%	33/34 97%	30/34 88%	33/34 97%
Time Manipulation	7/7 100%	0/7 0%	0/7 0%	4/7 57%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	3/7 43%	2/7 29%	7/ 7 100%
Unchecked Low Calls	62/75 83%	0/75 0%	0/75 0%	9/75 12%	60/75 80%	0/75 0%	0/75 0%	50/75 67%	51/75 68%	61/75 81%	70/75 93%
Other	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/ 5 0%
Total	120/224 54%	19/224 8%	0/224 0%	46/224 21%	107/224 48%	36/224 16%	48/224 21%	67/224 30%	93/224 42%	97/224 43%	143/224 64%
Delta	2%	8%	0%	19%	24%	0%	0%	1%	2%	1%	1%

Tabela 4.3: Resultados com o critério de acertar na categoria da vulnerabilidade mas com um intervalo de -5 a +5 em relação à linha exata

Na Tabela 4.3 os resultados são ligeiramente melhores para algumas ferramentas. A diferença para a Tabela 4.2 do aumento da percentagem da taxa de verdadeiros positivos de cada uma das ferramentas é apresentada na linha *Delta*. A Honeybadger, Manticore e Mythril beneficiaram bastante com este novo critério, no caso do Mythril o aumento foi de 24%. A Conkas continua a ser a ferramenta que deteta mais vulnerabilidades comparada com as restantes. Ainda nesta Tabela é possível verificar que a Smartcheck é a ferramenta que deteta mais categorias (7 categorias) tendo uma vantagem em relação às outras nomeadamente à Conkas. Esta deteta mais duas categorias que a Conkas, desta forma a sua taxa total de sucesso deveria ser maior. As ferramentas que detetam o mesmo número de categorias que a Conkas são a Manticore e a Mythril, detetando 5 categorias diferentes. A Conkas destaca-se bastante comparada à Manticore, tendo mais do dobro da taxa total de sucesso. Quando comparada com a Mythril, a Conkas é superior também, mas com uma diferença não tão grande, de 6%. Com estes resultados a Conkas é a que tem uma taxa total de sucesso superior a todas as outras, sendo até superior à Smartcheck que deteta mais categorias.

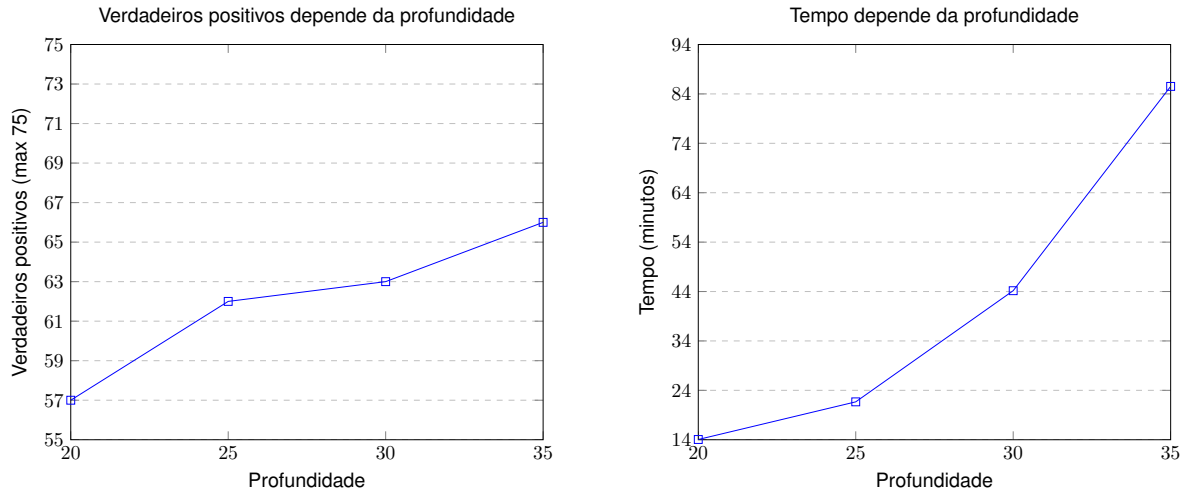
A justificação para a Conkas detetar mais 3 vulnerabilidades com o novo critério e tendo em conta que esta diz corretamente a linha exata, deve-se ao facto de, por exemplo, quando a vulnerabilidade está na linha de retorno da função, a linha que a Conkas indica é a linha onde é declarada a função. Isto acontece porque o mapeamento da instrução *assembly* é feito para o início da função. Se a função tiver 5 ou menos linhas então está no intervalo para ser considerada como verdadeiro positivo. Isto acontece nos dois contratos presentes no sub *dataset Time Manipulation*.

O que Falta para Chegar aos 100% de Taxa de Verdadeiros Positivos? Foi possível aferir que existem contratos da categoria *Arithmetic* que estão anotados com vulnerabilidades, no entanto estas só estão presentes no código fonte do contrato e quando compilados os contratos, as vulnerabilidades não são transferidas para o *bytecode*. Isto deve-se ao facto de as 4 vulnerabilidades que estão no código fonte estarem presentes em código morto e a Conkas como analisa o *bytecode* não as deteta, não chegando assim aos 100% de taxa de verdadeiros positivos nesta categoria.

A baixa percentagem relativamente à categoria *Front Running* deve-se ao facto de esta categoria ter vários sub tipos, tornando-se difícil de detetar todos, como foi descrito anteriormente na Secção 3.3.4. O mesmo acontece com as restantes ferramentas que detetam poucas vulnerabilidades desta categoria.

Em relação à categoria *Reentrancy* existe 1 contrato em que a Conkas indica que a vulnerabilidade está no modificador mas a vulnerabilidade está anotada na linha da função que usa esse modificador. Um outro caso é quando existe uma função que invoca outra e só neste cenário é que existe vulnerabilidade. A Conkas indica a linha onde a instrução *CALL* é executada mas a vulnerabilidade está anotada na função chamadora. Em ambos os casos, estas vulnerabilidades serão contadas como falsos positivos. Existem 2 vulnerabilidades presentes num contrato que depende de uma biblioteca e como a Conkas não suporta análise a contratos que dependam de bibliotecas, não irá analisar vulnerabilidades que possam haver nesse contrato, ver Subsecção 3.3.11.

Por fim, a Conkas não tem 100% de taxa de verdadeiros positivos na categoria *Unchecked Low Calls* devido à profundidade máxima definida para prevenir que a análise ficasse indefinidamente a ser executada. Assim, pode não chegar a analisar possíveis blocos que contenham vulnerabilidades. Outro cenário que existe é, uma vez mais, a eliminação de código proveniente da compilação do contrato, removendo do *bytecode* instruções *CALL*, removendo assim as vulnerabilidades. Uma vez que este *dataset* é maior que os restantes, não foi possível quantificar estes casos. Na Figura 4.1 é possível observar como a profundidade se relaciona com o tempo e com os verdadeiros positivos. Esta relação apenas considera a execução do sub *dataset Unchecked Low Calls*. Quando a profundidade é aumentada, a Conkas é capaz de detetar mais vulnerabilidades mas o tempo que demora a analisar também é maior. A profundidade definida pela Conkas é 25, pois considera-se estar num ponto que deteta um número razoável de vulnerabilidades ao mesmo tempo que não demora assim tanto tempo como acontece quando a profundidade é 30, que demora o dobro do tempo.



(a) Profundidade e verdadeiros positivos

(b) Profundidade e tempo

Figura 4.1: Relação entre profundidade e verdadeiros positivos e profundidade e tempo considerando apenas o sub *dataset Unchecked Low Calls*

Resumindo é possível observar que a Conkas acertou em todas as vulnerabilidades que analisou

existentes na categoria *Arithmetic*, considerando apenas as que estão no *bytecode*. Na categoria *Reentrancy* falha duas vulnerabilidades por dar a linha errada e duas vulnerabilidades não chega a analisar por estar num contrato que depende de uma biblioteca. Para a categoria *Unchecked Low Calls* não foi possível quantificar as vulnerabilidades, ou seja, as instruções *CALL* que foram removidas no processo de compilação. No entanto, se aumentar a profundidade a Conkas deteta um maior número de vulnerabilidades desta categoria, ver Figura 4.1.

4.1.1.2 Falsos Positivos

Na Tabela 4.4 são apresentadas todas as vulnerabilidades detetadas por cada ferramenta. Em cada linha é possível observar o número total de vulnerabilidades detetadas por cada ferramenta para uma dada categoria. De forma a obter o número de falsos positivos de uma ferramenta para uma dada categoria, basta subtrair a esse número, o número apresentado na Tabela 4.3 na mesma linha e na mesma coluna.

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	0	11	65	47	0	0	6	40	4	173
Arithmetic	325	0	0	22	219	213	395	0	0	25	1199
Denial Service	0	0	0	0	0	41	83	0	13	30	167
Front Running	26	0	0	0	55	0	62	204	0	0	347
Reentrancy	251	19	0	32	144	51	54	94	81	51	777
Time Manipulation	78	0	0	10	0	8	10	0	5	2	113
Unchecked Low Calls	97	0	0	14	90	0	0	146	96	74	517
Other	0	11	4	47	67	0	0	0	31	17	177
Total	777	30	15	190	622	313	604	450	266	203	3470

Tabela 4.4: Número total de vulnerabilidades detetadas (considerando a totalidade do *dataset*)

Os falsos positivos são apresentados na Tabela 4.5. Através desta é possível observar que a Conkas é a que apresenta o número de falsos positivos mais elevado. Não muito longe do número de falsos positivos da Conkas está a Oyente e a Mythril com números próximos, 556 e 515 respetivamente. No entanto, quando observada a categoria *Time Manipulation* o número de falsos positivos da Conkas é (percentualmente) bastante superior ao das restantes, apesar de a Conkas acertar em 100% das vulnerabilidades anotadas enquanto que as restantes acertam em pouco mais de metade, ver Tabela 4.3. Uma possível justificação para este elevado número de falsos positivos pode ser porque num contrato de uma categoria X podem haver vulnerabilidades de outras categorias, e estas outras vulnerabilidades não estão indicadas na SmartBugs e por isso serão contadas como falsos positivos, como foi referido anteriormente na Secção 4.1. Na Subsecção 4.1.1.3 será realizada uma análise em maior detalhe dos falsos positivos encontrados.

A ferramenta Oyente é a que tem um número mais alto de falsos positivos na categoria *Arithmetic*, mais 71 do que a Conkas. No entanto a ferramenta Mythril e a Osiris também têm um elevado número de falsos positivos. Na categoria *Unchecked Low Calls*, o número de falsos positivos é semelhante entre as várias ferramentas como a Conkas, Mythril, Slither e Smartcheck. A Securify é a que apresenta o número mais elevado, distanciando-se um pouco das outras ferramentas. Isto poderá querer dizer que talvez estes números não serão todos falsos positivos.

Na categoria *Reentrancy* o número de falsos positivos da Conkas é muito superior ao das restantes.

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	0	11	60	43	0	0	5	34	2	155
Arithmetic	306	0	0	9	203	200	377	0	0	24	1119
Denial Service	0	0	0	0	0	41	83	0	13	29	166
Front Running	24	0	0	0	53	0	60	202	0	0	339
Reentrancy	221	0	0	17	119	30	26	80	48	21	562
Time Manipulation	71	0	0	6	0	6	10	0	2	0	95
Unchecked Low Calls	35	0	0	5	30	0	0	96	45	13	224
Other	0	11	4	47	67	0	0	0	31	17	177
Total	657	11	15	144	515	277	556	383	173	106	2837

Tabela 4.5: Número de falsos positivos

Para além da justificação dada anteriormente, este número elevado pode ser devido à atualização do Ethereum que fez com que as funções *send* e *transfer* deixassem de ser consideradas seguras para mitigar este tipo de vulnerabilidade, ver Secção 2.5.1. As outras ferramentas devem considerar estas funções como seguras e por isso a disparidade do número da Conkas comparado com as restantes. Uma análise dos falsos positivos será descrita na Subsecção 4.1.1.3.

Na categoria *Front Running* a Conkas é a que tem o número mais baixo de falsos positivos e nenhuma das ferramentas deteta mais vulnerabilidades do que a Conkas detetou, ver Tabela 4.3.

Todas as justificações dadas são hipóteses, por isso existe a necessidade de verificar manualmente se realmente os falsos positivos são falsos positivos ou verdadeiros positivos. A estratégia usada para a seleção dos ficheiros foi escolher os *hashes* do nome dos ficheiros que terminassem com uns valores específicos. Desta forma a escolha dos ficheiros é aleatória e não existe manipulação dos resultados. A função de *hash* usada foi o MD5. O objetivo é analisar manualmente cerca de 20% dos ficheiros de cada sub *dataset* e mais do que 10 falsos positivos de cada sub *dataset*. A probabilidade do *hash* terminar em x é $1/16$ - pois apenas contém caracteres hexadecimais -, e por isso para ter cerca de 20% é necessário ter uma probabilidade de $3/16$ pelo que se selecionaram todos os *hashes* que terminam em 1, 2 ou 3. Sempre que a percentagem seja abaixo dos 15%, adiciona-se mais um caracter, por ordem crescente, sendo o próximo o caracter 4, de modo a ter uma percentagem perto dos desejados 20%. Se esta percentagem refletir que serão menos do que 10 falsos positivos a serem analisados manualmente, repete-se o processo de aumentar a percentagem até atingir pelo menos 10 falsos positivos. Foi escolhida uma função de *hash* porque espera-se que esta seja o mais uniforme possível.

4.1.1.3 Análise aos Falsos Positivos

Os nomes dos ficheiros em que a SmartBugs reportou falsos positivos para as ferramentas que foram analisadas manualmente estão presentes no Apêndice B.1. No Apêndice B.2 são apresentados os nomes dos ficheiros selecionados para serem verificados manualmente e os resultados dessa verificação manual para as ferramentas Conkas, Mythril, Slither e Smartcheck, identificando para cada uma o nome do ficheiro selecionado e a linha correspondente no código fonte que tem ou não um verdadeiro positivo ou um falso positivo. Foram consideradas apenas estas ferramentas por serem as que têm uma taxa de verdadeiros positivos acima dos 40%, sendo as mais próximas da Conkas.

Na Tabela 4.6 é apresentada a estatística da seleção dos ficheiros que foram verificados manualmente. O objetivo é analisar cerca de 20% dos ficheiros em que a Conkas reportou falsos positivos

para uma dada categoria e essa percentagem pode ser observada na Tabela 4.6 assim como o número de ficheiros que essa percentagem reflete. Se nos ficheiros selecionados não contiver pelo menos 10 falsos positivos então a percentagem será aumentada até haver no mínimo 10 falsos positivos para serem analisados manualmente.

Categoria	Percentagem dos Ficheiros	Número de Ficheiros
Arithmetic	18,28%	17
Front Running	54,55%	12
Reentrancy	17,35%	17
Time Manipulation	24,14%	7
Unchecked Low Calls	41,18%	7

Tabela 4.6: Estatística dos ficheiros selecionados para verificação manual

Na Tabela 4.7 é apresentada a percentagem de verdadeiros positivos que foram (erradamente) considerados falsos positivos por cada categoria e o rácio entre os falsos positivos verificados manualmente e todos os falsos positivos detetados pela SmartBugs por cada categoria, em relação à Conkas.

	Conkas	
Categoria	Verdadeiros Positivos	Rácio
Arithmetic	21/73 29%	24%
Front Running	6/13 46%	54%
Reentrancy	21/38 55%	17%
Time Manipulation	9/17 53%	24%
Unchecked Low Calls	10/10 100%	29%

Tabela 4.7: Percentagem de verdadeiros positivos considerados falsos positivos e rácio entre falsos positivos analisados e total, relativamente à Conkas

Em relação à categoria *Arithmetic* apenas 29% dos falsos positivos considerados pela SmartBugs são verdadeiros positivos, indicando que de facto a Conkas apresenta bastantes falsos positivos tal como todas as outras ferramentas. Isto quer dizer que existem vários *overflows/underflows* presentes no *bytecode* que são propositados pelo compilador de Solidity. Um problema para alguns falsos positivos da Conkas é que esta não tem em conta as funções que rodeiam a função a ser analisada, ou seja, se houver uma função que devolve o tamanho de um *array* subtraindo o valor 1, a Conkas indicará que poderá ocorrer um *Underflow* no caso do tamanho do *array* ser 0. Isto pode não ser válido se as outras funções não deixarem que o tamanho do *array* seja menor do que 1.

Na categoria *Front Running* a percentagem é de 46% indicando que existem algumas vulnerabilidades que não estão anotadas. Como foi escolhida uma função de *hash* com uma distribuição uniforme pode-se concluir que 46% dos falsos positivos são na realidade verdadeiros positivos. O mesmo acontece na categoria *Unchecked Low Calls* onde a percentagem é de 100%. No entanto, nas categorias *Reentrancy* e *Time Manipulation*, a Conkas tem um elevado número de falsos positivos. Pela análise feita manualmente, conclui-se que 55% dos falsos positivos na categoria *Reentrancy* são verdadeiros positivos e o mesmo acontece na categoria *Time Manipulation* com uma percentagem de 53%.

Não seria justo atualizar a Tabela 4.5 com a informação presente na Tabela 4.7 só para a ferra-

menta Conkas. Assim, o mesmo método usado pela Conkas para a seleção dos ficheiros a analisar manualmente foi usado nas ferramentas Mythril, Slither e Smartcheck. Foram escolhidas apenas estas ferramentas por serem as que têm uma taxa de verdadeiros positivos acima dos 40%, sendo as mais próximas da Conkas. Na Tabela 4.8 são apresentadas as estatísticas da seleção dos ficheiros onde foram reportados falsos positivos para uma dada categoria e que serão verificados manualmente para cada uma das ferramentas, sendo apresentado o número de ficheiros selecionados para cada categoria e para cada ferramenta assim como a percentagem que corresponde ao número de ficheiros selecionados. Na Tabela 4.9 são apresentadas as percentagens de verdadeiros positivos que foram (erradamente) considerados falsos positivos pela SmartBugs assim como o rácio entre os falsos positivos verificados manualmente e todos os falsos positivos detetados pela SmartBugs correspondentes às ferramentas Mythril, Slither e Smartcheck.

Categoria	Mythril Número de Ficheiros	Slither Número de Ficheiros	Smartcheck Número de Ficheiros
Arithmetic	18 (20,00%)	-	7 (50,00%)
Front Running	6 (20,00%)	-	-
Reentrancy	14 (19,44%)	2 (20,00%)	7 (70,00%)
Time Manipulation	-	-	-
Unchecked Low Calls	8 (57,14%)	9 (22,50%)	7 (77,78%)

Tabela 4.8: Estatística dos ficheiros selecionados para verificação manual relativamente à Mythril, Slither e Smartcheck

Categoria	Mythril		Slither		Smartcheck	
	Verdadeiros Positivos	Rácio	Verdadeiros Positivos	Rácio	Verdadeiros Positivos	Rácio
Arithmetic	4/33 12%	16%	-	-	4/10 40%	42%
Front Running	1/11 9%	21%	-	-	-	-
Reentrancy	3/21 14%	18%	2/26 8%	54%	6/10 60%	48%
Time Manipulation	-	-	-	-	-	-
Unchecked Low Calls	3/17 18%	57%	0/11 0%	11%	6/11 55%	85%

Tabela 4.9: Percentagem de verdadeiros positivos considerados falsos positivos e rácio entre falsos positivos analisados e total, relativamente à Mythril, Slither e Smartcheck

Onde está assinalado com "-" significa que não foram selecionados ficheiros para analisar manualmente por não existirem verdadeiros positivos correspondentes à ferramenta e à categoria em causa. Em relação à ferramenta Mythril podemos observar que as percentagens são baixas, inferiores a 50%, para as categorias *Arithmetic*, *Front Running*, *Reentrancy* e *Unchecked Low Calls*.

Quanto à ferramenta Slither, esta tem uma particularidade que é, para cada vulnerabilidade detetada indica as linhas todas do contrato, as linhas todas da função e as linhas das expressões onde ocorre essa vulnerabilidade. Foram apenas selecionados 2 ficheiros para verificação manual para a categoria *Reentrancy*. A Slither indica bastantes linhas onde ocorrem vulnerabilidades e para a categoria *Reentrancy* apenas 2 dos 26 falsos positivos são na realidade verdadeiros positivos. Uma das razões para o número de falsos positivos da Slither ser baixo é porque o *script* da SmartBugs-results que foi usado não considera falsos positivos se num conjunto de linhas uma delas for verdadeira positiva. Este *script* usado na análise percorre todas as linhas de uma função que o Slither considerou ter uma vul-

nerabilidade e se houver uma linha que satisfaça a condição para uma vulnerabilidade ser considerada verdadeiro positivo, as restantes são ignoradas não contando como falsos positivos. Para a categoria *Unchecked Low Calls* a percentagem de verdadeiros positivos é de 0%. Por esta análise foi possível perceber que esta ferramenta não acerta bem na linha do código fonte, uma vez que a maior parte das linhas indicadas nas funções eram comentários ou linhas sem código. Foi também possível perceber que o baixo número de falsos positivos apresentado na Tabela 4.5 deve-se ao facto desta ferramenta apresentar todas as linhas do contrato das vulnerabilidades detetadas. É possível também perceber que a maior parte dos falsos positivos reportados por esta ferramenta são de facto falsos positivos, dada a baixa percentagem de verdadeiros positivos, como mostrado na Tabela 4.9.

Em relação à Smartcheck, a percentagem da seleção dos ficheiros é mais alta do que os 20% porque em cada ficheiro selecionado existem poucos falsos positivos, devido ao baixo número de falsos positivos reportados como mostrado na Tabela 4.5. Pela Tabela 4.9, relativamente a esta ferramenta, a categoria mais penalizada pela analise manual dos falsos positivos é a categoria *Arithmetic*, tendo 40% de verdadeiros positivos. A categoria com a percentagem maior de verdadeiros positivos é a categoria *Reentrancy*, com 60%. Em relação à categoria *Unchecked Low Calls* a percentagem de verdadeiros positivos é de 55%.

Foi então atualizada a Tabela 4.5, apenas nas ferramentas mencionadas anteriormente, com base nas percentagens de verdadeiros positivos considerados falsos positivos como descrito anteriormente. Na Tabela 4.10 são apresentados os novos números de falsos positivos de cada ferramenta e na linha *Delta* é apresentada a diferença do número de falsos positivos relativamente à Tabela 4.5 para cada ferramenta.

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	-	-	-	43	-	-	-	34	2	79
Arithmetic	217	-	-	-	179	-	-	-	0	14	410
Denial Service	0	-	-	-	0	-	-	-	13	29	42
Front Running	13	-	-	-	48	-	-	-	0	0	61
Reentrancy	99	-	-	-	102	-	-	-	44	8	253
Time Manipulation	33	-	-	-	0	-	-	-	2	0	35
Unchecked Low Calls	0	-	-	-	25	-	-	-	45	6	76
Other	0	-	-	-	67	-	-	-	31	17	115
Total	362	-	-	-	464	-	-	-	169	76	1071
Delta	295	-	-	-	51	-	-	-	4	30	1766

Tabela 4.10: Número de falsos positivos normalizados pela análise manual de falsos positivos

O número de falsos positivos da Conkas baixou praticamente em todas as categorias como era de esperar, mas a categoria *Arithmetic* continua a ser a que penaliza mais a Conkas, sendo este um ponto a melhorar no futuro, por outro lado, é a que tem uma taxa de verdadeiros positivos maior nesta categoria. Os falsos positivos nas restantes categorias, à exceção da categoria *Time Manipulation*, são inferiores aos falsos positivos da Mythril, não sendo verdade para as ferramentas Slither e Smartcheck, à exceção da categoria *Unchecked Low Calls*. No entanto, como visto anteriormente, a ferramenta Slither indica todas as linhas do contrato, da função e da expressão onde esta reporta uma vulnerabilidade, tendo bastantes falsos positivos que não são reportados usando o atual *script* para obter os resultados.

Com estes resultados é possível concluir que existem de facto muitos contratos que não estão to-

talmente anotados com todas as vulnerabilidades presentes. Na categoria *Reentrancy* existem muitas vulnerabilidades que não estão anotadas devido à atualização do Ethereum, como foi dito anteriormente. Existem alguns falsos positivos reportados pela Conkas também devido à herança que possa haver entre os diferentes contratos. Isto porque se um contrato A contiver uma vulnerabilidade, e houver um contrato B que estenda do contrato A, então o B também irá ter presente a vulnerabilidade de A. A Conkas ao analisar o contrato B irá indicar uma vulnerabilidade presente numa função do contrato A, no entanto, será contabilizada como falso positivo uma vez que já foi contabilizada quando a Conkas analisou o contrato A.

4.1.1.4 Falsos Positivos Excluindo a Categoria *Arithmetic*

Como mencionado nas Subsecções anteriores, a categoria *Arithmetic* é a que mais penaliza a Conkas em termos de falsos positivos e é a categoria que mais os tem, ver Tabela 4.10. Desta forma, se removermos esta categoria da Tabela 4.10, é esperado que os resultados sejam substancialmente melhores para todas as ferramentas que detetam esta categoria de vulnerabilidades. Na Tabela 4.11 são apresentados os números de falsos positivos para cada ferramenta e para cada categoria removendo a categoria *Arithmetic*.

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	0	11	60	43	0	0	5	34	2	155
Denial Service	0	0	0	0	0	41	83	0	13	29	166
Front Running	13	0	0	0	48	0	60	202	0	0	323
Reentrancy	99	0	0	17	102	30	26	80	44	8	406
Time Manipulation	33	0	0	6	0	6	10	0	2	0	57
Unchecked Low Calls	0	0	0	5	25	0	0	96	45	6	177
Other	0	11	4	47	67	0	0	0	31	17	177
Total	145	11	15	135	285	77	179	383	169	62	1461
Delta	217	0	0	9	179	200	377	0	0	14	996

Tabela 4.11: Número de falsos positivos sem a categoria *Arithmetic*

A linha *Delta* na Tabela 4.11 mostra a diferença do total de falsos positivos detetados por cada ferramenta entre o total de falsos positivos detetados por cada ferramenta da Tabela 4.10. Estes números correspondem aos falsos positivos de cada ferramenta correspondentes à categoria *Arithmetic*. É possível observar que a Oyente, Conkas, Osiris e Mythril foram as que beneficiaram mais por ordem decrescente. A ferramenta Smartcheck não beneficia muito porque só deteta 4% das vulnerabilidades anotadas no *dataset* da categoria *Arithmetic*.

Tendo em conta apenas as ferramentas Conkas, Mythril, Slither e Smartcheck pelas razões mencionadas anteriormente, a Conkas tem um número de falsos positivos inferiores às ferramentas Mythril e Slither. Apenas a Smartcheck fica à frente da Conkas, tendo menos 83 falsos positivos que a Conkas.

É possível concluir que de facto a categoria Arithmetic é a categoria que mais penaliza a Conkas em termos de falsos positivos. No entanto, é esta ferramenta que tem a melhor taxa de verdadeiros positivos, e como foi mencionado anteriormente, só não tem 100% de taxa de verdadeiros positivos pois existem vulnerabilidades que não são transferidas para o *bytecode*. O módulo que deteta esta categoria deve ser trabalhado no futuro de modo a diminuir os falsos positivos. É possível também concluir que mesmo excluindo a categoria *Arithmetic*, a Smartcheck é a que tem um número de falsos

positivos mais baixo, seguido da Conkas. No entanto, em relação à taxa de verdadeiros positivos, a Conkas é superior tendo mais 11% que a Smartcheck.

4.1.1.5 Performance

A performance das ferramentas é um ponto a ter em conta pois se a ferramenta demorar muito tempo a verificar um contrato, o utilizador desta não irá ficar satisfeito. O ambiente de execução foi uma máquina virtual Ubuntu 18.04.5 LTS com 4 cores do CPU Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz e com 16GB de memória RAM. Na Tabela 4.12 é mostrado, em cada linha, o tempo médio de execução e o tempo total de execução correspondente a cada ferramenta.

#	Tool	Avg. Execution Time	Total Execution Time
1	Conkas	0:00:32	1:14:37
2	Honeybadger	0:01:12	2:49:03
3	Maian	0:03:47	8:52:25
4	Manticore	0:12:53	1 day, 6:15:28
5	Mythril	0:00:58	2:16:21
6	Osiris	0:00:21	0:50:25
7	Oyente	0:00:05	0:12:35
8	Securify	0:02:06	4:56:13
9	Slither	0:00:04	0:09:56
10	Smartcheck	0:00:15	0:35:23
Total		———	2 days, 4:12:25

Tabela 4.12: Tempo de execução de cada ferramenta

A ferramenta Manticore é de longe a que demora mais a ser executada seguida da Maian. A que tem melhor performance é o Slither com 4 segundos em média para analisar cada contrato. A Conkas está atrás da ferramenta Oyente mas à frente da Mythril. No entanto, a ferramenta Oyente não tem o passo extra de elevar o *bytecode* para uma representação intermédia. O Slither e a Smartcheck são ferramentas que analisam o código fonte em Solidity e não têm o passo extra de compilar o contrato nem de executar cada instrução simbolicamente. Assim, tendo em conta o que foi dito, o tempo médio da Conkas é um tempo razoavelmente bom.

4.1.1.6 Combinação entre as Ferramentas

Dado que a Conkas tem um número de falsos positivos mais alto que a Slither - no entanto isto não é verdade como visto na Subsecção 4.1.1.3 - e a Smartcheck considerando apenas as ferramentas que tenham uma taxa de verdadeiros positivos acima dos 40%, é boa ideia considerar a combinação entre as ferramentas. Na Tabela 4.13 são apresentadas as combinações das diferentes ferramentas tendo por base apenas a taxa de verdadeiros positivos de cada uma, usando como métrica a união das vulnerabilidades detetadas por cada uma.

Pelos resultados obtidos da união das vulnerabilidades detetadas por cada ferramenta, a melhor escolha vai para a Conkas com a Mythril ou a Conkas com a Slither, com uma percentagem de 58% em ambas as escolhas. No entanto, se forem considerados também os falsos positivos, a escolha vai

	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck
Conkas		120/224 54%	120/224 54%	125/224 56%	130/224 58%	120/224 54%	123/224 55%	127/224 57%	131/224 58%	128/224 57%
Honeybadger			19/224 8%	60/224 27%	113/224 50%	41/224 18%	48/224 21%	80/224 36%	93/224 42%	97/224 43%
Maian				46/224 21%	107/224 48%	36/224 16%	48/224 21%	67/224 30%	93/224 42%	97/224 43%
Manticore					114/224 51%	63/224 28%	70/224 31%	91/224 41%	114/224 51%	117/224 52%
Mythril						118/224 53%	117/224 52%	115/224 51%	126/224 56%	123/224 55%
Osiris							53/224 24%	91/224 41%	107/224 48%	111/224 50%
Oyente								99/224 44%	113/224 50%	116/224 52%
Securify									105/224 47%	105/224 47%
Slither										109/224 49%
Smartcheck										

Tabela 4.13: Combinação das ferramentas

para a Conkas com a Smartcheck. É feita esta escolha porque a Conkas é a que tem a maior taxa de verdadeiros positivos e a Smartcheck é a que tem o número mais baixo de falsos positivos com uma taxa de verdadeiros positivos acima dos 40%. Os falsos positivos da Conkas e da Smartcheck podem-se sobrepor, porém, sem esta informação assume-se que a Smartcheck será a melhor combinação quando é necessário ter os falsos positivos em conta. Estas duas ferramentas têm uma boa performance, sendo mais um ponto a favor da combinação destas ferramentas.

4.2 SolidiFI

A Conkas foi adicionada a esta framework, mas este processo não é muito simples comparando com a Smartbugs, ver Secção 4.1. Adicionar a Conkas ao *script* que irá executar a Conkas sobre os diferentes contratos é relativamente fácil, pois basta adicionar o comando que irá executar a Conkas. No entanto, para adicionar a Conkas ao *script* de análise dos resultados é algo que necessita de mais atenção porque o código está pouco estruturado. O código fonte utilizado na análise dos resultados obtidos pelo SolidiFI está presente publicamente no GitHub[4].

Depois de a Conkas ser adicionada à framework, esta começa por injetar os diferentes tipos de bugs que a Conkas deteta. Depois de injetados, a Conkas irá analisá-los à procura de vulnerabilidades causadas pela injeção desses bugs. Esta framework gerou erros a injetar bugs do tipo *Timestamp-Dependency*, o que resultou em contratos que quando compilados geravam erros.

Antes de ser possível obter os resultados da Conkas nesta framework, o *script* usado gerava erros que nada tiveram a ver com a adição da Conkas. Foi necessário corrigir estes erros, que consistiam na falta de um ficheiro Python necessário para a execução deste *script*, e foi necessária a criação de duas pastas específicas. Este *script* também não era capaz de realizar a análise dos resultados da ferramenta Oyente pelo que foi necessário modificar o caracter ":" para o caracter "_".

4.2.1 Análise da Conkas

Nesta secção serão apresentados os resultados obtidos através da comparação da ferramenta Conkas com outras ferramentas consideradas estado da arte usando a framework SolidiFI. Serão analisados os falsos negativos (os bugs que foram mal classificados e os que não foram detetados) e os falsos positivos. É de salientar que os resultados apresentados para as ferramentas que não a Conkas são

[4] `https://github.com/DependableSystemsLab/SolidiFI-benchmark`

os mesmos resultados apresentados no artigo desta framework [30].

4.2.1.1 Falsos Negativos

Na Tabela 4.14 são apresentados os falsos negativos reportados pelas diferentes ferramentas para os diferentes tipos de bugs e entre parênteses é indicado o número de bugs que não foram detetados. Os falsos negativos são bugs que não foram reportados pelas ferramentas ou que foram reportados pelas ferramentas em que acertam nas linhas onde está o bug mas identificam mal o tipo da vulnerabilidade. Para obter os bugs que foram mal classificados basta subtrair o número de falsos negativos com o número de bugs que não foram detetados (número dentro dos parênteses). Na Tabela 4.15 são apresentados os bugs que foram mal classificados.

	Conkas	Oyente	Securify	Mythril	Smartcheck	Manticore	Slither	Número de bugs injetados
Re-entrancy	176 (173)	1008 (844)	232 (232)	1085 (805)	1343 (106)	1250 (1108)	0 (0)	1343
Timestamp-Dependency	1389 (1389)	1381 (886)	-	810 (810)	902 (341)	-	537 (1)	1389
Unchecked-Send	-	-	499 (499)	389 (389)	-	-	-	1266
Unhandled-Exception	678 (325)	1052 (918)	673 (571)	756 (756)	1325 (1170)	-	457 (128)	1374
TOD	763 (168)	1199 (1199)	263 (263)	-	-	-	-	1336
Overflow-Underflow	758 (748)	898 (898)	-	1069 (932)	1072 (1072)	1196 (1127)	-	1333
tx.origin	-	-	-	455 (455)	1239 (1120)	-	0 (0)	1336

Tabela 4.14: Falsos negativos reportados pelas ferramentas. Entre parênteses é o número de bugs não reportados. Resultados parciais de [30]

	Conkas	Oyente	Securify	Mythril	Smartcheck	Manticore	Slither
Re-entrancy	3	164	0	280	1237	142	0
Timestamp-Dependency	0	495	-	0	561	-	536
Unchecked-Send	-	-	0	0	-	-	-
Unhandled-Exception	353	134	102	0	155	-	329
TOD	595	0	0	-	-	-	-
Overflow-Underflow	10	0	-	137	0	69	-
tx.origin	-	-	-	0	119	-	0

Tabela 4.15: Número de bugs que foram mal classificados. Resultados parciais de [30]

Na categoria *Re-entrancy* a Conkas apresenta menos falsos negativos do que todas as ferramentas à exceção da ferramenta Slither e em relação aos bugs não detetados a Conkas é outra vez a melhor à exceção da Smartcheck e da Slither. Em relação aos bugs mal classificados a Conkas é melhor que todas as ferramentas à exceção da Securify e da Slither, ficando muito perto destas.

Na categoria *Timestamp-Dependency*, a Conkas não realizou nenhuma análise devido ao problema que a SolidiFI teve em injetar este tipo de bugs nos diferentes contratos, daí ter tido todos os bugs como falsos negativos e não detetados, e nenhum mal classificado.

Em relação à categoria *Unhandled-Exception*, a Conkas fica atrás da Securify por muito pouco,

apenas mais 5 falsos negativos, e atrás da Slither. Teve menos bugs não reportados do que todas as ferramentas com exceção da Slither, no entanto é a que apresenta o maior número de bugs mal classificados.

Na categoria *TOD*, só há 2 ferramentas, Oyente e Securify, para além da Conkas que também detetam este tipo de vulnerabilidade. A Conkas apresenta menos falsos negativos do que a Oyente mas mais do que a Securify. Em relação aos bugs não reportados a Conkas é a que apresenta o menor número. Quando analisados os bugs mal classificados, só a Conkas é que tem um valor diferente de 0. Ambas as ferramentas não terem bugs mal classificados pode significar que alguma coisa pode ter falhado na análise destas duas ferramentas, uma vez que todos os bugs analisados foram não reportados.

Por fim, em relação à categoria *Overflow-Underflow*, a Conkas é a que apresenta menos falsos negativos do que as restantes ferramentas sendo também a que apresenta o menor número de bugs não reportados. As ferramentas que apresentam 0 bugs mal classificados, a Oyente e a Smartcheck, pode significar que alguma coisa pode ter falhado na análise destas duas ferramentas, uma vez que todos os bugs analisados foram não reportados. Não considerando estas duas ferramentas, a Conkas é a ferramenta que apresenta o menor número de bugs mal classificados. Nesta categoria é conhecido um bug que na realidade não apresenta nenhuma vulnerabilidade. O bug presente na Listagem 24 não introduz nenhuma vulnerabilidade uma vez que as variáveis têm o mesmo tipo e é inicializada com o valor 0. Isto pode indicar que a validade do *dataset* poderá estar comprometida.

```
1 function bug_intou20(uint8 p_intou20) public{
2     uint8 vundflw1=0;
3     vundflw1 = vundflw1 + p_intou20;   // overflow bug
4 }
```

Listagem 24: Bug injetado da categoria *Overflow-Underflow* que não apresenta nenhuma vulnerabilidade [30]

O facto de a ferramenta Slither ter sempre o número mais baixo de falsos negativos poderá ter a ver com o facto desta ferramenta ser bastante verbosa em relação ao número de linhas que indica, como visto anteriormente na Subsecção 4.1.1.3.

4.2.1.2 Falsos Positivos

Na Tabela 4.16 são apresentados os falsos positivos reportados pelas diferentes ferramentas para os diferentes tipos de bugs.

Quando analisados os falsos positivos de cada ferramenta, a Conkas é a que apresenta mais falsos positivos em todas as categorias à exceção das categorias *TOD* e *Overflow-Underflow*. Estes números são diferentes dos falsos positivos reportados pela SmartBugs, como mostra a Tabela 4.5. Nessa tabela, a Conkas na categoria *Unchecked Low Calls*, idêntica à categoria *Unhandled-Exception* na SolidiFI, é a que tem o menor número de falsos positivos em relação à Securify e à Slither. No entanto, pelos resultados obtidos e apresentados na Tabela 4.16, essas duas ferramentas, Securify e Slither, não apresentam nenhum falso positivo. Ainda nos resultados obtidos pela SmartBugs, a Conkas teve

	Conkas	Oyente	Securify	Mythril	Smartcheck	Manticore	Slither
Re-entrancy	375	0	12	54	0	6	79
Timestamp-Dependency	81	0	-	12	0	-	12
Unchecked-Send	-	-	7	14	-	-	-
Unhandled-Exception	29	10	0	0	6	-	0
TOD	69	32	121	-	-	-	-
Overflow-Underflow	346	947	-	17	3	9	-
tx.origin	-	-	-	0	3	-	4
other	0	0	318	144	1520	169	1807

Tabela 4.16: Falsos positivos reportados por cada ferramenta. Resultados parciais de [30]

apenas mais 5 falsos positivos que a ferramenta Mythril enquanto que nos falsos positivos reportados pela SolidiFI a ferramenta Mythril não apresenta nenhum falso positivo e a Conkas apresenta 29 falsos positivos. Os resultados obtidos pela framework SolidiFI são bastante dispares dos resultados obtidos pela framework SmartBugs.

Como referido anteriormente, os resultados apresentados nas Tabelas 4.14, 4.15 e 4.16 são os mesmos que os autores desta framework apresentam no artigo [30]. A razão desta escolha deveu-se ao facto de se ter realizado uma tentativa de executar a ferramenta Oyente sobre esta framework. Nesta tentativa não se obteve nenhum resultado como era de esperar, algo idêntico aos resultados apresentados em Ghaleb and Pattabiraman [30] para esta ferramenta, apenas foram obtidos erros. Uma vez que tal não foi possível, não se realizou o mesmo para as restantes ferramentas incluídas nesta framework e usou-se os resultados apresentados em Ghaleb and Pattabiraman [30]. Desta forma, esta análise usando a framework SolidiFI carece de alguma credibilidade, não devendo os resultados serem tidos muito em conta.

4.3 Limitações da Análise

Uma limitação desta análise é que não foram testados contratos que fossem escritos em Solidity que usassem versões maiores ou iguais à versão 0.6.0. Os *datasets* da SmartBugs nem da SolidiFI que foram usados nesta análise não fornecem nenhum contrato nestas condições. Foram testados apenas dois contratos com uma vulnerabilidade de reentrância. Cada contrato foi escrito usando três versões diferentes, a versão 0.4.25, 0.5.17 e a 0.6.7. Em ambos os contratos e para todas as versões a Conkas não gerou nenhum erro e apresentou as vulnerabilidades esperadas.

Capítulo 5

Conclusões

Neste Capítulo será descrito se os objetivos foram atingidos, quais as contribuições desta dissertação e também é dada uma perspetiva do trabalho futuro.

5.1 Objetivos Atingidos

O uso da blockchain e do Ethereum tem vindo a aumentar e consequentemente mais programadores serão necessários para satisfazer as necessidades de quem quer usar os benefícios da Ethereum, como os Smart Contracts. Mais programadores significa mais Smart Contracts, mais possíveis vulnerabilidades. Com o trabalho desenvolvido ao longo desta dissertação, os programadores têm uma ferramenta de análise de vulnerabilidades com uma taxa de verdadeiros positivos maior.

Inicialmente foi descrito o que é a blockchain bem como o Ethereum, os Smart Contracts e o funcionamento da Ethereum Virtual Machine. Foram também apresentadas as vulnerabilidades mais comuns no Ethereum segundo o DASP Top 10 e ainda foram apresentadas algumas ferramentas de análise estática consideradas estado da arte, depois de uma análise à literatura existente.

Foi desenvolvida a Conkas, uma ferramenta modular de análise estática, fácil de estender, ou seja, fácil de adicionar novas instruções EVM bem como novos sub módulos para detetar novas vulnerabilidades. Para erguer a Conkas foi necessário realizar algumas modificações à ferramenta Rattle que foi usada na Conkas. Com a contribuição da Conkas, os programadores de Smart Contracts podem usá-la para verificar os seus contratos.

Foi analisada a performance da Conkas comparada com outras ferramentas usando a framework SmartBugs. Foi também analisada a taxa de verdadeiros positivos da Conkas e esta é superior a todas as ferramentas já existentes usadas na comparação, no entanto também foi a que apresentou o número de falsos positivos mais elevado. Foram selecionados aleatoriamente alguns ficheiros para verificar manualmente se os falsos positivos detetados pela SmartBugs eram de facto falsos positivos ou verdadeiros positivos. Os ficheiros selecionados foram analisados manualmente para as ferramentas Conkas, Mythril, Slither e Smartcheck que foram as que apresentaram melhor performance em termos de deteção. Os falsos positivos da categoria *Arithmetic* eram muitos deles de facto falsos positivos,

mas para as restantes categorias mais de metade dos falsos positivos eram na realidade verdadeiros positivos, o que significa que os contratos não estão corretamente anotados no *dataset* em causa. Foi ainda analisada a combinação das ferramentas, usando como métrica a união das vulnerabilidades detetadas por cada ferramenta.

A framework SolidiFI também foi usada para comparar a performance da Conkas com outras ferramentas. Foi analisado os falsos negativos entre as várias ferramentas e a Conkas esteve quase sempre perto da ferramenta que apresentou o menor número de falsos negativos para uma dada categoria. Mais tarde foram analisados os falsos positivos e a Conkas foi a que apresentou sempre mais falsos positivos à exceção da categoria *Overflow-Underflow*. Estes números quando comparados com os resultados obtidos pela SmartBugs são bastante dispares.

Com todo o trabalho desenvolvido e realizado foram atingidos os objetivos propostos. A performance da Conkas é boa, é fácil adicionar novos sub módulos para detetar novas vulnerabilidades bem como adicionar novas instruções EVM e a taxa de verdadeiros positivos também foi superior a todas as ferramentas comparadas.

5.2 Trabalho Futuro

Algumas sugestões para trabalho futuro incluem melhorar o módulo da Conkas Motor de Execução Simbólica, aumentando o nível de informação das variáveis para incluir por exemplo onde foram declaradas. Outra melhoria seria poder usar estratégias diferentes na exploração de caminhos presentes no CFG. Outra sugestão seria melhorar o sub módulo que deteta as vulnerabilidades do tipo reentrância, melhorar o sub módulo que deteta as vulnerabilidades do tipo aritmética para diminuir os falsos positivos e adicionar novos sub módulos para detetar outros tipos de vulnerabilidades. De forma a diminuir também os falsos positivos seria interessante adicionar uma fase de validação semelhante ao que a ferramenta Maian faz que é executar o contrato concretamente numa blockchain privada para diminuir os falsos positivos. Mais tarde, teria de ser criado um módulo para cada vulnerabilidade que fosse capaz de identificar se realmente a vulnerabilidade existe, ou seja, se com as alterações ao estado da blockchain resultante da execução de um contrato, existe alguma propriedade desse estado que seja violada. No entanto, esta fase de validação teria certamente impacto na performance. Sempre que existam novas instruções na EVM estas devem ser adicionadas à Conkas. Por fim, outra sugestão seria eliminar as limitações da Rattle e consequentemente da Conkas.

Em relação ao *dataset* da SmartBugs, usado na comparação dos resultados da ferramenta Conkas, o trabalho futuro passaria por fazer uma revisão deste *dataset* para tentar anotar todas as vulnerabilidades presentes. Se houver vulnerabilidades diferentes no mesmo contrato, este contrato deveria estar presente em mais do que um sub *dataset*.

5.3 Outras Contribuições

Para além do trabalho realizado no âmbito desta dissertação foram feitas outras contribuições nomeadamente à framework SmartBugs. Uma das contribuições foi corrigir o *parser* da ferramenta Oyente assim como um contrato que continha um erro de escrita e que impossibilitava a sua análise. Outra contribuição foi remover uma vulnerabilidade de uma contrato da categoria *Arithmetic* presente no *dataset* manualmente anotado. Por fim, haviam 3 contratos com vulnerabilidades manualmente anotadas em que as linhas não correspondiam ao local onde a vulnerabilidade ocorria pelo que essa anotação foi atualizada.

Bibliografia

[1] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008. (Acedido em 15/09/2020).

[2] Ethereum. White paper · a next-generation smart contract and decentralized application platform. https://ethereum.org/en/whitepaper/, . (Acedido em 15/09/2020).

[3] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997. URL https://firstmonday.org/ojs/index.php/fm/article/view/548.

[4] G. Wood. Ethereum: A secure decentralised generalised transaction ledger petersburg version 3e2c089 – 2020-09-05. https://ethereum.github.io/yellowpaper/paper.pdf. (Acedido em 15/09/2020).

[5] Ethereum. Solidity, the contract-oriented programming language. https://github.com/ethereum/solidity, . (Acedido em 15/09/2020).

[6] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu. Smartbugs: A framework to analyze solidity smart contracts. *CoRR*, abs/2007.04771, 2020. URL https://arxiv.org/abs/2007.04771.

[7] A. Bahga and V. K. Madisetti. Blockchain platform for industrial internet of things. *Journal of Software Engineering and Applications*, 09(10):533–546, 01 2016. doi: 10.4236/jsea.2016.910036.

[8] R. Stortz. Rattle - an ethereum evm binary analysis framework. In reCON Montreal Conference, https://www.trailofbits.com/presentations/rattle/, 2018. (Acedido em 17/09/2020).

[9] N. Group. Decentralized application security project (or dasp) top 10. https://dasp.co/, 2018. (Acedido em 16/09/2020).

[10] S. Falkon. The story of the dao — its history and consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee. (Acedido em 16/09/2020).

[11] S. Marx. Stop using solidity's transfer() now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/. (Acedido em 16/09/2020).

[12] E. Rafaloff. Analyzing the erc20 short address attack. https://ericrafaloff.com/analyzing-the-erc20-short-address-attack/, 2017. (Acedido em 16/09/2020).

[13] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016. doi: 10.1145/2976749.2978309. URL https://doi.org/10.1145/2976749.2978309.

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

[15] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. URL http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf.

[16] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *CoRR*, abs/1802.06038, 2018. URL http://arxiv.org/abs/1802.06038.

[17] The parity wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/. (Acedido em 17/09/2020).

[18] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In M. Pinzger, G. Bavota, and A. Marcus, editors, *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 442–446. IEEE Computer Society, 2017. doi: 10.1109/SANER.2017.7884650. URL https://doi.org/10.1109/SANER.2017.7884650.

[19] C. F. Torres, J. Schütte, and R. State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 664–676. ACM, 12 2018. doi: 10.1145/3274694.3274737. URL https://doi.org/10.1145/3274694.3274737.

[20] J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In W. Enck and A. P. Felt, editors, *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.

[21] B. Mueller. Smashing ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 2018.

[22] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev. Securify: Practical security analysis of smart contracts. In D. Lie, M. Mannan, M. Backes, and X. Wang,

editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 67–82. ACM, 2018. doi: 10.1145/3243734.3243780. URL https://doi.org/10.1145/3243734.3243780.

[23] J. Feist, G. Grieco, and A. Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 8–15. IEEE / ACM, 2019. doi: 10.1109/WETSEB.2019.00008. URL https://doi.org/10.1109/WETSEB.2019.00008.

[24] J. Feist. Slithir, an intermediate representation of solidity to enable high precision security analysis. https://github.com/trailofbits/publications/tree/master/presentations/SlithIR, AnIntermediateRepresentationofSolidityтoenableHighPrecisionSecurityAnalysis, 2019. (Acedido em 17/09/2020).

[25] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1186–1189. IEEE, 2019. doi: 10.1109/ASE.2019.00133. URL https://doi.org/10.1109/ASE.2019.00133.

[26] Solidity. Yul. https://solidity.readthedocs.io/en/latest/yul.html. (Acedido em 17/09/2020).

[27] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In S. K. Lahiri and C. Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 513–520. Springer, 05 2018. doi: 10.1007/978-3-030-01090-4_30. URL https://doi.org/10.1007/978-3-030-01090-4_30.

[28] R. Stortz. Rattle - evm binary static analysis. https://github.com/crytic/rattle. (Acedido em 17/09/2020).

[29] D. Pérez and B. Livshits. Smart contract vulnerabilities: Does anyone care? *CoRR*, abs/1902.06710, 02 2019. URL http://arxiv.org/abs/1902.06710.

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

Apêndice A

Código Fonte

A.1 Instruções Suportadas pela Conkas

Na Tabela A.1 são apresentados os nomes de todas as instruções que a Conkas executa simbolicamente.

Instruções Executadas Simbolicamente			
STOP	NOT	RETURNDATACOPY	LOG0
ADD	BYTE	EXTCODEHASH	LOG1
MUL	SHL	BLOCKHASH	LOG2
SUB	SHR	COINBASE	LOG3
DIV	SAR	TIMESTAMP	LOG4
SDIV	SHA3	NUMBER	CREATE
MOD	ADDRESS	DIFFICULTY	CALL
SMOD	BALANCE	GASLIMIT	CALLCODE
ADDMOD	ORIGIN	MLOAD	RETURN
MULMOD	CALLER	MSTORE	DELEGATECALL
EXP	CALLVALUE	MSTORE8	CREATE2
SIGNEXTEND	CALLDATALOAD	SLOAD	STATICCALL
LT	CALLDATASIZE	SSTORE	REVERT
GT	CALLDATACOPY	JUMP	INVALID
SLT	CODESIZE	JUMPI	SELFDESTRUCT
SGT	CODECOPY	GETPC	CONDICALL
EQ	GASPRICE	MSIZE	ICALL
ISZERO	EXTCODESIZE	GAS	PHI
AND	EXTCODECOPY	JUMPDEST	SELFBALANCE
OR	RETURNDATASIZE	PUSH	CHAINID
XOR			

Tabela A.1: Todas as instruções que a Conkas suporta

A.2 *Parser* da Conkas

Nesta Secção é apresentado na Listagem 25 o *parser* da Conkas usado na framework SmartBugs que é responsável por realizar a normalização do *output* da Conkas.

```python
from src.output_parser.Parser import Parser
class Conkas(Parser):
    def __init__(self):
        pass
    @staticmethod
    def __parse_vuln_line(line):
        vuln_type = line.split('Vulnerability: ')[1].split('.')[0]
        maybe_in_function = line.split('Maybe in function: ')[1].split('.')[0]
        pc = line.split('PC: ')[1].split('.')[0]
        line_number = line.split('Line number: ')[1].split('.')[0]
        if vuln_type == 'Integer Overflow':
            vuln_type = 'Integer_Overflow'
        elif vuln_type == 'Integer Underflow':
            vuln_type = 'Integer_Underflow'
        return {
            'vuln_type': vuln_type,
            'maybe_in_function': maybe_in_function,
            'pc': pc,
            'line_number': line_number
        }
    def parse(self, str_output):
        output = []
        str_output = str_output.split('\n')
        for line in str_output:
            if 'Vulnerability' in line:
                try:
                    output.append(self.__parse_vuln_line(line))
                except:
                    continue
        return output
```

Listagem 25: *Parser* da Conkas

Apêndice B

Falsos Positivos

B.1 Falsos Positivos Detetados pelas Ferramentas Analisadas Manualmente

B.1.1 Conkas

Nesta Subsecção são apresentados, na Tabela B.1, os nomes de todos os ficheiros em que tenha sido detetado pelo menos um falso positivo e a categoria na qual foi detetado esse falso positivo, referente à ferramenta Conkas.

Categoria	Nome do Ficheiro	Nome do Ficheiro
Arithmetic	0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f	0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888
	0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e	0xbebbfe5b549f5db6e6c78ca97cac19d1fb03082c
	0x19cf8481ea15427a98ba3cdd6d9e14690011ab10	0xcead721ef5b11f1a7b530171aab69b16c5e66b6e
	0x23a91059fdc9579a9fbd0edc5f2ea0bfdb70deb4	0xd2018bfaa266a9ec0a1a84b061640faa009def76
	0x2972d548497286d18e92b5fa1f8f9139e5653fd2	0xd5967fed03e85d1cce44cab284695b41bc675b5c
	0x39cfd754c85023648bf003bea2dd498c5612abfa	0xdb1c55f6926e7d847ddf8678905ad871a68199d2
	0x3a0e9acd953ffc0dd18d63603488846a6b8b2b01	0xe09b1ab8111c2729a76f16de96bc86a7af837928
	0x3e013fc32a54c4c5b6991ba539dcd0ec4355c859	0xe4eabdca81e31d9acbc4af76b30f532b6ed7f3bf
	0x3f2ef511aa6e75231e4deafc7a3d2ecab3741de2	0xe82f0742a71a02b9e9ffc142fdcb6eb1ed06fb87
	0x4051334adc52057aca763453820cb0e045076ef3	0xe894d54dca59cb53fe9cbc5155093605c7068220
	0x4320e6f8c05b27ab4707cd1f6d5ce6f3e4b3a5a1	0xf015c35649c82f5467c9c74b7f28ee67665aad68
	0x4a66ad0bca2d700f11e1f2fc2c106f7d3264504c	0xf70d589d76eebdd7c12cc5eec99f8f6fa4233b9e
	0x4b71ad9c1a84b9b643aa54fdd66e2dec96e8b152	BECToken
	0x4e73b32ed6c35f570686b89848e5f39f20ecc106	FibonacciBalance
	0x561eac93c92360949ab1f1403323e6db345cbf31	FindThisHash
	0x5aa88d2901c68fda244f1d0584400368d2c8e739	arbitrary_location_write_simple
	0x610495793564aed0f9c7fc48dc4c7c9151d34fd6	blackjack
	0x627fa62ccbb1c1b04ffaecd72a53e37fc0e17839	dos_address
	0x663e4229142a27f00bafb5d087e1e730648314c3	dos_number
	0x70f9eddb3931491aab1aeafbc1e7f1ca2a012db4	ether_lotto
	0x7541b76cb60f4c60af330c208b0623b7f54bf615	etheraffle
	0x78c2a1e91b52bca4130b6ed9edd9fbcfd4671c37	etherpot_lotto

	0x7a4349a749e59a5736efb7826ee3496a2dfd5489	etherstore
	0x7a8721a9d64c74da899424c1b52acbf58ddc9782	governmental_survey
	0x7b368c4e805c3870b6c49a3f1f49f69af8662cf3	king_of_the_ether_throne
	0x7d09edb07d23acb532a82be3da5c17d9d85806b4	list_dos
	0x806a6bd219f162442d992bdc4ee6eba1f2c5a707	lottery
	0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	lucky_doubler
	0x89c1b3807d4c67df034fffb62f3509561218d30b	mapping_write
	0x8c7777c45481dba411450c228cb692ac3d550344	odds_and_evens
	0x8fd1e427396ddb511533cf9abdbebd0a7e08da35	old_blockhash
	0x93c32845fae42c83a70e5f06214c8433665c2ab5	proxy
	0x941d225236464a25eb18076df7da6a91d0f95e9e	reentrance
	0x958a8f594101d2c0485a52319f29b2647f2ebc06	reentrancy_bonus
	0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b	reentrancy_cross_function
	0x9d06cbafa865037a01d322d3f4222fa3e04e5488	reentrancy_dao
	0xa46edd6a9a93feec36576ee5048146870ea2c3ae	reentrancy_simple
	0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8	rubixi
	0xb0510d68f210b7db66e8c7c814f22680f2b8d1d6	short_address_example
	0xb11b2fed6c9354f7aa2f658d3b4d7b31d8a13b77	simple_dao
	0xb37f18af15bafb869a065b61fc83cfc44ed9cc27	spank_chain_payment
	0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12	timelock
	0xb620cee6b52f96f3c6b253e6eea556aa2d214a99	token
	0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7	wallet_02_refund_nosub
	0xb93430ce38ac4a6bb47fb1fc085ea669353fd89e	wallet_03_wrong_constructor
	0xbaa3de6504690efb064420d89e871c27065cdd52	wallet_04_confused_sign
	0xbaf51e761510c1a11bf48dd87c0307ac8a8c8a4f	
Front Running	0x89c1b3807d4c67df034fffb62f3509561218d30b	mishandled
	0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7	reen1
	0xe09b1ab8111c2729a76f16de96bc86a7af837928	reen2
	FibonacciBalance	reentrancy_bonus
	auction	reentrancy_cross_function
	ether_lotto	reentrancy_dao
	etherbank	reentrancy_insecure
	governmental_survey	reentrancy_simple
	king_of_the_ether_throne	rubixi
	list_dos	timelock
	lucky_doubler	wallet_02_refund_nosub
Reentrancy	0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f	0xbaa3de6504690efb064420d89e871c27065cdd52
	0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e	0xbaf51e761510c1a11bf48dd87c0307ac8a8c8a4f
	0x0cbe050f75bc8f8c2d6c0d249fea125fd6e1acc9	0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888
	0x19cf8481ea15427a98ba3cdd6d9e14690011ab10	0xbebbfe5b549f5db6e6c78ca97cac19d1fb03082c
	0x23a91059fdc9579a9fbd0edc5f2ea0bfdb70deb4	0xcead721ef5b11f1a7b530171aab69b16c5e66b6e
	0x2972d548497286d18e92b5fa1f8f9139e5653fd2	0xd2018bfaa266a9ec0a1a84b061640faa009def76
	0x39cfd754c85023648bf003bea2dd498c5612abfa	0xd5967fed03e85d1cce44cab284695b41bc675b5c
	0x3a0e9acd953ffc0dd18d63603488846a6b8b2b01	0xdb1c55f6926e7d847ddf8678905ad871a68199d2
	0x3e013fc32a54c4c5b6991ba539dcd0ec4355c859	0xe09b1ab8111c2729a76f16de96bc86a7af837928
	0x3f2ef511aa6e75231e4deafc7a3d2ecab3741de2	0xe4eabdca81e31d9acbc4af76b30f532b6ed7f3bf
	0x4051334adc52057aca763453820cb0e045076ef3	0xe82f0742a71a02b9e9ffc142fdcb6eb1ed06fb87
	0x4320e6f8c05b27ab4707cd1f6d5ce6f3e4b3a5a1	0xe894d54dca59cb53fe9cbc5155093605c7068220

	0x4a66ad0bca2d700f11e1f2fc2c106f7d3264504c	0xf015c35649c82f5467c9c74b7f28ee67665aad68
	0x4b71ad9c1a84b9b643aa54fdd66e2dec96e8b152	0xf2570186500a46986f3139f65afedc2afe4f445d
	0x4e73b32ed6c35f570686b89848e5f39f20ecc106	0xf29ebe930a539a60279ace72c707cba851a57707
	0x524960d55174d912768678d8c606b4d50b79d7b1	0xf70d589d76eebdd7c12cc5eec99f8f6fa4233b9e
	0x52d2e0f9b01101a59b38a3d05c80b7618aeed984	FibonacciBalance
	0x561eac93c92360949ab1f1403323e6db345cbf31	FindThisHash
	0x5aa88d2901c68fda244f1d0584400368d2c8e739	auction
	0x610495793564aed0f9c7fc48dc4c7c9151d34fd6	eth_tx_order_dependence_minimal
	0x627fa62ccbb1c1b04ffaecd72a53e37fc0e17839	ether_lotto
	0x663e4229142a27f00bafb5d087e1e730648314c3	etheraffle
	0x70f9eddb3931491aab1aeafbc1e7f1ca2a012db4	etherpot_lotto
	0x7541b76cb60f4c60af330c208b0623b7f54bf615	governmental_survey
	0x78c2a1e91b52bca4130b6ed9edd9fbcfd4671c37	guess_the_random_number
	0x7a4349a749e59a5736efb7826ee3496a2dfd5489	incorrect_constructor_name1
	0x7a8721a9d64c74da899424c1b52acbf58ddc9782	incorrect_constructor_name2
	0x7b368c4e805c3870b6c49a3f1f49f69af8662cf3	incorrect_constructor_name3
	0x7d09edb07d23acb532a82be3da5c17d9d85806b4	king_of_the_ether_throne
	0x806a6bd219f162442d992bdc4ee6eba1f2c5a707	list_dos
	0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	lottery
	0x89c1b3807d4c67df034fffb62f3509561218d30b	lotto
	0x8c7777c45481dba411450c228cb692ac3d550344	lottopollo
	0x8fd1e427396ddb511533cf9abdbebd0a7e08da35	lucky_doubler
	0x93c32845fae42c83a70e5f06214c8433665c2ab5	mapping_write
	0x941d225236464a25eb18076df7da6a91d0f95e9e	modifier_reentrancy
	0x958a8f594101d2c0485a52319f29b2647f2ebc06	multiowned_vulnerable
	0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b	mycontract
	0x9d06cbafa865037a01d322d3f4222fa3e04e5488	odds_and_evens
	0xa1fceeff3acc57d257b917e30c4df661401d6431	old_blockhash
	0xa46edd6a9a93feec36576ee5048146870ea2c3ae	phishable
	0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8	reentrancy_bonus
	0xb0510d68f210b7db66e8c7c814f22680f2b8d1d6	rubixi
	0xb11b2fed6c9354f7aa2f658d3b4d7b31d8a13b77	spank_chain_payment
	0xb37f18af15bafb869a065b61fc83cfc44ed9cc27	tokensalechallenge
	0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12	unchecked_return_value
	0xb620cee6b52f96f3c6b253e6eea556aa2d214a99	wallet_02_refund_nosub
	0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7	wallet_03_wrong_constructor
	0xb93430ce38ac4a6bb47fb1fc085ea669353fd89e	wallet_04_confused_sign
Time Manipulation	0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f	0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12
	0x23a91059fdc9579a9fbd0edc5f2ea0bfdb70deb4	0xb93430ce38ac4a6bb47fb1fc085ea669353fd89e
	0x4320e6f8c05b27ab4707cd1f6d5ce6f3e4b3a5a1	0xbaf51e761510c1a11bf48dd87c0307ac8a8c8a4f
	0x4e73b32ed6c35f570686b89848e5f39f20ecc106	0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888
	0x561eac93c92360949ab1f1403323e6db345cbf31	0xcead721ef5b11f1a7b530171aab69b16c5e66b6e
	0x663e4229142a27f00bafb5d087e1e730648314c3	0xf015c35649c82f5467c9c74b7f28ee67665aad68
	0x7541b76cb60f4c60af330c208b0623b7f54bf615	blackjack
	0x7a8721a9d64c74da899424c1b52acbf58ddc9782	etherstore
	0x7b368c4e805c3870b6c49a3f1f49f69af8662cf3	governmental_survey
	0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	king_of_the_ether_throne
	0x8c7777c45481dba411450c228cb692ac3d550344	list_dos
	0x93c32845fae42c83a70e5f06214c8433665c2ab5	lottopollo

	0x941d225236464a25eb18076df7da6a91d0f95e9e	roulette
	0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b	timelock
	0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8	
Unchecked	0x39cfd754c85023648bf003bea2dd498c5612abfa	governmental_survey
Low Calls	0x3a0e9acd953ffc0dd18d63603488846a6b8b2b01	list_dos
	0x627fa62ccbb1c1b04ffaecd72a53e37fc0e17839	lottopollo
	0x663e4229142a27f00bafb5d087e1e730648314c3	lucky_doubler
	0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	odds_and_evens
	0x8fd1e427396ddb511533cf9abdbebd0a7e08da35	reentrance
	0xb11b2fed6c9354f7aa2f658d3b4d7b31d8a13b77	rubixi
	0xbaa3de6504690efb064420d89e871c27065cdd52	simple_dao
	0xbebbfe5b549f5db6e6c78ca97cac19d1fb03082c	

Tabela B.1: Todos os nomes dos ficheiros que tenham um ou mais falsos positivos correspondentes à ferramenta Conkas

B.1.2 Mythril

Nesta Subsecção são apresentados, na Tabela B.2, os nomes de todos os ficheiros em que tenha sido detetado pelo menos um falso positivo e a categoria na qual foi detetado esse falso positivo, referente à ferramenta Mythril.

Categoria	Nome do Ficheiro	Nome do Ficheiro
Arithmetic	0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f	0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888
	0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e	0xbebbfe5b549f5db6e6c78ca97cac19d1fb03082c
	0x19cf8481ea15427a98ba3cdd6d9e14690011ab10	0xcead721ef5b11f1a7b530171aab69b16c5e66b6e
	0x23a91059fdc9579a9fbd0edc5f2ea0bfdb70deb4	0xd2018bfaa266a9ec0a1a84b061640faa009def76
	0x2972d548497286d18e92b5fa1f8f9139e5653fd2	0xd5967fed03e85d1cce44cab284695b41bc675b5c
	0x39cfd754c85023648bf003bea2dd498c5612abfa	0xdb1c55f6926e7d847ddf8678905ad871a68199d2
	0x3a0e9acd953ffc0dd18d63603488846a6b8b2b01	0xe09b1ab8111c2729a76f16de96bc86a7af837928
	0x3e013fc32a54c4c5b6991ba539dcd0ec4355c859	0xe4eabdca81e31d9acbc4af76b30f532b6ed7f3bf
	0x3f2ef511aa6e75231e4deafc7a3d2ecab3741de2	0xe82f0742a71a02b9e9ffc142fdcb6eb1ed06fb87
	0x4051334adc52057aca763453820cb0e045076ef3	0xe894d54dca59cb53fe9cbc5155093605c7068220
	0x4320e6f8c05b27ab4707cd1f6d5ce6f3e4b3a5a1	0xec329ffc97d75fe03428ae155fc7793431487f63
	0x4a66ad0bca2d700f11e1f2fc2c106f7d3264504c	0xf015c35649c82f5467c9c74b7f28ee67665aad68
	0x4b71ad9c1a84b9b643aa54fdd66e2dec96e8b152	0xf70d589d76eebdd7c12cc5eec99f8f6fa4233b9e
	0x4e73b32ed6c35f570686b89848e5f39f20ecc106	BECToken
	0x561eac93c92360949ab1f1403323e6db345cbf31	FibonacciBalance
	0x5aa88d2901c68fda244f1d0584400368d2c8e739	FindThisHash
	0x610495793564aed0f9c7fc48dc4c7c9151d34fd6	arbitrary_location_write_simple
	0x627fa62ccbb1c1b04ffaecd72a53e37fc0e17839	dos_address
	0x70f9eddb3931491aab1aeafbc1e7f1ca2a012db4	dos_number
	0x7541b76cb60f4c60af330c208b0623b7f54bf615	dos_simple
	0x78c2a1e91b52bca4130b6ed9edd9fbcfd4671c37	ether_lotto
	0x7a4349a749e59a5736efb7826ee3496a2dfd5489	etheraffle
	0x7a8721a9d64c74da899424c1b52acbf58ddc9782	etherpot_lotto
	0x7b368c4e805c3870b6c49a3f1f49f69af8662cf3	etherstore
	0x7d09edb07d23acb532a82be3da5c17d9d85806b4	king_of_the_ether_throne

	0x806a6bd219f162442d992bdc4ee6eba1f2c5a707	list_dos
	0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	lottery
	0x89c1b3807d4c67df034fffb62f3509561218d30b	mapping_write
	0x8c7777c45481dba411450c228cb692ac3d550344	odds_and_evens
	0x8fd1e427396ddb511533cf9abdbebd0a7e08da35	old_blockhash
	0x93c32845fae42c83a70e5f06214c8433665c2ab5	parity_wallet_bug_2
	0x941d225236464a25eb18076df7da6a91d0f95e9e	proxy
	0x958a8f594101d2c0485a52319f29b2647f2ebc06	reen1
	0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b	reen2
	0xa1fceeff3acc57d257b917e30c4df661401d6431	reentrance
	0xa46edd6a9a93feec36576ee5048146870ea2c3ae	reentrancy_cross_function
	0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8	reentrancy_dao
	0xb11b2fed6c9354f7aa2f658d3b4d7b31d8a13b77	reentrancy_simple
	0xb37f18af15bafb869a065b61fc83cfc44ed9cc27	rubixi
	0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12	short_address_example
	0xb620cee6b52f96f3c6b253e6eea556aa2d214a99	simple_dao
	0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7	spank_chain_payment
	0xb93430ce38ac4a6bb47fb1fc085ea669353fd89e	timelock
	0xbaa3de6504690efb064420d89e871c27065cdd52	token
	0xbaf51e761510c1a11bf48dd87c0307ac8a8c8a4f	wallet_04_confused_sign
Front Running	0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f	0xe4eabdca81e31d9acbc4af76b30f532b6ed7f3bf
	0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e	0xe82f0742a71a02b9e9ffc142fdcb6eb1ed06fb87
	0x3f2ef511aa6e75231e4deafc7a3d2ecab3741de2	0xf70d589d76eebdd7c12cc5eec99f8f6fa4233b9e
	0x4e73b32ed6c35f570686b89848e5f39f20ecc106	auction
	0x70f9eddb3931491aab1aeafbc1e7f1ca2a012db4	ether_lotto
	0x78c2a1e91b52bca4130b6ed9edd9fbcfd4671c37	etheraffle
	0x7a4349a749e59a5736efb7826ee3496a2dfd5489	governmental_survey
	0x7a8721a9d64c74da899424c1b52acbf58ddc9782	incorrect_constructor_name1
	0x7d09edb07d23acb532a82be3da5c17d9d85806b4	incorrect_constructor_name2
	0x806a6bd219f162442d992bdc4ee6eba1f2c5a707	incorrect_constructor_name3
	0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b	king_of_the_ether_throne
	0xb620cee6b52f96f3c6b253e6eea556aa2d214a99	list_dos
	0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888	lotto
	0xd2018bfaa266a9ec0a1a84b061640faa009def76	odds_and_evens
	0xdb1c55f6926e7d847ddf8678905ad871a68199d2	rubixi
Reentrancy	0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f	0xb0510d68f210b7db66e8c7c814f22680f2b8d1d6
	0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e	0xb11b2fed6c9354f7aa2f658d3b4d7b31d8a13b77
	0x0cbe050f75bc8f8c2d6c0d249fea125fd6e1acc9	0xb37f18af15bafb869a065b61fc83cfc44ed9cc27
	0x23a91059fdc9579a9fbd0edc5f2ea0bfdb70deb4	0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12
	0x2972d548497286d18e92b5fa1f8f9139e5653fd2	0xb620cee6b52f96f3c6b253e6eea556aa2d214a99
	0x39cfd754c85023648bf003bea2dd498c5612abfa	0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7
	0x3a0e9acd953ffc0dd18d63603488846a6b8b2b01	0xb93430ce38ac4a6bb47fb1fc085ea669353fd89e
	0x3e013fc32a54c4c5b6991ba539dcd0ec4355c859	0xbaa3de6504690efb064420d89e871c27065cdd52
	0x3f2ef511aa6e75231e4deafc7a3d2ecab3741de2	0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888
	0x4051334adc52057aca763453820cb0e045076ef3	0xbebbfe5b549f5db6e6c78ca97cac19d1fb03082c
	0x4a66ad0bca2d700f11e1f2fc2c106f7d3264504c	0xcead721ef5b11f1a7b530171aab69b16c5e66b6e
	0x4b71ad9c1a84b9b643aa54fdd66e2dec96e8b152	0xd2018bfaa266a9ec0a1a84b061640faa009def76
	0x4e73b32ed6c35f570686b89848e5f39f20ecc106	0xd5967fed03e85d1cce44cab284695b41bc675b5c

	0x524960d55174d912768678d8c606b4d50b79d7b1	0xdb1c55f6926e7d847ddf8678905ad871a68199d2
	0x52d2e0f9b01101a59b38a3d05c80b7618aeed984	0xe4eabdca81e31d9acbc4af76b30f532b6ed7f3bf
	0x5aa88d2901c68fda244f1d0584400368d2c8e739	0xe82f0742a71a02b9e9ffc142fdcb6eb1ed06fb87
	0x610495793564aed0f9c7fc48dc4c7c9151d34fd6	0xe894d54dca59cb53fe9cbc5155093605c7068220
	0x627fa62ccbb1c1b04ffaecd72a53e37fc0e17839	0xec329ffc97d75fe03428ae155fc7793431487f63
	0x663e4229142a27f00bafb5d087e1e730648314c3	0xf015c35649c82f5467c9c74b7f28ee67665aad68
	0x70f9eddb3931491aab1aeafbc1e7f1ca2a012db4	0xf2570186500a46986f3139f65afedc2afe4f445d
	0x7541b76cb60f4c60af330c208b0623b7f54bf615	0xf29ebe930a539a60279ace72c707cba851a57707
	0x78c2a1e91b52bca4130b6ed9edd9fbcfd4671c37	0xf70d589d76eebdd7c12cc5eec99f8f6fa4233b9e
	0x7a4349a749e59a5736efb7826ee3496a2dfd5489	etherbank
	0x7a8721a9d64c74da899424c1b52acbf58ddc9782	etherstore
	0x7b368c4e805c3870b6c49a3f1f49f69af8662cf3	modifier_reentrancy
	0x7d09edb07d23acb532a82be3da5c17d9d85806b4	parity_wallet_bug_2
	0x806a6bd219f162442d992bdc4ee6eba1f2c5a707	reen1
	0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	reen2
	0x89c1b3807d4c67df034fffb62f3509561218d30b	reentrance
	0x8fd1e427396ddb511533cf9abdbebd0a7e08da35	reentrancy_bonus
	0x93c32845fae42c83a70e5f06214c8433665c2ab5	reentrancy_cross_function
	0x958a8f594101d2c0485a52319f29b2647f2ebc06	reentrancy_dao
	0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b	reentrancy_insecure
	0x9d06cbafa865037a01d322d3f4222fa3e04e5488	reentrancy_simple
	0xa1fceeff3acc57d257b917e30c4df661401d6431	simple_dao
	0xa46edd6a9a93feec36576ee5048146870ea2c3ae	unchecked_return_value
Time Manipulation	-	
Unchecked	0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e	list_dos
Low Calls	0x19cf8481ea15427a98ba3cdd6d9e14690011ab10	lottopollo
	0x663e4229142a27f00bafb5d087e1e730648314c3	lucky_doubler
	0x7d09edb07d23acb532a82be3da5c17d9d85806b4	odds_and_evens
	0x89c1b3807d4c67df034fffb62f3509561218d30b	reentrance
	0xb620cee6b52f96f3c6b253e6eea556aa2d214a99	rubixi
	governmental_survey	simple_dao

Tabela B.2: Todos os nomes dos ficheiros que tenham um ou mais falsos positivos correspondentes à ferramenta Mythril

B.1.3 Slither

Nesta Subsecção são apresentados, na Tabela B.3, os nomes de todos os ficheiros em que tenha sido detetado pelo menos um falso positivo e a categoria na qual foi detetado esse falso positivo, referente à ferramenta Slither.

Categoria	Nome do Ficheiro	Nome do Ficheiro
Arithmetic	-	
Front Running	-	
Reentrancy	0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e	0x941d225236464a25eb18076df7da6a91d0f95e9e
	0x19cf8481ea15427a98ba3cdd6d9e14690011ab10	0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7
	0x663e4229142a27f00bafb5d087e1e730648314c3	0xe09b1ab8111c2729a76f16de96bc86a7af837928

	0x7d09edb07d23acb532a82be3da5c17d9d85806b4	parity_wallet_bug_2
	0x89c1b3807d4c67df034fffb62f3509561218d30b	spank_chain_payment
Time Manipulation	king_of_the_ether_throne	list_dos
Unchecked Low Calls	0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f	0xbaf51e761510c1a11bf48dd87c0307ac8a8c8a4f
	0x19cf8481ea15427a98ba3cdd6d9e14690011ab10	0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888
	0x23a91059fdc9579a9fbd0edc5f2ea0bfdb70deb4	0xcead721ef5b11f1a7b530171aab69b16c5e66b6e
	0x4320e6f8c05b27ab4707cd1f6d5ce6f3e4b3a5a1	0xf015c35649c82f5467c9c74b7f28ee67665aad68
	0x4e73b32ed6c35f570686b89848e5f39f20ecc106	FibonacciBalance
	0x52d2e0f9b01101a59b38a3d05c80b7618aeed984	etherbank
	0x561eac93c92360949ab1f1403323e6db345cbf31	etherstore
	0x627fa62ccbb1c1b04ffaecd72a53e37fc0e17839	parity_wallet_bug_2
	0x663e4229142a27f00bafb5d087e1e730648314c3	proxy
	0x7541b76cb60f4c60af330c208b0623b7f54bf615	reen1
	0x7a8721a9d64c74da899424c1b52acbf58ddc9782	reen2
	0x7b368c4e805c3870b6c49a3f1f49f69af8662cf3	reentrance
	0x89c1b3807d4c67df034fffb62f3509561218d30b	reentrancy_bonus
	0x8c7777c45481dba411450c228cb692ac3d550344	reentrancy_cross_function
	0x93c32845fae42c83a70e5f06214c8433665c2ab5	reentrancy_dao
	0x941d225236464a25eb18076df7da6a91d0f95e9e	reentrancy_insecure
	0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b	reentrancy_simple
	0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8	simple_dao
	0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12	spank_chain_payment
	0xb93430ce38ac4a6bb47fb1fc085ea669353fd89e	unchecked_return_value

Tabela B.3: Todos os nomes dos ficheiros que tenham um ou mais falsos positivos correspondentes à ferramenta Slither

B.1.4 Smartcheck

Nesta Subsecção são apresentados, na Tabela B.4, os nomes de todos os ficheiros em que tenha sido detetado pelo menos um falso positivo e a categoria na qual foi detetado esse falso positivo, referente à ferramenta Smartcheck.

Categoria	Nome do Ficheiro	Nome do Ficheiro
Arithmetic	0xb620cee6b52f96f3c6b253e6eea556aa2d214a99	king_of_the_ether_throne
	0xec329ffc97d75fe03428ae155fc7793431487f63	lucky_doubler
	arbitrary_location_write_simple	mapping_write
	dos_number	parity_wallet_bug_1
	ether_lotto	parity_wallet_bug_2
	etheraffle	rubixi
	etherpot_lotto	smart_billions
Front Running	-	
Reentrancy	0x0cbe050f75bc8f8c2d6c0d249fea125fd6e1acc9	0xb620cee6b52f96f3c6b253e6eea556aa2d214a99
	0x52d2e0f9b01101a59b38a3d05c80b7618aeed984	0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7
	0x89c1b3807d4c67df034fffb62f3509561218d30b	0xf29ebe930a539a60279ace72c707cba851a57707
	0xb0510d68f210b7db66e8c7c814f22680f2b8d1d6	reentrancy_bonus

	0xb37f18af15bafb869a065b61fc83cfc44ed9cc27	unchecked_return_value
Time Manipulation	-	
Unchecked Low Calls	0xe09b1ab8111c2729a76f16de96bc86a7af837928	reentrancy_cross_function
	auction	reentrancy_insecure
	governmental_survey	rubixi
	lottery	send_loop
	reentrancy_bonus	

Tabela B.4: Todos os nomes dos ficheiros que tenham um ou mais falsos positivos correspondentes à ferramenta Smartcheck

B.2 Resultado da Verificação Manual

Nesta Secção são apresentados os nomes de todos os ficheiros que tenham pelo menos um falso positivo identificado pela SmartBugs e que foram selecionados para verificar se realmente são falsos positivos. Na Tabela B.5 são apresentados os resultados da verificação manual relativamente à ferramenta Conkas. Cada linha da tabela refere-se a um dado ficheiro, sendo também apresentado na coluna Verdadeiros Positivos o número da linha onde foi detetado manualmente um verdadeiro positivo ao invés de um falso positivo e é apresentado na coluna Falsos Positivos o número da linha onde foi detetado manualmente um falso positivo. O mesmo acontece nas Tabelas B.6, B.7, B.8 e B.9 referentes às categorias *Front Running*, *Reentrancy*, *Time Manipulation* e *Unchecked Low Calls* respetivamente, considerando apenas a ferramenta Conkas. O mesmo acontece para as ferramentas Mythril, Slither e Smartcheck. Para a ferramenta Mythril, as Tabelas B.10, B.11, B.12 e B.13 são correspondentes às categorias *Arithmetic*, *Front Running*, *Reentrancy* e *Unchecked Low Calls* respetivamente. Para a ferramenta Slither, as Tabelas B.14 e B.15 correspondem às categorias *Reentrancy* e *Unchecked Low Calls* respetivamente. Por fim, para a ferramenta Smartcheck, as Tabelas B.16, B.17 e B.18 são correspondentes às categorias *Arithmetic*, *Reentrancy* e *Unchecked Low Calls* respetivamente. Há categorias em falta nas ferramentas que não a Conkas, porque não houveram ficheiros selecionados para a verificação manual, ou então a ferramenta não deteta essas categorias, ou não detetou nenhum falso positivo nas categorias em causa, como apresentado na Secção B.1.

Conkas		
Arithmetic		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x561eac93c92360949ab1f1403323e6db345cbf31	43	73,87,94
0x663e4229142a27f00bafb5d087e1e730648314c3	392,397,697,1386,1818, 1824,1825	385,516,522,526-528 907,984,985,1798, 1800,1808,2429,2432-2441
0x806a6bd219f162442d992bdc4ee6eba1f2c5a707		38
0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8		54,62,71,87,88
0x8fd1e427396ddb511533cf9abdbebd0a7e08da35	72	
0x93c32845fae42c83a70e5f06214c8433665c2ab5	16	62,76,83
0x941d225236464a25eb18076df7da6a91d0f95e9e	31	56,71,78
0xa46edd6a9a93feec36576ee5048146870ea2c3ae		11
0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8	43	72,86,93
0xb620cee6b52f96f3c6b253e6eea556aa2d214a99	118	27,28,73,86,110
0xcead721ef5b11f1a7b530171aab69b16c5e66b6e	16	62,76,83
0xdb1c55f6926e7d847ddf8678905ad871a68199d2		33
FibonacciBalance	32,59	38
old_blockhash	29	
reentrance	14	
token	23	
wallet_04_confused_sign	24,32	

Tabela B.5: Verdadeiros positivos e falsos positivos referentes à categoria *Arithmetic* da ferramenta Conkas

Conkas		
Front Running		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0xe09b1ab8111c2729a76f16de96bc86a7af837928	313	
FibonacciBalance	32	
auction		23
ether_lotto	52	
etherbank		21
governmental_survey	34	
lucky_doubler		119
mishandled		14
reen2		19
reentrancy_insecure		17
rubixi	85,103	
wallet_02_refund_nosub		36

Tabela B.6: Verdadeiros positivos e falsos positivos referentes à categoria *Front Running* da ferramenta Conkas

Conkas		
Reentrancy		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x0cbe050f75bc8f8c2d6c0d249fea125fd6e1acc9	12	
0x561eac93c92360949ab1f1403323e6db345cbf31		44,57
0x663e4229142a27f00bafb5d087e1e730648314c3	1496	819,1205,1214,1223,1961, 2058,2199
0x806a6bd219f162442d992bdc4ee6eba1f2c5a707	26,35,44	
0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	56	
0x8fd1e427396ddb511533cf9abdbebd0a7e08da35	44	
0x93c32845fae42c83a70e5f06214c8433665c2ab5		18,32
0x941d225236464a25eb18076df7da6a91d0f95e9e		32,47
0xa1fceeff3acc57d257b917e30c4df661401d6431	31	
0xa46edd6a9a93feec36576ee5048146870ea2c3ae	16	
0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8		44,57
0xb620cee6b52f96f3c6b253e6eea556aa2d214a99	81,100,106,133,137	
0xcead721ef5b11f1a7b530171aab69b16c5e66b6e		18,32
0xdb1c55f6926e7d847ddf8678905ad871a68199d2	21,30,39	
FibonacciBalance	32	
old_blockhash	39	
wallet_04_confused_sign	31,39	

Tabela B.7: Verdadeiros positivos e falsos positivos referentes à categoria *Reentrancy* da ferramenta Conkas

Conkas		
Time Manipulation		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x561eac93c92360949ab1f1403323e6db345cbf31	91	94
0x663e4229142a27f00bafb5d087e1e730648314c3	1343,1374	1235,1269
0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	88	
0x93c32845fae42c83a70e5f06214c8433665c2ab5	17,80	83
0x941d225236464a25eb18076df7da6a91d0f95e9e	75	78
0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8	90	93
0xcead721ef5b11f1a7b530171aab69b16c5e66b6e	17,80	83

Tabela B.8: Verdadeiros positivos e falsos positivos referentes à categoria *Time Manipulation* da ferramenta Conkas

Conkas		
Unchecked Low Calls		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x663e4229142a27f00bafb5d087e1e730648314c3 0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8	1496 56	
0x8fd1e427396ddb511533cf9abdbebd0a7e08da35 0xb11b2fed6c9354f7aa2f658d3b4d7b31d8a13b77	44 14	
governmental_survey odds_and_evens	34,35 38,41,50	
reentrance	24	

Tabela B.9: Verdadeiros positivos e falsos positivos referentes à categoria *Unchecked Low Calls* da ferramenta Conkas

Mythril		
Arithmetic		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x561eac93c92360949ab1f1403323e6db345cbf31 0x806a6bd219f162442d992bdc4ee6eba1f2c5a707		2,78,90 38
0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8 0x8fd1e427396ddb511533cf9abdbebd0a7e08da35		2,54,64,87,88 76
0x93c32845fae42c83a70e5f06214c8433665c2ab5 0x941d225236464a25eb18076df7da6a91d0f95e9e	16	 2,61,73
0xa1fceeff3acc57d257b917e30c4df661401d6431 0xa46edd6a9a93feec36576ee5048146870ea2c3ae		20 11
0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8 0xb620cee6b52f96f3c6b253e6eea556aa2d214a99		2,77,89 2
0xcead721ef5b11f1a7b530171aab69b16c5e66b6e 0xdb1c55f6926e7d847ddf8678905ad871a68199d2	16	 33
0xec329ffc97d75fe03428ae155fc7793431487f63 FibonacciBalance		28,40,68,74,84 60,63
old_blockhash reentrance	 14	32
token wallet_04_confused_sign	23	 32

Tabela B.10: Verdadeiros positivos e falsos positivos referentes à categoria *Arithmetic* da ferramenta Mythril

Mythril		
Front Running		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x806a6bd219f162442d992bdc4ee6eba1f2c5a707		25,36
0xb620cee6b52f96f3c6b253e6eea556aa2d214a99		140,144
0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888		53,68
0xd2018bfaa266a9ec0a1a84b061640faa009def76		25,36
0xdb1c55f6926e7d847ddf8678905ad871a68199d2	30	20
ether_lotto		54

Tabela B.11: Verdadeiros positivos e falsos positivos referentes à categoria *Front Running* da ferramenta Mythril

Mythril		
Reentrancy		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x0cbe050f75bc8f8c2d6c0d249fea125fd6e1acc9		13
0x663e4229142a27f00bafb5d087e1e730648314c3		1245,1269
0x806a6bd219f162442d992bdc4ee6eba1f2c5a707		46
0x84d9ec85c9c568eb332b7226a8f826d897e0a4a8		60
0x8fd1e427396ddb511533cf9abdbebd0a7e08da35		46
0x93c32845fae42c83a70e5f06214c8433665c2ab5		18,32
0xa1fceeff3acc57d257b917e30c4df661401d6431	31	
0xa46edd6a9a93feec36576ee5048146870ea2c3ae	16	
0xb620cee6b52f96f3c6b253e6eea556aa2d214a99		104,109,140
0xcead721ef5b11f1a7b530171aab69b16c5e66b6e		18,32
0xdb1c55f6926e7d847ddf8678905ad871a68199d2		40
0xec329ffc97d75fe03428ae155fc7793431487f63	30	98,116
reentrance		27
reentrancy_insecure		19

Tabela B.12: Verdadeiros positivos e falsos positivos referentes à categoria *Reentrancy* da ferramenta Mythril

Mythril		
Unchecked Low Calls		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x663e4229142a27f00bafb5d087e1e730648314c3		1536
0xb620cee6b52f96f3c6b253e6eea556aa2d214a99		140
governmental_survey	35	36
lucky_doubler		64,76,104,125
odds_and_evens		39,42,54
reentrance	24	
rubixi		76,89,99,107
simple_dao	19	

Tabela B.13: Verdadeiros positivos e falsos positivos referentes à categoria *Unchecked Low Calls* da ferramenta Mythril

Slither		
Reentrancy		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x663e4229142a27f00bafb5d087e1e730648314c3	1117-1197	428-515,428-515,829-838,1025-1046, 1025-1046,1054-1103,1054-1103,1117-1197, 1117-1197,1507-1518,1551-1577,1688-1714, 1771-1801,1801-1829,1840-1849,1929-1958, 1983-1995,1996-20031,2012-2057,2077-2086,
0x941d225236464a25eb18076df7da6a91d0f95e9e	38-55	2086-2107,2109-2117,2228-2251,2228-2251 26-37

Tabela B.14: Verdadeiros positivos e falsos positivos referentes à categoria *Reentrancy* da ferramenta Slither

Slither		
Unchecked Low Calls		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x561eac93c92360949ab1f1403323e6db345cbf31		49-66
0x663e4229142a27f00bafb5d087e1e730648314c3		2003-2007,2260-2264
0x93c32845fae42c83a70e5f06214c8433665c2ab5		21-39
0x941d225236464a25eb18076df7da6a91d0f95e9e		38-55
0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8		49-66
0xcead721ef5b11f1a7b530171aab69b16c5e66b6e		21-39
FibonacciBalance		27-35,37-42
reentrance		21-31
reentrancy_insecure		14-21

Tabela B.15: Verdadeiros positivos e falsos positivos referentes à categoria *Unchecked Low Calls* da ferramenta Slither

Smartcheck		
Arithmetic		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0xb620cee6b52f96f3c6b253e6eea556aa2d214a99		114
0xec329ffc97d75fe03428ae155fc7793431487f63	73	
arbitrary_location_write_simple	28	
ether_lotto		43
etheraffle		76,159
parity_wallet_bug_1	304	
smart_billions	416	676,679

Tabela B.16: Verdadeiros positivos e falsos positivos referentes à categoria *Arithmetic* da ferramenta Smartcheck

Smartcheck		
Reentrancy		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0x0cbe050f75bc8f8c2d6c0d249fea125fd6e1acc9 0xb37f18af15bafb869a065b61fc83cfc44ed9cc27	12 33	
0xb620cee6b52f96f3c6b253e6eea556aa2d214a99 0xb7c5c5aa4d42967efe906e1b66cb8df9cebf04f7	100,106,133 25	
0xf29ebe930a539a60279ace72c707cba851a57707 reentrancy_bonus		16 19
unchecked_return_value		12,17

Tabela B.17: Verdadeiros positivos e falsos positivos referentes à categoria *Reentrancy* da ferramenta Smartcheck

Smartcheck		
Unchecked Low Calls		
Nome do Ficheiro	Verdadeiros Positivos	Falsos Positivos
0xe09b1ab8111c2729a76f16de96bc86a7af837928 auction		235 23
governmental_survey lottery	34,35	 46
reentrancy_bonus reentrancy_insecure		19 17
rubixi	74,85,95,103	

Tabela B.18: Verdadeiros positivos e falsos positivos referentes à categoria *Unchecked Low Calls* da ferramenta Smartcheck