UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

FABIANO RODRIGO ALVES NASCIMENTO

GERAÇÃO AUTOMÁTICA DE CASOS DE TESTE PARA CONTRATOS INTELIGENTES DA REDE ETHEREUM

Rio de Janeiro
Dezembro de 2020

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

FABIANO RODRIGO ALVES NASCIMENTO

GERAÇÃO AUTOMÁTICA DE CASOS DE TESTE PARA CONTRATOS INTELIGENTES DA REDE ETHEREUM

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Orientador: Márcio de Oliveira Barros

Rio de Janeiro
Dezembro de 2020

CBIB Nascimento, Fabiano Rodrigo Alves

Geração automática de casos de teste para contratos inteligentes da rede Ethereum / Fabiano Rodrigo Alves Nascimento. – Dezembro de 2020.
121 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Estado do Rio de Janeiro. Programa de Pós-Graduação em Informática, Rio de Janeiro, BR–RJ, Dezembro de 2020.

Orientador: Márcio de Oliveira Barros.

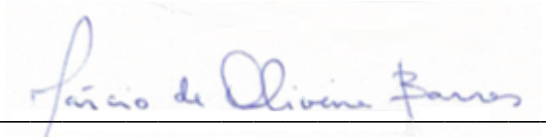
1. Blockchain. 2. Ethereum. 3. Solidity. 4. Truffle. 5. Testes. 6. Automação. – Teses. I. Barros, Márcio de Oliveira (Orient.). II. Universidade Federal do Estado do Rio de Janeiro, Centro de Ciências Exatas e Tecnologia, Programa de Pós-Graduação em Informática. III. Título

CDD

GERAÇÃO AUTOMÁTICA DE CASOS DE TESTE PARA CONTRATOS INTELIGENTES DA REDE ETHEREUM

DISSERTAÇÃO APRESENTADA COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE PELO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO (UNIRIO). APROVADA PELA COMISSÃO EXAMINADORA ABAIXO ASSINADA.

Aprovado por



Prof. Dr. Márcio de Oliveira Barros (Orientador)

Prof. Dr. Rodrigo Pereira dos Santos

Profa. Dra. Vânia de Oliveira Neves

Rio de Janeiro
Dezembro de 2020

AGRADECIMENTOS

Em primeiro lugar, preciso agradecer a uma mulher. Uma mulher que teve que enfrentar batalhas desde muito cedo para correr atrás da própria educação, fazer escolhas difíceis como entre ficar na tranquilidade do seio familiar em uma rocinha lá no interior das Minas Gerais ou sair cedo da casa dos pais em busca do saber, mas a custo de se passar por muitas dificuldades. As pessoas a chamam de Paula, eu a chamo de mãe.

Dona Paula é minha base, minha inspiração, é um alicerce sem o qual eu jamais poderia construir coisa alguma. É o meu maior exemplo. A renda familiar não lhe permitia pagar uma boa educação aos filhos. Ela então madrugava nas filas de matrícula daquelas que seriam as melhores escolas públicas possíveis para que pudesse cumprir a missão que se imputara: dar uma boa educação aos filhos.

Quando iniciei o segundo grau, ela começou a trabalhar em uma Universidade da nossa cidade. Contribuía com o sindicato já pensando em ter acesso a bolsa de estudos para quando chegasse a hora de eu entrar na faculdade. Contudo, quando passei no vestibular, o benefício da bolsa já não era tão relevante quanto antes. Ainda que reuníssemos os rendimentos de todos os membros da família, não seria suficiente. Ela não pensou duas vezes, pegou a caneta e escreveu de próprio punho uma carta ao reitor. Naquelas folhas, misturadas às palavras, ela colocou a frustração de um sonho que lhe fora arrancado, a esperança de virar uma situação que já parecia perdida, a crença na educação como instrumento de transformação e, sem dúvida alguma, o amor pelos seus filhos. E ... deu certo! Conseguimos uma bolsa que, apesar de muita dificuldade, viabilizou meu ensino superior. Por essas e tantas outras, que eu não posso nunca me esquecer de associar esta mulher que Deus colocou na minha vida a cada uma das minhas conquistas.

Ainda no meu núcleo familiar, gostaria de agradecer também ao meu irmão, Kassius. Se tem uma certeza que sempre carrego comigo, é que, nas horas mais erradas, o Katisoca é o cara mais certo para estar ao seu lado. É companheiro, é parceiro, e vai contigo para o que der e vier. Nesse ano tão atípico com essa pandemia, ele teve um papel fundamental para que eu conseguisse passar por esse período tão difícil de uma forma menos traumática. Foi meu apoio, minha companhia, um refúgio no meio de tudo isso.

Finalmente, agradeço a cada um dos meus professores ao longo de toda minha jornada de aprendizado. Desde a minha alfabetização nos primeiros anos de vida até este ponto mais alto da minha vida acadêmica. Representando muito bem todos esses mestres, agradeço nominalmente ao meu orientador Márcio. Além da notável generosidade de compartilhar conhecimento, um dom nato dos educadores, o Márcio foi um orientador muito presente! Sempre atuando, contribuindo e me puxando para elevar o nível desta pesquisa. E quando as coisas não saíam exatamente como planejado ou o cansaço batia, o Márcio sempre tinha uma palavra de incentivo e encorajamento.

RESUMO

Nascimento, Fabiano Rodrigo Alves. **Geração automática de casos de teste para contratos inteligentes da rede Ethereum**. Dezembro de 2020. 109 f. Dissertação (Mestrado em Informática) - PPGI, Departamento de Informática Aplicada, Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro, Dezembro de 2020.

A execução de contratos inteligentes em uma rede *blockchain* vem acompanhada de um grande potencial de viabilizar negócios e políticas públicas de forma eficiente e inovadora. Entretanto, o desenvolvimento de software em um ambiente que tem como uma de suas características principais a imutabilidade traz desafios ainda maiores ao processo de garantia e controle de qualidade. Diversas pesquisas nos últimos anos abordam a automação da atividade de testes de contratos inteligentes, mas focam majoritariamente na identificação de problemas de vulnerabilidades previamente conhecidos. Há, portanto, uma lacuna na literatura acerca da automação da geração de testes automatizados voltados à identificação de defeitos de natureza semântica. O objetivo deste estudo é preencher esta lacuna abordando a geração automática de testes automatizados com base nas expressões relacionais encontradas no código-fonte dos contratos inteligentes. Uma análise experimental foi conduzida utilizando-se um universo de 40 projetos Solidity disponíveis na plataforma GitHub. Entre outros resultados, ao se comparar os índices de cobertura de ramos alcançados pela proposta deste estudo contra os obtidos pelos testes automatizados encontrados originalmente nos projetos selecionados, pode-se observar que a cobertura dos testes gerados de forma automatizada foi superior na maioria dos projetos.

Palavras-chave: Blockchain, Ethereum, Solidity, Truffle, Testes, Automação.

ABSTRACT

Nascimento, Fabiano Rodrigo Alves. **Geração automática de casos de teste para contratos inteligentes da rede Ethereum**. Dezembro de 2020. 109 f. Dissertação (Mestrado em Informática) - PPGI, Departamento de Informática Aplicada, Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro, Dezembro de 2020.

Running smart contracts in a *blockchain* network comes with a great potential to enable business and public policies in an efficient and innovative way. However, software development in an environment that has immutability as one of its main characteristics brings even greater challenges to the quality assurance and control process. Several researches in recent years have addressed the automation of smart contract testing activity, but mostly focus on identifying previously known vulnerability issues. There is, therefore, a gap in the literature about the automation of the generation of automated tests aimed at identifying defects of semantic nature. The purpose of this study is to fill this gap by addressing the automatic generation of automated tests based on the relational expressions found in the source code of smart contracts. An experimental analysis was conducted using a set of 40 Solidity projects available on the GitHub platform. Among other results, when comparing the branch coverage rates achieved by the proposal of this study against those obtained by the automated tests originally found in the selected projects, it can be observed that the coverage of the tests generated in an automated way was higher in most projects.

Keywords: Blockchain, Ethereum, Solidity, Truffle, Tests, Automation.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contexto

A crise financeira mundial de 2008, que foi precedida por uma bolha no mercado imobiliário residencial americano, desencadeou em um período de estagnação econômica em diversos países ao redor do globo. A avaliação subestimada de risco dos complexos produtos financeiros sintéticos no cerne desta crise expôs a fragilidade do sistema financeiro mundial, incluindo não somente bancos e demais instituições financeiras do mercado privado, mas também os governos, suas agências reguladoras e as agências internacionais de avaliação de crédito (BENMELECH; DLUGOSZ, 2010).

Em um contexto de imensa desconfiança sobre o sistema financeiro e seus agentes, nasce a criptomoeda Bitcoin (BALDWIN, 2018), a primeira e, no momento de publicação desta pesquisa, a mais popular aplicação da tecnologia *blockchain*. Sua proposta é a eliminação de intermediários nas transações financeiras, substituindo a confiança em instituições, que já haviam provado não ser tão confiáveis como se pensara, por um protocolo computacional baseado em desafios criptográficos e realizado de forma descentralizada.

Desde a mineração do primeiro bloco da rede Bitcoin, no início de 2009, surgiram diversas outras criptomoedas (COINMARKETCAP, 2020). Além disso, emergiu também uma segunda geração da tecnologia *blockchain* com a proposta de se expandir as fronteiras para além das transações financeiras com criptomoedas, sendo a plataforma Ethereum a mais notável representante desta segunda geração. Desde já, é importante esclarecer que, ao referenciarmos a Ethereum como uma "plataforma" ao longo do texto, estamos tratando de todo o arcabouço que a acompanha. Este arcabouço inclui sua es-

pecificação, a criptomoeda Ether, as linguagens de programação e seus respectivos compiladores, as implementações do software cliente[1] (e.g. Geth e OpenEthereum), e toda variedade de ferramentas para acesso aos serviços de uma rede Ethereum (e.g. Metamask, MyEtherWallet, web3). Por sua vez, ao referenciarmos a Ethereum ou qualquer outra implementação da tecnologia *blockchain* como "rede", estamos tratando de conjunto de instâncias de software cliente em execução, conectadas entre si e aplicando os protocolos estabelecidos na especificação da respectiva plataforma. Nos casos em que o termo "rede" vier precedido do artigo definido, estamos referenciando à *mainnet*[2] da respectiva plataforma.

A plataforma Ethereum nasce com a proposta de uso das transações seguras (com base em criptografia) e do modelo de computação descentralizada inerente à tecnologia *blockchain*, porém, de uma maneira generalista. Isto é, as transações e dados compartilhados entre os nós não se limitam à representação de uma moeda e à transferência de unidades e frações desta moeda entre as contas de usuários da rede. A Ethereum oferece uma linguagem de programação Turing-completa integrada, tornando possível a implementação de algoritmos para execução dentro de suas redes. Estes algoritmos são chamados de *contratos inteligentes*. Eles têm suas próprias regras de propriedade de ativos, formatos de transação e funções de transição de estado (WOOD et al., 2019) e com todas as características próprias da tecnologia *blockchain*.

Em uma rede *blockchain* de plataformas que implementem recursos como os propostos pela plataforma Ethereum, passa-se a ter software executando no âmbito de rede, implementando regras de negócio e requisitos técnicos em suas linhas de código. É importante lembrar que os contratos inteligentes têm um diferencial em relação ao software convencional: eles podem ser responsáveis não apenas por lidar com transações de mon-

[1] Cada instância em execução de um software cliente da rede Ethereum será um nó em uma rede Ethereum

[2] Toda plataforma *blockchain* tem uma rede principal. Trata-se da rede onde as transações ocorrem e a criptomoeda possui um valor monetário real. Por outro lado, pode haver diversas redes alternativas, que são usadas para fins de teste e onde as transações não têm um valor monetário real

tantes significativos em criptoativos, mas também por carregar em si próprios tais ativos. Falhas de implementação podem causar prejuízos financeiros vultuosos. Quando aplicado a cenários alheios a transações financeiras ou registro de bens, como um processo eleitoral, por exemplo, os prejuízos podem ser imensuráveis.

A capacidade de implementar e executar software em uma rede *blockchain* vem acompanhada de questões intrínsecas à produção de software. Entre estas questões estão não apenas os desafios, mas também todo o conhecimento gerado e a experiência acumulada ao longo de décadas na área de Engenharia de Software. Esse conhecimento abrange processos, métodos e ferramentas, conforme apresentado na Figura 1.1. Segundo PRESSMAN; MAXIM (2016), o software, em todas as suas formas e campos de aplicação, deve passar pelos processos de Engenharia de Software que, como qualquer abordagem de engenharia, deve estar fundamentada em um comprometimento com a qualidade.

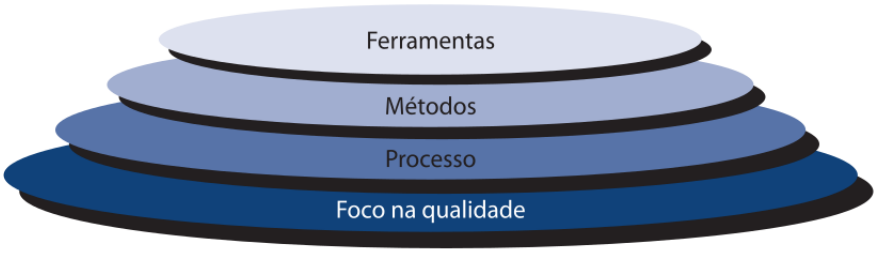


Figura 1.1: Camadas da Engenharia de Software (PRESSMAN; MAXIM, 2016)

Entretanto, conforme PORRU et al. (2017) declaram em seu trabalho, o sentimento de muitos engenheiros de software é que o desenvolvimento em torno das várias implementações de *blockchain* tem sido descontrolado e apressado, uma espécie de competição baseada na ordem de chegada que não garante nem a qualidade do software, nem que todos os conceitos básicos da Engenharia de Software sejam levados em consideração. PORRU et al. (2017) chamam a atenção para a necessidade da Engenharia de Software propor ferramentas e técnicas especializadas para o software orientado a *blockchain*. Este trabalho de pesquisa se posiciona neste contexto.

1.2 Definição do Problema

Entre os conceitos da Engenharia de Software que o desenvolvimento de software para *blockchain* deve empregar a fim de se alcançar um software de qualidade, encontra-se o processo de Gestão de Qualidade do Software. Afinal, todo software, inevitável e inerentemente, possuirá falhas (LIBICKI; ABLON; WEBB, 2015). Alguns eventos na recente história da Ethereum evidenciam esta realidade. Por exemplo, o ataque *The DAO* ocorrido em junho de 2016, no qual um ofensor explorou uma falha conhecida como *Reentrancy Vulnerability*[3] levando à perda de US$60 milhões. Outros exemplos são os dois eventos da *Parity Wallet* em julho e em novembro de 2017. No primeiro evento, *hackers* exploraram um defeito no código que permitiu aos ofensores modificar uma lista de endereços de contas destinatárias de fundos do contrato atacado e, assim, saquearam mais de US$30 milhões. No segundo evento, um outro defeito: uma biblioteca compartilhada não atribuía valor à variável que deveria guardar o endereço da conta proprietária da carteira. Os ofensores atribuíram seu endereço como proprietário e então invocaram o comando `kill`. Com isso, todos os contratos que referenciavam tal biblioteca compartilhada ficaram impedidos de transferir fundos, provocando o congelamento de aproximadamente US$150 milhões.

No estudo realizado por TAN et al. (2014), foi identificado que entre 70% e 87% das causas raiz dos problemas reportados nos softwares analisados eram de natureza semântica. Entre estes problemas, podemos citar funcionalidades que estariam supostamente implementadas mas na realidade não estavam, cenários de uso não cobertos em uma funcionalidade, cenários de uso limítrofes tratados incorretamente ou ignorados, controle de fluxo implementado de forma incorreta, entre outros que não cumprem os requisitos de projeto. Em um estudo mais recente, com o objetivo de mapear as características

[3] Ocorre quando uma função no contrato atacado faz chamada externa a outro contrato não conhecido (ofensor) e este contrato faz repetidas invocações à função original antes mesmo da primeira chamada ser concluída

de problemas reportados em sistemas *blockchain*, WAN et al. (2017) confirmaram que a maioria dos problemas reportados eram de natureza semântica (67,23%).

Motivadas pelos eventos relacionados a perda de criptoativos em função de falhas no software, algumas pesquisas foram empreendidas com o objetivo de identificar estas vulnerabilidades no código-fonte de contratos inteligentes. Entre estas vulnerabilidades, alguns trabalhos buscaram por padrões indesejados de programação, conhecidos como *code smells* ou anti-padrões de desenvolvimento (DURIEUX et al., 2020). Entretanto, problemas de natureza semântica nos contratos inteligentes dificilmente seriam detectados por análise estática de código focada na detecção de anti-padrões de desenvolvimento.

O teste é o principal método pelo qual se estabelece confiança na correção de um software (DEMILLI; OFFUTT, 1991). As potenciais perdas resultantes de falhas em contratos inteligentes e a representatividade dos problemas de natureza semântica evidenciam o que Fraser e Arcuri constataram e reportaram: o teste é uma atividade essencial no processo de desenvolvimento de software (FRASER; ARCURI, 2012). Em um ambiente como a *blockchain*, em que uma das principais características é a imutabilidade (GREVE et al., 2018), a etapa de testes torna-se ainda mais crítica no processo de desenvolvimento.

1.3 Objetivo

O teste de software é uma atividade cara. Ela é responsável por cerca de 50% do tempo e mais de 50% do custo total da produção de um software (MYERS; SANDLER; BADGETT, 2011). Por este motivo, muito esforço de pesquisa foi colocado na automação da geração e execução de casos de teste (FRASER; ARCURI, 2012).

O objetivo deste trabalho é propor uma técnica para identificação de requisitos de teste para as funções de contratos inteligentes com base nas expressões relacionais pre-

sentes no código-fonte destes projetos. Um segundo objetivo consiste em construir uma ferramenta que, além de identificar os requisitos de teste utilizando a técnica proposta, automatize a geração do código-fonte JavaScript para execução destes testes. A automação da geração do código de testes passa pela geração dos dados de teste, de forma que estes valores atendam às condições necessárias para exercitar a condição de teste pretendida, bem como pela identificação de chamadas a outras funções do contrato que mudem seu estado a fim de atender às pré-condições do respectivo teste quando necessário. Embora a área de testes já disponha de diversos trabalhos acerca da automação de testes de software em outras linguagens e ambientes, o desconhecimento das peculiaridades do ambiente da tecnologia *blockchain* no primeiro estágio da pesquisa inviabiliza o julgamento da viabilidade de aplicação destas mesmas técnicas sobre os contratos inteligentes. Assim, optou-se pela criação desta técnica baseada na técnica de execução simbólica.

A proposta será submetida a um processo de avaliação que inclui a execução da técnica de geração de testes unitários para um conjunto de projetos selecionados. Para apresentar evidências sobre a eficácia da técnica proposta, a cobertura de ramos (*branch coverage*) obtida por essa série de execuções em cada projeto será comparada com a cobertura alcançada pelos testes unitários encontrados no repositório destes projetos.

Embora o código-fonte resultante contemple tanto cenários de execuções bem sucedidas das funções quanto cenários em que se espera o lançamento de uma exceção, o foco principal desta pesquisa está na identificação e automação daqueles testes cujo resultado esperado seja o lançamento de uma exceção provocado pelo não atendimento de validações sobre parâmetros de entrada e/ou variáveis de estado do contrato inteligente, codificadas pelo desenvolvedor nas funções do contrato inteligente ou dos modificadores (*modifier*) aplicados a essas funções. Para este grupo de testes, o resultado esperado (ou seja, a geração da exceção) é conhecido pela ferramenta. Para os testes em que não há erro de validação, o resultado esperado é desconhecido, dependente do contexto do projeto e, consequentemente, está fora do contexto desta pesquisa.

1.4 Questões de Pesquisa

RQ1: Há melhoria significativa na cobertura de ramos pelos testes gerados através da técnica proposta em comparação com o percentual de cobertura de ramos alcançado pelos testes unitários presentes nos projetos de contratos inteligentes?

> Confirmando-se uma melhoria significativa na cobertura de ramos pelos testes gerados pela proposta deste trabalho, os desenvolvedores de contratos inteligentes seriam poupados de uma parte do trabalho de escrita dos testes unitários.
>
> Esta automação tem potencial para trazer impacto positivo em projetos que envolvam o desenvolvimento de contratos inteligentes, não apenas em termos de custos e cronograma, mas também em termos de qualidade por reduzir as chances de erro humano na identificação e implementação dos testes.

1.5 Contribuições Esperadas

É possível destacar as seguintes contribuições da presente pesquisa:

- Definição de um método para a identificação de requisitos de teste para contratos inteligentes escritos na linguagem de programação Solidity e que usam o *framework* Truffle, baseando-se nas expressões relacionais presentes em seu código-fonte;
- Criação de uma ferramenta que automatiza a geração do código-fonte de testes unitários para os contratos inteligentes supracitados;
- Avaliação experimental do método e da ferramenta com sua aplicação em 40 projetos de contratos inteligentes com código-fonte disponível na plataforma GitHub.

1.6 Metodologia de Pesquisa

Para desenvolver a técnica de geração de testes unitários proposta, primeiramente foram realizadas consultas à literatura sobre a tecnologia *blockchain*, a plataforma Ethereum, seus contratos inteligentes e a linguagem de programação Solidity, bem como pesquisas relacionadas à automação de processos de controle de qualidade dos contratos inteligentes.

Em seguida, foi construído um utilitário para realizar buscas no GitHub, procurando por repositórios cuja linguagem de programação principal estivesse estabelecida como Solidity. Ao analisar este conjunto de repositórios e os projetos de contratos inteligentes contidos neles, identificou-se a oportunidade para a geração automatizada de testes unitários para os contratos inteligentes escritos em Solidity.

Foi então iniciada a construção de uma ferramenta para geração automática dos testes com o objetivo de realizar uma prova de conceito. Esta ferramenta cobriu um conjunto reduzido de situações em que os testes poderiam ser aplicados: a validação de parâmetros realizada pelo comando `require` nas funções dos contratos. Tão logo se chegou à conclusão de que a construção da ferramenta era viável, o escopo de geração de testes foi ampliado, passando a abarcar expressões também em *modifiers*, em estruturas de decisão, variáveis de estado, entre outros constituintes da linguagem.

Concluída a implementação da técnica proposta, prosseguiu-se com sua aplicação sobre um conjunto selecionado de projetos de contratos inteligentes, realizando uma série de execuções para apuração da capacidade dos testes gerados cobrirem o código-fonte dos contratos. Os resultados obtidos foram então analisados e comparados com aqueles alcançados pelos testes encontrados nos repositórios dos projetos selecionados.

Por fim, redigiu-se este documento, que relata os detalhes da técnica proposta e da sistemática utilizada na sua avaliação.

1.7 Estrutura do Texto

Este trabalho está organizado em cinco capítulos. O primeiro capítulo compreende esta introdução, que inclui o contexto, a definição do problema, o objetivo, as questões de pesquisa, as contribuições esperadas e a metodologia de pesquisa.

O Capítulo 2 apresenta os fundamentos teóricos que nortearam este trabalho, apresentando a literatura acerca da tecnologia *blockchain*, de contratos inteligentes, da plataforma Ethereum e componentes do seu ecossistema relevantes a esta pesquisa, como a linguagem Solidity e o *framework* Truffle, além dos trabalhos relacionados ao controle de qualidade de contratos inteligentes da plataforma Ethereum.

No Capítulo 3, é apresentada a técnica para geração dos testes automatizados para contratos inteligentes. São também discutidas as limitações da técnica e as estratégias de preparação dos contratos para que os testes sejam executados.

O Capítulo 4 apresenta a avaliação experimental da proposta, descrevendo o processo e os critérios utilizados para seleção dos projetos para aplicação da técnica de geração de testes. Em seguida, são apresentados e discutidos os resultados da cobertura de ramos obtida por uma série de execuções da técnica proposta, que é comparada com a cobertura dos testes originais dos projetos analisados.

O Capítulo 5 fecha a Dissertação com as conclusões e perspectivas de trabalhos futuros relacionados ao tema.

2 REFERENCIAL TEÓRICO

2.1 Introdução

Este capítulo tem como objetivo apresentar uma revisão da literatura sobre os principais temas relacionados a testes de contratos inteligentes (*smart contracts*) escritos usando a linguagem de programação Solidity. A Seção 2.2 apresenta a tecnologia *blockchain*, onde são armazenados os contratos inteligentes e os dados que podem ser manipulados por estes contratos. A Seção 2.3 apresenta os contratos inteligentes. A Seção 2.4 apresenta a plataforma *blockchain* Ethereum, uma implementação do conceito de *blockchain* onde os contratos inteligentes podem ser armazenados e ativados. Trabalhos relacionados à automação de testes de contratos inteligentes são apresentados na Seção 2.5. Finalmente, a Seção 2.6 apresenta as considerações finais deste capítulo.

2.2 Blockchain

Organizações tradicionalmente registram transações contábeis em livros razão. Esses livros são tipicamente isolados fisicamente para proteger sua exatidão e inviolabilidade e cada organização mantém seu próprio registro separadamente. As *blockchains* podem ser compreendidas como livros razão distribuídos, ao mesmo tempo compartilhados e confiáveis (WHITE; KILLMEYER; CHEW, 2017). GREVE et al. (2018) definem *blockchain* como "um livro razão distribuído que provê uma forma de a informação ser gravada e compartilhada por uma comunidade" e afirmam que essas informações podem representar transações, contratos, ativos, identidades ou praticamente qualquer outra coisa que possa ser descrita em formato digital.

Segundo SHEN; PENA-MORA (2018), uma *blockchain* típica consiste em uma rede ponto-a-ponto de nós de computador que mantém um banco de dados compartilhado e descentralizado de registros. Podemos definir uma *blockchain*, portanto, como uma estrutura de dados distribuída que é replicada e compartilhada entre os membros de uma rede (CHRISTIDIS; DEVETSIKIOTIS, 2016). Assim, não existe uma base de dados central e o histórico completo das transações é armazenado por todos os nós da rede (ANDESTA; FAGHIH; FOOLADGAR, 2019). Esta base distribuída é organizada como uma lista ordenada de blocos, onde os blocos salvos (*committed*) são imutáveis (CASINO; DASAKLIS; PATSAKIS, 2019). Novos blocos podem ser anexados, mas blocos salvos não podem ser alterados ou excluídos. Estas características fazem com que o banco de dados compartilhado seja representado por uma lista crescente de registros imutáveis e irreversíveis (SHEN; PENA-MORA, 2018). A Figura 2.1 apresenta uma representação visual da cadeia de blocos de um *blockchain*.

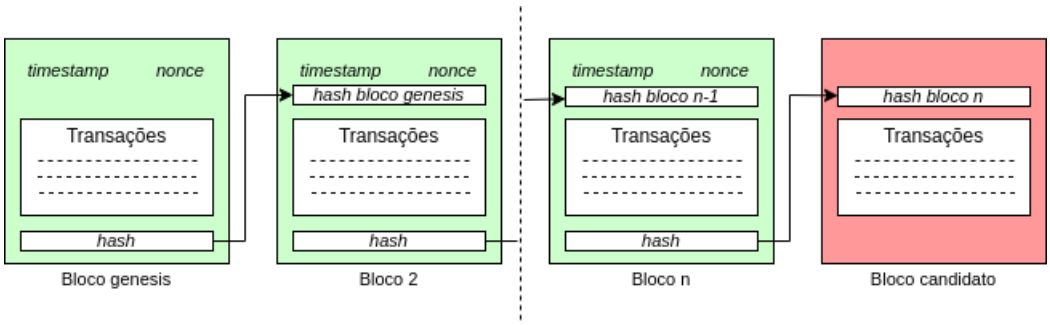


Figura 2.1: Representação visual da cadeia de blocos de um *blockchain*.

Cada bloco contém múltiplas transações. Além das transações, cada bloco contém um *timestamp*, um valor de *hash* criptográfico do bloco anterior e um "nonce". Considerando-se que a função utilizada para gerar o *hash* do bloco é determinística, ou seja, produzirá o mesmo resultado toda vez que for chamada com os mesmos parâmetros de entrada, o *hash* do bloco poderá ser verificado chamando esta função e passando suas transações, o *hash* do bloco anterior, o *timestamp* e o "nonce" que foi utilizado originalmente e se encontra no bloco.

A geração de um *hash* de um conjunto de transações de um bloco é uma operação trivial para um computador moderno. Considerando isso, a rede *blockchain* estipula um nível de dificuldade para a criação de novos blocos, especificando um critério para que o *hash* do próximo bloco seja válido. Em algumas implementações, é estipulado um número de 256 bits, chamado de *hash* alvo, e o critério de validade do próximo bloco é que seu *hash* seja menor ou igual ao *hash* alvo. O processo de mineração consiste na descoberta do valor de "nonce" que, junto com os demais dados do bloco, gerará um *hash* que atenda ao critério de validade estipulado.

A aplicação destes conceitos garante a integridade da cadeia de blocos. Os valores de *hash* são únicos e fraudes podem ser evitadas, porque as alterações de um bloco na cadeia alterariam o valor do seu *hash*, que está registrado no bloco seguinte, que também não pode ser alterado (NOFER et al., 2017). Assim, a adulteração de qualquer informação em um bloco pode ser detectada por outros nós componentes da *blockchain* (SHEN; PENA-MORA, 2018).

O uso intensivo de criptografia, uma característica fundamental das *blockchains*, traz autoridade para as interações na rede (CHRISTIDIS; DEVETSIKIOTIS, 2016). Se a maioria dos nós na rede concordar, através de um mecanismo de consenso, sobre a validade das transações em um novo bloco e sobre a validade do próprio bloco, o bloco pode ser adicionado à cadeia (NOFER et al., 2017). SWANSON (2015) define esse mecanismo de consenso como o processo no qual a maioria dos validadores da rede chega a um acordo sobre o estado da *blockchain*. Esse processo consiste em um conjunto de regras e procedimentos que permite manter um conjunto coerente de fatos entre vários nós participantes da rede. No mecanismo de consenso *Proof-of-Work*, por exemplo, quando há divergências sobre qual é o próximo bloco válido, a regra é que a cadeia de blocos que consumiu maior poder de processamento para ser gerada deve ser acatada pela rede.

Segundo CHRISTIDIS; DEVETSIKIOTIS (2016), depois que uma transação é enviada por algum usuário para os nós adjacentes da rede, estes nós verificam a validade da transação, retransmitem a transação para os próximos nós adjacentes, e assim sucessivamente. As transações que foram recebidas e validadas pela rede em um determinado intervalo de tempo são então ordenadas e empacotadas em um bloco candidato. Em seguida, os nós mineradores calculam o *hash* do bloco candidato e enviam de volta para a rede. Os nós então verificam se o bloco sugerido contém transações válidas e se contém o *hash* do bloco anterior em sua cadeia de blocos. Caso positivo, eles adicionam o bloco na sua cadeia de blocos; do contrário, o bloco é descartado.

De acordo com SHEN; PENA-MORA (2018), o mecanismo de consenso distribuído é fundamental para a *blockchain*, pois determina qual bloco pode ser aceito e inserido na cadeia de blocos. Entretanto, podem ocorrer situações em que dois mineradores gerem blocos simultaneamente e, assim, gera-se uma bifurcação ou *fork*. Nestes casos, considerando o mecanismo de consenso *Proof-of-Work*, a divergência é geralmente resolvida automaticamente já no próximo bloco gerado. Conforme citado anteriormente, este mecanismo de consenso estabelece que os nós devem acatar o *fork* com o maior consumo de poder de processamento, sendo pouco provável que dois mineradores também gerem o próximo bloco de forma simultânea. Assim, o *fork* que acumular o maior número de blocos gerados será adotado por toda a rede como o correto. A Figura 2.2 apresenta uma representação visual da ocorrência de um *fork* com a geração simultânea do bloco "n + 1" e, na sequência, a rede se resolvendo e acatando o primeiro ramo a gerar o bloco "n + 2" e, consequentemente, descartando o outro.

CHRISTIDIS; DEVETSIKIOTIS (2016) afirmam que a tecnologia *blockchain* permite redes transacionais sem confiança, pois as partes envolvidas podem registrar transações mesmo que não confiem umas nas outras. Transações que antes eram realizadas por meio de um intermediário confiável (como a transferência de dinheiro entre duas pessoas através de um banco) agora podem acontecer de maneira descentralizada,

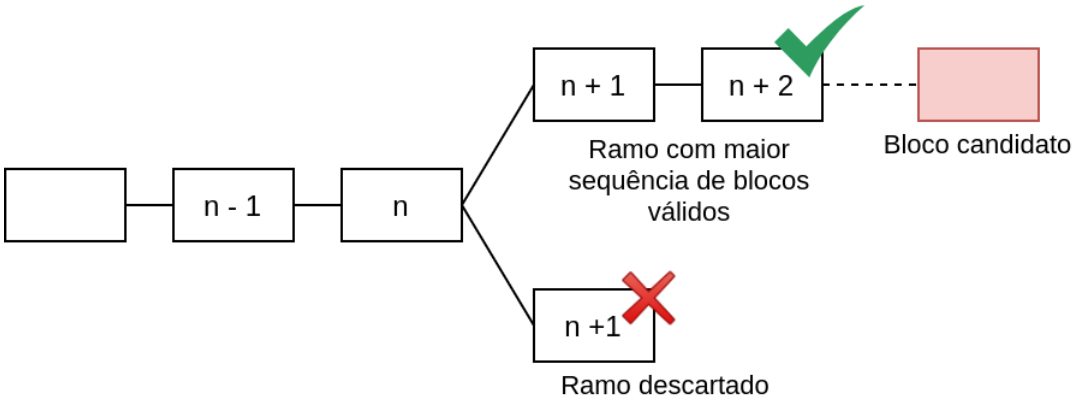


Figura 2.2: Representação visual da ocorrência de um *fork* em um *blockchain*.

sem a necessidade de uma autoridade central. As características inerentes à arquitetura e *design* da *blockchain* oferecem transparência, robustez, auditabilidade e segurança.

ATZORI (2015) destaca que a inovação introduzida por esta tecnologia é que as partes envolvidas não precisam conhecer ou confiar uns nos outros para interagir: as transações eletrônicas podem ser automaticamente verificadas e registradas pelos nós da rede através de algoritmos criptográficos, sem intervenção humana, sem uma autoridade central, um ponto de controle ou a interferência de terceiros. Mesmo que alguns nós da rede sejam desonestos ou mal-intencionados (por tentar alterar os dados e interferir nas transações), a rede é capaz de verificar as transações e proteger os dados de adulteração.

O primeiro caso de sucesso (e também mais conhecido) da aplicação da tecnologia *blockchain* é a rede Bitcoin, que implementa o mecanismo de transação da criptomoeda que leva o mesmo nome da rede. As criptomoedas são desenvolvidas com o objetivo de servir como meio de troca de valores, podendo ser utilizadas tanto para compra de bens e serviços quanto para reserva de valor ou investimento, da mesma forma que qualquer moeda fiduciária. Nos dias atuais, o Bitcoin é comercializado como ativo financeiro em todo o mundo, assim como também é aceito em diversos estabelecimentos como forma de pagamento na compra de bens e serviços. Utilizando a tecnologia *blockchain*, foi possível

transformar a nova moeda em um meio de troca sem precisar confiar em instituições financeiras ou mesmo em governos. A confiança, indispensável para transações desta natureza, é garantida pela tecnologia.

2.3 Contratos Inteligentes

Um contrato é um acordo entre duas ou mais pessoas (ou entidades jurídicas) com a intenção de criar uma obrigação legal entre elas e ser legalmente executável (WALKER, 1980). Por legalmente executável, entende-se que os contratos são cumpridos pela força da Lei, ou seja, eles têm valor legal – sua validade é garantida pelo Estado através do Poder Judiciário. Os contratos são, portanto, a personificação de um acordo, que define um conjunto de termos aceitos por ambas as partes, de modo que cada parte compreenda, possa proteger e fazer valer seus direitos (ZACK, 2009).

Nick Szabo introduziu o conceito de contratos inteligentes em 1994, definindo-os como "um protocolo de transação computadorizado que executa os termos de um contrato" (SZABO, 1994). SZABO (1997) sugeriu traduzir cláusulas contratuais (garantias, títulos, entre outras) em código-fonte e incorporá-las em propriedades (hardware ou software) que possam se auto aplicar, de modo a minimizar a necessidade de intermediários confiáveis entre as partes envolvidas em uma transação, assim como evitar a ocorrência de exceções maliciosas ou acidentais. Na visão de WOOD et al. (2019), ficava claro que a redação de acordos na forma de algoritmos poderia se tornar uma força significativa para intensificar a cooperação entre agentes humanos ou suas personificações jurídicas.

PINNA et al. (2019) definem contrato inteligente como um programa de computador que implementa uma sequência lógica de etapas de acordo com um conjunto de cláusulas e regras. Conceitualmente, os contratos inteligentes consistem de três partes:

- o código de um programa que se torna a expressão da lógica contratual;
- o conjunto de mensagens que o programa acima pode receber e que representam os eventos que ativam o contrato;
- o conjunto de métodos que ativam as reações previstas pela lógica contratual.

A primeira engloba todo o código do programa, incluindo suas funções privadas. A segunda estabelece a interface pela qual o mundo externo se comunica com o próprio contrato, ou seja, o formato das mensagens e dados que este precisará para executar suas funcionalidades. Por fim, a terceira parte é a porta de entrada das mensagens enviadas por agentes do mundo externo: ela é composta por funções públicas e visíveis a todos os nós da rede, que poderão ser invocadas a qualquer tempo para execução da lógica contratual.

De acordo com SHEN; PENA-MORA (2018), a *blockchain* é uma infraestrutura adequada para o registro e execução de contratos inteligentes, pois fornece uma plataforma transparente e rastreável que permite que as partes realizem transações sem confiança entre si e sem intermediários. Neste contexto, os contratos inteligentes são programas de computador imutáveis, rastreáveis, hospedados e executados em uma rede *blockchain*. Eles são escritos em uma linguagem suportada pela rede *blockchain* subjacente, com o objetivo de expressar a lógica de negócios para gerenciar ativos também registrados nesta rede (HARTEL; SCHUMI, 2019).

Nesta segunda geração da tecnologia *blockchain*, é possível não apenas a execução de transações simples, mas a realização de computação em uma rede, onde, por exemplo, pagamentos podem ser condicionados ao estado de variáveis internas ou externas à rede (PETERS; PANAYI, 2016). Isso torna possível fluxos de trabalho eficientes (dependentes de menos recursos como, por exemplo, a terceira parte confiável), distribuídos e automatizados (CHRISTIDIS; DEVETSIKIOTIS, 2016). Os contratos inteligentes estendem a funcionalidade de uma rede *blockchain*, permitindo que ela passe de um consenso sobre

dados para um consenso sobre computação (KOSBA et al., 2016). No primeiro, o consenso entre os nós é realizado sobre as transações e seus respectivos dados, enquanto no segundo o consenso é realizado sobre a lógica que determina o processamento que levou a cadeia de blocos para um novo estado.

SHEN; PENA-MORA (2018) resumem em três aspectos as características dos contratos inteligentes baseados em *blockchain* em comparação com os contratos tradicionais. Primeiro, os contratos inteligentes executados em uma *blockchain* são gerenciados por código de computador e não estão sujeitos ao controle de nenhuma entidade central. Segundo, a única maneira de modificar um contrato inteligente implantado em uma *blockchain* é criar um novo contrato sob o consentimento de todas as partes envolvidas. O antigo contrato não pode ser simplesmente retirado da rede como se nunca tivesse existido. Terceiro, se comparado a contratos tradicionais, estabelecer acordos complexos que exigem várias condições entre múltiplas partes é mais econômico.

Os contratos inteligentes baseados em *blockchain* possuem um endereço exclusivo. Para que um contrato inteligente seja executado, uma transação deve ser endereçada a ele (CHRISTIDIS; DEVETSIKIOTIS, 2016). Uma transação que envolve um contrato inteligente também é chamada de mensagem e contém as instruções necessárias para executar uma função do contrato (PINNA et al., 2019). A partir do envio desta mensagem, o contrato é executado de forma independente e automática, da maneira prescrita em todos os nós da rede de acordo com os dados que foram incluídos na transação (CHRISTIDIS; DEVETSIKIOTIS, 2016). Após sua execução, os estados nos nós da rede serão atualizados de forma consistente através do processo de consenso da *blockchain* (SHEN; PENA-MORA, 2018).

2.4 Ecossistema Ethereum

Ethereum é uma plataforma *blockchain* global, de código aberto e com suporte a contratos inteligentes. Pode ser considerada como uma máquina de estado baseada em transações, de acordo com WOOD et al. (2019). Uma vez que uma nova transação é reconhecida pela *blockchain*, ela causa uma transição de estado. Segundo SHEN; PENA-MORA (2018), é a plataforma *blockchain* para execução de contratos inteligentes mais conhecida e, de acordo com PINNA et al. (2019), trata-se da mais importante plataforma baseada na tecnologia *blockchain* em termos de número de transações. No momento em que esta pesquisa foi redigida, a rede Ethereum contava com mais de 82 milhões de contas criadas (ETHERCHAIN.ORG, 2020).

Segundo PINNA et al. (2019), a proposta de execução de aplicativos descentralizados da plataforma é nova e disruptiva: uma máquina virtual programável *Turing-completa* baseada em *blockchain* para executar código de software escrito especificamente para o ambiente *blockchain*. Segundo o mesmo autor, os recursos oferecidos pelas plataformas marcaram o avanço da tecnologia *blockchain* para uma segunda geração, sendo a primeira caracterizada pelo *Bitcoin*.

A plataforma Ethereum adota um modelo baseado em contas com dois tipos distintos: contas externas, pertencentes a usuários do Ethereum, e contas de contrato. Cada contrato inteligente é uma conta Ethereum que possui seu código-fonte, armazenamento privado e um saldo de dinheiro em *Ethers*, a criptomoeda da plataforma Ethereum (ANDESTA; FAGHIH; FOOLADGAR, 2019). As contas externas não possuem código de programação e podem iniciar a execução de uma operação criando uma transação digitalmente assinada e enviando-a a outra conta externa ou a uma conta de contrato. Por sua vez, uma conta de contrato tem seu código executado toda vez que recebe uma transação. Neste momento, ela pode realizar leituras e escritas no armazenamento interno, transferência de fundos para outras contas ou até mesmo a criação de novas contas de contrato.

As contas de contrato são utilizadas tanto para criar aplicativos descentralizados (*Dapps*) quanto para criar novos *tokens* digitais (PINNA et al., 2019).

Todo processamento computacional na rede Ethereum está sujeito a taxas, isto é, cada fragmento de computação, seja criação de contratos, chamadas de funções, armazenamento ou acesso a dados de contas ou qualquer operação executada na Ethereum *Virtual Machine* (EVM), possuirá um custo acordado universalmente e que é especificado na unidade de medida *gas* de uma rede Ethereum (WOOD et al., 2019). Ao enviar uma transação para a rede, a conta de origem informa o valor em Ether que pagará por cada unidade de *gas* despendida para efetivar sua transação, o `gasPrice`. A soma das unidades de *gas* das operações de processamento realizadas para efetivar a transação multiplicada pelo `gasPrice` será a taxa paga ao minerador do bloco onde ela será incluída. Como é possível deduzir, as transações com um `gasPrice` maior tendem a ser efetivadas mais rapidamente. Existe também um parâmetro da transação que estabelece um limite de consumo de *gas*, o `gasLimit`. Ao enviar uma transação, a conta estabelece a quantidade máxima de *gas* que a transação poderá consumir. Caso a quantidade utilizada efetivamente seja menor que o `gasLimit`, o excedente é devolvido à conta de origem. Caso a quantidade utilizada seja maior do que o `gasLimit` estabelecido pela conta origem, uma exceção é lançada, a transação é revertida e todo o *gas* é devolvido à conta origem.

PINNA et al. (2019) dividem o processo de implantação de um contrato inteligente na Ethereum em três fases: (1) a escrita do código-fonte; (2) a compilação local do código-fonte para o *bytecode* da Ethereum *Virtual Machine* (EVM); e (3) a criação de uma transação na *blockchain* que realiza a implantação do contrato inteligente na rede. Nesta última etapa, a *blockchain* associa um endereço ao contrato inteligente. Acessando tal endereço, é possível visualizar alguns dados do contrato, tais como seu endereço, seu saldo em Ether e sua *Application Binary Interface* (ABI)[1]. Concluída a terceira etapa, qualquer usuário da Ethereum pode usar o endereço para chamar qualquer função do

[1]Descrição da interface do contrato incluindo nomes das funções e tipos dos parâmetros

contrato através de uma transação (ANDESTA; FAGHIH; FOOLADGAR, 2019). Uma transação normalmente incluirá pagamento ao contrato pela execução (em *Ethers*) e/ou dados de entrada para a chamada (LUU et al., 2016).

2.4.1 Solidity

De acordo com o site oficial da Ethereum[2], as linguagens de programação mais ativas para desenvolvimento de contratos inteligentes nesta plataforma são Solidity e Vyper. Contudo, a linguagem Vyper sequer aparece como opção na lista de linguagens exibida na busca avançada da plataforma GitHub[3]. Segundo NGUYEN et al. (2020), Solidity é a linguagem de programação mais utilizada no desenvolvimento de contratos inteligentes. Solidity é uma linguagem de programação semelhante ao JavaScript, porém tipificada. O código-fonte dos contratos escritos em Solidity se assemelham às classes nas linguagens orientadas a objetos. Eles podem conter declarações de variáveis de estado, definições de funções, modificadores, construtores, estruturas de dados (*structs*), entre outros (WANG et al., 2019). Solidity suporta tipos tradicionais, como números inteiros, *strings*, *arrays*, *structs* e enumeradores. Além disso, possui tipos específicos, como o *address*, que identifica usuários e contratos (PINNA et al., 2019).

```
pragma solidity ^0.5.3;

import "./ERC20Basic.sol";
import "./SafeMath.sol";

contract Token is ERC20Basic {
  using SafeMath for uint256;

  mapping(address => uint256) balances;

  function transfer(address _to, uint256 _value) public returns (bool) {
    require(_to != address(0));
    require(_value <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    emit Transfer(msg.sender, _to, _value);
    return true;
  }
```

[2]https://ethereum.org/en/developers/docs/smart-contracts/languages/
[3]https://github.com/search/advanced

```
19
20    function balanceOf(address _owner) public view returns (uint256 balance) {
21      return balances[_owner];
22    }
23  }
```

Algoritmo 2.1: Exemplo de contrato em Solidity

Um exemplo de contrato escrito na linguagem Solidity pode ser encontrado no Algoritmo 2.1. Esse algoritmo implementa um contrato que aplica um padrão de implementação de *tokens* de uma rede Ethereum, o ERC-20, estendendo o contrato ERC20Basic. Nota-se que o referido contrato possui duas funções públicas:

- `transfer`: transfere uma quantidade de *tokens* (indicada no parâmetro `_value`) da conta que invocou o contrato para outra conta (indicada no parâmetro `_to`). Percebe-se que algumas validações são realizadas utilizando o comando nativo `require` antes de se efetivar a transferência. Por exemplo, o código verifica a validade do endereço de destino e se a conta de origem detém uma quantidade de *tokens* maior ou igual a da transferência solicitada;

- `balanceOf`: retorna o saldo de *tokens* de uma conta cujo endereço é recebido através do parâmetro `_owner`.

Conforme percebido no Algoritmo 2.1, a linguagem Solidity dispõe de algumas particularidades, como a falha de transação de forma programática através de chamadas aos comandos `require()` ou `assert()`. O `require()` é usado para checar consistência externa, como, por exemplo, para garantir que os valores dos parâmetros de entrada satisfazem a requisitos específicos. O `assert()` é utilizado para checar consistência interna, como para garantir que o saldo de uma conta não sofreu um *underflow* (HARTEL; STAALDUINEN, 2019).

2.4.2 Truffle Framework

Conforme constatações apresentadas no capítulo 4, o *framework* Truffle tem se tornado popular entre os desenvolvedores de contratos inteligentes para a plataforma Ethereum. Ao utilizá-lo, os testes dos contratos inteligentes podem ser escritos em Solidity ou JavaScript e o *framework* Truffle garante que os testes possam se comunicar com o contrato quando ele for implantado em uma das redes de teste ou mesmo na rede Ethereum principal (HARTEL; STAALDUINEN, 2019). O Truffle é um ambiente de desenvolvimento, *framework* de teste e *pipeline* de ativos para o Ethereum que tem como objetivo facilitar a vida dos desenvolvedores na plataforma. Algumas das facilidades oferecidas pelo *framework* são:

- compilação e *linking* de contratos inteligentes e gerenciamento de binários utilizados pelo *framework* durante o ciclo de vida do projeto, tais como as versões dos compiladores utilizadas para compilar os contratos inteligentes ou pacotes de contratos inteligentes publicados no gerenciador de pacotes ethPM[4]. Neste sentido, o *framework* controla as dependências dos contratos inteligentes com versões específicas de outros artefatos do ambiente Ethereum;
- teste automatizado de contratos com uso das bibliotecas JavaScript Mocha[5] e Chai[6];
- migração e implantação de contratos via scripts, permitindo uma maior flexibilidade do comportamento desejado na implantação dos contratos inteligentes. Por exemplo, implantar um determinado contrato e utilizar seu endereço como parâmetro para um segundo contrato;
- gerenciamento de diferentes redes Ethereum, para implantação dos contratos inteligentes em redes públicas e privadas;

[4]http://www.ethpm.com/
[5]https://mochajs.org
[6]https://www.chaijs.com

- console interativo para comunicação direta com os contratos inteligentes.

Ao utilizar o Truffle, o projeto de contratos inteligentes deve seguir uma estrutura de diretórios definidas pelo *framework*. O diretório "contracts" contém o código do contrato inteligente escrito em Solidity, bem como um contrato de migração usado pelo Truffle na fase de implantação. O diretório de "test" contém os *scripts* de testes dos contratos inteligentes, que podem ser escritos tanto em Solidity quanto em Javascript.

O Algoritmo 2.2 apresenta um teste escrito na linguagem JavaScript que exercita as funções do Algoritmo 2.1. Na primeira e segunda linhas, tem-se a importação de uma biblioteca de asserção do *framework* Truffle e a importação do contrato Solidity a ser testado, respectivamente. Na linha 4, percebe-se uma ligeira modificação na função de testes em relação à biblioteca Mocha padrão: enquanto no Mocha padrão a função que identifica o teste possui o nome `describe` e recebe como segundo parâmetro uma função sem parâmetros de entrada, no Truffle essa função possui o nome `contract` e seu segundo parâmetro é uma função que recebe um *array* com as contas externas criadas na rede e que estão disponíveis para utilização nos testes. Em seguida, é declarada uma função `beforeEach`, que é executada antes de cada teste. Nota-se que ela cria uma nova instância do contrato que está sendo testado para que ele seja submetido à função de teste sem efeitos colaterais para os demais testes. Por fim, a partir da linha 12 estão as funções de testes `it`, que executam as funções de teste do contrato Solidity.

Os testes apresentados no Algoritmo 2.2 representam a estratégia de gerar um caso de teste para exercitar o não atendimento da condição indicada em cada comando `require()`, testando se o comportamento observado do contrato foi uma falha (capturada através do comando `truffleAssert.fails`). Esta estratégia foi utilizada nesta Dissertação como um meio para geração automática de testes para os contratos inteligentes, conforme descrito no Capítulo 3.

```
1 const truffleAssert = require('truffle-assertions');
```

```
2  const Token = artifacts.require('Token');
3
4  contract('Token', (accounts) => {
5
6      let contractToken = null;
7
8      beforeEach(async () => {
9          contractToken = await Token.new({from: accounts[0]});
10     });
11
12     it('Should execute transfer', async () => {
13         let result = await contractToken.methods["transfer(address,uint256)"](
               accounts[6], 17,{from: accounts[0]});
14     });
15     it('Should fail transfer when NOT comply with: _to != 0
           x0000000000000000000000000000000000000000', async () => {
16         let result = await truffleAssert.fails(contractToken.methods["transfer(
               address,uint256)"]("0x0000000000000000000000000000000000000000", 4,{
               from: accounts[0]}),'revert');
17     });
18     it('Should execute balanceOf', async () => {
19         let result = await contractToken.methods["balanceOf(address)"](accounts
               [8],{from: accounts[0]});
20     });
21     it('Should execute totalSupply', async () => {
22         let result = await contractToken.methods["totalSupply()"]({from: accounts
               [0]});
23     });
24 })
```

Algoritmo 2.2: Exemplo de teste unitário de um contrato inteligente utilizando Truffle

2.5 Trabalhos Relacionados

De acordo com CHEN et al. (2019), da mesma forma que programas desenvolvidos em outras linguagens e que executam em outros tipos de plataforma e dispositivos, os contratos inteligentes escritos em Solidity também são propensos a falhas que podem causar perdas financeiras. No intuito de reduzir a propensão a falhas, várias pesquisas relacionadas ao processo de garantia e controle da qualidade dos contratos inteligentes foram realizadas nos últimos anos. Uma parte significativa dessas pesquisas abordou a análise estática do código-fonte escrito em Solidity ou mesmo do EVM *bytecode* dos contratos para identificar padrões de vulnerabilidades conhecidos. Geralmente, fazem parte da contribuição dessas pesquisas o desenvolvimento de algum utilitário ou *framework*.

LUU et al. (2016) propuseram uma ferramenta chamada Oyente, que analisa o *bytecode* dos contratos inteligentes e os executa simbolicamente para detectar padrões em quatro categorias de vulnerabilidade de segurança:

- *Transaction-Ordering Dependence*: consiste no fato de que a ordem das transações pode alterar o seu estado final. Por exemplo, ao receber ofertas de bens ou serviços a uma determinada quantia de Ether, um nó malicioso pode mandar uma transação de mudança do valor anunciado para um valor menor (zero, por exemplo), colocar essa transação de mudança antes das ofertas, lembrando que os mineradores podem arbitrar a ordem das transações ao minerar um bloco e, assim, arrematar os bens pelo valor mais baixo. O contrato deveria registrar o valor negociado como parte da transação para evitar esta vulnerabilidade;
- *Timestamp Dependence*: vulnerabilidade na qual o *timestamp* do bloco é utilizado como condição de gatilho para executar operações críticas, como enviar dinheiro para uma conta. Normalmente, o *timestamp* do bloco é definido como a hora local do sistema do minerador. Contudo, o minerador pode variar este valor por cerca de 900 segundos e ainda manter a validade do bloco. Quando recebem um novo bloco, os mineradores checam se o *timestamp* do bloco recebido é maior do que o do último bloco e se está a até 900 segundos de diferença da sua hora local. Desta maneira, um minerador mal intencionado pode escolher diferentes *timestamps* para manipular os contratos com esta vulnerabilidade;
- *Mishandled Exceptions*: pode ocorrer quando um contrato chama uma função de outro e, na sequência, o chamador executa uma operação crítica sem avaliar o retorno da função chamada do outro contrato. No caso de ocorrer alguma exceção na função do contrato chamado, como a insuficiência de *gas* por exemplo, a operação crítica será executada sem a execução da função que foi previamente chamada;
- *Reentrancy Vulnerability*: quando um contrato chama a função de outro, a execução corrente espera pelo fim da chamada. Dependendo de como o contrato foi imple-

mentado, isso pode levar a um problema em que o contrato ofensor se aproveita do estado intermediário no qual o contrato atacado se encontra. Por exemplo, considere um contrato que funcione como um banco, armazene o saldo em Ether de suas contas clientes e possua uma função `sacarTudo` que realiza a transferência desses valores para a mesma conta cliente que a chamou. Essa função a) obtém o valor de saldo da conta que a invocou; b) realiza a transferência através da chamada `msg.sender.call.value(valorSaldo)();` e, após a transferência, c) altera o valor de saldo da conta cliente para zero. Caso esta conta cliente seja um contrato ofensor, ela poderá implementar sua função *fallback* (função com o modificador `payable` e que é invocada quando alguém manda Ether para o contrato sem informar nenhum dado adicional) com uma nova chamada à função `sacarTudo`. Perceba que, com essa nova chamada a `sacarTudo`, o contrato atacado ainda está aguardando a conclusão do passo (b) e, logo, o saldo da conta cliente ainda não foi atualizado para zero. Essa reexecução da função `sacarTudo` persistirá até que o contrato atacado esteja sem nenhum Ether ou que o *gas* disponível para a operação acabe, o que ocorrer primeiro (LUU et al., 2016). O código parcial do exemplo descrito está disponível no Algoritmo 2.3.

```
1  //CONTRATO ATACADO
2  mapping (address => uint) private saldoContas;
3
4  function sacarTudo() public {
5      uint valorSaldo = saldoContas[msg.sender];
6      (bool sucesso, ) = msg.sender.call.value(valorSaldo)(""); // Neste ponto, o
           codigo do ofensor sera executado e chamara sacarTudo novamente
7      require(sucesso);
8      saldoContas[msg.sender] = 0;
9  }
10
11
12 //CONTRATO OFENSOR
13 function() external payable {
14     Atacado a;
15     a.sacarTudo();
16 }
```

Algoritmo 2.3: Exemplo de código em Solidity com *Reentrancy Vulnerability*

Os autores executaram a ferramenta em todos os contratos dos primeiros 1.459.999 blocos da rede Ethereum, representando cerca de 19.366 contratos. Dentre os contratos analisados, 8.833 (46%) possuíam ao menos uma das vulnerabilidades cobertas pela ferramenta. As vulnerabilidades mais comuns encontradas entre os contratos analisados são: *Mishandled Exceptions*, constando em 27,9% dos contratos analisados, e *Transaction-Ordering Dependence*, encontrada em 15,7% dos contratos analisados.

Após publicação da pesquisa, a ferramenta Oyente continuou sendo desenvolvida e passou a abordar outras categorias de vulnerabilidade de segurança. Entre elas, a *Integer Overflow*, que consiste no fato de que uma variável do tipo `uint256` da linguagem Solidity ter um limite de até 256 bits e, portanto, comportar um valor máximo de 2^{256}-1. Quando a variável detém esse valor máximo e a ela é somado o valor um, o valor da variável é automaticamente modificado para zero.

Posteriormente, SHIGEMURA et al. (2019) propuseram uma evolução da Oyente, a qual batizaram de Oyente-NG, que amplia o escopo de categorias de vulnerabilidade detectadas, adicionando a cobertura de cinco novos tipos de vulnerabilidade através das seguintes capacidades de detecção:

- *Call to the unknown detection*: analisa cada função chamada a fim de inferir possíveis incompatibilidades entre a assinatura da função e os parâmetros enviados pelo chamador;
- *Gasless send detection*: considerando que cada instrução de *bytecode* consome um certo volume de *gas*, a detecção de um contrato potencialmente exigente é realizada somando-se o valor de cada instrução executada simbolicamente. Se a quantidade de *gas* for maior que 2.300 (valor padrão de `gasLimit` para funções *fallback* quando a transferência de Ether é realizada via `transfer` ou `send`), o contrato será sinalizado como altamente exigente;

- *Typecast casts detection*: detecta conversões de tipo perigosas verificando a assinatura da função chamada e os parâmetros enviados pelo chamador durante a execução simbólica;
- *Ether lost detection*: os endereços encontrados nos arquivos de código-fonte são verificados de acordo com os padrões Ethereum. Se o contrato possuir algum endereço inválido, será marcado com uma potencial vulnerabilidade *Ether lost*;
- *Randomness bug detection*: verifica se o contrato executa alguma instrução envolvendo números aleatórios e, em caso positivo, marca-o com esta potencial vulnerabilidade.

Os autores executaram a ferramenta sobre a primeira versão dos contratos da criptomoeda *Wibx* e encontraram duas ocorrências da vulnerabilidade *Integer Overflow* e uma ocorrência potencial da vulnerabilidade *Gasless send*. Em seguida, executaram sobre a segunda versão dos contratos da criptomoeda (uma versão mais elaborada e complexa segundo os próprios autores) e encontraram três ocorrências da vulnerabilidade *Gasless send* e uma vulnerabilidade *Integer Overflow*.

CHEN et al. (2017) identificaram sete padrões de consumo excessivo de *gas*. Os padrões foram classificados pelos autores em duas categorias: padrões relacionados a código sem utilidade, que introduzem custo adicional ao aumentar o tamanho do *bytecode* para implantação, e padrões relacionados a *loops*, que envolvem a utilização de operações caras dentro de *loops*. Os padrões cobertos pela pesquisa foram:

- Padrões relacionados a código sem utilidade
  - *Dead code*: código cujas condicionais não podem ser atendidas;
  - *Opaque predicate*: condicionais no código que, considerando as condicionais atendidas previamente, sempre terão o mesmo resultado.

- Padrões relacionados a *loops*
  - *Expensive operations in a loop*: código que faz operações de gravação e leitura nos blocos da rede dentro de *loops*;
  - *Constant outcome of a loop*: situação em que a saída de um *loop* é constante e pode ser inferida durante a compilação;
  - *Loop fusion*: caso em que é possível combinar vários *loops* em um único, reduzindo o tamanho do *bytecode* e, talvez, o número de operações;
  - *Repeated computations in a loop*: situação em que há expressões no *loop* que produzem o mesmo resultado, sendo possível reutilizar o valor ao invés de recalculá-lo nas interações subsequentes;
  - *Comparison with unilateral outcome in a loop*: caso em que uma comparação é executada em cada interação do *loop*, mas o resultado é sempre o mesmo, ainda que não seja possível determinar durante a compilação.

A mesma pesquisa contribuiu com uma ferramenta chamada GASPER, também baseada no Oyente. Ela analisa o *bytecode* dos contratos e realiza sua execução simbólica em busca de três dos sete padrões de consumo excessivo de *gas*: *Dead Code*, *Opaque Predicates* e *Expensive Operations in a Loop*.

Na etapa de avaliação da pesquisa, os autores identificaram 27.290 contratos existentes na rede Ethereum até o dia 05 de novembro de 2016. Parte desses contratos possuía o mesmo *bytecode* e tiveram as duplicidades excluídas pelos autores. Ao executar a ferramenta sobre os contratos restantes, parte deles apresentou problemas devido a erros internos ou *timeouts* apontados pelo Oyente, restando 4.240 contratos inspecionados com sucesso. Os autores identificaram os três padrões de consumo excessivo de *gas* em mais de 70% dos contratos. Mais de 90% dos contratos continham os padrões *Dead code* ou *Opaque predicate* e cerca de 80% continham o padrão *Expensive operations in a loop*.

TIKHOMIROV et al. (2018) propuseram um utilitário chamado *SmartCheck* que, através de análise léxica e sintática do código-fonte dos contratos, busca por 21 padrões de codificação com potencial vulnerabilidade:

- *Balance equality*: pontos de decisão no código que tomam alguma ação baseando-se na condição de igualdade do valor de saldo da conta do contrato com um valor específico (e.g. `if(this.balance == 42)`). Um ofensor pode forçar o envio de Ether para qualquer conta através de mineração ou via função `selfDestruct`;

- *Unchecked external call*: falta de validação do valor de retorno ao chamar funções externas. O retorno deve ser validado de forma a tratar casos de erro devidamente;

- *DoS by external contract*: condicionais (e.g. `if`, `for`, `while`) com dependência de chamada externa. A função chamada pode falhar e impedir a conclusão da execução do código que contém a condicional;

- *Send insteadof transfer*: utilização da função `send` ao invés da função `transfer` para realização de pagamentos. A segunda é a forma recomendada, pois lança uma exceção automaticamente em caso de falha;

- *Re-entrancy*: código que deixa o contrato vulnerável a chamadas subsequentes via função *fallback*, conforme já apresentado no Algoritmo 2.3, e que pode levar à perda de todo o saldo de Ether do contrato atacado;

- *Malicious libraries*: utilização de bibliotecas de terceiros. O desconhecimento sobre o que a biblioteca utilizada faz traz consigo o risco de vulnerabilidade. Por outro lado, a ferramenta verifica apenas a existência da expressão `library`, o que pode levar a falso positivos;

- *Using tx.origin*: utilização de `tx.origin` na autenticação dentro do contrato, ao invés de usar a expressão recomendada `msg.sender`;

- *Transfer forwards all gas*: utilização da função `addr.call.value()` ao invés da função `transfer` para realização de transferência de Ether. A primeira é conhecida pela vulnerabilidade de permitir o consumo de todo o *gas*;

- *Integer division*: divisão onde o numerador e o denominador são números literais. Como Solidity não suporta números em ponto flutuante, quando ocorre divisão de inteiros, o quociente é arredondado para baixo;

- *Locked money*: existência de função com modificador `payable`, mas sem nenhuma função que possibilite a transferência do Ether acumulado pelo contrato para outra conta;

- *Unchecked math*: operações aritméticas que estejam fora de uma declaração condicional no intuito de prevenir problemas do tipo *Integer overflow* e *underflow*;

- *Timestamp dependence*: uso da variável de ambiente `now`;

- *Unsafe type inference*: atribuições onde o lado esquerdo é uma declaração de variável e o lado direito é um número inteiro literal;

- *Byte array*: utilização de `bytes[]` ao invés de `bytes`. A segunda forma leva a um consumo menor de *gas*;

- *Costly loop*: existência de `for` ou `while` com chamadas a funções ou com identificador dentro de sua condição. No caso do identificador, variável ou atributo, uma vulnerabilidade pode ocorrer caso o ofensor consiga manipular seu valor. Por exemplo, caso o `for` tenha como condição a repetição até que a varíavel de controle seja menor que o tamanho de um array e o ofensor consiga manipular este array através de alguma função aberta à execução por qualquer usuário, tal ofensor pode fazer com que uma transação se torne significativamente custosa;

- *Token API violation*: existência de contrato com a expressão 'token' em seu nome, que herda de outro contrato e que lança exceções de dentro das seguintes funções

previstas no padrão ERC20 e cujo comportamento deveria ser o retorno de booleano indicando sucesso ou não em sua execução: `approve`, `transfer` e `transferFrom`;

- *Compiler version not fixed*: declaração da diretiva que indica a versão do compilador Solidity contendo o caracter de acento circunflexo, que permitiria que um intervalo de versões sejam utilizadas ao invés de apenas uma única versão específica (e.g. ^0.5.0);

- *private modifier*: utilização do modificador `private` em variáveis de estado que, ao contrário do que possa se presumir, não torna a variável invisível;

- *Redundant fallback function*: para contratos na versão 0.4.0 ou superior da linguagem Solidity, verifica a existência de função *fallback* com lançamento de exceção. Esse padrão de codificação é utilizado com a intenção de que o contrato não receba e acumule Ether. Contudo, a partir da versão 0.4.0 do Solidity, basta a inexistência da função com modificador `payable` para que qualquer envio de valor ao contrato lance uma exceção. Assim, o padrão de código utilizado até a versão 0.4.0 é considerado redundante: `function () payable { throw; }`

- *Style guide violation*: ocorrência de nomes de função iniciando com letras maiúsculas e eventos com nome iniciando com letra minúscula, o que, segundo os autores, diminui a legibilidade do código;

- *Implicit visibility level*: funções e definições de variáveis sem um modificador de visibilidade.

Os autores executaram a ferramenta no código-fonte de 4.600 contratos inteligentes disponíveis no Etherscan[7] no dia 04 de outubro de 2017. Dentre os contratos analisados, 99,9% continham algum problema e 63,2% tinham vulnerabilidades críticas. Dentre as vulnerabilidades críticas, aquelas que apresentaram um maior número de ocorrências

[7] http://etherscan.io

foram: *DoS by external account* (7.864), *Timestamp dependency* (7.692), *Re-entrancy* (4.015), *send instead of transfer* (3.370) e *Costly loop* (2.610).

TORRES; SCHÜTTE; STATE (2018) propuseram Osiris, um *framework* baseado no Oyente que realiza a execução simbólica dos contratos para a identificação de problemas conhecidos no tratamento de números inteiros. A pesquisa também faz uso de análise *taint*, uma técnica que consiste em rastrear a propagação de dados através do fluxo de controle de um programa com o objetivo de reduzir a quantidade de falso-positivos. Os autores classificaram os problemas com números inteiros na Ethereum em três categorias:

- *Arithmetic bugs*: encontram-se nessa categoria problemas como *Integer overflow* e *Integer underflow*, que ocorrem quando uma expressão aritmética resulta em um valor maior ou menor do que pode ser representado pelo tipo resultante;
- *Truncation bugs*: erros que ocorrem na conversão de um valor de um tipo para um outro tipo com capacidade de armazenar um intervalo menor de valores. Tais erros podem levar a perda de precisão e, consequentemente, perda de Ether;
- *Signedness bugs*: erros que ocorrem quando se converte um tipo inteiro para um tipo inteiro do mesmo tamanho, porém sem sinal. Tal conversão pode mudar um número negativo para positivo ou vice-versa.

O Etherscan provê um ranking dos *tokens* baseados no seu mercado de capitalização. Em junho de 2018, este ranking continha 509 *tokens* diferentes, dos quais 495 tinham seu código-fonte público. Os autores obtiveram o código-fonte e *bytecode* desses 495 contratos inteligentes e executaram a ferramenta sobre eles. Dentre os contratos analisados, 164 (33%) apresentaram vulnerabilidades, dos quais, 126 apresentaram *Integer overflow* e 54 *Integer underflow*. Apesar de todos as ocorrências dessas vulnerabilidades serem semanticamente possíveis, a análise dos pesquisadores concluiu que, na prática,

a maioria deles seria improvável de ser explorada considerando-se que: a) uma grande parte delas só poderia ser exercitada pelo dono do contrato; e b) grande parte é devido a implementações que não verificam se o saldo de uma conta destinatária pode extrapolar a capacidade após a transferência ou o inverso pode ocorrer com a conta remetente.

KRUPP; ROSSOW (2018) desenvolveram o teEther, uma ferramenta que realiza análise estática do *bytecode* em busca de falhas de segurança que possibilitam o roubo de Ether armazenado em contratos inteligentes. Além disso, a ferramenta cria de forma automática instruções com objetivo de explorar essas falhas. Duas vulnerabilidades são cobertas pela pesquisa:

- *Direct value transfer*: caso o ofensor consiga ter controle dos parâmetros que determinam o destinatário nas instruções da EVM que permitem a transferência de valores para um endereço específico, `CALL` ou `SELFDESTRUCT`, ele poderá transferir os valores para uma conta de sua propriedade;
- *Code injection*: caso o ofensor consiga ter controle dos parâmetros das instruções da EVM que permitem a execução de código de terceiros no contexto do contrato atual, `CALLCODE` e `DELEGATECALL`, ele poderá injetar um código arbitrário no contrato atacado e, consequentemente, incluir um código que transfira todo o Ether do contrato atacado para o ofensor.

Na avaliação da pesquisa, os autores executaram a ferramenta em 38.757 contratos inteligentes estipulando um tempo limite. Para 33.195 dos contratos analisados, o processamento ocorreu dentro do tempo limite estipulado pelos pesquisadores e foram exploradas as vulnerabilidades de 815 deles (2,10%) de forma automática.

JIANG; LIU; CHAN (2018) desenvolveram um *framework* chamado *ContractFuzzer*, que analisa o *bytecode* dos contratos, gera dados de testes usando a técnica

Fuzzing com base nas especificações ABI (*Application Binary Interface*) de contratos inteligentes, define oráculos de teste para detectar as vulnerabilidades de segurança, instrumenta a EVM para monitorar o comportamento dos contratos inteligentes em tempo de execução e analisa esses *logs* para relatar vulnerabilidades de segurança. A ferramenta é capaz de detectar os seguintes tipos específicos de vulnerabilidades de segurança:

- *Gasless Send*: conforme a extensão de SHIGEMURA et al. (2019) para a ferramenta Oyente;

- *Exception Disorder*: acontece quando um contrato chama uma função de outro e podem ocorrer falhas devido a diferentes tipos de exceção. Quando essas exceções ocorrem, o mecanismo de tratamento é determinado em razão da forma que as chamadas foram realizadas. Em uma cadeia de chamadas aninhadas, onde cada chamada invoca diretamente a função de um contrato, quando ocorre exceção, todas as transações serão revertidas. Entretanto, para uma cadeia de chamadas aninhadas onde ao menos uma delas é feita através de métodos de baixo nível (`address.call()`, `address.delegateCall()` ou `address.send()`), a reversão da transação encerrará apenas função chamada e retornará `false`. A partir daí, nenhum outro efeito colateral será revertido e nenhuma exceção será relançada;

- *Re-entrancy*: conforme implementações das ferramentas Oyente de LUU et al. (2016) e SmartCheck de TIKHOMIROV et al. (2018);

- *Timestamp Dependency*: ocorre quando um contrato inteligente utiliza o *timestamp* do bloco com parte de condições para executar operações críticas como transferir Ether ou como semente para gerar números aleatórios. Considerando-se que os mineradores têm liberdade para estabelecer o *timestamp* com um intervalo de até 900 segundos, um ofensor pode estabelecer um *timestamp* que leve a perdas indevidas do contrato vulnerável;

- *Block Number Dependency*: vulnerabilidade semelhante à *Timestamp Dependency*, mas que faz uso do `block.number` em condições ou como semente de números aleatórios. Da mesma forma que *timestamp*, o número do bloco também pode ser manipulado pelos mineradores;

- *Dangerous DelegateCall*: a instrução `delegateCall` é similar à instrução `call`, com a diferença de que o código do endereço alvo é executado no contexto do contrato que executou a instrução. Isso significa que um contrato pode carregar código de um outro endereço em tempo de execução, enquanto as operações de leitura e escrita estão sendo executadas. Esta é a forma com que bibliotecas são executadas em Solidity. Entretanto, quando `msg.data` é passado como parâmetro para `delegateCall`, um ofensor pode manipular o `msg.data` com a assinatura de uma função de forma que o contrato atacado invoque qualquer função que o ofensor provê, podendo levar a perdas;

- *Freezing Ether*: ocorre quando um contrato inteligente pode receber Ether e enviar Ether para outro endereço via `delegateCall`. Contudo, eles mesmos não possuem funções para transferência de Ether para outros endereços. Ou seja, eles confiam tão somente no código de outros contratos para transferir Ether para outras contas. Caso os contratos que proveem código para manipulação de Ether realizem uma operação `selfDestruct`, o contrato vulnerável não possui outra maneira de transferir seus Ether, que ficam congelados.

O fluxo de trabalho do *ContractFuzzer* para realizar a avaliação de contratos inteligentes ocorre através das seguintes etapas:

- análise estática da interface ABI e do *bytecode* do contrato sob testes, para obter a assinatura das suas funções públicas e os tipos dos parâmetros destas funções;

- análise da assinatura das funções de todos os contratos e indexação dos contratos pela assinatura das funções que eles suportam;
- geração aleatória de entradas válidas para cada função, de acordo com o domínio dos parâmetros especificados na interface ABI;
- invocação aleatória das funções, utilizando as entradas geradas anteriormente, o que levará à geração de *logs* com dados sobre a execução das funções;
- detecção de vulnerabilidades de segurança através da análise dos *logs* de execução gerados durante a invocação das funções dos contratos.

Como parte da pesquisa, os autores encontraram 9.960 contratos disponíveis no Etherscan com código-fonte verificado. Parte desses contratos não puderam ser implantados no ambiente de teste utilizado pela pesquisa devido a utilizarem endereços Ethereum inválidos e foram removidos do escopo, restando 6.991 contratos para ser utilizados na avaliação da pesquisa. Embora o *ContractFuzzer* utilize apenas o *bytecode* para testar os contratos, os pesquisadores escolheram contratos com código-fonte disponível no intuito de facilitar a verificação manual dos resultados de seu experimento.

Foram encontradas 459 ocorrências das vulnerabilidades abarcadas pela pesquisa. Dos contratos analisados, 138 (2,06%) apresentaram a vulnerabilidade *Gasless Send* que, segundo os pesquisadores, não apresentou nenhuma ocorrência de falso positivo. Trinta e seis contratos (0,54%) foram diagnosticados com a vulnerabilidade *Exception Disorder*, dado também verificado e no qual foi constatada a ausência de falso positivos. A vulnerabilidade *Re-entrancy* foi detectada em 14 contratos (0,21%), também sem falso positivos. As vulnerabilidades *Timestamp Dependency* e *Block Number Dependency* foram detectadas em 152 (2,17%) e 82 (1,17%) contratos respectivamente. Para estas, porém, foram detectados seis e três falso positivos. A vulnerabilidade *Dangerous DelegateCall* foi encontrada em sete contratos e a vulnerabilidade *Freezing Ether* em 30 contratos, em ambos os casos sem falso positivos.

Em um trabalho mais recente, NGUYEN et al. (2020) criaram o *sFuzz*, um motor de *fuzzing* construído com base na Ethereum VM *Aleth* e que combina a estratégia do fuzzer AFL (ZALEWSKI, 2014) com uma estratégia adaptativa multi-objetivo. O *sFuzz* monitora a execução dos contratos por meio de um mecanismo de *hooking* suportado pela EVM. Neste mecanismo, a cada *opcode*[8] executado é criado um evento com informações sobre a execução que serão utilizadas para melhorar o conjunto de casos de teste. Além disso, as informações são utilizadas para detectar um vulnerabilidades já abordadas por outros pesquisadores como *Gasless Send* (SHIGEMURA et al., 2019), *Exception Disorder* (JIANG; LIU; CHAN, 2018), *Re-entrancy* (LUU et al., 2016) (TIKHOMIROV et al., 2018) (JIANG; LIU; CHAN, 2018), *Timestamp Dependency* (LUU et al., 2016) (JIANG; LIU; CHAN, 2018), *Block Number Dependency* (JIANG; LIU; CHAN, 2018), *Dangerous DelegateCall* (JIANG; LIU; CHAN, 2018), *Freezing Ether* (JIANG; LIU; CHAN, 2018) e *Integer Overflow/Underflow* (TORRES; SCHÜTTE; STATE, 2018) (TIKHOMIROV et al., 2018). O *sFuzz* avalia os contratos inteligentes executando as seguintes etapas:

- geração de scripts *bash* com comandos para análise da ABI dos contratos sob testes;
- configuração da rede a ser utilizada para execução dos testes dos contratos, incluindo a publicação de um *pool* de contas externas com saldo aleatório de *Ethers* e de dois contratos especiais que farão papel de ofensores: *Normal Attacker*, que lança exceção quando sua função *fallback* é chamada, e *Reentrancy Attacker*, que na função *fallback* reinvoca a função que disparou a execução desta função *fallback*;
- geração de uma população inicial com múltiplos casos de teste com valores *default*, seguido pela geração de casos de teste com valores de parâmetros aleatórios;
- execução da sequência de transações geradas anteriormente, o que levará ao disparo dos *hooks* da EVM com geração de dados sobre a execução das funções;

[8]Código de instrução operacional da Ethereum *Virtual Machine*

- avaliação dos dados gerados utilizando sua função *fitness* para empregar sua estratégia adaptativa multi-objetivo e, com base nesta avaliação, criação de uma nova geração de casos de teste. O item anterior é repetido até que o tempo definido para execução dos testes seja esgotado;

- análise dos *logs* gerados durante a execução dos testes para detecção do conjunto de vulnerabilidades de segurança suportados.

Os pesquisadores selecionaram 4.112 contratos com código-fonte disponível no Etherscan implementados utilizando Solidity 0.4.20, a versão mais popular do Solidity de acordo com autores do estudo. Esses contratos foram então submetidos ao processo de teste realizado pela ferramenta *sFuzz*. Devido ao grande número de vulnerabilidades identificadas durante o experimento, os pesquisadores relatam que seria impossível verificar todas manualmente. Assim, o procedimento adotado foi selecionar aleatoriamente 50 contratos em cada categoria de vulnerabilidade para validar se o problema encontrado é um falso positivo. Quando a categoria continha menos de 50 contratos vulneráveis, todos foram verificados.

Dos contratos analisados, 764 (18,58%) apresentaram a vulnerabilidade *Gasless Send*. Segundo os pesquisadores, nenhum dos 50 contratos analisados manualmente apresentou ocorrência de falso positivo. Trinta e seis contratos (0,88%) foram diagnosticados com a vulnerabilidade *Exception Disorder*, dado também verificado e no qual foi constatada a ausência de falso positivos. A vulnerabilidade *Re-entrancy* foi detectada em 29 contratos (0,71%), também sem falso positivos. A vulnerabilidade *Timestamp Dependency* foi detectada em 243 contratos (5,91%). Entretanto, dos 50 contratos analisados manualmente, 7 (14%) apontaram falso positivo. A vulnerabilidade *Block Number Dependency* foi detectada em 59 contratos (1,43%). Dentre os 50 contratos analisados manualmente, 10 (20%) foram identificados como falso positivo. A vulnerabilidade *Dangerous DelegateCall* foi encontrada em 17 contratos (0,41%) sem ocorrências de falso

positivo. A vulnerabilidade *Freezing Ether* foi identificada em 30 contratos, sem falso positivos. A vulnerabilidade *Integer Overflow* foi detectada em 98 contratos (2,38%) também sem ocorrências de falso positivos nos contratos analisados manualmente. A vulnerabilidade *Integer Underflow* foi detectada em 224 contratos (5,45%) apresentando uma taxa de 20% de falso positivos nas amostras analisadas. Finalmente, a vulnerabilidade *Freezing Ether* foi identificada em quinze contratos (0,36%) com a maior taxa de falso positivos: 40%.

2.6 Considerações Finais

Este capítulo apresentou os conceitos acerca da tecnologia *blockchain*, incluindo suas características e vantagens, definições relacionadas a contratos inteligentes, assim como foi discutido o valor e o potencial de aplicabilidade que essa nova geração da tecnologia *blockchain* traz consigo. Foi apresentado o Ecossistema Ethereum, em pleno uso hoje no mercado, incluindo a própria plataforma Ethereum com sua linguagem de programação de contratos inteligentes Solidity e o Truffle Framework, um *framework* amplamente utilizado para construção de testes de contratos inteligentes para a plataforma Ethereum. Finalmente, foi apresentado um conjunto de trabalhos relacionados ao processo de controle da qualidade dos contratos inteligentes produzidos por diferentes pesquisadores. É possível perceber que essas pesquisas têm como foco os testes de um conjunto de vulnerabilidades de segurança previamente conhecidas, enquanto este trabalho visa gerar testes que explorem a lógica de negócio representada no código-fonte de cada contrato inteligente.

O próximo capítulo apresenta a proposta de automação da geração de testes unitários automatizados para contratos inteligentes da plataforma Ethereum desenvolvidos na linguagem Solidity, seu conjunto de estratégias e limitações atuais.

3 PROPOSTA DE SOLUÇÃO

O objetivo deste capítulo é apresentar a proposta para automação da geração de testes de unidade para contratos inteligentes da plataforma Ethereum desenvolvidos na linguagem Solidity e utilizando o *framework* Truffle. A Seção 3.1 apresenta um exemplo de contrato inteligente escrito na linguagem. A Seção 3.2 aborda a importância das atividades de controle de qualidade no processo de desenvolvimento de contratos inteligentes. A Seção 3.3 apresenta a técnica proposta para construção dos testes dos contratos inteligentes incluindo cada uma de suas etapas. Finalmente, a Seção 3.4 apresenta as considerações finais deste capítulo.

3.1 Um exemplo de contrato inteligente

A fim de orientar a apresentação da técnica de geração de testes proposta nesta Dissertação, é tomado como exemplo um projeto Truffle fictício que tem como objetivo gerenciar uma linha pública de financiamento. Este projeto é composto por dois contratos: `Gerenciavel`, cujo código-fonte é apresentado no Algoritmo 3.1, e `LinhaFinanciamento`, apresentado no Algoritmo 3.2.

O contrato `Gerenciavel` é uma implementação de uma aplicação descentralizada (`dApp`) na qual uma conta específica na rede Ethereum atua como 'gerente'. Inicialmente, o gerente do contrato é a própria conta que criou o contrato na rede, conforme pode ser visto na atribuição realizada no construtor (linha 8). O contrato possui dois modificadores de função: `somenteGerente` e `naoGerente`. As funções marcadas com o modificador `somenteGerente` só poderão ser executadas a partir da conta da rede Ethereum cujo endereço seja o mesmo armazenado na variável de estado `gerente`. É o caso da

função `mudarGerente`, que permite que o gerente atual designe outra conta para assumir este papel. Caso outra conta tente executar esta função, uma exceção será lançada. As funções marcadas com o modificador `naoGerente` terão o comportamento oposto, ou seja, não poderão ser executadas a partir da conta da rede Ethereum cujo endereço esteja armazenado na variável de estado `gerente`.

```
1  pragma solidity 0.5.7;
2
3  contract Gerenciavel {
4      /** atual gerente da dApp */
5      address private gerente;
6
7      constructor() public {
8          gerente = msg.sender;
9      }
10
11     modifier somenteGerente {
12         require(
13             msg.sender == gerente,
14             "Somente o gerente desta dApp pode executar esta operacao"
15         );
16         _;
17     }
18     modifier naoGerente {
19         require(
20             msg.sender != gerente,
21             "O gerente desta dApp nao pode executar esta operacao"
22         );
23         _;
24     }
25
26     function mudarGerente(address novoGerente) public somenteGerente {
27         gerente = novoGerente;
28     }
29 }
```

Algoritmo 3.1: Código-fonte do contrato `Gerenciavel`

O contrato `LinhaFinanciamento` é uma especialização do `Gerenciavel`. A intenção com esta herança é que o contrato `LinhaFinanciamento` herde a característica de designar uma conta da rede à qual são reservadas certas operações e, ao mesmo tempo, outras são vedadas. Em função do seu propósito didático, a lógica de negócio deste contrato é simples, consistindo em apenas duas funções.

A função `solicitar` possui o modificador `naoGerente`, ou seja, a conta gerente da linha de financiamento não poderá fazer solicitações de financiamento. A função recebe a idade do proponente, pessoa física a quem se solicita a concessão de financiamento, e o valor de financiamento solicitado. Percebe-se que, na linha 24, a função faz uso do comando `require` para validar se o proponente é maior de idade. Ultrapassada esta exigência, é criada uma instância da *struct* `Financiamento` com os dados dos parâmetros de entrada, mais alguns atributos com valor padrão que indicam que a solicitação de financiamento ainda não foi apreciada. Em seguida, armazena-se esta instância em uma variável de estado do contrato.

Por sua vez, a função `deliberar` possui modificador `somenteGerente`, isto é, apenas o gerente poderá deliberar quanto à aprovação ou não de uma solicitação de financiamento. A função recebe como parâmetros o índice da solicitação de financiamento no *array* que armazena todas as solicitações recebidas e a indicação sobre a aprovação ou não da solicitação. O código na linha 29 verifica se o índice informado é válido. O código nas linhas 30 a 33 verifica se a solicitação já foi deliberada, confirmando se os atributos que guardam as informações acerca de qual gerente deliberou sobre a solicitação e qual a taxa do financiamento foi adotada continuam com os valores padrão, respectivamente, `address(0)` e 0 (zero). Superadas essas verificações, a taxa de juros anual para a solicitação é estabelecida com base na idade do proponente. Por fim, o endereço da conta do gerente da `dApp` é armazenado no atributo `gerente` da *struct*.

```
1  pragma solidity 0.5.7;
2
3  import "./Gerenciavel.sol";
4
5  contract LinhaFinanciamento is Gerenciavel {
6    struct Financiamento {
7      address origem;
8      uint8 idade;
9      uint256 valor;
10     bool aprovado;
11     uint8 taxa;
12     address gerente;
13   }
14
15   uint256 private capital;
16
```

```
17    constructor(uint256 capitalLF) public Gerenciavel() {
18      capital = capitalLF;
19    }
20
21    Financiamento[] private financiamentos;
22
23    function solicitar(uint8 idade, uint256 valor) public naoGerente {
24      require(idade >= 18, "O proponente deve ser maior de idade");
25      financiamentos.push(Financiamento(msg.sender, idade, valor, false, 0, address
            (0)));
26    }
27
28    function deliberar(uint256 indiceArray, bool aprovado) public somenteGerente {
29      require(financiamentos.length > indiceArray, "A solicitacao nao existe");
30      require(
31        financiamentos[indiceArray].gerente == address(0) && financiamentos[
              indiceArray].taxa == 0,
32        "Solicitacao ja deliberada"
33      );
34      Financiamento memory f = financiamentos[indiceArray];
35      if (aprovado) {
36        uint8 taxa;
37        if (f.idade > 65) {
38          f.taxa = 15; //1,5% aa
39        } else {
40          f.taxa = 20; //2,0% aa
41        }
42        f.aprovado = true;
43        capital = capital - f.valor;
44      } else {
45        f.aprovado = false;
46      }
47      f.gerente = msg.sender;
48      financiamentos[indiceArray] = f;
49    }
50  }
```

Algoritmo 3.2: Código-fonte do contrato `LinhaFinanciamento`

3.2 Controle de qualidade utilizando testes

Como em qualquer tipo de software, o controle da qualidade por meio da execução de testes automatizados também se aplica aos contratos inteligentes. Alguns exemplos de testes que devem ser realizados no registro das solicitações de financiamento estão listados a seguir:

- solicitar um financiamento com a conta do gerente da linha de financiamento e confirmar se a exceção esperada é lançada;

- solicitar um financiamento informando uma idade menor do que 18 anos e confirmar se ocorre a exceção conforme esperado;

- solicitar um financiamento com uma conta que não seja a mesma do gerente da linha de financiamento, informando idade maior ou igual a 18 anos, e confirmar se a solicitação de financiamento foi registrada ao final da execução da função.

Idealmente, os contratos inteligentes deveriam passar por baterias de testes que exercitassem e validassem todos os cenários de uso possíveis, a fim de se atestar a sua qualidade com maior grau de certeza. Isso passa, inevitavelmente, pela identificação de tais cenários e de seus respectivos requisitos de teste. Em um projeto pequeno e simples, como o exemplo de linha de financiamento acima, o esforço necessário para a identificação manual dos requisitos de testes pode ser considerado pequeno, e, levando em conta sua baixa complexidade, a probabilidade de erro humano é reduzida. Entretanto, à medida que a lógica de negócio gerenciada pelos contratos inteligentes se torna mais complexa, o esforço para a identificação dos requisitos de teste também aumenta, assim como as chances de algum cenário não ser exercitado durante as baterias de teste e, consequentemente, do contrato ser publicado na rede com defeitos que possam acarretar prejuízos.

A proposta deste trabalho de pesquisa é, através da análise estática do código-fonte dos contratos inteligentes, automatizar:

- a identificação dos requisitos de teste;

- a geração dos dados de teste;

- a identificação de chamadas a funções necessárias antes de se invocar a função sob teste a fim de se colocar os contratos inteligentes em um estado que atenda às pré-condições do teste desejado.

3.3 Testes baseados em restrições

Esta seção apresenta as etapas propostas neste trabalho de pesquisa para a geração de testes para código Solidity. A primeira delas consiste na construção de um grafo representando as relações entre os contratos e entre os seus componentes, tais como funções, variáveis de estado, enumeradores, *structs*, entre outros, bem como a identificação de um conjunto de atributos destes elementos do grafo. A segunda etapa consiste na geração do código responsável pela publicação dos contratos a serem testados em uma rede Ethereum. A etapa seguinte utiliza as expressões relacionais presentes no código-fonte para identificar os requisitos de teste de cada função. A etapa subsequente é responsável por gerar os dados de teste, ou seja, por gerar os valores dos parâmetros de entrada das funções invocadas durante a execução dos testes. A última etapa identifica as pré-condições de um teste, isto é, o estado em que o contrato precisa estar para que o teste seja executado, e define o conjunto de chamadas a funções do contrato para alcançar esse estado. A figura 3.1 apresenta uma abstração do processo de geração dos testes empregado pela técnica proposta.

3.3.1 Construção de grafo de referências

A primeira etapa consiste na varredura de todos os arquivos com código-fonte Solidity do projeto com o propósito de identificar os contratos e realizar um mapeamento da estrutura de cada um deles. Para cada contrato, são identificadas suas funções, modificadores, estruturas de dados, variáveis de estado, dependências com outros contratos e suas relações

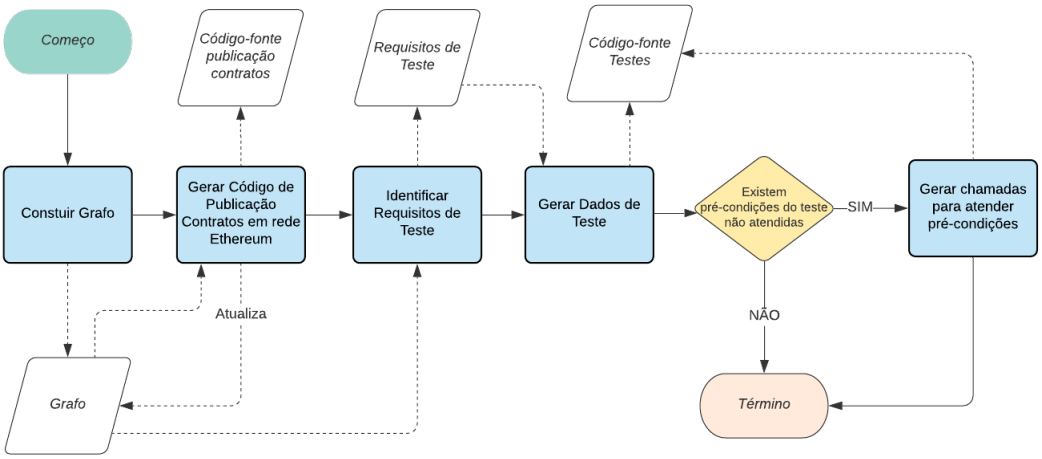


Figura 3.1: Processo de geração de testes de unidade da proposta da pesquisa

de herança. Para cada função do contrato, são mapeados seus parâmetros de entrada e saída, seus modificadores, outras funções que a função inicial invoca e variáveis de estado do contrato sobre as quais a função executa operação de escrita. Além disso, também são extraídas algumas informações adicionais sobre essas operação de escrita, por exemplo, se ela é executada incondicionalmente ou se está condicionada a alguma restrição. Para cada variável de estado do contrato, são identificados atributos como visibilidade, tipo, se é uma constante ou não, qual seu valor inicial e quais as funções do contrato que executam operação de escrita nesta variável. A figura 3.2 apresenta uma ilustração simplificada deste grafo gerado com base no projeto de exemplo descrito na seção 3.1.

A construção começa por adicionar à estrutura do grafo apenas os contratos, enumeradores e as estruturas de dados, pois esses podem, potencialmente, ser referenciados por qualquer uma das estruturas subjacentes. Neste ponto, aqueles contratos que não têm funções, com alguma função sem implementação[1] ou, ainda, com um construtor com visibilidade *internal*[2] serão sinalizados como abstratos, não sendo possível instanciá-los.

[1]https://solidity.readthedocs.io/en/v0.5.0/contracts.html#abstract-contracts
[2]https://solidity.readthedocs.io/en/v0.5.0/contracts.html#constructors

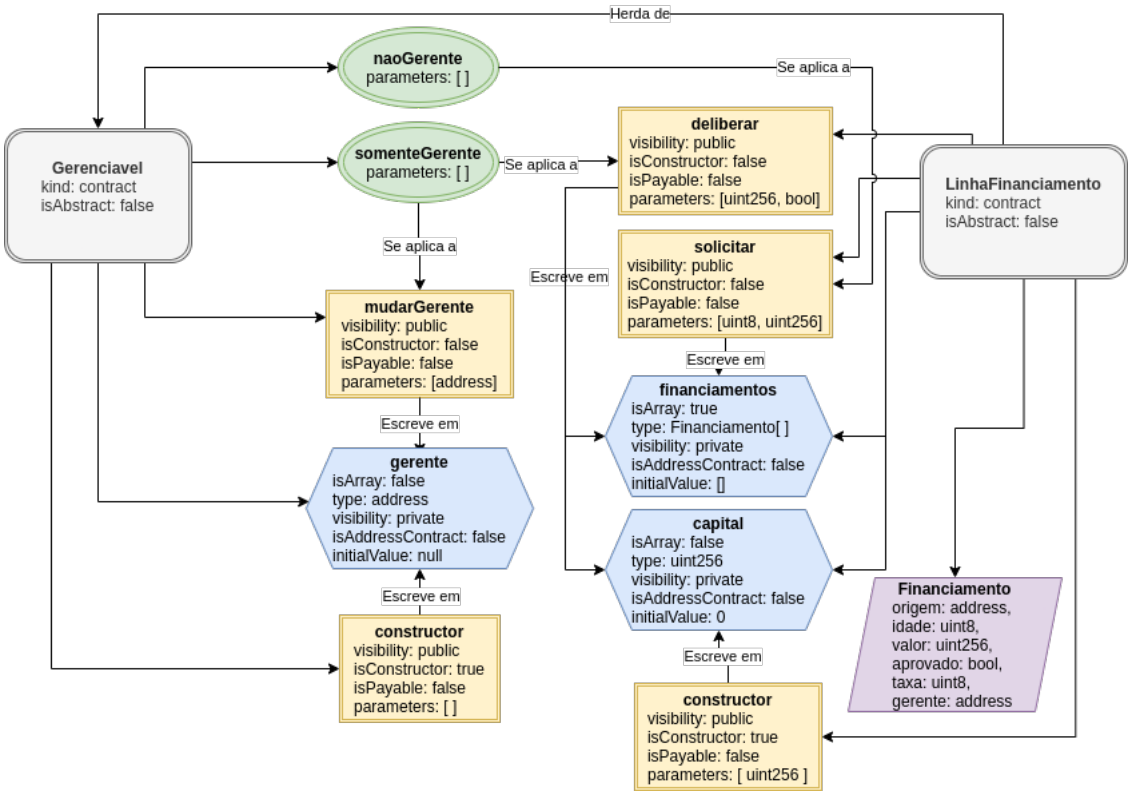


Figura 3.2: Representação visual do grafo de referências do projeto Truffle apresentado na seção 3.1. As formas retangulares com cantos arredondados em cinza representam os contratos. As formas elípticas em verde representam os modificadores de função. As formas retangulares em amarelo representam as funções. As formas hexagonais em azul representam as variáveis de estado. A forma de paralelogramo em magenta representa a única *struct* do exemplo.

A partir deste estágio, são identificados e adicionados ao grafo as relações de herança e outras relações de dependência entre os contratos, os modificadores de função declarados nos contratos, as variáveis de estado dos contratos, o construtor e as funções de cada contrato.

Baseando-se nas informações apuradas sobre as relações de herança entre os contratos, é feita uma segunda análise quanto à abstração dos contratos. Para os contratos que possuam implementação de um construtor e herdem de outros contratos e, ao mesmo

tempo, os contratos que lhes servem de superclasse também implementem construtores que recebam parâmetros de entrada, é averiguado se o contrato filho especifica ou repassa valores para todos os parâmetros do construtor pai. Em caso negativo, o contrato filho é sinalizado como abstrato[3].

Com o conjunto de informações acumuladas no grafo até este estágio, é realizada uma análise especificamente sobre os parâmetros e variáveis de estado do tipo *address*. A linguagem Solidity define o tipo *address* para armazenar o endereço de uma conta na rede, não sendo possível distinguir se essa conta é externa ou pertence ao contrato. Contudo, no código Solidity, é possível obter a instância de um contrato publicada em um determinado endereço invocando o nome do contrato como se fosse uma função, passando como parâmetro de entrada o respectivo endereço, como consta na função `funcaoX` do Algoritmo 3.3. A fim de melhorar os resultados na etapa de geração dos dados de teste, realiza-se neste momento uma busca no código-fonte por ocorrências deste tipo, onde uma variável de estado do tipo *address* é passada como único parâmetro de entrada para um contrato. Sinaliza-se no grafo que esta variável de estado é um endereço de contrato e também armazena-se os contratos identificados neste tipo de utilização da variável. O mesmo é feito quando parâmetros de entrada são assim utilizados, como ocorre na função `funcaoY` do mesmo algoritmo, ou ainda quando é identificado que o parâmetro de entrada é utilizado indiretamente desta forma, por exemplo, quando é atribuído a uma variável de estado que será utilizada como parâmetro para buscar o contrato, como pode ser observado no *constructor* do Algoritmo 3.3.

```
1  contract ContratoA {
2
3      address variavelEstadoAddress;
4
5      constructor(address paramAddress) public{
6          variavelEstadoAddress = paramAddress;
7      }
8
9      function funcaoX() public {
10         ContratoB b = ContratoB(variavelEstadoAddress);
11     }
```

[3]https://solidity.readthedocs.io/en/v0.5.0/contracts.html#arguments-for-base-constructors

```
12
13      function funcaoY(address paramAddress) public{
14          ContratoB b = ContratoB(paramAddress);
15      }
16  }
```

Algoritmo 3.3: Código-fonte de um contrato que busca a instância de outro com uma variável de estado do tipo *address*.

Por fim, percorre-se as instruções das funções de cada contrato e identifica quais variáveis de estado são afetadas por operações de escrita por quais funções e vice-versa. Esse conjunto de informações sobre os contratos, suas relações, seus elementos estruturais e sobre a relação entre estes elementos será fundamental na execução dos demais procedimentos para geração dos testes automatizados.

3.3.2 Geração de código validado de publicação de contratos inteligentes

O objetivo nesta etapa é produzir o código JavaScript capaz de publicar as novas instâncias dos contratos na rede Ethereum utilizada durante os testes e guardar referências para tais instâncias em variáveis utilizadas no código das funções de teste. Um requisito na geração do código de publicação dos contratos é que sua execução não falhe. Do contrário, os testes não poderão ser executados pela falta de uma instância do contrato testado na rede.

Primeiramente, é feita uma seleção que inclui apenas os contratos do projeto no grafo que não estejam sinalizados como abstratos. Além disso, os construtores de contratos podem possuir parâmetros de entrada cujos tipos sejam outros contratos ou interfaces do projeto, constituindo-se assim uma dependência da existência prévia de uma instância do tipo referenciado. Para resolver estas dependências, os contratos são ordenados de tal forma que os contratos dos quais se depende sejam instanciados antes dos seus dependentes.

Em seguida, itera-se pelos contratos selecionados para realizar a geração da função `beforeEach` de forma incremental e acumulativa. Esta função será chamada antes da execução de cada teste e nela os contratos utilizados nos casos de teste serão publicados na rede. Para cada contrato, identifica-se os requisitos de teste existentes para seu construtor. Assim como qualquer função, o construtor pode ser composto de ramos e restrições, resultando em requisitos de testes cujo resultado esperado seja o lançamento de exceção e outros não. Para a publicação de uma instância do contrato na rede, a seleção de um único requisito de teste do construtor, cujo resultado esperado não seja o lançamento de uma exceção, é suficiente. Assim, seleciona-se o primeiro requisito de teste com esta característica, bem como os seus dados de teste. Além destas, o processo também fará uso de informações sobre a dependência do contrato em relação a bibliotecas Solidity, haja vista que quando um contrato faz uso de alguma biblioteca, este vínculo precisa ser estabelecido no momento da publicação do contrato na rede.

De posse dessas informações, utiliza-se uma cópia temporária do projeto sob teste, gravando no subdiretório 'test' um arquivo de teste contendo apenas o código da função `beforeEach` acumulado até o momento, conforme apresentado no Algoritmo 3.4.

```
1  const truffleAssert = require('truffle-assertions');
2  const Gerenciavel = artifacts.require("Gerenciavel");
3  const LinhaFinanciamento = artifacts.require("LinhaFinanciamento");
4
5  contract("validation beforeEach",(accounts)=>{
6    let contractGerenciavel = null;
7
8    beforeEach(async () => {
9      contractGerenciavel = await Gerenciavel.new({from:accounts[0]});
10   });
11
12 });
```

Algoritmo 3.4: Código-fonte da validação do beforeEach

Realiza-se então a execução do comando `truffle test`. Se o comando for executado com sucesso, significa que o código `beforeEach` construído é válido até o momento e, assim, pode-se seguir para geração do incremento de código que instanciará

o próximo contrato da lista. Contudo, se a execução do comando retornar um erro, há algum problema no incremento do código do `beforeEach`. Neste caso, retorna-se para último estado válido do código `beforeEach` e avalia-se a causa do erro para determinar o próximo passo.

Por exemplo, caso o erro tenha ocorrido por falta de Ether na conta sendo utilizada para executar o teste que tem apenas o `beforeEach` ou porque a execução não encontrou a instalação do Truffle, o processo é interrompido imediatamente. Caso a causa seja a falta de alguma biblioteca Solidity, passa-se para a geração do código de publicação do próximo contrato da lista. Contudo, caso nenhuma condição anormal conhecida tenha sido identificada, realiza-se uma nova tentativa de geração do código para instanciar o contrato da iteração que falhou, regerando também, de forma aleatória[4], valores de parâmetros de entrada potencialmente distintos daqueles utilizados na execução mal sucedida.

Para cada contrato, também são identificadas as variáveis de estado que recebem atribuição de valor proveniente de algum parâmetro de entrada ou de um valor literal no construtor do contrato. Tais valores são armazenados em memória para utilização no processo de geração dos valores de parâmetros das funções que tenham restrições relacionadas a estas variáveis de estado.

Esse processo resultará em um código validado da função `beforeEach` que instancia os contratos do projeto na rede e realiza o vínculo entre os contratos e as bibliotecas utilizadas. Este código será executado antes de cada um dos teste de unidade. Assim, todos os testes são iniciados com uma instância do contrato no mesmo estado, estado esse que pode ser alterado dentro do próprio código do teste quando necessário para exercitar o requisito de teste pretendido. Adicionalmente, resultará em uma lista de variáveis de estado cujo valor inicial é modificado pelo construtor e seus respectivos novos valores.

[4] A geração aleatória de valores é realizada via software utilizando recursos oferecidos pela linguagem JavaScript. Portanto, trata-se de uma pseudo aleatoriedade

3.3.3 Identificação de requisitos de teste baseada em expressões relacionais

Dependendo dos requisitos e da sua implementação, uma função pode apresentar múltiplos caminhos de execução. Em outros casos, a função pode ser simples a ponto de ter um único caminho, como ocorre em uma função que apenas atribua o valor de um parâmetro de entrada a uma variável de estado.

Esta etapa da técnica de identificação de requisitos de teste proposta por esta pesquisa consiste em percorrer o código-fonte de cada função, mapear todos os ramos de decisão (*IF*, *ELSE IF* e *ELSE*)[5] com suas respectivas condições e, com base neste mapeamento, estabelecer os caminhos de execução da função. Por exemplo, na função `deliberar` do Algoritmo 3.2, o primeiro caminho é definido pela condição na linha 35. A condição para execução deste caminho é que o parâmetro de entrada '`aprovado`' seja igual a `true`. Presumivelmente, o outro caminho nesta função é aquele em que o parâmetro '`aprovado`' seja `false`. Em decorrência desta condição, já é possível afirmar que os testes da função `deliberar` deverão contemplar, no mínimo, um caso em que o parâmetro '`aprovado`' seja `true` e outro caso em que o mesmo parâmetro seja `false`.

Com os caminhos de execução e suas respectivas condições mapeadas, é dado início a uma segunda etapa em que o código-fonte da função é percorrido com o objetivo de mapear restrições de validação presentes no código. Essas restrições consistem na utilização do comando `require`, como pode ser visto nas linhas 24, 29 e 30 do Algoritmo 3.2. Esse comando recebe como primeiro parâmetro uma expressão *booleana* e, se esta expressão for falsa, lança uma exceção. Desse modo, para que um caminho de execução seja exercitado, além das condições associadas a ele serem atendidas, também deverá passar por todas as restrições de validação realizadas neste caminho.

[5] As decisões contidas em laços *FOR* e *WHILE* estão fora do escopo da pesquisa

Isto posto, para cada caminho de execução identificado em uma função, faz-se o mapeamento das restrições existentes no código-fonte ao longo do caminho. A análise das restrições será realizada não somente pelo corpo da própria função, mas também pelo corpo de outras funções do próprio contrato invocadas por ela e dos modificadores aplicados a ela. Por exemplo, o conjunto de restrições da função `solicitar` é formado pela validação da idade existente no corpo da função, mas também pela validação presente no modificador `naoGerente`. Ao longo deste percurso, a análise de código-fonte acumula as restrições encontradas para cada caminho de execução.

De posse de todos os caminhos de execução da função a ser testada e de todas as restrições de validação presentes em cada um destes caminhos, o algoritmo utiliza esses dados para estabelecer os requisitos de teste da função. Serão gerados os requisitos de testes ordinários e requisitos de teste de exceção na execução de cada caminho. No contexto desta pesquisa, entende-se por "teste ordinário" aquele com a intenção de se executar um caminho da função de forma completa sem lançar exceções, ou seja, os valores de entrada da função são gerados de forma que se passe com sucesso por toda e qualquer restrição de validação que exista no caminho de execução. Um "teste de exceção", por sua vez, é aquele cujo objetivo é exercitar o lançamento de uma exceção em uma restrição de validação específica.

Cada caminho de execução encontrado no código da função dará origem a um requisito de teste ordinário. Este teste consiste na chamada da função com valores de parâmetros que estejam em conformidade com todas as suas restrições, isto é, atenderá à expressão *Booleana* de todas as chamadas ao comando `require` encontradas no caminho de execução. Para cada restrição encontrada no caminho de execução, será gerado um requisito de teste de exceção, que consistirá na chamada da função com valores de parâmetros que levarão a respectiva restrição a lançar a exceção prevista.

Nota-se que a técnica para identificação dos requisitos de teste aqui apresentada é fundamentada na identificação das expressões relacionais presentes no código-fonte dos contratos inteligentes. Por definição, esse tipo de expressão realiza a comparação de dois elementos e retorna um valor *Booleano*. Os ramos de decisão encontrados na função determinam o número de caminhos de execução, enquanto as restrições de validação definem o que é entendido como requisito de teste ordinário e requisito de teste de exceção em cada um destes caminhos. Cabe destacar que tais expressões relacionais, algumas vezes, poderão ser formadas pela composição de um conjunto de expressões relacionais, isto é, várias expressões relacionais concatenadas com operadores lógicos `AND` ou `OR` que resultarão em um único resultado *Booleano*. Nestes casos, a pesquisa responde de forma diferenciada de acordo com o operador lógico envolvido. Quando se tratar do operador `AND`, todas as expressões relacionais da composição devem ser atendidas para se satisfaça à condição ou restrição. Quando se tratar do operador `OR`, o algoritmo buscará atender a apenas uma destas expressões relacionais da composição durante a geração de dados de teste e ignorará as demais. A escolha de qual das expressões relacionais será atendida é realizada de forma aleatória.

Ao fim deste processo, portanto, tem-se como resultado a identificação de um conjunto de requisitos de teste com suas respectivas condições e restrições para cada uma das funções dos contratos do projeto sob teste. Tomando como exemplo a função `deliberar` do Algoritmo 3.2, teríamos os seguintes requisitos de teste ordinários:

- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, cujos atributos `gerente` e `taxa` estejam zerados e cujo proponente tenha informado estar com mais de 65 anos de idade;

- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, cu-

jos atributos `gerente` e `taxa` estejam zerados e cujo proponente tenha informado estar com 65 anos de idade ou menos;

- reprovar um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento e cujos atributos `gerente` e `taxa` estejam zerados.

Por sua vez, considerando as restrições de validação existentes no corpo da função e do seu único modificador, `somenteGerente`, os requisitos de teste de exceção seriam:

- aprovação de um financiamento utilizando uma conta não gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, cujos atributos `gerente` e `taxa` estejam zerados e cujo proponente tenha informado estar com mais de 65 anos de idade;
- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice inválido no *array* de solicitações de financiamento, cujos atributos `gerente` e `taxa` estejam zerados e cujo proponente tenha informado estar com mais de 65 anos de idade;
- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice inválido no *array* de solicitações de financiamento;
- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, seu atributo `gerente` já tenha sido atribuído um endereço diferente de zero, seu atributo `taxa` esteja zerado e cujo proponente tenha informado estar com mais de 65 anos de idade;
- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, seu

atributo `gerente` esteja zerado, seu atributo `taxa` tenha um valor diferente de zero e cujo proponente tenha informado estar com mais de 65 anos de idade;

- aprovação de um financiamento utilizando uma conta não gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, cujos atributos `gerente` e `taxa` estejam zerados e cujo proponente tenha informado estar com 65 anos de idade ou menos;

- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice inválido no *array* de solicitações de financiamento, cujos atributos `gerente` e `taxa` estejam zerados e cujo proponente tenha informado estar com 65 anos de idade ou menos;

- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, seu atributo `gerente` já tenha sido atribuído um endereço diferente de zero, seu atributo `taxa` esteja zerado e cujo proponente tenha informado estar com 65 anos de idade ou menos;

- aprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, seu atributo `gerente` esteja zerado, seu atributo `taxa` tenha um valor diferente de zero e cujo proponente tenha informado estar com 65 anos de idade ou menos;

- reprovação de um financiamento utilizando uma conta não gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, cujos atributos `gerente` e `taxa` estejam zerados;

- reprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice inválido no *array* de solicitações de financiamento, cujos atributos `gerente` e `taxa` estejam zerados;

- reprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, cujo atributo `gerente` já tenha sido atribuído um endereço diferente de zero e o atributo `taxa` esteja zerado;

- reprovação de um financiamento utilizando a conta gerente da linha de financiamento, informando um índice válido no *array* de solicitações de financiamento, cujo atributo `gerente` esteja zerado e cujo atributo `taxa` tenha sido atribuído um valor diferente de zero.

3.3.4 Limitações de escopo do tratamento das condições e restrições mapeadas

Conforme mencionado na seção anterior, a definição das condições e restrições de cada requisito de teste é fundamentada na identificação das expressões relacionais existentes no código. Em um contrato inteligente, os elementos de uma expressão relacional podem ser dos mais variados tipos, como um valor literal, uma variável local, um parâmetro de entrada da função onde se encontra a expressão, um atributo de um parâmetro de entrada, uma variável de estado do contrato, um atributo de uma variável de estado, o resultado de uma operação matemática, um acesso indexado a um elemento de `array` ou `mapping`, o resultado da chamada de uma função do próprio contrato ou de outro contrato e, presumivelmente, uma composição complexa de vários desses elementos, como o exemplo apresentado no Algoritmo 3.5.

```
1 function funcaoComOperacaoRelacionalComplexa(uint indexadorArray) public {
2     require(funcaoRetornaArray(variavelEstadoMapping[variavelEstado].
          atributoStructDoMapping)[indexadorArray].atributoStructDoArray >
          outraVariavelEstado / variavelEstadoArray.length + indexadorArray);
3     ...
4 }
```

Algoritmo 3.5: Código-fonte de uma operação relacional complexa

Importante destacar que, no procedimento de geração de dados de teste, esta pesquisa trata apenas de um subconjunto dos tipos de elementos encontrados em expressões relacionais dos contratos inteligentes Solidity, a saber:

- valor literal;
- membro de uma enumeração;
- parâmetro de entrada de função;
- atributo de parâmetro de entrada de função;
- variável de estado de contrato;
- o atributo *length* para a variável de estado de contrato que seja um *array*;
- resultado da chamada à função `address(0)`, bastante utilizada para validar se um endereço foi informado;
- resultado de função que somente retorne um valor literal ou variável de estado (função *getter*).

Desta forma, quando for mencionado o atendimento das condições e restrições dos requisitos de teste, apenas as expressões relacionais que envolvam somente elementos do escopo atendido são consideradas. Estão fora do escopo desta pesquisa as expressões relacionais em que ao menos um dos membros não esteja dentre os listados acima.

3.3.5 Geração de dados de teste baseada em restrições

Identificados os caminhos de execução de uma função e filtrados aqueles que envolvem apenas elementos indicados no escopo desta pesquisa, o próximo passo consiste em gerar

os dados de teste, isto é, produzir os valores de parâmetros de entrada das funções a serem invocadas durante a execução dos testes. Para tal, esta etapa recebe o requisito de teste com suas respectivas condições e restrições.

Para a geração dos valores de parâmetros de entrada da função que exercitem um requisito de teste ordinário, o conjunto de parâmetros será iterado seguindo os passos abaixo para cada parâmetro:

1. obtém-se as condições e restrições do requisito de teste que se refiram ao parâmetro e cuja contraparte na expressão relacional esteja no escopo atendido pela pesquisa, conforme citado na Subseção 3.3.4;

2. quando o operador de alguma das condições ou restrições for a igualdade (==) e esta comparação envolvendo o parâmetro é realizada contra um valor literal, contra uma variável de estado ou contra um parâmetro de entrada que já tenha seu valor estabelecido, o valor com o qual está sendo comparado será retornado. Entretanto, caso nenhum dos operadores seja a igualdade ou se a variável de estado ou outro parâmetro de entrada com o qual o parâmetro corrente está sendo comparado ainda não possua um valor conhecido, será gerado aleatoriamente um valor que atenda a todas as condições e restrições do requisito de teste resultantes do passo anterior;

3. armazena-se o valor gerado para o parâmetro corrente em memória para que, posteriormente, possa ser utilizado caso alguma condição ou restrição faça comparação de outro parâmetro com o parâmetro corrente.

Perceba que, no primeiro passo do processo acima descrito, são desconsideradas as condições e restrições cuja expressão relacional de origem faça referência a tipos de elementos não tratados pela pesquisa. Com isto, o atendimento destas condições e restrições será não determinístico: ele poderá ocorrer ou não em decorrência do fator aleatório

utilizado na geração dos valores dos parâmetros de entrada das funções que compõem o requisito de teste. Isso significa que, na função `deliberar`, os valores de parâmetros de entrada das chamadas de funções serão gerados de forma determinística para atender ou violar a restrição presente na linha 29 e as condições presentes nas linhas 35 e 44. Contudo, o atendimento ou violação das restrições das linhas 30 a 33 e as condições das linhas 37 e 39 são não determinísticos, dependem do fator de aleatoriedade da geração dos valores.

Para a geração dos valores de parâmetros de entrada com fim de produzir um teste que viole uma restrição, parte-se dos valores de parâmetros de entrada definidos para o o teste ordinário e da restrição cuja falha o teste se propõe a cobrir. Com base nestes dados, os valores dos parâmetros de entrada para atender tal requisito de teste serão os mesmos utilizados para executar o teste ordinário, com exceção do parâmetro referenciado na restrição que será violada. O valor para este parâmetro será um valor que leve o contrato a lançar uma exceção por inconformidade com a expressão *Booleana* passada como argumento para o `require`.

Por exemplo, uma das restrições que o teste ordinário da função `solicitar` do Algoritmo 3.2 precisa atender é que a idade do proponente seja maior do que 18 anos. A partir do teste ordinário, que já possui todos os dados necessários para atender a todas as restrições, será modificado apenas o valor do parâmetro "idade" com o objetivo de violar tal restrição. A geração deste valor desconforme parte do valor e do operador da restrição, neste caso, 18 e '>', respectivamente. Quando se tratar dos operadores '>' (maior), '<' (menor) ou '!=' (diferente), o valor retornado é exatamente o valor da restrição, neste exemplo, 18. Quando se tratar de um operador '==' (igualdade) ou '<=' (menor ou igual), soma-se 1 (um) ao valor da restrição. Por fim, para um operador '>=' subtrai-se 1 (um) do valor da restrição.

Nota-se que o exemplo acima tratou de uma restrição cujo valor é do tipo numérico. Para outros tipos de dados, uma abordagem semelhante é utilizada conforme sintetizado nas Tabelas 3.1, 3.2, 3.3, 3.4 e 3.5.

Tabela 3.1: Tipos Numéricos: Valor de retorno de violação de restrições por operador

Operador	**Valor retornado**
!=, >, <	o próprio valor de comparação da restrição
==, <=	valor de comparação da restrição + 1
>=	valor de comparação da restrição - 1

Tabela 3.2: Tipo *String*: Valor de retorno de violação de restrições por operador.

Operador	**Valor retornado**
!=, >, <	o próprio valor de comparação da restrição
==, <=	'Z' concatenado ao valor de comparação da restrição
>=	'a' concatenado ao valor de comparação da restrição

Tabela 3.3: Tipo *Address*: Valor de retorno de violação de restrições por operador.

Operador	**Valor retornado**
!=	o próprio valor de comparação da restrição
==	uma referência a qualquer outra conta disponibilizada pelo Truffle na rede de testes que seja distinta do valor da comparação
>, <, >=, <=	Não se aplica

3.3.6 Definição das chamadas para atendimento às pré-condições do teste

Quando uma instância do contrato é publicada na rede, suas variáveis de estado possuem um valor inicial. Este valor pode ter sido atribuído no corpo de sua função construtora ou das funções construtoras dos contratos dos quais este herda, na declaração da própria variável ou, na ausência de uma definição explícita, valores *default* são estabelecidos pela própria linguagem Solidity[6].

[6]https://solidity.readthedocs.io/en/v0.5.0/control-structures.html

Tabela 3.4: Tipo *Boolean*: Valor de retorno de violação de restrições por operador

Operador	**Valor retornado**
!=	o próprio valor de comparação da restrição
==	valor inverso da restrição (se *true*, *false*. Se *false*, *true*.)
>, <, >=, <=	Não se aplica

Tabela 3.5: Tipo Bytes: Valor de retorno de violação de restrições por operador

Operador	**Valor retornado**
!=	o próprio valor de comparação da restrição
==	Um valor aleatório distinto do valor da restrição
>, <, <=, >=	Não se aplica

Quando uma condição ou restrição do requisito de teste envolve uma validação contra uma variável de estado e seu valor inicial não atende às condições para realizar o teste em questão, faz-se necessária a execução de funções no contrato que possam modificar a variável de estado de forma que seu valor atenda às pré-condições do requisito de teste, isto é, que leve o contrato para um estado em que seja possível exercitar o teste pretendido. Para tanto, é necessário identificar tais pré-condições do requisito de teste, determinar o conjunto de chamadas a funções necessário para atendê-las e incorporar esse conjunto de chamadas ao código-fonte do teste de unidade em questão.

A partir das condições e restrições de um requisito de teste, restringe-se o conjunto de funções que podem ser chamadas a aquelas que alterem o valor de alguma variável de estado e que o valor atual desta variável de estado não atenda às pré-condições necessárias para realizar o teste em questão. Neste processo, são avaliadas também eventuais restrições de estado que a própria função que modifica a variável de estado possa ter e, desta forma, pode dar origem à chamada de outra função em cadeia.

Por exemplo, a função `deliberar` do Algoritmo 3.2 tem como sua primeira restrição a condição de que seja invocada apenas com a conta gerente da linha de financiamento. Como a conta que realiza a operação que instancia o contrato na rede já é

atribuída à variável de estado `gerente` no construtor do contrato `Gerenciavel`, essa restrição já é atendida pelo seu valor inicial. A segunda restrição, entretanto, estabelece que o tamanho do *array* `financiamentos` seja maior que o parâmetro entrada `indiceArray`. O tamanho inicial da variável `financiamentos` é zero e como o menor valor possível para o tipo `uint` também é zero, será necessária a invocação de uma função que altere o tamanho do *array* de `financiamentos`, como a função `solicitar`. Assim, será realizada uma chamada à função `solicitar`, atendendo tanto à sua restrição do parâmetro `idade` quanto de ser invocada por uma conta distinta da conta `gerente`.

Ao final deste procedimento, um conjunto de chamadas de função que coloque o contrato no estado necessário para iniciar o teste é incorporado ao próprio teste unitário e será executado imediatamente antes da função alvo.

3.4 Considerações Finais

Este capítulo apresentou a técnica proposta neste trabalho de pesquisa para automação da geração de teste unitários automatizados para contratos inteligentes da plataforma Ethereum desenvolvidos na linguagem Solidity.

A técnica inclui a identificação dos requisitos de teste de cada função do contrato inteligente através do reconhecimento de expressões relacionais presentes no seu código-fonte, a geração dos dados de teste, a identificação das pré-condições que precisam ser atendidas para cada teste e a geração do código responsável pela publicação dos contratos inteligentes na rede Ethereum onde serão executados os testes.

O próximo capítulo apresenta a avaliação experimental desta proposta, onde será descrito o processo e critérios utilizados para seleção dos projetos analisados, bem como

os resultados de cobertura de ramos alcançados pela versão original dos testes existentes nestes projetos e os resultados alcançados após aplicação da proposta desta pesquisa.

4 AVALIAÇÃO EXPERIMENTAL DA PROPOSTA

4.1 Introdução

Este capítulo tem como objetivo apresentar a análise experimental da solução proposta para a geração de testes para contratos inteligentes da plataforma Ethereum. Na Seção 4.2, é apresentada a sistemática aplicada para a seleção de um conjunto inicial de projetos de contratos inteligentes com código-fonte aberto e disponível na rede mundial de computadores. A Seção 4.3 aborda os passos realizados para filtrar os projetos obtidos na etapa anterior e selecionar o conjunto final de projetos utilizados nos estudos experimentais. Na Seção 4.4, são apresentados os resultados obtidos após uma série de 30 execuções para cada projeto utilizando a proposta da pesquisa. Finalmente, a Seção 4.5 apresenta as considerações finais deste capítulo.

4.2 Seleção inicial de projetos para análise

O primeiro passo da análise experimental aqui conduzida foi catalogar o maior número possível de projetos de contratos inteligentes da plataforma Ethereum desenvolvidos em Solidity e com código-fonte aberto e disponível na rede mundial de computadores. Considerando sua relevância, popularidade e a disponibilidade de uma API para consultas, a busca por projetos realizada como parte desta pesquisa ficou restrita a projetos disponíveis de forma pública na plataforma de hospedagem de código-fonte GitHub.

Primeiramente, foi utilizada a API REST v3[1] do GitHub para obter todos os repositórios cuja linguagem de programação principal estivesse estabelecida como Solidity[2]. É importante registrar que, mesmo que o repositório tenha arquivos de contratos em Solidity, ele pode não ser retornado nesta consulta. É o caso, por exemplo, do projeto BNDESToken[3], que tem arquivos Solidity em sua solução, mas que possui como linguagem principal o JavaScript.

Uma consulta executada em 15 de julho de 2019 encontrou um total de 1.054 repositórios com projetos Solidity. A partir desta lista de repositórios, foi realizada uma verificação para contar quantos deles possuíam ao menos um diretório "test" ou "tests", chegando ao número de 502 repositórios (48%). Durante a condução desta análise, a frequente ocorrência de arquivos de configuração do *framework* Truffle nos repositórios motivou uma nova consulta, desta vez para contabilizar quantos dos repositórios possuíam ao menos um arquivo "truffle-config.js" ou "truffle.js". A nova consulta retornou um total de 360 repositórios (34%).

Na sequência, foi contabilizado o conjunto interseção destes dois grupos, isto é, todos os repositórios que se baseavam no *framework* Truffle e possuíam ao menos um diretório "test" ou "tests", chegando-se a 269 repositórios (26%). O resumo dos números apurados nestas análises pode ser visualizado na Figura 4.1. Tomando por base o número de repositórios Solidity em que se verifica a sua presença, é possível constatar que o *framework* Truffle é utilizado com muita frequência em projetos de contratos inteligentes na linguagem Solidity.

Conforme citado no Capítulo 2, o *framework* Truffle oferece um conjunto de facilidades aos desenvolvedores que o adotam em seus projetos. Além disso, projetos que

[1] https://developer.github.com/v3/

[2] https://api.github.com/repositories?language=Solidity

[3] https://github.com/bndes/bndestoken

utilizam um mesmo *framework* costumam seguir algum nível de padronização, tal como formato dos arquivos de configuração e estrutura de diretórios similar. Considerando-se os benefícios citados, além de representativo número de repositórios que fazem referência ao Truffle, a seleção dos projetos utilizados na análise experimental conduzida para avaliar a proposta de geração de testes apresentada no Capítulo 3 consistiu em identificar quais dentre os projetos Truffle contidos nos 269 repositórios possuem evidência de preocupação por parte dos desenvolvedores com a criação de testes automatizados.

Importante destacar que alguns dos repositórios do GitHub contêm mais de um projeto Truffle[4]. Por exemplo, o repositório `willitscale/learning-solidity` é composto por mais de 30 tutoriais divididos em diretórios distintos. Alguns destes tutoriais possuem seu próprio arquivo de configuração do *framework* Truffle, caracterizando-os como projetos distintos. Do conjunto de 269 repositórios selecionados, foram contabilizados 324 projetos Truffle. Dentre estes, a análise foi restrita aos projetos que apresentaram alguma evidência de implementação de testes automatizados. Para isso, o critério utilizado foi a existência do diretório "test" com ao menos um arquivo. Por fim, chegou-se a uma lista de 272 projetos que foram estabelecidos como seleção inicial desta pesquisa.

4.3 Seleção final de projetos para análise

Finda a seleção inicial de projetos, deu-se início à preparação de um ambiente para execução dos testes dos projetos e apuração dos resultados desta execução. O primeiro passo foi obter o código disponível nos respectivos repositórios através da execução do comando `git clone`, ou seja, para cada um dos 269 repositórios foi executada a linha de comando `git clone <url do repositório>`. Com esta execução, o código-fonte do repositório é baixado para a máquina local.

[4]Entende-se por 'projeto Truffle' todo diretório dentro do repositório que possuir um arquivo de configuração 'truffle-config.js' ou 'truffle-js'

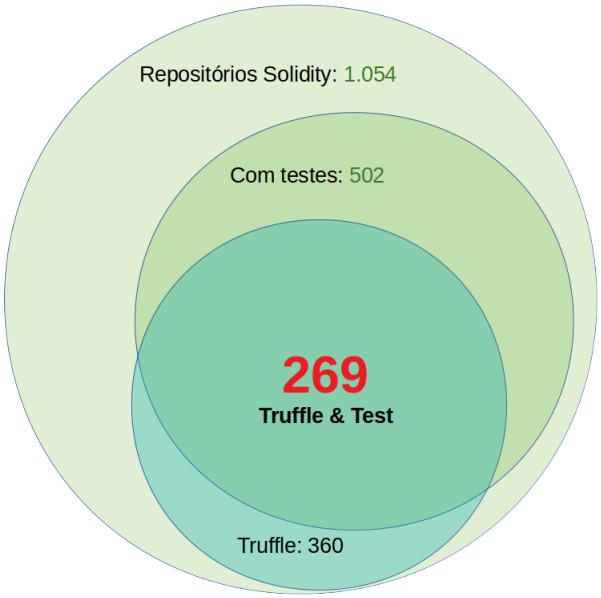


Figura 4.1: Conjuntos de repositórios Solidity no GitHub.

Observou-se que parte dos projetos dentro dos repositórios possui um arquivo `package.json`, arquivo de configuração do gerenciador de pacotes `npm`. Este arquivo contém, entre outros dados, referências às dependências de pacotes armazenados no repositório `npm`[5] e suas respectivas versões. O passo seguinte foi a execução do comando `npm install` no diretório de cada um dos projetos que continham o arquivo `package.json` para que suas dependências fossem obtidas e instaladas localmente.

Neste estudo, foi utilizada a versão 5 do *framework* Truffle. Esta versão tem como *default* a versão 0.5.* do compilador `solc`, que não mantém compatibilidade com versões anteriores. Parte dos contratos Solidity encontrados nos repositórios selecionados foram escritos para a versão 0.4.* da linguagem. Desta forma, foi realizado um procedimento de configuração da versão do compilador Solidity a ser utilizada durante a compilação de cada um dos projetos selecionados para que o *framework* Truffle utilize a versão correta do compilador em cada projeto.

[5]http://npmjs.com

O procedimento consiste em percorrer todos os arquivos de contratos de cada projeto, identificar a versão de Solidity mais recente declarada através da diretiva `pragma solidity` e incluir no arquivo `truffle-config.js` a propriedade que define de forma explícita a versão do compilador a ser utilizada como aquela mais recente declarada entre os contratos do projeto, conforme consta na linha 18 do Algoritmo 4.1, por exemplo. Assumiu-se que, havendo versões divergentes entre os contratos do mesmo projeto, estas seriam compatíveis. Essa modificação do arquivo `truffle-config.js` só é executada para os arquivos que não tenham uma propriedade "compilers".

```
1  module.exports = {
2    networks: {
3      ganache: {
4        host: '127.0.0.1',
5        port: '7545',
6        network_id: "*",
7      },
8      coverage: {
9        host: "localhost",
10       network_id: "*",
11       port: 8555,
12       gas: 0xfffffffffff,
13       gasPrice: 0x01
14     }
15   },
16   compilers: {
17     solc: {
18       version: "0.4.21",
19     }
20   }
21 };
```

Algoritmo 4.1: Exemplo de arquivo de configuração truffle-config.js

Concluída a preparação e configuração do ambiente, foi dado início ao trabalho de avaliação dos 272 projetos encontrados na seleção inicial. Esta avaliação se deu em etapas que se comportam como um funil, isto é, a cada etapa um conjunto de projetos é submetido a um determinado procedimento e somente os projetos que passam por tal procedimento de forma bem sucedida seguem para a próxima etapa para serem submetidos ao procedimento seguinte. A evolução sucessiva da execução destes procedimentos pode ser visualizada na Figura 4.2.

A primeira etapa foi a compilação dos 272 projetos Truffle utilizando o utilitário de linha de comando `truffle compile` em cada um deles. Após este procedimento, apurou-se que 200 projetos foram compilados com sucesso, o que representa 74% dos projetos selecionados inicialmente.

A etapa seguinte consistiu no procedimento de migração ou publicação dos contratos inteligentes compilados em uma *blockchain* por meio da execução da linha de comando `truffle migration`. Para esta pesquisa, utilizou-se um utilitário da suíte Truffle que simula uma *blockchain* localmente para fins de testes e desenvolvimento: `Ganache`. Os duzentos projetos que tiveram a compilação bem sucedida foram então submetidos ao procedimento de migração. Deste conjunto, 154 projetos tiveram sua migração para a *blockchain* local bem sucedida, ou seja, 77% dos projetos compilados ou 57% dos projetos inicialmente selecionados. Alguns destes projetos apresentaram falha de compilação dos artefatos de migração. Vale lembrar que este procedimento é realizado através de código escrito dentro do projeto que, como todo código-fonte, pode conter falhas que impeçam a sua publicação na rede para posterior execução dos testes.

Com os contratos inteligentes compilados e migrados para a *blockchain* emulada pelo Ganache, a etapa de execução dos testes automatizados pode ser iniciada através da linha de comando `truffle test`, para então se coletar os resultados da execução dos testes automatizados de cada projeto e diagnosticar a qualidade dos testes automatizados de contratos inteligentes dos projetos restantes. Os 154 projetos compilados e migrados seguiram para o procedimento de execução dos testes automatizados. Parte deles não foi executada, pois também tiveram problemas de compilação dos artefatos de teste, causados por falhas de programação, problemas com dependências não satisfeitas, entre outros. Logo, não tiveram sequer seus testes invocados. Assim, esta etapa resultou em 118 projetos em que os testes foram executados, ou seja, 77% dos projetos migrados, ou 43% dos projetos inicialmente selecionados. Dos 118 projetos, 65 tiveram todos os seus testes

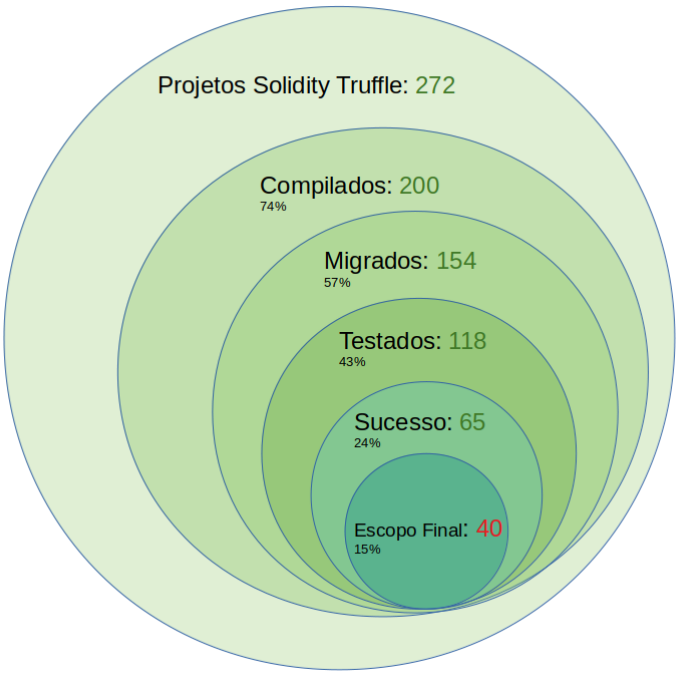


Figura 4.2: Resultados das etapas para seleção dos projetos que participarão na pesquisa.

executados sem nenhuma falha, enquanto os outros 53 tiveram ao menos um teste em que o resultado esperado divergiu do resultado obtido e, logo, falhou.

Em seguida, foi medida a cobertura de ramos (*branch coverage*) alcançada pelos testes automatizados dos 65 projetos que tiveram todos os seus testes executados com sucesso. Para isto, foi utilizado o utilitário `solidity-coverage`[6]. Novamente, alguns projetos apresentaram problemas na medição de cobertura e foram removidos do conjunto que será analisado nos experimentos reportados a seguir, restando 44 projetos que representam 16% dos projetos inicialmente selecionados. Por fim, verificou-se que alguns projetos eram cópias idênticas em repositórios distintos, enquanto outros possuíam

[6]https://github.com/sc-forks/solidity-coverage

zero ramos em seu código-fonte e, assim, também foram removidos do conjunto final de projetos selecionados para a pesquisa.

A lista de projetos analisados no contexto desta pesquisa, com o número total de ramos no código-fonte dos contratos de cada projeto (coluna *Total*), o número de ramos cobertos pelos testes implementados pelos desenvolvedores do projeto (coluna *Cobertos*) e o percentual de cobertura de ramos alcançado pela versão original de seus testes (coluna *%Cobertura*), é apresentada na Tabela 4.1, enquanto um resumo dos resultados destes 40 projetos pode ser observado na Tabela 4.2.

4.4 Análise experimental da proposta

A fim de avaliar a cobertura de ramos dos testes gerados pela proposta apresentada no Capítulo 3, o procedimento de geração de testes foi repetido 30 vezes para cada um dos projetos que compõem a seleção final para análise. As trinta repetições são necessárias para caracterizar a cobertura em diversos cenários de sorteio de números aleatórios utilizados como parte do processo de geração de testes. Os resultados dessas execuções estão sintetizados nas Figuras 4.3, 4.4, 4.5, 4.6 e 4.7 e na Tabela 4.3.

Os resultados alcançados pela pesquisa mostram que a média do percentual de cobertura de ramos dos testes gerados pela técnica proposta igualaram ou superaram a cobertura oferecida pelos testes originais em 22 dos 40 projetos. Em 19 projetos, os testes gerados pela pesquisa superaram os originais; destes, em 10 projetos o índice de cobertura dos testes gerados atingiu ou superou o dobro da cobertura dos testes originais. Aplicando-se um teste de Mann-Whitney com $\alpha = 0.05$, observamos que as diferenças de cobertura de ramos são estatisticamente significativas para os 19 projetos.

Tabela 4.1: Seleção final de projetos para análise experimental da técnica de geração de testes proposta.

Identificador	Ramos			Caminho no GitHub
	Cobertos	Total	% Cobertura	
06	12	36	33,3	/Dev43/ethinitium/modules/challenges/token_challenge/06
Action	4	4	100,0	/chaitanyapotti/Action
actus-solidity	182	354	51,4	/atpar/actus-solidity
aiakos-contracts	20	30	66,7	/PegaSysEng/aiakos-contracts
button	10	34	29,4	/abcoathup/button
crowdsale	2	2	100,0	/worldofatlantis/crowdsale
curve-bonded-tokens	59	98	60,2	/tarrencev/curve-bonded-tokens
debug-solidity	2	4	50,0	/kern/debug-solidity
EIP712	4	6	66,7	/willjgriff/solidity-playground/EIPsERCs/EIP712
equilibrium-bonding-curve	39	76	51,3	/gongf05/equilibrium-bonding-curve
ERC165	4	4	100,0	/willjgriff/solidity-playground/EIPsERCs/ERC165
final-project-hav-noms	1	222	0,5	/hav-noms/final-project-hav-noms
golem-contracts	87	174	50,0	/golemfactory/golem-contracts
harberger-ads-contracts	0	28	0,0	/bin-studio/harberger-ads-contracts
king	12	36	33,3	/abcoathup/king
left-gallery-token	6	64	9,4	/harmvandendorpel/left-gallery-token
liquid	11	32	34,4	/convergentcx/liquid
MarketSample	8	14	57,1	/cds-blog-code-samples/MarketSample
MembershipVerificationToken	12	24	50,0	/chaitanyapotti/MembershipVerificationToken
MetaTransactions	7	12	58,3	/willjgriff/solidity-playground/Misc/MetaTransactions
minime	36	70	51,4	/Onther-Tech/minime
mvp	6	8	75,0	/Dev43/ethinitium/advanced_modules/plasma/mvp
noia-token	37	52	71,2	/noia-network/noia-token
ponyBadges	10	34	29,4	/abcoathup/ponyBadges
real-estate-marketplace	30	260	11,5	/brenj/real-estate-marketplace
SafeERC20	10	16	62,5	/nachomazzara/SafeERC20
secretstore-acl	6	6	100,0	/parity-contracts/secretstore-acl
securities	73	166	44,0	/tokenfoundry/securities
simple-shared-wallet	44	60	73,3	/CYBRToken/simple-shared-wallet
SimpleCoin	0	10	0,0	/jackleslie/SimpleCoin
Solidity-RLP	36	44	81,8	/hamdiallam/Solidity-RLP
solidity-sigutils	5	8	62,5	/dsys/solidity-sigutils
splitter	16	30	53,3	/hilmarx/splitter
swaps	39	56	69,6	/MyWishPlatform/swaps
swaps2	46	92	50,0	/MyWishPlatform/swaps2
token	15	42	35,7	/relevant-community/token
trickle-smart-contracts	24	26	92,3	/dreamteam-gg/trickle-smart-contracts
truffle	16	170	9,4	/nicholashc/MerkleShip/truffle
try-catch-solidity	10	26	38,5	/maxsam4/try-catch-solidity
vvisp-klaytn-sample	5	24	20,8	/HAECHI-LABS/vvisp-klaytn-sample

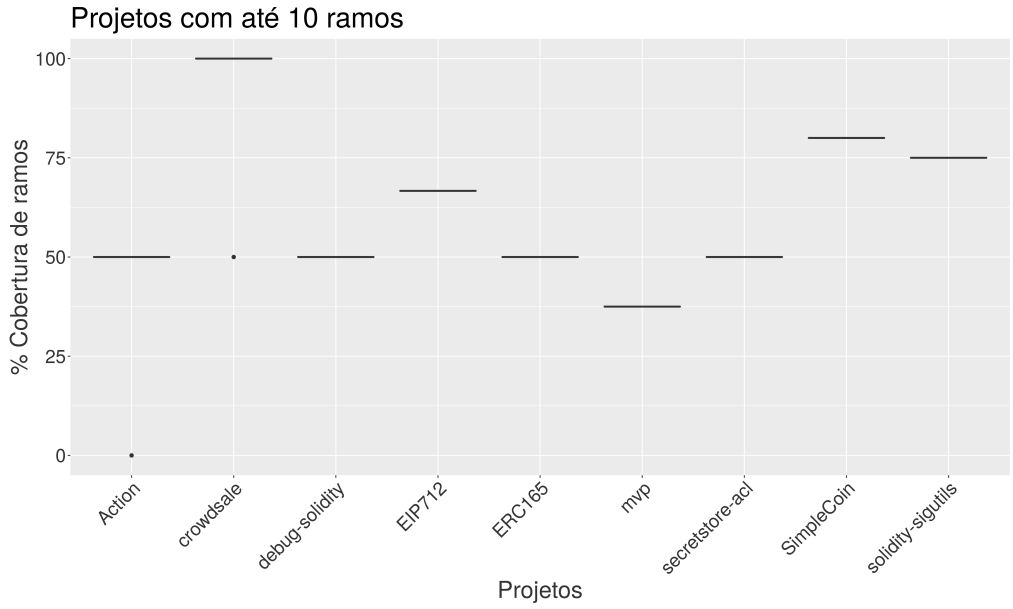


Figura 4.3: Percentual de cobertura de ramos para projetos com até 10 ramos.

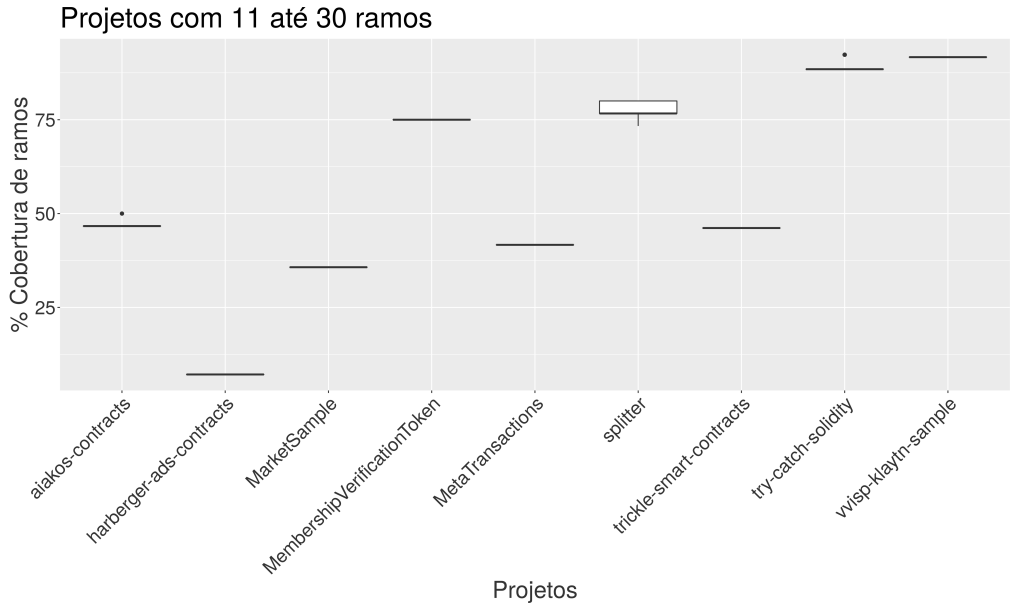


Figura 4.4: Percentual de cobertura de ramos para projetos com 11 até 30 ramos.

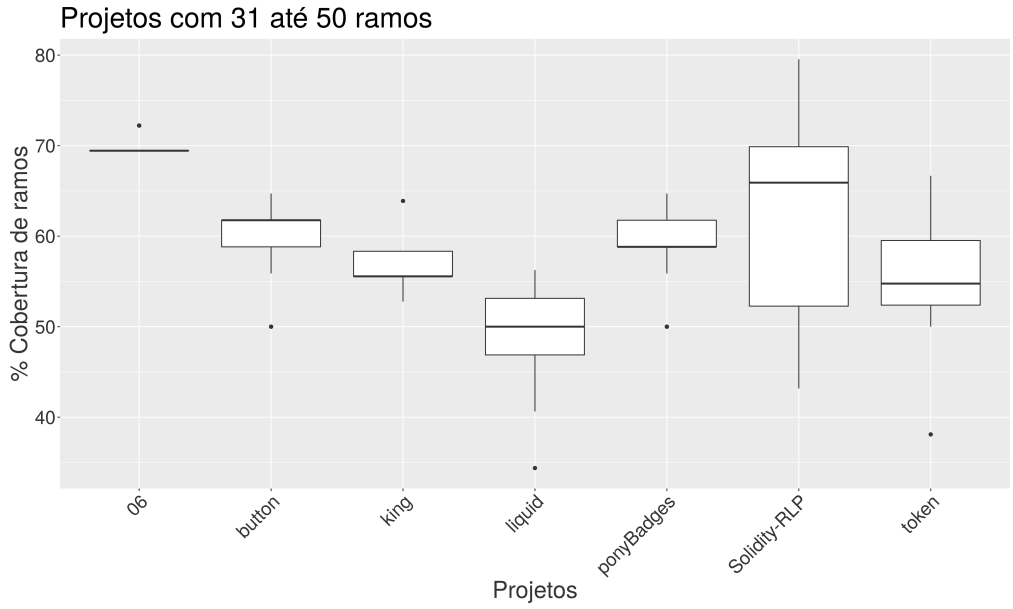


Figura 4.5: Percentual de cobertura de ramos para projetos com 31 até 50 ramos.

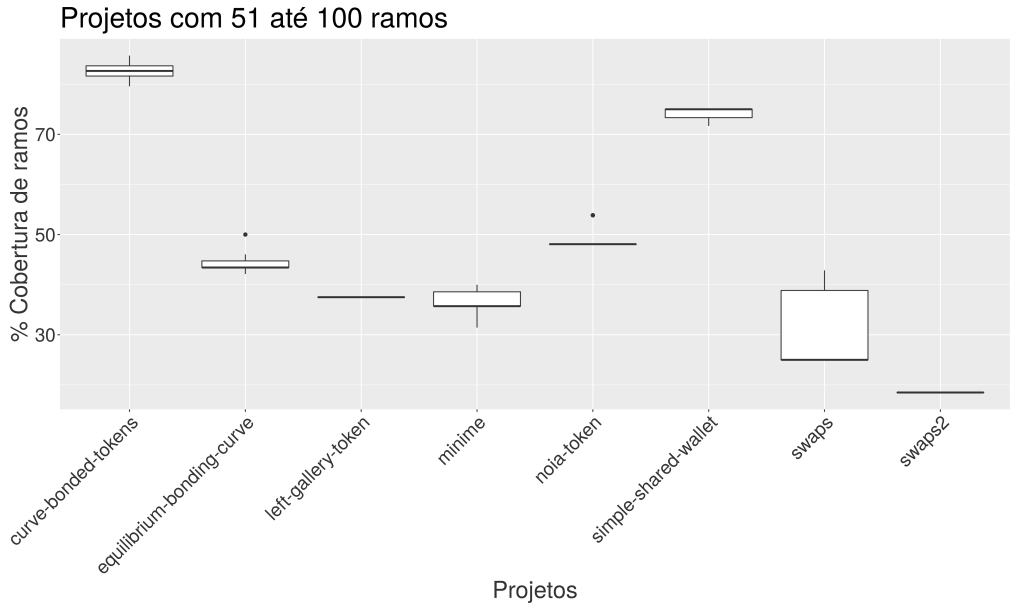


Figura 4.6: Percentual de cobertura de ramos para projetos com 51 até 100 ramos.

Tabela 4.2: Faixa de cobertura de ramos dos testes disponíveis nos projetos selecionados para analisar a técnica de geração de testes.

Faixa do percentual de cobertura	Número de projetos
Cobertura maior que 75% dos ramos	7
Cobertura entre 50% e 75% dos ramos	18
Cobertura entre 25% e 50% dos ramos	8
Cobertura menor que 25% dos ramos	7

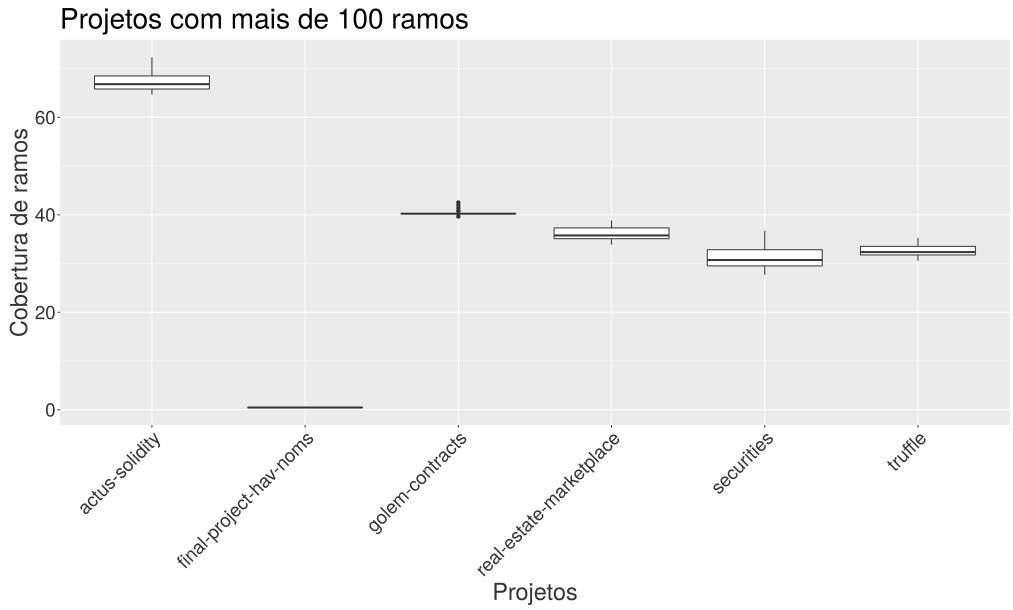


Figura 4.7: Percentual de cobertura de ramos para projetos com mais de 100 ramos.

Analisando-se o percentual de cobertura máximo alcançado entre as 30 rodadas, a cobertura dos testes originais é igualada ou superada em 23 projetos. Ainda que seja considerado o menor percentual de cobertura obtido pela pesquisa, em 21 projetos o percentual de cobertura dos testes gerados é pelo menos igual aos testes original do projeto. Um resumo do resultado da execução dos testes gerados de forma automatizada para os 40 projetos analisados pode ser observada na Tabela 4.4.

A Figura 4.8 apresenta a distribuição dos projetos de acordo com o percentual médio de cobertura de ramos dos testes gerados e observado nos testes originais dos projetos.

Tabela 4.3: Percentuais de cobertura de ramos obtidos pelas execuções do gerador de testes para os projetos selecionados. A primeira coluna apresenta o menor percentual de cobertura entre as 30 execuções do gerador de testes, a segunda coluna apresenta o percentual médio de cobertura, a terceira coluna apresenta o desvio padrão observado entre as 30 execuções, a quarta coluna apresenta a mediana da cobertura, a quinta coluna apresenta o percentual de cobertura máximo entre as 30 execuções e a última coluna apresenta o percentual de cobertura alcançado pelos testes originais, encontrados no repositório do projeto no GitHub. Valores em negrito representam projetos em que o gerador de testes atingiu resultados iguais ou melhores na média que os testes originais do projeto.

Identificador	% Mínimo	% Médio	Desvio Padrão	% Mediana	% Máximo	% Original
06	69,44	**69,90**	1,05	69,44	72,22	33,33
Action	0,00	45,00	15,26	50,00	50,00	100,00
actus-solidity	64,69	**67,24**	2,03	66,81	72,32	51,41
aiakos-contracts	46,67	46,78	0,61	46,67	50,00	66,67
button	50,00	**60,00**	3,51	61,76	64,71	29,41
crowdsale	50,00	96,67	12,69	100,00	100,00	100,00
curve-bonded-tokens	79,59	**82,55**	1,61	82,65	85,71	60,20
debug-solidity	50,00	**50,00**	0,00	50,00	50,00	50,00
EIP712	66,67	**66,67**	0,00	66,67	66,67	66,67
equilibrium-bonding-curve	42,11	44,21	1,57	43,42	50,00	51,32
ERC165	50,00	50,00	0,00	50,00	50,00	100,00
final-project-hav-noms	0,45	**0,45**	0,00	0,45	0,45	0,45
golem-contracts	39,66	40,42	0,59	40,23	42,53	50,00
harberger-ads-contracts	7,14	**7,14**	0,00	7,14	7,14	0,00
king	52,78	**56,21**	2,38	55,56	63,89	33,33
left-gallery-token	37,50	**37,50**	0,00	37,50	37,50	9,38
liquid	34,38	**48,65**	4,83	50,00	56,25	34,38
MarketSample	35,71	35,71	0,00	35,71	35,71	57,14
MembershipVerificationToken	75,00	**75,00**	0,00	75,00	75,00	50,00
MetaTransactions	41,67	41,67	0,00	41,67	41,67	58,33
minime	31,43	36,81	1,86	35,71	40,00	51,43
mvp	37,50	37,50	0,00	37,50	37,50	75,00
noia-token	48,08	48,85	1,99	48,08	53,85	71,15
ponyBadges	50,00	**59,02**	3,78	58,82	64,71	29,41
real-estate-marketplace	33,85	**36,18**	1,42	35,77	38,85	11,54
secretstore-acl	50,00	50,00	0,00	50,00	50,00	100,00
securities	27,71	31,37	2,46	30,72	36,75	43,98
simple-shared-wallet	71,67	**74,22**	1,14	75,00	75,00	73,33
SimpleCoin	80,00	**80,00**	0,00	80,00	80,00	0,00
Solidity-RLP	43,18	61,82	10,40	65,91	79,55	81,82
solidity-sigutils	75,00	**75,00**	0,00	75,00	75,00	62,50
splitter	73,33	**78,00**	2,07	76,67	80,00	53,33
swaps	25,00	30,72	7,06	25,00	42,86	69,64
swaps2	18,48	18,48	0,00	18,48	18,48	50,00
token	38,10	**55,63**	5,75	54,76	66,67	35,71
trickle-smart-contracts	46,15	46,15	0,00	46,15	46,15	92,31
truffle	30,59	**32,71**	1,11	32,35	35,29	9,41
try-catch-solidity	88,46	**89,10**	1,46	88,46	92,31	38,46
vvisp-klaytn-sample	91,67	**91,67**	0,00	91,67	91,67	20,83

Tabela 4.4: Faixa de cobertura média alcançada pelos testes gerados pela pesquisa

Percentual de cobertura	Número de projetos
Cobertura maior que 75%	9
Cobertura entre 50% e 75%	13
Cobertura entre 25% e 50%	15
Cobertura maior que 25%	3

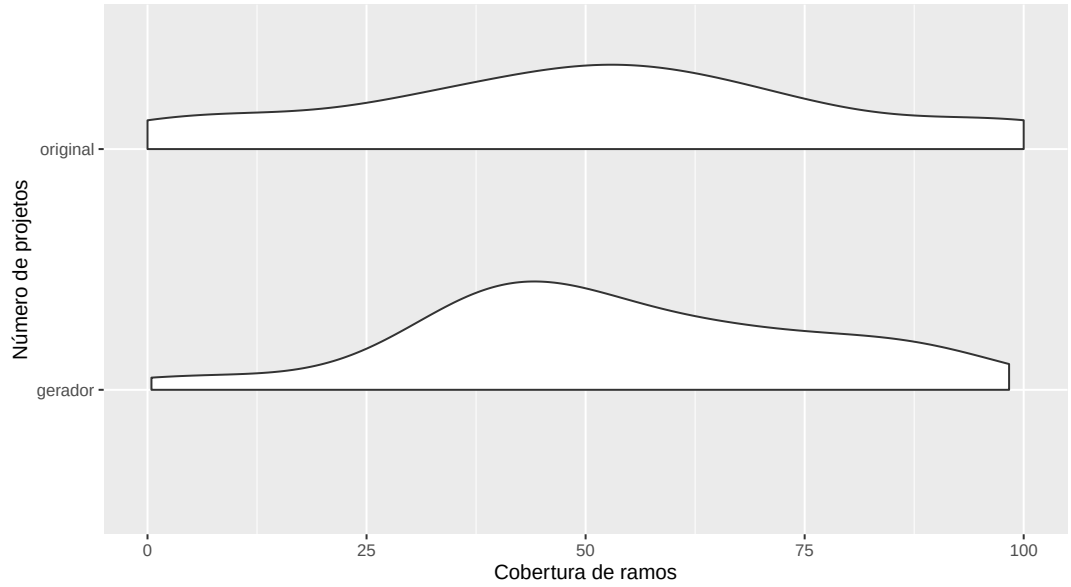


Figura 4.8: Histogramas do percentual médio de cobertura de ramos obtidos pela técnica proposta e percentual de cobertura de ramos obtido pelos testes originais dos projetos.

Percebe-se uma concentração de projetos em torno de 40% de cobertura para os testes gerados, enquanto a concentração se encontra em torno de 55% para os testes originais. No entanto, observa-se também uma maior concentração de projetos com alta concentração de cobertura de ramos nos testes gerados. Assim, a técnica geradora de testes com frequência gera cobertura um pouco abaixo da média observada nos testes originais, mas alcança alta cobertura em um maior número de projetos do que os testes escritos pelos desenvolvedores.

Outro aspecto interessante da técnica proposta é a sua baixa sensibilidade aos fatores aleatórios envolvidos na geração dos testes. Como pode ser visto na Tabela 4.3, o desvio padrão da distribuição do percentual de cobertura dos testes é, em geral, pequeno, ficando abaixo de 10% em 37 dos 40 projetos. Assim, embora a técnica não seja determinística, para uma grande parte dos projetos analisados não houve grande influência do fator aleatório na cobertura de ramos dos testes gerados.

Calculando a correlação entre o percentual de cobertura de ramos e o número de ramos de cada projeto, observamos um valor baixo e negativo de $\rho = -0.37$ (correlação de Spearman). Este valor indica uma fraca relação entre o tamanho do projeto e a capacidade da técnica gerar testes com maior cobertura de ramos, sinalizando para uma relação inversa neste sentido: quanto mais ramos tiver o código-fonte do projeto, menor será a capacidade da técnica em atingir uma boa cobertura (percentual).

Além disso, dos 22 projetos em que a técnica proposta ultrapassou ou igualou os testes originais em cobertura de ramos, 10 projetos foram desenvolvidos na versão 0.5.* da linguagem Solidity e doze projetos foram desenvolvidos na versão 0.4.*. Dos demais projetos, sete foram desenvolvidos na versão 0.4.* do Solidity e onze na versão 0.5.*. Assim, não há uma predominância de versão da linguagem Solidity em que a técnica de geração de testes apresente uma vantagem sobre os testes originais dos projetos.

Também foram encontradas correlações fracas entre o ganho de cobertura de ramos da técnica de geração de testes (ou seja, a média da cobertura de ramos nas 30 execuções da técnica menos a cobertura de ramos dos testes originais do projeto) e a idade do repositório do projeto no GitHub ($\rho = -0.26$), o número de estrelas do projeto no GitHub ($\rho = -0.01$), o número de observadores do projeto ($\rho = -0.01$), o número de *forks* do projeto ($\rho = -0.13$) e o número de *issues* registrados do projeto ($\rho = +0.09$). Sendo assim, o ganho oferecido pela técnica é independente da atenção recebida pelo projeto, ao menos do ponto de vista da rede de desenvolvedores do GitHub.

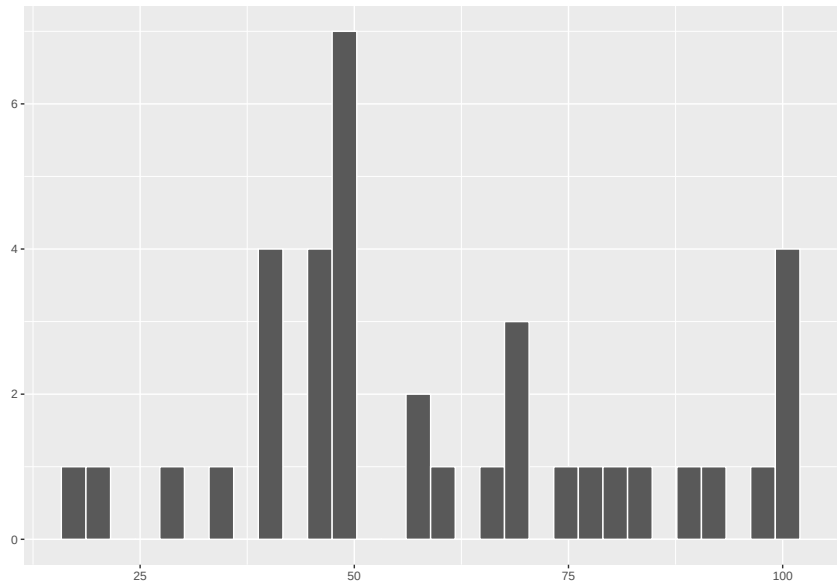


Figura 4.9: Interseção da cobertura de ramos entre os testes gerados automaticamente e os testes originais do projeto.

Examinamos também a interseção da cobertura de ramos entre os testes gerados e os testes originais dos projetos. O histograma na Figura 4.9 apresenta a distribuição do percentual médio dos ramos cobertos pelos testes originais que também são cobertos pelos testes gerados. Observa-se uma interseção de 50% ou mais dos ramos para a maior parte dos projetos, o que indica que muito do esforço de codificação investido na construção dos testes originais poderia ter sido substituído, ao menos em parte, pelo processo de geração automática dos testes. Em média, temos 60.3% de intersecção na cobertura dos ramos, com um desvio padrão de 23.3%.

4.4.1 Análise qualitativa de resultados negativos

O primeiro ponto a destacar é a ausência de resultados para o projeto `SafeERC20`, que sequer aparece na Tabela 4.3. Este projeto é composto por uma única biblioteca de mesmo nome e cujas funções recebem como parâmetro de entrada uma interface IERC20, impor-

tada do pacote *npm openzeppelin-eth*. Por ser uma interface, não é possível criar instâncias de IERC20. Entretanto, essa condição não seria um limitador, pois as relações de herança são identificadas durante o processo de construção do grafo do projeto e, assim, identifica-se que contratos implementam esta interface. Com isso, seria possível instanciar qualquer um desses contratos para suprir a necessidade de uma instância de IERC20 ao gerar os valores dos parâmetros para as funções do projeto `SafeERC20`.

A limitação se impõe pelo fato de que, para instanciar um dos contratos que implementam a IERC20, eles precisariam passar pelo processo de compilação do *framework* Truffle, gerando, para cada contrato, um arquivo no formato JSON no subdiretório "build/contracts". Este arquivo guarda informações cruciais para o funcionamento do *framework*, incluindo o recurso utilizado nos testes para se importar dentro do código JavaScript uma referência de um contrato, como o apresentado no Algoritmo 4.2. Sem este arquivo, uma exceção ocorre ao tentar obter essa referência.

```
1 const ContratoA = artifacts.require("ContratoA");
2 const ERC20Burnable = artifacts.require("openzeppelin-eth/contracts/token/ERC20/
      ERC20Burnable.sol");
```

Algoritmo 4.2: Código-fonte JavaScript de importação de referência de contrato.

Durante o processo de compilação, o Truffle gera um arquivo JSON no diretório "build/contracts" para cada contrato do projeto. Esse arquivo contém metadados essenciais para funcionamento do *framework*. Além de cada contrato do próprio projeto, o Truffle gera um arquivo semelhante para contratos que estejam dentro de pacotes obtidos via gerenciador de pacotes *npm*. Para este segundo grupo, ele apenas gerará o arquivo JSON com metadados para os contratos que forem referenciados no código Solidity dos contratos do projeto através do comando *imports*.

Como no código da biblioteca SafeERC20 o *import* é feito somente para a interface IERC20, apenas o JSON da própria SafeERC20 e da interface IERC20 (importada)

constarão no diretório "build/contracts" após a compilação. Desta forma, não é possível instanciar outros contratos de pacotes importados via *npm*. A abordagem utilizada pelo projeto original é criar contratos que implementam a IERC20 específicos para a realização dos testes no diretório "test". Tal abordagem foge do escopo da proposta desta pesquisa porque exigiria a geração de código na linguagem de programação Solidity para criar os contratos usados nos testes.

Outro resultado negativo que se destaca entre os projetos analisados é o obtido no projeto `final-project-hav-noms`: dos 222 ramos identificados neste projeto, os testes gerados cobriram apenas um ramo (0,45%). Nota-se que este é o mesmo resultado obtido pelos testes originais encontrados no repositório do projeto no GitHub. Ao investigar as causas para uma cobertura tão inexpressiva, foi encontrada uma causa raiz comum: uma exceção *"VM Exception while processing transaction: out of gas"* gerada durante o processo de publicação dos contratos na rede *blockchain* que é realizada durante os testes. Desta forma, apenas parte dos contratos do projeto foi publicada e, consequentemente, somente uma parte dos testes foi executada. É importante reforçar que este erro não ocorre quando executamos o comando `truffle test`, mas apenas quando executamos o utilitário de medição de cobertura de testes `npx solidity-coverage`. Este comportamento é justificado pela documentação do `solidity-coverage`, onde os desenvolvedores afirmam que o processo de instrumentação para o cálculo da cobertura de ramos aumenta os custos com *gas*[7].

Outro resultado que também se destaca pela baixa cobertura foi obtido no projeto `harberger-ads-contracts`. Dos 28 ramos contidos neste projeto, os testes cobriram apenas dois ramos (7,14%) em todas as suas execuções, isto é, o fator de aleatoriedade aplicado na geração dos valores dos parâmetros de entrada das funções não teve impacto nos resultados. Uma das explicações para isto se encontra na limitação da pesquisa em relação à avaliação de acesso indexado a parâmetros e variáveis de estado. O contrato

[7]https://github.com/sc-forks/solidity-coverage/blob/0.6.x-final/docs/faq.md

`HarbergerAds`, que contém os 28 ramos, possui uma variável de estado que é um *array* de *struct*, similar ao exemplo do contrato `LinhaFinanciamento` apresentado no Capítulo 3. Em quatro das nove funções que este contrato contém, é realizado um acesso indexado a este *array* na primeira linha de código, atribuindo-se o valor a uma variável local. O código destas funções adota como premissa que o *array* terá uma quantidade de elementos superior ao índice informado via parâmetro de entrada, entretanto, ao contrário do que acontece no código do contrato `LinhaFinanciamento`, não há no código nenhuma instrução para validar esta condição. Apesar disso, é notória a presença de uma pré-condição que precisa ser satisfeita para este teste ser executado: que haja ao menos um elemento no referido *array*. Como a técnica proposta só é capaz de identificar esse tipo de pré-condição de estado dos *arrays* com base em expressões relacionais que envolvam sua propriedade *length*, não foi identificada a necessidade de inserir ao menos um item neste *array* antes de se executar o teste efetivamente, logo, uma exceção é lançada na primeira instrução dessas funções sem exercitar os ramos presentes nas linhas seguintes. As outras cinco funções contêm apenas quatro dos 28 ramos do contrato e como suas expressões relacionais também envolvem elementos não cobertos pela pesquisa, os ramos foram apenas parcialmente cobertos.

Fechando a lista de projetos que tiveram uma média de cobertura de ramos abaixo de 30%, encontram-se os resultados alcançados no projeto `swaps2`. Dos seus 92 ramos, os testes cobriram apenas 17 (18,48%) em todas as suas execuções. Mais uma vez, a cobertura baixa está atrelada à limitação da pesquisa em relação aos tipos de elementos encontrados nas expressões relacionais. Dos 92 ramos citados, 86 estão localizados no contrato `Swaps`. Este contrato possui 28 funções[8] e quatro modificadores. O modificador `onlyInvestor` executa um `require` que avalia o retorno de uma função que retorna um acesso indexado a *mapping* em cadeia, como a que pode ser vista no Algoritmo 4.3. Neste algoritmo, um `mapping` é indexado por `bytes32`, tendo como valor um outro `mapping`, que por sua vez é indexado por `address` e seu valor é também outro `mapping`. O mo-

[8] Ao contrário do relatório do `solidity-coverage`, não contabilizamos modificadores como funções.

dificador `onlyWhenVaultDefined` executa um `require` que faz uso da função nativa `address` para obter o endereço de uma variável de estado cujo tipo é um contrato. Os modificadores `onlyOrderOwner` e `onlyWhenOrderExists` também executam `requires` comparando um acesso indexado a um mapeamento do contrato. As três funções são afetadas por ao menos um desses quatro modificadores, enquanto outras sete funções, apesar de não serem afetadas por esses modificadores, em alguma parte do seu código-fonte possuem expressões relacionais com operações não suportadas por esta pesquisa. As dezoito funções restantes, apesar de representarem um conjunto mais numeroso, são funções *getters* e *setters* que não possuem ramos.

```
1     mapping(bytes32 => mapping(address => mapping(address => uint))) public
          investments;
```

Algoritmo 4.3: Exemplo de utilização da estrututura `mapping` em cadeia

4.4.2 Análise qualitativa de resultados positivos

No outro extremo, encontram-se os projetos que tiveram resultados mais favoráveis. É o caso do projeto `crowdsale`, que obteve em média 96,67% de cobertura dos seus dois únicos ramos. Nas 30 execuções realizadas sobre o projeto, em duas delas um dos ramos deixou de ser executado. A expressão relacional que dá origem aos dois ramos do projeto realiza a comparação entre um parâmetro de entrada da função e uma variável de estado, para garantir que o primeiro seja maior ou igual à segunda. Em duas das execuções da técnica proposta, durante a geração dos valores de entrada do construtor do contrato foi gerado o valor zero e este foi atribuído à variável de estado. Desta forma, no momento de gerar o valor do parâmetro com o propósito de violar a restrição, foi subtraído 1 do valor da variável de estado, conforme citado em 3.1. Contudo, o parâmetro de entrada é do tipo `uint256`, que só aceita valores iguais ou maiores do que zero. Assim, o Truffle fez a conversão do parâmetro para `uint256`, modificando seu valor e impossibilitando o exercício do ramo que violava a restrição citada.

Dos 24 ramos do projeto `vvisp-klaytn-sample`, 22 (91,67%) foram cobertos pelos testes automatizados. Os únicos dois ramos não executados estão relacionados com tipos de elementos de expressões relacionais não cobertos pela pesquisa: ambos envolvem resultados de expressões matemáticas com variáveis locais.

Um resultado próximo ao anterior foi obtido no projeto `try-catch-solidity`. Dos seus 26 ramos, em média, 89,10% foram cobertos pelos testes gerados de forma automatizada. Nas 30 execuções realizadas nesta análise experimental, a moda foi de 23 ramos, alcançando em cinco oportunidades a cobertura de 24 ramos. Este ramo adicional alcançado em algumas execuções é resultante do fator da aleatoriedade que levou à mudança de valor de uma variável local Booleana, a qual é atribuído o resultado de chamada a uma função de outro contrato. Os outros dois ramos deste contrato sistematicamente não cobertos envolvem comparações entre expressões matemáticas e variáveis locais.

4.4.3 Resultados inesperados

Conforme mencionado no Capítulo 1, o objetivo da pesquisa é a identificação e a construção de testes cujo resultado esperado seja o lançamento de exceção. Essa exceção deve ser provocada pelo não atendimento das condições expressas nas validações realizadas sobre parâmetros de entrada e/ou variáveis de estado do contrato inteligente, codificadas pelo desenvolvedor nas funções do contrato ou dos modificadores (*modifier*) aplicados a elas.

Na Tabela 4.5, está listado o número de testes gerados de forma automática pela técnica proposta para cada projeto. Estes testes se dividem em testes ordinários e testes de exceção. Testes ordinários são aqueles em que a intenção é atender a todas as validações expressas nos comandos `require` identificados no caminho de execução coberto pelo requisito de teste. Por outro lado, os testes de exceção têm como objetivo exercitar

Tabela 4.5: Número de testes gerados pela técnica proposta para os projetos selecionados. A primeira coluna apresenta o número total de testes gerados. A segunda coluna apresenta o número de testes ordinários gerados, a terceira coluna apresenta o número de testes ordinários que lançaram alguma exceção (inesperada) e a quarta coluna apresenta o percentual de falso positivos. A quinta coluna apresenta o número de testes que deveriam lançar exceção, a sexta coluna apresenta o número de testes de exceção que não geraram a exceção esperada e a sétima coluna apresenta o percentual de falso negativos.

Projeto	Número de Testes	Ordinários	Falso Positivos	%	Exceção	Falso Negativos	%
06	27	19	2	10,53	8	0	-
Action	3	3	2	66,67	0	0	-
actus-solidity	200	186	19	10,22	14	1	7,14
aiakos-contracts	25	17	8	47,06	8	0	-
button	29	19	14	73,68	10	0	-
crowdsale	7	3	1	33,33	4	0	-
curve-bonded-tokens	122	54	19	35,19	68	8	11,76
debug-solidity	10	10	2	20,00	0	0	-
EIP712	8	8	0	-	0	0	-
equilibrium-bonding-curve	48	39	18	46,15	9	2	22,22
ERC165	6	6	0	-	0	0	-
final-project-hav-noms	0	0	0	-	0	0	-
golem-contracts	110	69	28	40,58	41	6	14,63
harberger-ads-contracts	17	16	10	62,50	1	0	-
king	29	19	13	68,42	10	0	-
left-gallery-token	63	44	21	47,73	19	2	10,53
liquid	47	31	19	61,29	16	8	50,00
MarketSample	6	5	2	40,00	1	0	-
MembershipVerificationToken	35	19	5	26,32	16	1	6,25
MetaTransactions	11	11	2	18,18	0	0	-
minime	34	21	12	57,14	13	0	-
mvp	7	4	2	50,00	3	0	-
noia-token	51	32	16	50,00	19	2	10,53
ponyBadges	30	20	14	70,00	10	0	-
real-estate-marketplace	183	167	96	57,49	16	0	-
secretstore-acl	4	3	1	33,33	1	0	-
securities	76	29	28	96,55	47	14	29,79
simple-shared-wallet	52	26	6	23,08	26	0	-
SimpleCoin	12	9	2	22,22	3	0	-
Solidity-RLP	33	27	8	29,63	6	0	-
solidity-sigutils	8	8	2	25,00	0	0	-
splitter	28	15	1	6,67	13	1	7,69
swaps	35	27	15	55,56	8	7	87,50
swaps2	68	43	19	44,19	25	0	-
token	49	37	11	29,73	12	2	16,67
trickle-smart-contracts	21	10	6	60,00	11	1	9,09
truffle	52	36	18	50,00	16	0	-
try-catch-solidity	45	21	6	28,57	24	6	25,00
vvisp-klaytn-sample	46	22	6	27,27	24	6	25,00

as exceções em tais chamadas. Com exceção dos testes de exceção, onde o resultado esperado é o lançamento de uma exceção, o presente trabalho não tem por objetivo gerar os oráculos de teste.

É esperado que alguns dos testes ordinários gerem exceções decorrentes de validações que estão fora do escopo desta pesquisa, como comandos `require` que dependem de vetores, estruturas complexas e outros construtos de linguagem apresentados no Capítulo 3. Chamamos estes testes ordinários que falham de *falso positivos* e seu número e percentual são apresentados na Tabela 4.5. Percebe-se que, em função desta pesquisa considerar somente um conjunto limitado de escopos e tipos de dados na geração dos testes, o número de falso positivos é alto.

Para exemplificar este cenário, o projeto `securities`, em sua pior execução, tem uma exceção lançada em 96% dos seus testes ordinários. Analisando a execução dos testes, percebe-se que as causas estão atreladas às limitações da pesquisa em relação aos tipos e escopo das variáveis utilizadas nas expressões relacionais. Exemplos deste cenário encontrados no projeto `securities` podem ser visualizados nos Algoritmos 4.4 a 4.7.

No Algoritmo 4.4, um `require` verifica se o resultado da chamada à função `allowance` de uma variável de estado do tipo `IERC20` (`paymentToken`) é maior ou igual ao resultado da multiplicação da variável de estado `totalToRepurchase` pelo parâmetro de entrada `_newPaymentPerSecurity`. O resultado da chamada de uma função e o resultado de uma operação matemática são dois tipos de elementos encontrados nas expressões relacionais que estão fora do escopo da pesquisa.

```
1 require(paymentToken.allowance(owner(), address(this)) >= totalToRepurchase *
      _newPaymentPerSecurity,"Redemption contract does not have access to enough
      tokens");
```

Algoritmo 4.4: Projeto securities: `require` com validação de retorno de uma função contra o resultado de uma operação matemática.

No Algoritmo 4.5, um `require` verifica se o resultado da chamada à função `allowance` de uma variável de estado do tipo `IERC20` é maior ou igual ao valor da variável local `totalPaymentNeeded`. Além do resultado de chamada de função, a variável

local é um tipo de elemento encontrado nas expressões relacionais que está fora do escopo desta pesquisa.

```
uint256 totalPaymentNeeded = totalToRepurchase * paymentPerSecurity;
require(paymentToken.allowance(owner(), address(this)) >= totalPaymentNeeded, "
    Redemption contract does not have access to enough tokens");
```

Algoritmo 4.5: Projeto securities: `require` com validação do retorno de uma função contra uma variável local.

No Algoritmo 4.6, um comando `require` verifica se o atributo *length* do vetor local `holderPayments` é maior do que zero. Da mesma maneira que as variáveis locais são um tipo de elemento encontrado nas expressões relacionais que está fora do escopo da pesquisa, também está o acesso a seus atributos.

```
Payment[] memory holderPayments = payments[_securityHolder];
require (holderPayments.length > 0, "The holder has no payment history.");
```

Algoritmo 4.6: Projeto securities: `require` com validação de um atributo de uma variável local.

No Algoritmo 4.7, um `require` valida se o atributo *length* do vetor armazenado na variável de estado `payments` (que é do tipo *mapping*, tem índice do tipo *address* e valores do tipo vetor) é maior do que o valor passado através do parâmetro de entrada `_index`. O acesso indexado a variáveis de estado é um outro tipo de elemento encontrado nas expressões relacionais que está fora do escopo da pesquisa.

```
require(payments[_payee].length > _index && _index >= 0, "Payment index not in
    range for message sender");
```

Algoritmo 4.7: Projeto securities: `require` com validação de um atributo de um valor retornado através de acesso indexado a uma variável de estado.

Mesmo que os testes ordinários funcionem, a técnica proposta não gera os `asserts` que confeririam os valores retornados com o oráculo desconhecido. Para estes casos, o desenvolvedor precisa complementar os testes gerados com os valores do oráculo, ficando a contribuição da proposta limitada à identificação dos caminhos independentes presentes

no código-fonte do contrato e a geração de valores para os seus parâmetros e variáveis de estado que permitam o teste percorrer estes caminhos.

A principal contribuição desta pesquisa está nos testes de exceção, permitindo a geração de testes que simulem a tentativa de quebra nas regras de negócio codificadas nos comandos `require` e permitindo que os desenvolvedores rodem os testes de forma regressiva depois de uma alteração para garantir que as regras de negócio continuam ativas e protegendo o contrato contra o uso com parâmetros e variáveis de estado com valores indevidos.

No entanto, existem testes de exceção que terminam sem gerar a exceção esperada. A coluna "Falso negativos" apresenta o número de testes de exceção, ou seja, para os quais se esperava uma exceção do tipo *revert*[9] mas foi obtido um comportamento diferente. Este comportamento diferente pode ser a) nenhuma exceção foi lançada ao invocar a função testada, b) outro tipo de exceção foi lançado ao invocar a função testada ou c) uma exceção foi lançada por alguma chamada de função de teste antes de chamar a função do contrato que se desejava testar (por exemplo, uma exceção ocorreu em uma chamada a uma função para definir o estado do contrato a fim de atender a uma pré-condição do caso de teste). Embora consideravelmente menor que o número de falso positivos, o número de falso negativos ainda é expressivo em alguns projetos.

Um dos projetos que chama atenção pelo número de falso negativos é o `swaps`, para o qual foram gerados oito testes de exceção. Na sua melhor execução, seis (75%) destes testes não geraram a exceção esperada. Na sua pior execução, sete (87,5%) dos seus testes de exceção não geram a exceção esperada. A análise dos resultados mostrou que o falso negativo ocorre, em diversos casos, em decorrência de falhas nas chamadas de função para atender às pré-condições do teste. Por exemplo, no teste da função `_refund` pelo Algoritmo 4.8, uma exceção é lançada na invocação da função `sendTransaction`

[9]Tipo de exceção gerada pelo comando `require`

(linha 3) e o teste falha com a seguinte mensagem: "*Error: Returned error: VM Exception while processing transaction: revert Currency amount must be positive – Reason given: Currency amount must be positive*".

Essa exceção ocorre porque `sendTransaction` invoca a função `_depositEther` do mesmo contrato e passa como parâmetro o valor recebido nesta transação. Perceba que o objeto passado como parâmetro na invocação da função `sendTransaction` não possui a propriedade *value* de forma explícita. Essa propriedade só é passada de forma explícita quando é identificada uma restrição relacionada a ela. Portanto, o valor passado para a função `_depositEther` será o valor *default* para o tipo `uint`, no caso, zero. A função `_depositEther`, por sua vez, invoca outra função do contrato, a `_deposit`, repassando o mesmo parâmetro. Ao examinar a função `_deposit`, cujas primeiras linhas estão demonstradas no Algoritmo 4.9, encontramos na linha 4 o comando `require` com a mensagem de saída do erro. Importante destacar que este estudo seria capaz de identificar a restrição para o *value* do parâmetro caso a expressão relacional do `require` utilizasse diretamente o parâmetro `_amount` de `_deposit`. Contudo, ao utilizar uma variável local (linha 2 do Algoritmo 4.9), elemento ainda não coberto pela pesquisa, a rastreabilidade é perdida. Situação idêntica foi encontrada em outros cinco testes falso negativos deste projeto, que testam as funções `refundQuote`, `refundBase`, `cancel` e `_swap`.

```
1 it('Should fail test_refund(address) when NOT comply with: isSwapped != true',
      async () => {
2     let localcontractProxyBaseSwaps = await ProxyBaseSwaps.new(accounts[5],accounts
          [6],"0x0000000000000000000000000000000000000000",5,10000,(await web3.eth.
          getBlock(await web3.eth.getBlockNumber())).timestamp+768,{from:accounts
          [0]});
3     await localcontractProxyBaseSwaps.sendTransaction({from:accounts[0]});
4     let result = await truffleAssert.fails(localcontractProxyBaseSwaps.test_refund(
          accounts[2],{from: accounts[0]}),'revert');
5 });
```

Algoritmo 4.8: Projeto swaps: teste da função `test_refund` quando a variável `isSwapped` é igual a `true`.

```
1 function _deposit(address _token, address _from, uint _amount) internal {
2     uint amount = _amount;
3     require(baseAddress == _token || quoteAddress == _token, "You can deposit only
          base or quote currency");
```

```
4       require(amount > 0, "Currency amount must be positive");
5       require(raised[_token] < limits[_token], "Limit already reached");
6       ...
```

Algoritmo 4.9: Projeto swaps: validações com comando `require` no início da função `_deposit`.

Outro projeto que chama a atenção pelo alto percentual de exceções que eram esperadas mas não se realizaram é o `liquid`, atingindo na sua pior execução 8 falso negativos (50% dos testes). Destes, seis ocorrências se referem aos testes das funções `sell`, `buy` e `requestService` e possuem em comum o fato de que a restrição encontrada, direta ou indiretamente, na função testada é que o *sender* no *transaction parameter* seja diferente do valor inicial implícito de uma variável do tipo *address*: '0x0000000000000000000000000000000000000000'. Com isto, para forçar a violação de tal restrição, quando lidamos com restrições do tipo *address* e o operador "diferente", atribui-se exatamente este valor conforme mencionado na Tabela 3.3. Entretanto, a Ethereum não permite que uma transação seja originada de um endereço inexistente e o endereço zero não existe na rede Ethereum utilizada para testes durante a pesquisa. Isto faz com que outro tipo de exceção seja lançado sem nem mesmo iniciar a execução da função que se pretendia testar. Esse mesmo tipo de ocorrência é observado em outros sete projetos: *try-catch-solidity* (6 casos), *vvisp-klaytn-sample* (6 casos), *curve-bonded-tokens* (5 casos), *golem-contracts* (1 caso), *MembershipVerificationToken* (1 caso), *noia-token* (1 caso) e *trickle-smart-contracts* (1 caso).

Os outros dois falso negativos do projeto `liquid` foram casos similares ao reportado anteriormente: o problema ocorre em uma exceção retornada em chamadas de funções para configuração do ambiente que ocorrem antes da chamada à função que se pretende testar. Neste caso, a função `buy` possui um comando `require` que compara uma variável local, que previamente recebeu o resultado de uma função, contra um dos parâmetros de entrada da função.

Na Tabela 4.6 estão compiladas as ocorrência de resultados falso negativos para cada uma das principais causas identificadas. Trata-se do número máximo de ocorrências para cada causa identificada entre as 30 execuções realizadas para cada projeto, o que pode fazer com que, em alguns casos, a soma do número de ocorrências de cada causa supere o número máximo de falso negativos total.

As causas "*Sender 0x0*" (endereço zerado não existe na rede Ethereum utilizada nos testes) e "Pré-condições" (lançamento de uma exceção nas funções invocadas antes da chamada à função sob testes) foram descritos acima. A causa "*Invalid Opcode*" está associada ao lançamento de uma exceção *invalid opcode* pela Ethereum Virtual Machine. Segundo o site oficial da Ethereum, essa exceção é lançada quando uma validação realizada com o comando Solidity `assert` não é bem sucedida. De fato, foi constatada a utilização do comando `assert` em todos os projetos onde ocorreu este tipo de exceção. Por fim, a causa "Sem exceção" está associada a situações em que, apesar do teste esperar uma exceção do tipo *revert*, nenhuma exceção ocorreu ao chamar a função sob testes. Trata-se de casos imprevistos, com causas ainda não esclarecidas, podendo ser até mesmo um defeito de codificação da ferramenta desenvolvida como parte desta pesquisa.

A Tabela 4.6 mostra que a concentração dos falso negativos se dá principalmente pelas limitações do ambiente de teste (causa "Sender 0x0", com 27 ocorrências) ou pelas limitações do escopo da pesquisa (causa "Pré-condições", também com 27 ocorrências). A falta de tratamento para os comandos `assert` também provoca 8 falso negativos.

4.4.4 Submissão dos testes gerados automaticamente aos repositórios originais no GitHub

Como um esforço adicional de validação da proposta apresentada, submetemos os testes gerados pela pesquisa aos seus repositórios originais no GitHub por meio do recurso

Tabela 4.6: Número de falso negativos dos testes de exceção por causa identificada. A primeira coluna apresenta o número total de testes que deveriam lançar exceção, a segunda coluna apresenta o número de testes de exceção que não geraram a exceção esperada (falso negativo), a terceira coluna apresenta o número de falso negativos que teve como causa o envio de um endereço zerado como remetente da transação, a quarta coluna indica o número de falso negativos que teve como causa uma exceção lançada em chamadas de função anteriores à chamada da função sob testes, a quinta coluna mostra o número de falso positivos que teve como causa uma exceção *invalid opcode* lançada pela EVM e a sexta coluna apresenta o número de falso negativos cuja causa foi não lançar qualquer tipo de exceção.

Projeto	Exceção	Falso Negativos	Causas			
			Sender 0x0	Pré-condições	*% Invalid Opcode*	Sem exceção
actus-solidity	14	1	-	-	-	1
curve-bonded-tokens	68	8	5	-	3	1
equilibrium-bonding-curve	9	2	-	-	1	1
golem-contracts	41	6	1	4	1	-
left-gallery-token	19	2	-	-	2	-
liquid	16	8	6	2	-	-
MembershipVerificationToken	16	1	1	-	-	-
noia-token	19	2	1	2	-	-
securities	47	14	-	13	-	1
splitter	13	1	-	-	-	1
swaps	8	7	-	6	-	1
token	12	2	-	-	1	2
trickle-smart-contracts	11	1	1	-	-	-
try-catch-solidity	24	6	6	-	-	-
vvisp-klaytn-sample	24	6	6	-	-	-

de *pull request* oferecido pela própria plataforma. Inicialmente, foram selecionados para esta etapa de validação os 19 projetos nos quais a média de cobertura de ramos alcançada pelos testes gerados na análise experimental foi superior à cobertura de ramos oferecida pelos testes originais.

Definiu-se um procedimento padrão para submissão do *pull request* com as seguintes atividades sequenciais:

- geração de *fork* do repositório do projeto para a conta do autor deste estudo na plataforma GitHub[10];
- clone do repositório *fork* recém criado para a máquina local;
- execução do comando `npm install` para obtenção das dependências do projeto via gerenciador de pacotes *npm*;
- execução do comando `truffle test` para verificação dos resultados da versão atual dos testes do projeto;
- criação de uma *branch* no repositório sob o nome "solidity-test" para realização das alterações;
- seleção do conjunto de arquivos de testes gerados pela pesquisa, dentre as 30 execuções realizadas na análise experimental, que alcançaram a maior cobertura de ramos e, em caso de empate, a que teve menor número de falso negativos;
- adição do conjunto de arquivos selecionado no item anterior à *branch* de trabalho;
- edição dos arquivos adicionados à *branch* de trabalho para remoção dos testes ordinários e também dos testes de exceção que deram falso negativo (se houver), ou seja, o *pull request* conterá apenas os testes de exceção bem sucedidos;

[10]https://github.com/fabianorodrigo

- reexecução do comando `truffle test` para confirmação de que o novo conjunto de testes do projeto continua funcionando;
- realização do *commit* do código e envio (*push*) da *branch* para o repositório nos servidores do GitHub;
- solicitação do *pull request* com as mudanças realizadas para o repositório original do projeto via interface web da plataforma GitHub.

O procedimento foi seguido conforme planejado e o *pull request* foi submetido para 15 dos 19 projetos. A execução do primeiro comando `truffle test` retornou sem erros para 14 deles, a exceção foi o projeto `token`, que apresentou dois erros pela versão atual dos testes originais presentes no projeto. Após a execução do segundo `truffle test` previsto no procedimento, apenas os mesmos dois erros continuavam retornando. Desta forma, decidiu-se por fazer a submissão do *pull request* com os testes gerados de forma automatizada pela pequisa para este projeto.

Os 4 projetos que apresentaram problemas, não passaram pelo procedimento de forma completa, e, logo, não tiveram *pull request* submetido ao repositório original, estão listados a seguir com as suas respectivas justificativas:

- actus-solidity: o projeto sofreu mudanças relevantes desde a versão obtida em 15/07/2019, com exclusão ou mudança nos nomes originais dos contratos. Desta forma, os testes gerados na pesquisa não eram mais compatíveis com a versão mais atual do projeto;
- left-gallery-token: na data de conclusão deste estudo, o repositório já não estava mais disponível na plataforma GitHub;
- solidity-sigutils: dos oito testes gerados de forma automática por este estudo, nenhum deles era um teste de exceção;

- splitter: a segunda execução do comando `truffle test` do procedimento levou um dos testes originais a falhar. O ponto de falha era a verificação de saldo de uma das contas. Com esta quebra do teste original, optou-se por não se fazer o *pull request* com os novos testes gerados de forma automatizada pela pesquisa.

A submissão dos *pull requests* foi realizada na mesma data de conclusão deste estudo. Até o momento da conclusão, não foi recebido nenhum retorno dos proprietários dos repositórios originais sobre a contribuição enviada.

4.4.5 Ameaças à validade da pesquisa

A identificação e avaliação das ameaças à validade dos resultados de uma pesquisa é indispensável à condução de um processo de experimentação. Cientes das limitações do processo experimental, os pesquisadores deverão adotar soluções para eliminar ou mitigar tais riscos, quando possível. De acordo com WOHLIN et al. (2012), as ameaças à validade de uma pesquisa estão organizadas em quatro categorias: ameaças a conclusão, ameaças internas, ameaças de construção e ameaças externas.

As ameaças à validade de conclusão questionam a capacidade de se chegar a conclusões corretas quanto ao relacionamento estatístico entre os tratamentos e o resultado observado. No que se refere a procedimentos de mitigação de ameaças desta categoria, o experimento empreendido nesta pesquisa realizou o procedimento de geração automatizada dos testes 30 vezes para cada um dos projetos do escopo de análise, medindo assim o efeito dos sorteios aleatórios realizados durante a aplicação da técnica. Os dados obtidos em todas as execuções são sintetizados tanto em gráficos *boxplot* quanto por representação tabular (Tabela 4.3) e apresentam cobertura de ramos mínima, média, mediana e máxima, bem como o desvio padrão apurado na série de execuções para cada projeto.

Ameaças à validade interna estão relacionadas a influência de outros fatores alheios ao controle do estudo que possam afetar os resultados. A fim de eliminar o risco de influência dos testes originais encontrados nos repositórios dos projetos ou de execuções anteriores durante a medição da cobertura dos testes gerados pela técnica proposta, esta pesquisa adotou como prática que o subdiretório "test" fosse apagado no início do procedimento da geração automatizada de testes. Com isso, os testes originais não poderiam afetar os resultados dos testes gerados pela técnica sob avaliação.

As ameaças à validade de construção questionam se objetos e participantes do estudo refletem a questão que está sendo abordada. Por exemplo, a utilização de uma métrica equivocada para se avaliar os resultados da pesquisa poderia afetar diretamente as conclusões. Neste sentido, o presente estudo adotou como indicador de eficácia o percentual de cobertura de ramos obtido pelos testes, uma métrica reconhecida e utilizada amplamente tanto em pesquisas científicas quanto na indústria. Outro exemplo desta categoria é quando o pesquisador, de forma consciente ou não, influencia os participantes de uma pesquisa. Como os projetos selecionados para aplicação da proposta já existiam nos repositórios do GitHub antes da pesquisa, não houve qualquer contato entre o pesquisador e os desenvolvedores que participam em tais projetos. Logo, a influência nas soluções e na forma de desenvolvimento dos contratos é nula.

Quanto às ameaças externas, que se referem ao risco de generalização indevida dos resultados obtidos no estudo experimental para uma população maior que os participantes do experimento, a presente pesquisa adotou um processo sistemático na seleção dos projetos que comporiam o universo de observação da análise experimental. Este processo e os critérios de seleção utilizados são descritos em detalhes nas Seções 4.2 e 4.3. Um dos critérios adotados, e que pode representar uma realidade distorcida em relação a outras populações, é que foram analisados somente projetos com código aberto na plataforma GitHub. Há, sem dúvida, um conjunto significativo de projetos que fogem deste universo de observação, como, por exemplo, projetos abertos em outras plataformas ou

projetos que não tenham código aberto. Por outro lado, no intuito de selecionar um grupo de comparação mais qualificado, foram escolhidos apenas projetos que implementassem testes automatizados e que não apresentassem falhas durante a sua execução. Um aspecto positivo que se pode observar no universo selecionado é a presença de projetos dos mais variados tamanhos e complexidades, desde projetos com apenas dois ramos de decisão até projetos com mais de 350 ramos.

4.5 Considerações Finais

Este capítulo apresentou o processo e critérios utilizados na seleção do conjunto de projetos para composição do escopo de aplicação e avaliação experimental da proposta de automação de geração de testes unitários para contratos inteligentes apresentada no Capítulo 3. Foram apresentados os índices de cobertura de ramos obtidos com a versão original dos testes dos 40 projetos selecionados, bem como os resultados de cobertura de ramos obtidos após um conjunto de execuções utilizando a técnica proposta, seguindo por uma análise qualitativa dos resultados obtidos pela técnica em projetos com resultados menos favoráveis e também os que alcançaram um desempenho mais significativo.

O próximo capítulo apresenta a conclusão que se chega do trabalho empreendido nesta pesquisa, os resultados alcançados, suas limitações e os trabalhos futuros que possam levar à evolução do presente trabalho.

5 CONCLUSÃO

Após os resultados apresentados no Capítulo 4, o presente capítulo tem como objetivo a apresentação das considerações finais, das contribuições alcançadas pelo estudo, da resposta à questão de pesquisa, das limitações do trabalho e das perspectivas de trabalhos futuros relacionados ao tema desta pesquisa.

5.1 Considerações Finais

Este trabalho apresentou uma técnica para identificação de requisitos de teste para as funções de contratos inteligentes escritos na linguagem Solidity com base nas expressões relacionais presentes no código-fonte destes projetos, além de construir uma ferramenta que automatize a geração do código-fonte JavaScript para execução destes testes.

A busca por soluções que possam automatizar o trabalho de geração de testes é justificada pela proporção, em termos de custo e cronograma, que as atividades de teste ocupam no processo de desenvolvimento de software. Além disso, os contratos inteligentes não apenas são responsáveis por gerenciar transações com montantes significativos em criptoativos, mas eles carregam em si próprios tais ativos. Tal fato configura um incentivo à exploração de defeitos do software por participantes mal intencionados da rede.

O estudo da literatura acerca da automação dos testes de contratos inteligentes mostra que o problema já foi abordado por um conjunto de pesquisadores. Contudo, os estudos encontrados focam na localização de um conjunto de padrões de codificação dos contratos inteligentes já conhecidos por seu relacionamento com algumas das vulnerabilidades de segurança do ambiente *blockchain*.

Embora realize o mapeamento dos requisitos de teste e a geração do código-fonte que executa tanto teste ordinários quanto testes de exceção, a ênfase deste trabalho está nos casos de teste cujo resultado esperado seja o lançamento de uma exceção oriunda de validações sobre parâmetros e variáveis de estado não atendidas. Este foco se justifica porque os testes de exceção possuem um oráculo conhecido – a geração da exceção –, enquanto o oráculo dos casos de teste ordinário só pode ser determinado, em geral, com conhecimento do domínio da aplicação sob testes.

Uma análise experimental foi conduzida e apresentada no Capítulo 4 a fim de validar a eficácia da técnica proposta. Foram selecionados 40 projetos de contratos inteligentes com código-fonte disponível na plataforma GitHub. Sobre eles, foi executada uma série com 30 repetições do processo de geração automática do código-fonte dos testes. Este experimento comparou os resultados de cobertura de ramos obtidos com a técnica desenvolvida como parte da pesquisa contra a cobertura de ramos dos testes encontrados nos repositórios dos projetos selecionados. Além do ganho por demandar um esforço mínimo e um tempo significativamente menor que o empregado na produção manual dos testes, os testes gerados pela técnica proposta alcançaram uma cobertura igual ou superior em 22 dos 40 projetos.

5.2 Contribuições

As principais contribuições deste estudo são:

- Uma técnica para a identificação de requisitos de teste para contratos inteligentes escritos em Solidity com base nas expressões relacionais presentes em seu código;
- Uma ferramenta que automatiza a geração do código de teste, em JavaScript e com base no *framework* Truffle, para os contratos inteligentes escritos em Solidity;

- A avaliação experimental da técnica e da ferramenta, com sua aplicação em projetos de contratos inteligentes com código-fonte disponível na plataforma GitHub.

5.3 Resposta para a questão da pesquisa

Após a análise dos resultados do estudo experimental apresentado no Capítulo 4, é possível responder a questão de pequisa proposta neste trabalho:

RQ1: Há melhoria significativa na cobertura de ramos pelos testes gerados através da técnica proposta em comparação com o percentual de cobertura de ramos alcançado pelos testes unitários presentes nos projetos de contratos inteligentes?

> Os testes gerados pela pesquisa de forma automatizada alcançaram, na média, uma cobertura igual ou superior em 22 dos 40 projetos. Em 19 destes 22 projetos, todos os testes gerados pela pesquisa superaram a cobertura de ramos alcançada pelos testes originais. Em dez projetos, os teste gerados alcançaram, no mínimo, o dobro de cobertura de ramos alcançada pelos testes originais. Em contrapartida, não foi possível gerar os testes para um projeto e em outros quatro projetos a cobertura de ramos ficou abaixo da metade da alcançada pelos testes originais.

Assim, conclui-se que os testes gerados apresentam resultados de cobertura de ramos iguais ou melhores na maioria dos projetos analisados, sendo maior do que o dobro em alguns destes projetos.

5.4 Limitações

A principal limitação da pesquisa se encontra nos tipos de elementos encontrados nas expressões relacionais que ainda não são tratados pela pesquisa. Com isso, quando a análise do código-fonte se depara com uma expressão relacional em que um dos elementos seja de um desses tipos (como uma variável local, um operação matemática, um acesso indexado como acontece nos casos de *arrays* ou *mappings* ou uma chamada a uma função), o algoritmo não é capaz de identificar quais são os valores necessários para que aquele ramo seja coberto pelos testes. Para estes casos, a cobertura de ramos dependerá do fator aleatório utilizado na geração dos parâmetros de entrada da função testada.

Durante a análise experimental, alguns projetos apresentaram falso negativos, isto é, o teste foi gerado tendo como resultado esperado o lançamento de uma exceção em determinado ponto e durante sua execução a exceção não ocorreu ou não ocorreu no ponto previsto. Parte desses falso negativos está diretamente associada à limitação apontada no parágrafo anterior; outra parte é atribuída a geração de testes que simulam o envio de uma transação a partir do endereço: '0x0000000000000000000000000000000000000000', devido a inexistência deste endereço na rede de testes utilizada, a transação é rejeitada sem nem mesmo executar a função sob testes; alguns falso negativos tem como causa a rejeição de uma validação realizada através do comando `assert`, o que gera uma exceção do tipo *invalid opcode*. Por fim, um outro conjunto residual de falso negativos simplesmente não geraram exceção e precisariam ter sua causa investigada de forma mais minuciosa, é possível até que estejam associados a alguma falha de codificação da ferramenta desenvolvida durante a pesquisa. De toda forma, acredita-se que seja tecnicamente possível a eliminação de todos esses casos de falso negativos, independentemente do grupo de causas no qual foi classificado.

5.5 Trabalhos Futuros

Algumas sugestões de evolução desta pesquisa incluem:

- Ampliar o escopo e os tipos de elementos utilizados nas expressões relacionais e tratados na identificação das condições e restrições a serem atendidas pelos testes gerados pela técnica proposta, reduzindo-se assim os falso negativos;
- Avaliação de técnicas distintas da geração aleatória para a geração dos valores de parâmetros da função que não possam ser definidos de forma determinística. Por exemplo, algoritmos de busca mais sofisticados, como algoritmos genéticos, podem ser utilizados para aprimorar o processo de geração de valores para os parâmetros;
- Implementação da técnica proposta para geração de testes para contratos inteligentes que sejam executados em plataformas concorrentes da Ethereum;
- Submissão dos testes gerados de forma automatizada a uma avaliação de eficácia com a aplicação de ferramentas de testes de mutação;
- Tratamento dos casos em que a geração dos testes depende da criação de contratos que implementem determinada interface para que possa ter instâncias passadas como parâmetro para as funções testadas;
- Geração de oráculos para os testes ordinários.

REFERÊNCIAS

ANDESTA, E.; FAGHIH, F.; FOOLADGAR, M. Testing Smart Contracts Gets Smarter. **arXiv preprint arXiv:1912.04780**, [S.l.], 2019.

ATZORI, M. Blockchain technology and decentralized governance: is the state still necessary? **Available at SSRN 2709713**, [S.l.], 2015.

BALDWIN, J. In digital we trust: bitcoin discourse, digital currencies, and decentralized network fetishism. **Palgrave Communications**, [S.l.], v.4, n.1, p.1–10, 2018.

BENMELECH, E.; DLUGOSZ, J. The credit rating crisis. **NBER macroeconomics annual**, [S.l.], v.24, n.1, p.161–208, 2010.

CASINO, F.; DASAKLIS, T. K.; PATSAKIS, C. A systematic literature review of blockchain-based applications: current status, classification and open issues. **Telematics and Informatics**, [S.l.], v.36, p.55–81, 2019.

CHEN, T. et al. Under-optimized smart contracts devour your money. In: IEEE 24TH INTERNATIONAL CONFERENCE ON SOFTWARE ANALYSIS, EVOLUTION AND REENGINEERING (SANER), 2017. **Anais...** [S.l.: s.n.], 2017. p.442–446.

CHEN, T. et al. A Large-Scale Empirical Study on Control Flow Identification of Smart Contracts. In: ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), 2019. **Anais...** [S.l.: s.n.], 2019. p.1–11.

CHRISTIDIS, K.; DEVETSIKIOTIS, M. Blockchains and smart contracts for the internet of things. **Ieee Access**, [S.l.], v.4, p.2292–2303, 2016.

COINMARKETCAP. **Cryptocurrency Market Capitalizations**. 2020.

DEMILLI, R.; OFFUTT, A. J. Constraint-based automatic test data generation. **IEEE Transactions on Software Engineering**, [S.l.], n.9, p.900–910, 1991.

DURIEUX, T. et al. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. **arXiv preprint arXiv:1910.10601v2**, [S.l.], 2020.

ETHERCHAIN.ORG. **Evolution of the total number of Ethereum accounts**. 2020.

FRASER, G.; ARCURI, A. Sound empirical evidence in software testing. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), 2012. **Anais...** [S.l.: s.n.], 2012. p.178–188.

GREVE, F. G. et al. Blockchain e a Revolução do Consenso sob Demanda. **Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)-Minicursos**, [S.l.], 2018.

HARTEL, P.; SCHUMI, R. Gas limit aware mutation testing of smart contracts at scale. **arXiv preprint arXiv:1909.12563**, [S.l.], 2019.

HARTEL, P.; STAALDUINEN, M. van. Truffle tests for free–Replaying Ethereum smart contracts for transparency. **arXiv preprint arXiv:1907.09208**, [S.l.], 2019.

JIANG, B.; LIU, Y.; CHAN, W. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: ACM/IEEE INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 33. **Proceedings...** [S.l.: s.n.], 2018. p.259–269.

KOSBA, A. et al. Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY (SP), 2016. **Anais...** [S.l.: s.n.], 2016. p.839–858.

KRUPP, J.; ROSSOW, C. teether: gnawing at ethereum to automatically exploit smart contracts. In: USENIX} SECURITY SYMPOSIUM ({USENIX} SECURITY 18), 27. **Anais...** [S.l.: s.n.], 2018. p.1317–1333.

LIBICKI, M. C.; ABLON, L.; WEBB, T. **The defender's dilemma**: charting a course toward cybersecurity. [S.l.]: Rand Corporation, 2015.

LUU, L. et al. Making smart contracts smarter. In: ACM SIGSAC CONFERENCE ON COMPUTER AND COMMUNICATIONS SECURITY, 2016. **Proceedings...** [S.l.: s.n.], 2016. p.254–269.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011.

NGUYEN, T. D. et al. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. **arXiv preprint arXiv:2004.08563**, [S.l.], 2020.

NOFER, M. et al. Blockchain. Business & Information Systems Engineering 59, 3, 183–187. **DOI: http://dx. doi. org/10.1007/s12599-017-0467-3**, [S.l.], 2017.

PETERS, G. W.; PANAYI, E. Understanding modern banking ledgers through blockchain technologies: future of transaction processing and smart contracts on the internet of money. In: **Banking beyond banks and money**. [S.l.]: Springer, 2016. p.239–278.

PINNA, A. et al. A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics. **IEEE Access**, [S.l.], v.7, p.78194–78213, 2019.

PORRU, S. et al. Blockchain-oriented software engineering: challenges and new directions. In: IEEE/ACM 39TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING COMPANION (ICSE-C), 2017. **Anais...** [S.l.: s.n.], 2017. p.169–171.

PRESSMAN, R.; MAXIM, B. **Engenharia de Software-8ª Edição**. [S.l.]: McGraw Hill Brasil, 2016.

SHEN, C.; PENA-MORA, F. Blockchain for Cities—A Systematic Literature Review. **IEEE Access**, [S.l.], v.6, p.76787–76819, 2018.

SHIGEMURA, R. A. L. et al. Wibx: making smart contracts even smarter. In: 2017 . **Anais...** [S.l.: s.n.], 2019.

SWANSON, T. Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. **Report, available online, Apr**, [S.l.], 2015.

SZABO, N. Smart contracts in Essays on Smart Contracts. **Commercial Controls and Security**, [S.l.], 1994.

SZABO, N. The idea of smart contracts. **Nick Szabo's Papers and Concise Tutorials**, [S.l.], v.6, 1997.

TAN, L. et al. Bug characteristics in open source software. **Empirical software engineering**, [S.l.], v.19, n.6, p.1665–1705, 2014.

TIKHOMIROV, S. et al. Smartcheck: static analysis of ethereum smart contracts. In: INTERNATIONAL WORKSHOP ON EMERGING TRENDS IN SOFTWARE ENGINEERING FOR BLOCKCHAIN, 1. **Proceedings...** [S.l.: s.n.], 2018. p.9–16.

TORRES, C. F.; SCHÜTTE, J.; STATE, R. Osiris: hunting for integer bugs in ethereum smart contracts. In: ANNUAL COMPUTER SECURITY APPLICATIONS CONFERENCE, 34. **Proceedings...** [S.l.: s.n.], 2018. p.664–676.

WALKER, D. M. **The Oxford Companion to the Law**. Oxford, UK: Oxford University Press, 1980.

WAN, Z. et al. Bug characteristics in blockchain systems: a large-scale empirical study. In: IEEE/ACM 14TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES (MSR), 2017. **Anais...** [S.l.: s.n.], 2017. p.413–424.

WANG, X. et al. Basis Path Coverage Criteria for Smart Contract Application Testing. In: INTERNATIONAL CONFERENCE ON CYBER-ENABLED DISTRIBUTED COMPUTING AND KNOWLEDGE DISCOVERY (CYBERC), 2019. **Anais...** [S.l.: s.n.], 2019. p.34–41.

WHITE, M.; KILLMEYER, J.; CHEW, B. Will blockchain transform the public sector? **Blockchain basics for government, A report from the Deloitte Center for Government Insights, Deloitte UniversityPress. Wunsche, A.(2016). Technological Disruption of Capital Markets and Reporting**, [S.l.], 2017.

WOHLIN, C. et al. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012.

WOOD, G. et al. **Ethereum**: a secure decentralised generalised transaction ledger. 2019.

ZACK, M. H. **Knowledge and strategy**. [S.l.]: Routledge, 2009.

ZALEWSKI, M. **Technical"whitepaper" for afl-fuzz**. [Online; acessado em 11/11/2020], `https://lcamtuf.coredump.cx/afl/technical_details.txt/`.