UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Gustavo Garcia Gava

FILTRO DE PACOTES COM AUXÍLIO DE SMART CONTRACTS

Florianópolis

2019

Gustavo Garcia Gava

FILTRO DE PACOTES COM AUXÍLIO DE SMART CONTRACTS

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciência da Computação para a obtenção do Grau de Bacharel em Ciência da Computação.
Orientador: Prof. Dr. Jean Everson Martina

Florianópolis

2019

Gustavo Garcia Gava

FILTRO DE PACOTES COM AUXÍLIO DE SMART CONTRACTS

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciência da Computação", e aprovado em sua forma final pelo Programa de Graduação em Ciência da Computação.

Florianópolis, 01 de julho 2019.

Prof. José Francisco Danilo De Guadalupe Correa Fletes
Coordenador do Curso

Prof. Dr. Jean Everson Martina
Orientador

Banca Examinadora:

Douglas Marcelino Beppler Martins

Lucas Pandolfo Perin

Ao meu avô Walter Gava, o Vô Velho. Obrigado pelas longas conversas e por me apresentar ao bom cinema.

AGRADECIMENTOS

Agradeço à minha família por estar sempre ao meu lado, seja para dividir bons momentos quanto para me apoiar quando necessário, e tornarem possível eu ser o homem que hoje sou. Por desde sempre me ensinarem o valor do estudo. Por me ensinarem o que é amor, companheirismo e honestidade.

Aos amigos do tempo de escola, que me acompanharam por praticamente toda minha vida e não perdiam a paciência comigo mesmo quando eu sumia para fazer este trabalho, obrigado Simão.

Aos amigos que fiz em Relações Internacionais, por me tornarem um bom jogador de truco e me proporcionarem estórias da graduação para a vida inteira, espero que um dia vocês me perdoem pelo churrasco que queimei em 2013.

Aos amigos da Computação, pelo suporte durante à graduação e para reclamar dela. Aos amigos do futebol por me consagrarem com chutes fáceis.

Aos amigos de Sidra, pelas músicas de bom gosto e me acolherem como membro da banda mesmo que eu só apareça de vez em quando para tocar uma música ou outra.

Ao pessoal do Reza a Lenda pelas cervejas de toda sexta-feira. Ao pessoal do Café Patagônia por sempre me receberem muito bem quando precisava de um lugar para me concentrar. Ao pessoal da sala individual da BU, já que ela foi quase minha segunda casa.

Aos amigos do LABSec, por toda a ajuda e conversas durante esses anos. Principalmente aos amigos Juliano Zatta, Pablo Montezano e Lucas Palma pela a ajuda em realizar este trabalho.

Ao professor Jean Martina e aos membros da banca, pela orientação e me acompanharem durante o desenvolvimento deste trabalho, me aconselhando e me cobrando quando necessário.

RESUMO

Este trabalho apresenta a proposta de um modelo de filtro de pacotes que utiliza um ou mais *smart contracts* para a realização do veredicto. Um filtro de pacotes é um tipo de *firewall* que funciona nas camadas de Rede e Transporte do modelo TCP/IP, atuando de forma independente às aplicações. Suas regras são estáticas e utilizam informações dos cabeçalhos de cada pacote para verificar a aderência à cada regra, portanto, questões que vão além destas informações devem ser tratadas na camada de aplicação. Ao acoplar um *smart contract* ao filtro de pacotes, desejamos dar a capacidade ao usuário de realizar acordos sobre o acesso à sua rede ou computador independentemente de suas aplicações, expandindo a capacidade do filtro padrão. Portanto, apresentamos um modelo de filtro dividido em três módulos, de modo a permitir total controle e personalização para o usuário. Mostramos também a implementação de um protótipo funcional capaz de realizar os objetivos propostos, demonstrando o funcionamento do modelo e seus resultados frente a um filtro de pacotes padrão.

Palavras-chave: Firewall. Filtro de Pacotes. iptables. Smart Contract.

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A Internet está se tornando cada vez mais presente na vida da população. Hoje, estima-se que aproximadamente metade da população mundial tenha acesso à internet, número que chega a 81% em países desenvolvidos segundo a *International Telecommunications Union*(2018). Além disso, para as pessoas que possuem acesso, a internet vem tomando espaço em diferentes aspectos da rotina humana, seja para entretenimento, trabalho, comunicação, questões burocráticas, armazenamento de dados, etc. Portanto, quanto mais a internet se torna presente em nossa rotina, maior se torna a importância da segurança em nossas redes, questão presente em diferentes formas e níveis em toda a comunicação pela rede, seja pela privacidade dos nossos dados, garantia da qualidade de serviço ou até mesmo para nos livrarmos de empecilhos como o *spam*.

Nesse contexto, adotamos algumas ferramentas e práticas em conjunto para proteger nossos dados e comunicações. Aqui citamos os protocolos de comunicação com autenticação, criptografia, *softwares* antivírus e o próprio design seguro de cada aplicação(BISHOP, 2003). Além destes citados, também utilizamos os *firewalls*, que são como barreiras posicionadas entre duas redes, servindo para controlar o que entra e sai da rede que protege.

No entanto, os *firewalls* a nível de rede, os filtros de pacote, não possuem conhecimento sobre o conteúdo das mensagens, eles só são capazes de enxergar os aspectos técnicos da conexão, mas não analisar o contexto desta. Assim, só temos um conjunto limitado de opções para decidir que dados podem entrar e sair. Então, se o administrador de uma rede desejar verificar algo mais, por exemplo um pagamento, para aceitar uma mensagem ou iniciar uma conexão? A princípio ele recorre às suas aplicações para verificar essa informação.

Como exemplo, vamos imaginar um servidor que presta um serviço pago, em que o cliente deve efetuar um pagamento para ter acesso ao serviço. Quando o servidor recebe uma requisição de serviço, esta passa pelo filtro de pacotes independentemente se o serviço foi pago ou não, e então essa avaliação é feita a nível de aplicação. Para cada diferente serviço adicionado para funcionar nesse servidor, cada uma dessas aplicações vão necessariamente ter que realizar essa checagem de pagamento.

Portanto, nesse trabalho propomos um modelo de filtro de pacotes que permita o usuário expandir a capacidade de decisão de seu

filtro. No modelo proposto, a decisão de entrada e saída de pacotes é realizada por um *smart contract* escrito pelo administrador da rede. Ao utilizar um *smart contract* o usuário é capaz de definir quaisquer acordos que desejar para reger o acesso à sua rede ou computador, como por exemplo incluir a necessidade de um pagamento às suas regras. Assim, qualquer pessoa que desejar se comunicar com esse usuário deve cumprir as regras de acordo definidas pelo *smart contract*. Deste modo, possibilitamos a criação de uma conexão regida por um contrato, que estabelece regras e custos definidas pelo contratado.

O usuário pode desejar utilizar este modelo para diferentes tarefas, como a cobrança de serviços(independente de quais sejam as aplicações) ou recebimento de mensagens, como email. Seguindo no exemplo de um servidor, mas agora pensando em um serviço a princípio gratuito, mas limitado por uso, em que os usuários podem requisitar o serviço gratuitamente, desde que sua demanda não ultrapasse um dado limite. A partir do momento em que um usuário passa a demandar além do limite, seus pacotes podem ser direcionados à regra de consulta ao *smart contract*, que demandará um pagamento pelo uso. Com o *smart contract* é possível aplicar diferentes modelos de precificação para as requisições. Como citado, uma abordagem semelhante pode ser tomada para o recebimento de mensagens, com o exemplo de emails, aplicando um modelo "me pague para lhe ouvir".

Podemos citar também a possibilidade de vários usuários utilizarem o mesmo *smart contract*, criando assim uma sub-rede regida por contrato entre eles, em que somente os pagantes podem entrar e se comunicar nessa rede. As mensagens enviadas por membros dessa rede ainda seriam acessíveis a membros externos, mas diferentes formas de tornar essas mensagens privadas podem ser aplicadas, mas que não são o escopo deste trabalho.

1.1 OBJETIVOS

1.1.1 Objetivos Gerais

Projetar e propor um modelo de filtro de pacotes capaz de decidir sobre a entrada de pacotes IP em um computador com base na decisão de um *smart contract*. O sistema deve fornecer as informações necessárias sobre o pacote para contrato efetuar sua decisão. O remetente que tiver seu pacote negado deve receber um aviso indicando o erro e deve receber uma informação sobre o *smart contract* acoplado

ao sistema. Implementar um protótipo para verificar o funcionamento do modelo e testá-lo, observando seu funcionamento e eficiência.

1.1.2 Objetivos Específicos

1. Desenvolver o sistema de veredicto sobre os pacotes capturados. O sistema deve extrair as informações do cabeçalho de cada pacote e consultar o veredicto do *smart contract*, e aplicá-lo sobre o pacote. O sistema deve ser capaz de ser integrada com diferentes *smart contracts* para cada caso de uso.

2. O sistema deve ser capaz de alertar o remetente em caso de veredicto negativo, enviando uma mensagem de alerta informando o erro e indicando o *smart contract* relacionado.

3. Para fins de validação, implementar um *smart contract* e acoplá-lo ao filtro. O *smart contract* deve ser capaz de decidir sobre a entrada de cada pacote. Além disso, deve ser desenvolvido um módulo de comunicação com o contrato.

4. Utilizar o sistema em um computador e verificar a eficácia, analisando se os veredictos dados estão corretos e se a filtragem está sendo efetiva. Avaliar também se a mensagem de erro está sendo enviada corretamente. Verificar o atraso causado pelo uso da aplicação no recebimento das mensagens.

5. Documentar o funcionamento do sistema e as etapas do modelo, demonstrando também seus resultados e apresentando todos os conceitos necessários para a proposta.

1.2 ESTRUTURA DO TEXTO

Primeiramente, faremos um estudo de fundamentação teórica, passando por todos os aspectos empregados na proposta e prototipação do trabalho. Então, explicaremos o modelo proposto de forma independente de tecnologias. Por fim, realizaremos a apresentação e o estudo de um protótipo implementado, com a demonstração e análise de resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 HASH CRIPTOGRÁFICO

Uma função de *hash* é utilizada para transformar uma mensagem, de tamanho qualquer, em um resumo de tamanho definido, o *hash*. (SCHNEIER, 1996). Uma função *hash* pode ser definida como $h(x) = y$, onde h é a função, x uma mensagem de tamanho qualquer, e y o seu resultado de tamanho definido.

Por outro lado, as funções de hash criptográficos, são funções que possuem algumas propriedades que tornam uma função de hash mais segura, de modo que seja fácil obter o resultado a função, mas extremamente difícil revertê-la. Portanto, mesmo que o hash de uma mensagem seja publico, é impossível obter essa mensagem com base no hash. Segundo Stinson (2006), uma função de *hash* seguro atende às seguintes propriedades:

i Resistência à pré-imagem: Dada uma função $h(x) = y$, se for extremamente difícil obter x, dado y, a função é resistente à pré-imagem.

ii Resistência à segunda pré-imagem: Uma função é chamada de resistente à segunda imagem caso, dado x, é extremamente difícil encontrar x' em que $h(x) = h(x')$ e $x \neq x'$.

iii Resistência à colisão: Para uma função h ser considerada resistente à colisão, deve ser extremamente difícil encontrar x e x', em que $h(x) = h(x')$ e x ≠ x'. Note que, diferente da propriedade ii, x não é conhecido.

Funções *hash* criptográfico são muito utilizadas para checar a integridade de dados, já que qualquer modificação no dado faz com que o resultado da função seja diferente. Outro uso atribuído a essas funções é a verificação de equidade entre arquivos e mensagens, já que entradas iguais geram resultados iguais.

2.2 SMART CONTRACT

Smart contract é uma forma de executar e validar um contrato digitalmente, por meio de um protocolo de transação. Um contrato é

um conjunto de acordos entre duas ou mais partes mutuamente interessadas, que ditam direitos e obrigações das partes envolvidas. Um contrato possui diversos usos, como compra, venda e doação de bens, reger as regras sob uma prestação de serviço ou até mesmo reger a relação entre pessoas. Os contratos geralmente são validados pela lei, que garante sua execução e a aplicação de suas regras previstas em caso de quebra contratual.

A base de um contrato é a sua execução. O ato de criar um contrato é a forma das partes envolvidas garantirem que vão executar as tarefas concordadas. Caso as partes envolvidas concluam suas obrigações, então o contrato é concluído. Pode haver também casos em que para a conclusão do contrato seja necessário que primeiro uma parte execute sua parte, para que então outra possa concluir (por exemplo, o pagamento de um serviço prestado). Caso alguma parte não cumpra com os termos do contrato, então executa-se o que for previsto neste contrato para este caso. Isso pode incluir somente o encerramento do acordo, ou a imposição de uma punição.

O termo *smart contract* foi proposto por Nick Szabo (SZABO, 1996) como uma forma de expandir a ideia e funcionamento dos contratos convencionais. Sua proposta é por as clausulas de um contrato em *software* ou *hardware*, automatizando o processo de execução do contrato, e tornando-o seguro computacionalmente, de forma que seja custoso(ou impossível) para uma das partes quebrar o acordo. Deste modo, ele deve ser capaz de resistir tanto quebras não intencionais, como também quebras intencionais, de alguém que possua recursos e conhecimento para buscar alguma falha no protocolo.

Smart contracts funcionam por meio de respostas ao ambiente em que estão inseridos. Isso significa que quando uma situação prevista acontece, a sua programação emite uma resposta. O exemplo mais simples para isso é um pagamento, que quando realizado, pode gerar diferentes respostas do contrato(dependendo do valor, horário, ou qualquer outra especificação feita). Um evento que causa uma mudança no estado do contrato pode ser chamado de *gatilho*.

A princípio, podemos enxergar a ideia básica de um *smart contract* em qualquer sistema automático de vendas, como uma máquina de refrigerantes ou um sistema de compra online, por exemplo. No entanto, tais sistemas geralmente não englobam questões contratuais, obrigações e direitos das partes envolvidas.

2.2.1 Características e Objetivos de um Contrato

Para garantir maior executabilidade, todo *smart contract* deve buscar quatro objetivos identificados por Szabo, que constituem a fundação de como todos os contratos(incluindo os convencionais) são formados(SZABO, 1996):

- **Observabilidade** é a capacidade de que cada parte do contrato consiga observar a execução da tarefa das outras partes envolvidas. No contexto de *smart contracts* a observabilidade pode operar de forma automatizada, já que é possível utilizar um trecho de código para verificar se uma parte foi cumprida(ARCARI, 2018). Podemos citar como exemplo o caso de um hotel, em que o *smart contract* disponibiliza a abertura do quarto somente quando o contrato detecta o pagamento do cliente, e por outro lado, pode verificar se o *check-out* foi feito dentro do prazo, e assim terminar o contrato.

- **Verificabilidade** é a capacidade de se provar o desfecho do contrato, de modo que se possa verificar se o contrato foi executado ou quebrado. Este conceito está muito relacionado à arbitragem de um contrato tradicional pela Justiça. A verificabilidade de *smart contracts* tende a depender da plataforma em que o *smart contract* funciona, já que é esta que define o nível de acesso às informações, como também é o que determina a confiança da informação fornecida. A verificabilidade tende a ser maior em plataformas em que os dados são mais confiáveis e não podem ser alterados ou excluídos com facilidade.

- **Privacidade**, sob a ótica de contratos normais, significa que os deveres e direitos do contrato só são aplicados sob as partes envolvidas nele. Ou seja, nenhuma terceira parte possui influência ou é influenciado pelo contrato(ARCARI, 2018). Szabo define privacidade como "[...] o princípio que conhecimento e controle sobre o conteúdo e execução de um contrato devem ser distribuídos entre as partes somente o quão necessário for para sua execução."(SZABO, 1996, tradução nossa). Portanto, para *smart contracts*, privacidade é além de blindar o código do contrato à ações externas, é prover o máximo de privacidade às partes envolvidas.

- **Exigibilidade**, que podemos entender como a garantia que as clausulas do contrato serão executadas. Em contratos convencionais o que tende a dar tal garantia é a lei e justiça. Para *smart*

contracts, os contratos tendem a se auto-garantir por meio de protocolos e segurança computacional. Além disso, quanto maior o nível dos objetivos anteriores, maior é a auto-exigibilidade do contrato, pois este passa a dar maior confiabilidade para as partes envolvidas.

Apesar de serem diferenciados, os objetivos se relacionam e afetam quanto um contrato necessita de envolvimento com uma terceira parte, como um mediador na duração do contrato ou um árbitro para julgar a sua execução. Enquanto a Privadade retira de terceiros acesso à informação, a Observabilidade é necessária a um mediador e a Verificabilidade pode depender de um árbitro (SZABO, 1996).

Portanto, o principal objetivo por trás dos *smart contracts* é permitir a oficialização de um acordo, se afastando da necessidade de um sistema legal(leis e justiça) como forma de garantia para o contrato. Assim, se torna mais simples e menos centralizado para que indivíduos possam realizar negócios e criar acordos, removendo a barreira burocrática imposta por um sistema jurídico.

2.2.2 Smart Contracts sobre Blockchains

Seguindo as características expostas, *smart contracts* podem ser aplicados utilizando diferentes métodos ou plataformas. No entanto, o conceito ganhou extrema popularidade quando relacionados à *blockchains*, tanto que se tornaram termos praticamente dissociáveis. Uma *blockchain* é um registro distribuído de informações organizadas em blocos. As características de *blockchains* garantem algumas propriedades por vezes desejáveis aos *smart contracts*, como imutabilidade dos dados e registro, confiabilidade, e descentralização. No entanto, os dados da *blockchain* são públicos, o que pode afetar a privacidade do contrato.

É importante notar que *blockchains* são somente uma tecnologia usada para a criação e execução de *smart contracts*, de modo que estes podem existir fora do contexto de *blockchains*. Sendo vistas como uma plataforma, as características das *blockchains* afetam diretamente os objetivos citados na seção 2.3.1, já que todos os contratos com base em uma *blockchain* assumem tais características.

Em *blockchains* descentralizadas, toda informação nova precisa ser aprovada pelos nodos participantes por meio de consenso. Portanto, se uma informação relacionada ao estado de execução de um contrato é descreditada por parte dos nodos, por qualquer motivo que seja, este *gatilho* não chegaria ao contrato. Além disso há outra questão em

relação à confiabilidade dessa informação, pois é necessário garantir que ela venha de uma parte confiável. Essas questões afetam diretamente a **Observabilidade** do contrato, já que se relacionam com a capacidade de acompanhar a execução dos acordos(ARCARI, 2018). Uma solução para obter informações externas à *blockchain* (como câmbio de moedas e valor de ações) de forma confiável são os chamados *oráculos*, uma fonte de informações em que os nodos concordam em confiar. Exemplos de *oráculos* muito utilizados atualmente são Oraclize e Augur. No entanto, a validação de informações e a confiança em dados externos ainda é um desafio em andamento aos usuários(ARCARI, 2018).

Como citado anteriormente, as *blockchains* possuem a características de serem imutáveis, de modo que dados já incluídos na corrente são confiáveis. Portanto, contratos funcionando em *blockchains* tendem a possuir alta **Verificabilidade** pois uma vez que o estado o *smart contract* é atualizado, é simples verificá-lo, além de que essa nova informação é confiável. Porém, a questão se torna mais complicada no aspecto jurídico do contrato, caso haja a necessidade de uma arbitragem, pois se torna mais difícil obter informações sobre os lados envolvidos no acordo devido ao alto nível de privacidade ao usuário fornecida pela maioria das *blockchains*. Para a arbitragem ser possível, todas as partes deveriam concordar em fornecer informações. Isso é uma das principais barreiras ao reconhecimento jurídico dos *smart contracts* como verdadeiros contratos (ARCARI, 2018).

A natureza de publicidade dos dados afeta diretamente a **Privacidade** do contrato. Os dados e o código do contrato são públicos e acessíveis a todos, no entanto é possível limitar quem pode entrar ou modificar o estado do contrato. A falta de privacidade para os dados é um grande problema para possíveis usuários de *smart contracts*, o que desestimula o seu uso, principalmente por grandes empresas. Para tal, existem algumas propostas de solução, como *On-Chain Encryption*, que são métodos de cifragem dos dados dentro da *blockchain*, tornando-os legíveis somente para as partes desejadas.(POPEJOY, 2016).

2.2.3 Exemplos de Plataformas

Atualmente, grande parte dos smart contracts são criados e executados sob alguma plataforma que utiliza *blockchain*. Algumas plataformas comuns são:

- **Ethereum**: Uma plataforma de *blockchain* pública, que permite executar *smart contracts* sem quedas, censura, fraude ou

interferência de terceiros(WANG et al., 2018). A *Ethereum* utiliza uma máquina virtual(*Ethereum Virtual Machine*), uma máquina Turing-completa, possibilitando o uso de linguagens de alto nível e, teoricamente, a execução de praticamente qualquer programa na plataforma, se exlcuído problemas de tempo e memória (BUTERIN, 2014). Sua principal linguagem para os contratos é chamada de *Solidity*.

- **Hyperledger Fabric**[1]: É uma *framework* de *blockchains* mantida pela *Linux Foundation*. Seu diferencial é permitir *smart contracts* em linguagens comuns, como *Java* e *Go*, não dependendo de uma linguagem específica como a *Solidity* da *Ethereum*.

- **NEM**[2]: É outra plataforma com base em *blockchain*, mas diferentemente das acima citadas, o código do contrato não é mantido dentro da *blockchain*. Isso permite maior escalabilidade, desempenho e facilidade de manutenção, as custas de menor descentralização. Outro diferencial é a capacidade de fazer mais operações em menor tempo, se comparado às plataformas tradicionais.

Neste trabalho, optou-se por utilizar a *Ethereum* para demonstração devido à sua maior popularidade entre usuário de *smart contracts*. No entanto, a proposta não se prende a somente essa plataforma, podendo ser empregado qualquer tipo de *smart contract*.

2.2.4 Propostas e Casos de Uso de Smart Contracts

O emprego de *smart contracts* ainda é um campo em desenvolvimento e passando por adaptações, sendo que seu uso, levando em conta o estado da arte, ainda é bastante propositivo, de modo que o conceito ainda não é claro e visível à população geral (PETERSSON, 2018). Apesar disso, existe uma pletora de ideias e implementações se difundindo além do uso clássico de **venda de bens e serviços**, sendo que algumas estão ganhando maior popularidade e uso concreto.

A capacidade de descentralização e segurança computacional dos *smart contracts* permite o seu uso para a criação de **identidades eletrônicas** seguras, e também em alguns casos auto-soberanas. Utilizando uma identidade eletrônica a verificação de identidade se torna mais segura, possibilita criação um registro de atividades, checar o

[1] <hyperledger-fabric.readthedocs.io>. Acesso em 12/07/2019.
[2] <https://nem.io/enterprise/>. Acesso em 12/07/2019.

acesso a serviços, realizar o controle de saúde(histórico e vacinação), como também efetuar a função de passaporte. As identidades eletrônicas também podem ser emitidas sem a necessidade de reconhecimento estatal, baseando sua validade em uma rede de confiança, permitindo maior privacidade, como também segurança e direito à identidade para grupos marginalizados ou reprimidos. O principal exemplo do emprego de identidade eletrônica, aliado a *smart contracts* é o programa *e-Estonia*, em que o governo Estoniano fornece um cartão digital aos seus residentes que garante mantém registros e dá acesso a serviços públicos[3].

Recentemente, também observamos a popularização no campo de **transações financeiras**. Em 2018, o grupo Banco Bilbao Vizcaya Argentaria - BBVA criou um sistema piloto de empréstimo com um contrato sob uma *blockchain*[4], facilitando e desburocratizando o processo, com ganhos também na transparência. O *Commonwealth Bank of Australia* também já empregou *smart contracts* para gerenciar transações internacionais de grande valor[5]. Tais exemplos são relacionados a bancos, mas também é possível automatizar transações, empréstimos, apostas e seguros de forma independente entre partes interessadas, sem uma agência central. Além disso, é possível trabalhar com o mercado de ações, tanto de compra e venda, como previsões e estatísticas de mercado(WANG et al., 2018).

Além dos casos citados, ainda há uma gama de propostas e campos de uso em pesquisa e desenvolvimento, algumas possuindo maior atenção, e outras sendo descartadas. *Internet of Things* e *Supply Chain* outros campos que também possuem destaque.

2.2.5 Desafios

Os desenvolvedores de *smart contracts* ainda devem superar alguns desafios para que a tecnologia se concretize. Além disso, como o conceito anda praticamente junto ao desenvolvimento de *blockchains*, muitos dos problemas e desafios destas são também limitantes aos *smart contracts.*Alguns problemas e desafios são levantados nos trabalhos de Wang et al. (2018) e Alharby e Moorsel (2017), sendo aqui expostos

[3] <e-estonia.com/>. Acessado em 12/07/2019.

[4] <www.bbva.com/en/bbva-signs-world-first-blockchain-based-syndicated-loan-arrangement-with-red-electrica-corporacion/> Acesso em 16/04/2019.

[5] <www.commbank.com.au/guidance/newsroom/commonwealth-bank-completes-new-blockchain-enabled-global-trade–201807.html>. Acesso em 16/04/2019.

alguns deles:

- **Desenvolvimento:** Alguns problemas surgem da dificuldade para os desenvolvedores criarem *smart contracts* corretamente. Além disso, *smart contracts* são mais complexos de se criar do que contratos tradicionais, já é necessário se adaptar à uma linguagem de programação. Erros de programação nos contratos podem acarretar em falhas de segurança e privacidade e defeitos na execução do contrato, causando prejuízos para as partes. O principal exemplo é o *The DAO attack*, em que um *hacker* se aproveitou de uma falha de programação de um contrato para desviar aproximadamente 60 milhões de dólares(MEHAR et al., 2019). Além disso, esse problema se agrava quando leva-se em conta a questão de imutabilidade das *blockchains*, pois uma vez que o contrato é colocado na *blockchain* não é possível mais alterá-los(salvo algumas exceções, como a plataforma NEM, citado na seção 2.3.3), o que dificulta a correção de *bugs*, e a criação de atualizações e modificações. Isso requer que os *smart contracts* só sejam implantados em sua forma final.

- **Segurança:** Além de problemas de segurança derivados de erros de implementação dos contratos, estes também podem estar sujeitos a falhas de segurança da própria plataforma em que são executados. No caso do ataque ao *The DAO*, o *hacker* usou a capacidade de realizar sucessivas chamadas recursivas de uma função para realizar vários saques sem deduzir o balanço. Este problema é chamado de **vulnerabilidade à reentrância**. Outro problema é a **dependência de timestamp**, já que os blocos de *blockchains* são marcados com um *timestamp*. Contratos que utilizam o *timestamp* como gatilhos podem ser afetados por um minerador mal-intencionado que pode manipular essa informação. Outro desafio recorrente em contratos na *blockchain* é a necessidade de oráculos confiáveis.

- **Prividade e Privacidade:** Conforme citado na seção 2.3.3, contratos em *blockchains* possuem problemas em garantir a privacidade dos usuários, o que desestimula o seu uso. Além do código do contrato ser público, também existe a exposição dos dados. Além disso, as *blockchains* não conferem total privacidade aos usuários, pois é possível tentar descobrir informações privadas do usuário por meio do seu histórico de transações.

Além dos desafios técnicos, também há o entrave legal do uso dos

contratos. Apesar de sua proposta ser eliminar a necessidade de envolvimento burocrático, o cenário atual ainda requer o reconhecimento jurídico de algumas questões(posse de alguns bens, por exemplo). Portanto ainda há questões abertas sobre o reconhecimento de *smart contracts* como contratos tradicionais em termos jurídicos(RASKIN, 2017), que também dependem da solução dos desafios técnicos citados.

2.3 MODELO TCP/IP

O modelo TCP/IP(*Transmission Control Protocol/Internet Protocol*) é o conjunto de protocolos de comunicação entre redes utilizado atualmente na Internet(TANENBAUM, 2003). Sua função é padronizar a forma que os dados são transmitidos da fonte ao destino. Seu nome deriva dos dois principais protocolos definidos pelo modelo, mas também possível encontrar referências que utilizam o termo Modelo Internet. O modelo é separado em quatro camadas, que interagem entre si de forma que cada camada recebe os dados com o serviço de sua camada superior. As camadas são:

i Camada de Aplicação: Se refere aos diferentes protocolos de serviços que utilizam a internet. Alguns protocolos definidos nessa camada são o protocolo de transferência de arquivos(*File Transfer Protocol* - FTP), de correio eletrônico(*Simple Main Transfer Protocol* - SMTP), o *Domain Name System* - DNS, utilizado para mapeamento de nomes, e o *Hypertext Transfer Protocol* - HTTP, usado para navegação na internet.

ii Camada de Transporte: Camada que separa os dados em pacotes menores e organiza a forma de transmissão desses pacotes, levando em consideração a ordem e confiabilidade. Os principais protocolos são o *Transmission Control Protocol* - TCP e o *User Datagram Protocol*- UDP.

iii Camada de Rede: Responsável pelo transporte de um pacote em uma rede, ou seja, que o pacote chegue de uma máquina à outra. É nesta camada em que o pacote recebe o seu endereço. O principal protocolo é o *Intenet Protocol*- IP, mas também o *Internet Control Message Protocol* - ICMP é definido aqui.

iv Camada de Enlace: Trata dos aspectos físicos envolvidos na transmissão de dados, como equipamentos e padrões de cabos, além da topologia de rede. Tanenbaum (2003) cita que esta camada

não é muito especificada pelo modelo TCP/IP, só deixando claro a sua necessidade de permitir o fluxo de pacotes IP.

Neste trabalho focaremos nas camadas de Rede(Protocolo IP) e Transporte(Protocolos TCP e UDP), mas também citaremos a camada de Aplicação por ser importante para o estudo de *firewalls*. Nas seções seguintes, adotaremos uma abordagem *top-down* para descrever as camadas, ou seja, descreveremos o caminho dos dados a partir da aplicação, passando por como ele é preparado, até chegarem na etapa em que são enviados.

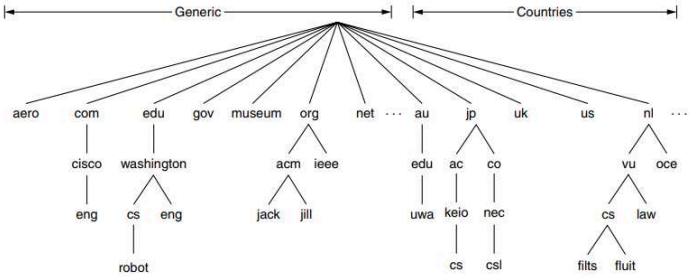
2.3.1 Camada de Aplicação

A camada de aplicação é onde todas as aplicações de rede do usuário funcionam. "As camadas situadas abaixo da camada de aplicação têm a função de oferecer um serviço de transporte confiável mas, na verdade, elas não executam qualquer tarefa para o usuário. "(TANENBAUM, 2003, p. 616). Mesmo se referindo a qualquer aplicação do usuário, existem protocolos definidos para vários serviços, dos quais alguns estamos bem acostumados a lidar.

É nesta camada que é definida a internet como conhecemos, a Web(*World Wide Web*). A Web pode ser vista como uma aplicação que facilitou a forma de acesso ao conteúdo disponibilizado em outro local(uma página da internet) e também facilitou que esse conteúdo seja disponibilizado(KUROSE; ROSS, 2010). A base do funcionamento da Web é o **Protocolo de Transferência de Hipertexto - HTTP** que é a linguagem em que o usuário e o servidor de um site se comunicam (TANENBAUM, 2003), de modo que ele funciona tanto no cliente como no servidor, em dois programas. Assim, as mensagens HTTP do usuário são utilizadas para requisitar objetos ou serviços do servidor e contam com métodos padronizados que indicam sua requisição ao servidor. Já as mensagens de resposta, enviados pelo servidor, indicam ao usuário o que ocorreu com a solicitação. Por exemplo, se o usuário requisitou algum objeto, a mensagem pode contê-lo(se ele existe e o usuário possui acesso), ou pode conter um aviso que não foi possível atender a requisição. Portanto, as respostas HTTP possuem também códigos padronizados para indicar o status. O melhor exemplo é o *404 Not Found*, que encontramos quando tentamos acessar uma página indisponível na internet.

O **Sistema de Nomes de Domínio - DNS** também é definido e executado na camada de aplicação. O DNS é o serviço responsável

Figura 1 – Uma parte do espaço de nomes de domínios da Internet



Fonte: Tanenbaum, (2003)

por traduzir o nome de um site ou endereço, que é mais intuitivo e fácil para nós, para o endereço numérico que é realmente usado na transmissão. Por exemplo, o acesso ao site *estetrabalho.com*, seria traduzido para 123.123.12.12, que é o seu endereço hipotético. O DNS funciona de maneira hierárquica, em que um domínio pode fazer parte de outro, de modo que *trabalho.org* e trabalho.net representam endereços diferentes(o primeiro está incluido no nomínio *org*, e o outro no domínio *net*). A Fig.1 mostra uma típica árvore de hierarquia dos domínios. Para realizar a tradução é necessário um servidor DNS, que recebe o domínio e realiza uma busca na árvore, e então retorna o endereço.

O DNS e o HTTP são os exemplos mais comuns de serviços padronizados que funcionam na camada de de aplicação, mas existem vários outros, como os citados no início dessa seção(FTP, SMTP, etc.). Além disso, é nessa camada que as aplicações do usuário executam, portanto é nela que os dados a serem transmitidos "surgem"(isso inclui os serviços citados). Para se referir a esses dados usaremos a palavra *payload*, a carga dos pacotes de transmissão.

2.3.2 Camada de Transporte

A camada de transporte é responsável pela comunicação entre os processos sem se preocupar em como os diferentes hospedeiros estão ligados. Segundo Tanenbaum (2003, p. 512) "sua função é promover uma transferência de dados confiável e econômica entra a máquina de origem e a máquina de destino, independente das redes". Nesta camada, os dados que chegam da camada de aplicação são separados em

segmentos e recebem um cabeçalho. Aqui são definidos protocolos que são utilizados para determinar como será feita o transporte entre os processos, sendo que cada protocolo possui um cabeçalho diferente. Os dois principais protocolos são o TCP e o UDP, que serão os tratados neste trabalho.

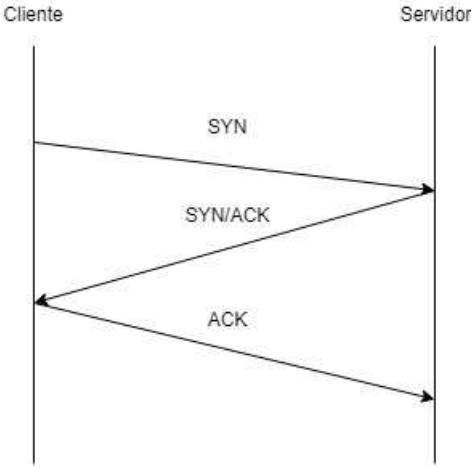
Um conceito importante para tratar da Camada de Transporte são as **portas**, que são números de 16 bits presentes no cabeçalhos da camada de transporte e são utilizados para identificar qual é o serviço envolvido em cada *host*. Algumas portas (as chamadas portas baixas) são reservadas para o uso de aplicações bem conhecidas, como a porta 53 é usada pelo serviço de DNS e a porta 80 para HTTP. Já as portas acima de 1024 são de livre uso, de modo que as aplicações do usuário podem se ligar a qualquer uma delas.

2.3.2.1 Transmission Control Protocol - TCP

O **TCP**(*Transmission Control Protocol*) é um protocolo de comunicação orientado à conexão ponto-a-ponto, que garante a entrega em sequência dos pacotes, sendo utilizada quando a confiabilidade é o foco. Uma comunicação orientada a conexão indica que ambas as partes devem concordar em quando começar um fluxo de dados(*handshaking*. A conexão possui início e fim, portanto quando um ponto encerra a conexão(ou há um *timeout*), o fluxo de dados só pode voltar a acontecer com um novo *handshaking*. Portanto, para uma aplicação utilizar o TCP para enviar seus dados, é necessário primeiro realizar esse passo, e só assim seus dados serão transmitidos. A Fig.2 mostra a sequência de comunicação que ocorre no *handshaking*. Nesse caso, o cliente deseja iniciar uma conexão com o servidor, então ele envia um pacote TCP com a *flag* SYN ativa, que indica sua intenção. O servidor, que deseja aceitar a conexão, então responde com um outro pacote com as *flags* SYN e ACK ativas. Por último, o cliente envia outro pacote com a *flag* ACK ativa, avisando ao servidor que irá começar a enviar dados. A partir desse momento, tanto cliente e servidor podem enviar dados um para o outro(serviço *full-duplex*). Para encerrar a conexão, um processo semelhante ocorre, mas utilizando a *flag* FIN.

Para garantir que todos os pacotes cheguem, e cheguem em ordem, cada pacote TCP possui um número de sequência. Quando um *host* envia um pacote a outro, esse pacote é marcado com um número, que quando recebido é comparado ao número de sequência que era esperado. O *host* destino então responde com um ACK e o número do

Figura 2 – Início de uma conexão TCP



Fonte: Autoria própria.

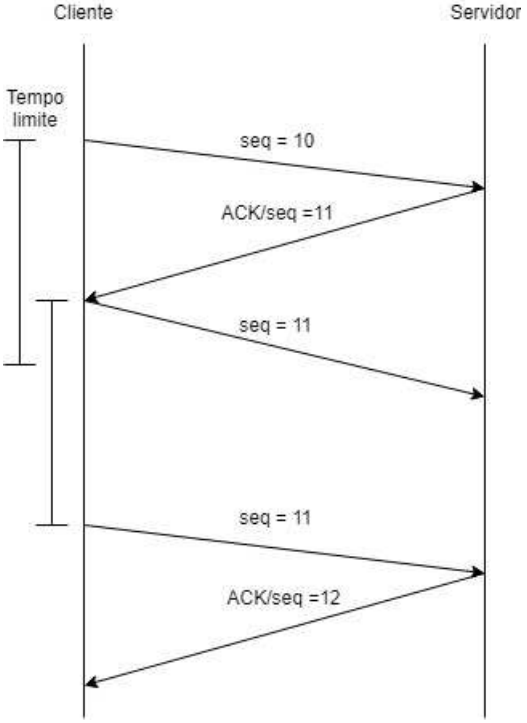
próximo pacote que espera receber[6]. Além disso, ao enviar o pacote, o *host* fonte ativa um *timer* com um tempo limite estabelecido, e caso ele não receba o ACK em tempo, o pacote é reenviado. A Fig.3 indica essa dinâmica.

2.3.2.2 User Datagram Protocol - UDP

O outro protocolo bastante usado é o **UDP**(*User Datagram Protocol*), que é muito mais simples que o TCP, pois não possui controle de entrega ou de sequência. Em suma, os únicos serviços do UDP são informar as portas envolvidas e checar a integridade do pacote(KUROSE; ROSS, 2010). Essa característica não faz com que o UDP seja pior que o TCP, pois algumas aplicações podem desejar a simplicidade. Por exemplo, transmissões de video "ao vivo"não desejam que caso um pacote seja perdido a transmissão se atrase para reenviar o pacote. Além disso, não é necessário manter uma conexão, portanto não existe o atraso inerente a iniciá-la e mantê-la, como também o cabeçalho é

[6]Na verdade, o número de sequência utilizado é o número do primeiro byte do pacote em relação à transmissão inteira, mas adotamos como número do pacote para simplificar a explicação.

Figura 3 – Envio e Reenvio - TCP



Fonte: Autoria própria.

muito menor(UDP possui 8 *bytes*, já o TCP possui no mínimo 20 *bytes*), o que diminui a quantidade de bytes gastos em um pacote. Além de alguns serviços de multimídia, outros exemplos de uso são o DNS e SNMP, utilizado para gerenciamento de rede.

Além das portas de origem e destino, existe um campo para verificação no cabeçalho UDP. Esse campo*checksum* é preenchido ao enviar o pacote, e é utilizado pelo *host* destino para verificar se nenhum bit do pacote foi alterado(por algum erro na transmissão) e verificar se esse pacote é válido. O UDP não prevê nenhuma ação em caso de erro e o pacote é somente descartado. No entanto, nada impede que uma verificação mais precisa seja realizada por parte da aplicação, que também pode requisitar o reenvio, mas isso ocorreria por parte da aplicação e foge do escopo do UDP.

2.3.3 Camada de Rede

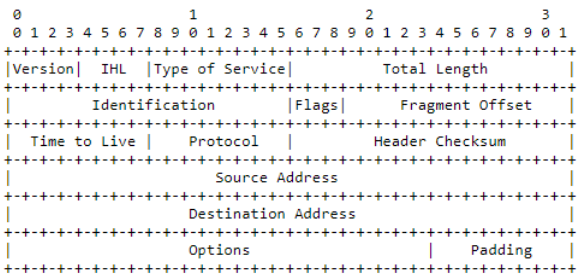
Como vimos, a camada de transporte é responsável por realizar a comunicação entre os processos. No entanto, ela não leva em consideração como um pacote chega no *host* em que o processo destino está executando. Portanto, a função da camada de rede é justamente garantir que um pacote, saindo de um *host*, chegue no *host* destino, independente da topologia de rede. Sendo assim, diferente da camada de transporte, os serviços da camada de rede são usados não só nos *hosts* envolvidos, mas também em roteadores. Aqui, os segmentos de dados que chegam da camada de transporte são encapsulados com mais um cabeçalho, referente aos serviços prestados nessa camada.

2.3.3.1 Internet Protocol - IP

O protocolo **IP**(*Internet Protocol*), definido nessa camada, é a base para o funcionamento da Internet. Na Fig. 4 podemos ver o cabeçalho IP, do qual mostraremos alguns campos resumidamente. O campo de Versão(*Version*) é usado para identificar qual a versão do protocolo usado, IPv4 ou IPv6 (Neste trabalho usaremos o IPv4 como padrão). O campo de tempo de vida (*Time to Live*) é um indicador de quantos *hops* o pacote pode dar sem ser descartado. Um *hop* é contado quando o pacote passa por algum lugar no seu caminho até o seu destino. Por exemplo, quando ele passa por um roteador é contado um *hop*. Isso impede que pacotes circulem infinitamente em uma rede. O campo de protocolo(*Protocol*) indica qual o protocolo utilizado na camada de transporte.

Já os campos de endereço são os mais importante para nós, pois são eles que representam o endereço do *host* remetente e do *host* destino. No IPv4, os endereços IP possuem 32 bits e são representados com números decimais separados por pontos, isso é, cada byte é representado por um número decimal seguido por um ponto. Portanto, o menor endereço possível é representado como 0.0.0.0 e o maior é 255.255.255.255. O que ocorre geralmente, é que os três primeiros números decimais(ou os 24 bits mais significativos) são utilizados para determinar o endereço de rede sub-rede, e o último número é usado para identificar o *host*. Por exemplo, dois computadores pertencentes à mesma sub-rede poderiam ter o endereço 192.1.1.1 e 192.1.1.2, enquanto um outro computador em outra sub-rede poderia ter o endereço 192.1.2.1. Uma sub-rede pode ser definida como uma subdivisão de uma rede, por exemplo, cada de-

Figura 4 – Cabeçalho IP



Fonte: RFC 791.

partamento de uma universidade possui uma sub-rede referente à rede da universidade.

2.3.3.2 Internet Control Message Protocol - ICMP

O **ICMP** é um protocolo utilizado para *hosts* e roteadores se comunicarem e obterem informações a respeito da rede. Por exemplo, ao tentar enviar uma mensagem a um IP inválido, um roteador ao não encontrar tal endereço, envia uma mensagem ICMP ao remetente avisando o ocorrido. O uso mais conhecido de ICMP é no programa *ping*, que envia um pacote ICMP ao *host* desejado, e aguarda outro ICMP como resposta, com o fim de verificar o estado da rede ou verificar a disponibilidade do alvo.

Apesar de também ser definido na camada de rede, o ICMP utiliza serviços do protocolo IP, e funciona de forma semelhante a um protocolo da camada de transporte, pois a mensagem ICMP faz parte do *payload* de um pacote IP, assim como é realizado com TCP e UDP(KUROSE; ROSS, 2010). Existem diferentes tipos de mensagens ICMP(definidos pelo primeiro campo do cabeçalho ICMP), que ditam o objetivo da mensagem. No caso do *ping*, a mensagem é enviada com um ICMP de tipo 8, uma solicitação de eco. Já a resposta para a solicitação é um ICMP de tipo 0.

Além do tipo, também existe um campo de código que é utilizado para detalhar a situação do ICMP. Na Tab.1 vemos alguns exemplos, com destaque para o tipo 3, que significa "Destino inalcançável"e é criado quando uma mensagem não pode ser entregue, e o código é

Tabela 1 – Tipo de ICMP

Tipo	Código	Descrição
0	0	Resposta *echo*
3	0	Rede inalcançável
3	1	Hospedeiro inalcançável
3	2	Protocolo inalcançável
3	3	Porta inalcançável
8	0	Requisição de *echo*
11	0	TTL expirado

Fonte: Autoria própria.

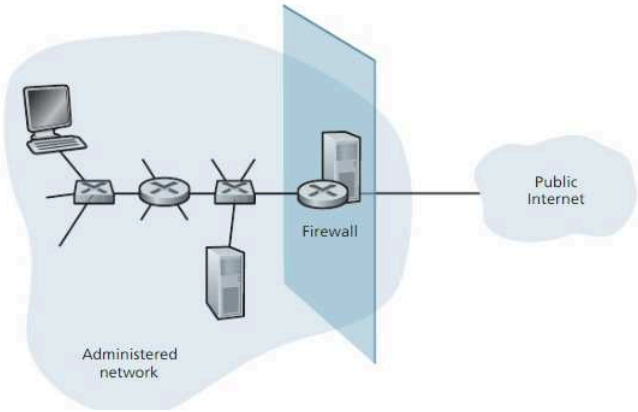
usado para especificar o motivo. Todos os tipos e códigos são definidos no RFC 792(POSTEL, 1981a). Nos casos de ICMP de erro, o pacote é enviado com os primeiro 64 bits da mensagem original(que causou o erro) para que seja possível identificar sobre qual mensagem o ICMP se refere.

2.4 FIREWALL

Um *firewall* é um sistema que controla o fluxo de dados que entram e saem de uma rede interna, de modo a isolar a rede interna da Internet ou de outra rede(KUROSE; ROSS, 2010). Ele pode ser usado para impedir que pessoas não desejadas acessem sua rede ou enviem dados maliciosos, como também pode ser usado para impedir que uma informação sigilosa seja enviado para a rede externa. Os *firewalls* podem ser usados de diferentes maneiras dependendo do seu objetivo. O modelo mais simples é demonstrado na Fig. 2, em que um único *firewall* está colocado entre duas redes. Além desse modelo, Skoudis(2006, apud KUROSE; ROSS. 2010) menciona que podem ser empregados *firewalls* em camadas ou também *firewalls* distribuídos.

Kurose e Ross (2010) classificam os *firewalls* em três modos, que geralmente são utilizados em conjunto: filtros de pacotes, filtros de pacote com controle de estado e os *gateways*.

Figura 5 – Firewall entre uma rede interna e a Internet.



Fonte: Kurose; Ross, (2012)

2.4.1 Filtros de Pacote

Os filtros de pacote verificam cada pacote e tomam sua decisão com base nas informações presentes nos cabeçalhos. Mais precisamente, o filtro funciona nas **camadas de Rede e de Transporte** do protocolo TCP/IP, levando em conta **endereços de origem e destino**, **protocolo de transporte**(TCP, UDP, ICMP, etc.), **portas de origem e destino**, além de outras informações específicas dos cabeçalhos dos diferentes protocolos de transporte(NAKAMURA; GEUS, 2017). Esse tipo de *firewall* também é chamado de *static packet filtering*, já que suas decisões são estáticas e não se adaptam sem interferência humana.

O filtro de pacotes pode ser configurado de diferentes maneiras dependendo da necessidade do administrador da rede. Por exemplo, uma companhia pode desejar que os computadores de sua rede não tenham acesso a *Voice over Internet Protocol* - VoIP e outros serviços semelhantes, o filtro pode ser configurado para rejeitar todo pacote UDP que não seja de DNS. Em outro caso, se o administrador desejar que não haja contato HTTP com a *web*, ele pode bloquear todo pacote TCP na porta 80. É possível também que seja indesejável receber *ping* na rede, portanto pode ser bloqueado pacotes ICMP do tipo *Echo*. Enfim, utilizando o filtro de pacotes é possível criar diferentes tipos de regras para diferentes motivos, baseando-se sempre na combinação de

informações do cabeçalho dos pacotes, mas sem levar em consideração o conteúdo do pacote.

As regras do filtro são mantidas em uma lista(ou tabela), de modo que cada pacote tem seus dados do cabeçalho checados na lista, de forma sequencial. Assim, a primeira regra que engloba as informações do pacote é aplicada. A Tab.2 demonstra uma simplificação de uma lista de regras para os dois primeiros exemplos indicados anteriormente. Todo pacote que passar pelo filtro será comparado sequencialmente pelas quatro regras, e o filtro aplicará a ação da primeira regra em que o pacote se encaixar. Nota-se que a sequência das regras é de suma importância, pois se a ultima regra estivesse no topo, todo pacote seria permitido e as demais regras jamais seriam aplicadas.

Tabela 2 – Exemplo de lista de regras do filtro de pacotes

IP:Porta Destino	Protocolo	Ação
IP rede:53	UDP	Permitir
IP rede:Todas	UDP	Negar
IP rede:80	TCP	Negar
IP rede:Todas	Todos	Permitir

Fonte: Autoria própria.

Este modelo é o mais simples e fácil de administrar, além de consumir poucos recursos e ter um bom desempenho para controle do tráfego. No entanto, este modelo não oferece tanta segurança quanto os outros, já que o filtro não consegue detectar pacotes fraudados ou e com informações do cabeçalho modificados, como também é suscetível a ataques sobre suas regras. A combinação de regras pode ser utilizada para realizar um ataque *backdoor*, como exemplificado por Nakamura e Geus (2017) da seguinte maneira: A Tab.3 demonstra um *firewall* que só permite conexões dos usuários da rede com páginas *web*(porta 80). Neste caso, um atacante pode abrir uma conexão com a rede utilizando a Regra 2, e utilizar um Cavalo de Tróia para obter informações privadas. Para resolver esse problema, é possível utilizar uma regra adicional na Regra 1, em que somente pacotes TCP SYN são permitidos, de modo que somente a rede possa abrir conexões. No entanto, os filtros com controle de estado oferecem uma solução diferente para esse tipo de problema.

Tabela 3 Exemplo de filtro de pacotes suscetível a *backdoor*

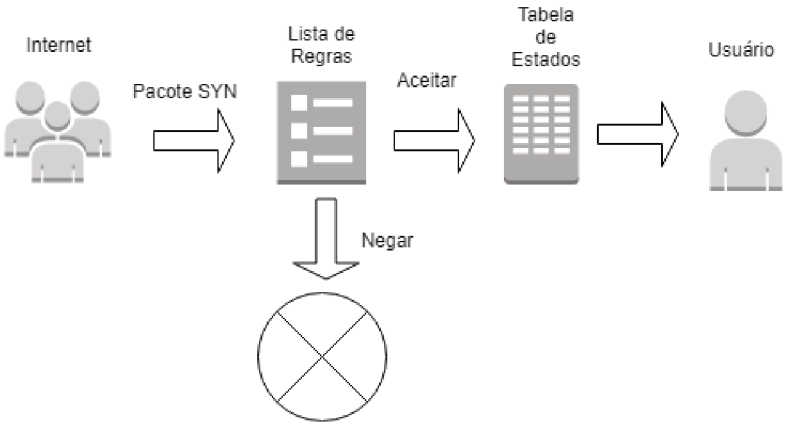
Regra	IP:Porta Origem	IP:Porta Destino	Ação
1	IP rede:Porta alta	Qualquer IP:80(HTTP)	Permitir
2	Qualquer IP:80(HTTP)	Ip rede:porta alta	Permitir
3	Qualquer IP:Todas	Qualquer IP:Todas	Negar

Fonte: Nakamura e Geus (2017), modificada

2.4.2 Filtros de Pacote com Controle de Estado

Também chamado de *stateful packet filter* ou *dynamic packet filter*, este modelo de *firewall* funciona semelhantemente ao filtro de pacotes estático, porém também conta com o auxílio de uma tabela de estados, que indica quais conexões estão abertas(NAKAMURA; GEUS, 2017). Assim, quando o administrador de rede for criar suas regras, ele só precisa levar em conta as mensagens de início de conexão(SYN).

Figura 6 Filtro de Estado - Pacotes SYN



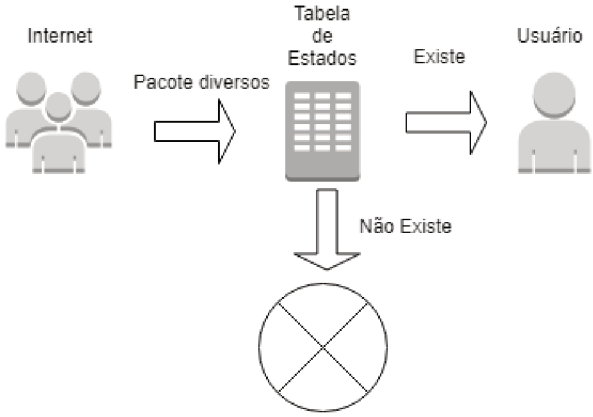
Fonte: Autoria própria.

O funcionamento deste filtro é diferente para alguns tipos específicos de pacote. Pacotes SYN, que iniciam uma conexão, primeiro são checados na lista de regras. Caso haja uma regra que aceite-o, então ele é aprovado e essa conexão é registrada na tabela de estados, por outro lado, se não houver uma regra para ele, então ele é rejeitado, conforme

a Fig. 3, que indica o caso em que algum computador da rede externa tenta abrir uma conexão com um usuário da rede interna

Já os demais pacotes não são verificados na tabela de regras e são somente checados na tabela de estados: Se houver um registro sobre a conexão a qual faz parte, o pacote é aceito, se não, ele é rejeitado. Esse fluxo é indicado pela Fig. 4. A sessão pode ser encerrada por um pacote FIN ou por um *timeout* definido, acarretando na exclusão da sessão da tabela de estados.

Figura 7 Filtro de Estado - Demais Pacotes



Fonte: Autoria própria.

Utilizando essa abordagem, a solução para o exemplo do *backdoor* se torna mais simples, já que a solução de incluir a *flag* SYN na regra se torna implícita, de maneira que basta o usuário criar uma única regra permitindo que sua rede envia pacotes para a porta 80 de qualquer endereço, que o resto será negado. Assim, o administrador garante que somente sua rede possa abrir conexões, garantindo também o fluxo em qualquer sentido dos dados dessa sessão.

Já em casos de transmissões que não são orientados a conexão, como o protocolo UDP, a tabela de estado cria uma conexão virtual utilizando dados de contexto, como portas e endereços, para determinar sobre qual sessão cada pacote se refere. Como não é possível detectar quando a sessão se encerrou, ela só é excluída da tabela em caso de *timeout*, o que é uma vulnerabilidade.

2.4.3 Gateways

Os *gateways*(ou *proxies*) são utilizados como intermediários na comunicação entre um usuário e a rede externa. Ou seja, para abrir o usuário se comunicar com o exterior, primeiro ele abre uma conexão com o *gateway*, que então abre uma conexão com a rede externa. Isso significa que o usuário e a rede externa nunca possuem comunicação externa. Os *gateways* podem funcionar na camada de transporte(*circuit level gateways*) ou na camada de aplicação(*application level gateway*), que fornecem diferentes níveis de segurança(NAKAMURA; GEUS, 2017).

O *gateway* a nível de transporte só funciona como intermediário, impedindo conexões diretas e registrando o tráfego. Já os *application level gateways* fornecem maior segurança, já que possuem acesso ao *payload* dos pacotes, e assim podem realizar a filtragem de mensagens de forma mais específica e rigorosa, o que é chamado de inspeção profunda de pacotes. Deste modo, é possível utilizar os *gateways* para filtragem voltada ao *conteúdo*, como por exemplo filtragem de emails e HTTP.

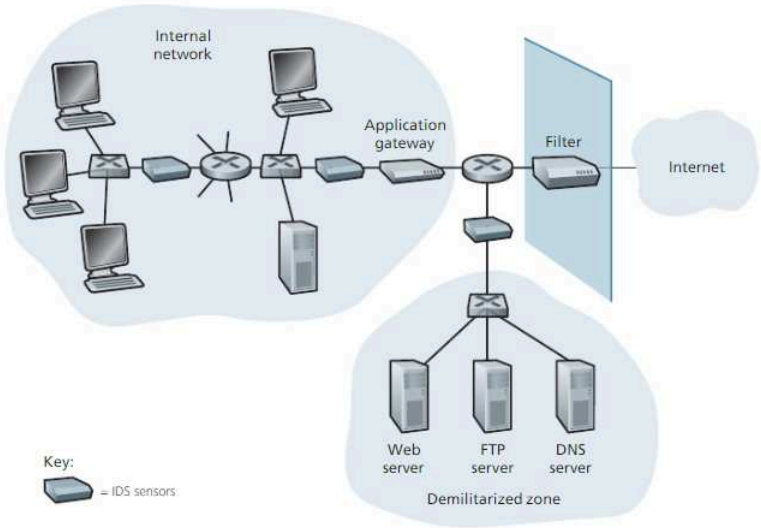
Como são voltados ao conteúdo, deve existir um *gateway* específico para cada aplicação, o que traz problemas para a escalabilidade. Além disso, existe um efeito sobre o desempenho, já que há a necessidade de tratar de todos os *payloads*, o que pode ser problemático quando vários *gateways* de aplicação estão funcionando em um único *host*.

2.4.4 Firewalls Híbridos

Como mostrado, os filtros de pacote e os *gateways* de aplicação possuem vantagens e desvantagens para casos diferentes.Assim, é muito comum que as 3 tecnologias sejam utilizadas em conjunto, de modo que os benefícios de cada tecnologia se somem. Os filtros de pacotes, por possuírem melhor desempenho, podem ser usados para se livrar de tráfegos indesejados, deixando os *gateways* de aplicação responsáveis por filtragens mais específicas, por exemplo.

Existem também os **sistemas de detecção de intrusão - IDS** que aliam a checagem do cabeçalho com a inspeção profunda de pacotes para detectar possíveis intrusos e emitir um alerta ao administrador da rede. É importante diferenciar que esses dispositivos não realizam decisões de entrada e saída de pacotes e são usados somente como um adicional à segurança da rede como um todo, não fazendo

Figura 8 – Rede com *firewall* híbrido e IDS



Fonte: Kurose e Ross(2010)

parte especificamente do *firewall*. Ataques que podem ser detectados por IDS incluem DoS, escaneamento (TCP e portas), mapeamento e vírus(KUROSE; ROSS, 2010). Os dispositivos de IDS podem ser espalhados em diferentes locais na topologia da rede, de modo a dividir a carga de pacotes a serem verificados e reduzir o escopo de possíveis ataques a procurar, aumentando assim a chance de detecção.

A Figura 4 mostra uma corporação hipotética que utiliza um filtro de pacotes em conjunto com um *gateway* de aplicação e vários dispositivos de IPS para proteger sua rede. A zona desmilitarizada(do ingles *demilitarized* zone - DMZ) indica uma região de menor segurança, protegida somente pelo filtro de pacotes e por um dispositivo IDS, que mantém serviços que devem possuir acesso direto com computadores externos. Já mais a esquerda podemos ver a rede interna, que além da proteção do filtro de pacotes, também é protegida por um *gateway* de aplicação e dois sensores IDS em diferentes locais.

2.4.5 Iptables

Iptables é a aplicação utilizada para configurar a tabela de regras do filtro de pacotes incluído no *kernel* do Linux em espaço de usuário. O iptables faz parte do projeto Netfilter, que é responsável pela filtragem de pacotes no Linux, sendo suas funcionalidades: filtro de pacotes estático, filtro de pacotes baseado em estados, tradução de endereços e portas, entre outras funções relacionadas à rede[7]. O iptables funciona sobre IPv4, possuindo uma aplicação relacionada para IPv6, o ip6tables, com funcionamento semelhante.

Suas três principais tabelas são *filter*, *nat* e *mangle*, das quais a *filter* é a padrão e geralmente mais usada para configuração de *firewall*, cuja qual trata do fluxo normal de pacotes e só não é utilizada caso outra tabela seja especificada.

A tabela *filter* possui três correntes, nas quais são especificadas as regras para cada tipo de comunicação. Estas são:

- Input: Nesta corrente estão as regras referentes a todos os pacotes recebidos no servidor em que ele é o destino final, ou seja, todos os pacotes que entram no computador.
- Output: Regras referentes a todos os pacotes que saem deste computador destinados à outro endereço.
- Forward: Corrente que trata os pacotes que passam pelo computador mas não são destinados a ele. Pode ser usada no controle de fluxo de uma rede externa à uma rede privada.

Deste modo, as correntes *input* e *output* são mais utilizadas para determinar regras localmente, para um determinado computador. Já a corrente *forward* é mais usada para regras de um *firewall* de rede, que controla a comunicação de uma rede local (SILVA, 2008).

Em cada corrente é possível determinar um conjunto de regras, que são compostas por uma ou mais condições e um veredicto(ação, ou *target*), exatamente como explicado sobre os filtros de pacotes[8]. Quando um pacote está passando por essa corrente, e todas as condições de uma regra são cumpridas, o veredicto é aplicado. As condições podem ser as informações presentes nos cabeçalhos da camada de transporte e de rede.

Os veredictos mais comuns são ACCEPT, que aceita o pacote, DROP, que descarta o pacote, e REJECT, que descarta o pacote e envia

[7] <www.netfilter.org/>. Acesso em 23/04/2019.
[8] Seção 2.6.1 e 2.6.2

uma mensagem ao endereço de origem indicando o ocorrido. Além disso, é possível redirecionar o pacote para outras correntes, ou enviá-los para uma fila em que a decisão é feita por uma aplicação em espaço de usuário.

3 TRABALHOS RELACIONADOS

A precificação de comunicações na internet não é novidade no campo da computação. Falkner, Devetsikiotis e Lambadaris (2000) citam diferentes trabalhos que propõem formas de precificação no contexto de *Quality of Service*(QoS) e *Internet Service Providers*(ISPs). A forma mais comum é o modelo *flat*, um modelo fixo em que não se leva em conta o consumo, os recursos alocados e nem estado da banda.Reichl e Hammer (2006) também levanta trabalhos em que a precificação é usada como forma de controle de congestionamento. O primeiro trabalho que propõe precificação com este fim é de MacKie-Mason (1995), onde se define o termo "Smart Market", onde cada pacote possui um campo de "lance"(como em um leilão) e em caso de congestionamento na rede, são aceitos os pacotes com maior valor. Roberts (2004) vai além do controle de congestionamento para explorar o preço como retorno de investimento.

Elovici, Ben-Shimol e Shabtai (2003) apresentam um modelo de cobrança de serviços de roteadores. Cada pacote possui campos adicionais que levam uma identificação do usuário, um campo de custo máximo e um campo de custo acumulado. A cada roteador que o pacote passa, o roteador adiciona em um campo próprio o seu custo e adiciona esse valor no campo de custo acumulado no pacote. Assim que o pacote chega em seu destino, o custo acumulado do pacote e a identificação do usuário é enviado a uma entidade central que realiza a cobrança, assim como todos os roteadores enviam seu valor de serviço acumulado para receber.

Apesar de haver relação com este trabalho, tais estudos estão mais voltados a um problema dentro de um dos casos de uso deste trabalho. Não foram encontrados trabalhos voltados à precificação em *firewalls*, que é a principal proposta deste trabalho, nem à utilização de *smart contracts* em *firewalls*.

4 FILTRO DE PACOTES COM DECISÃO DE UM SMART CONTRACT

Aqui será apresentada a proposta deste trabalho e será detalhada a arquitetura do filtro proposto e seu funcionamento, independente de tecnologias utilizadas.

4.1 PROPOSTA

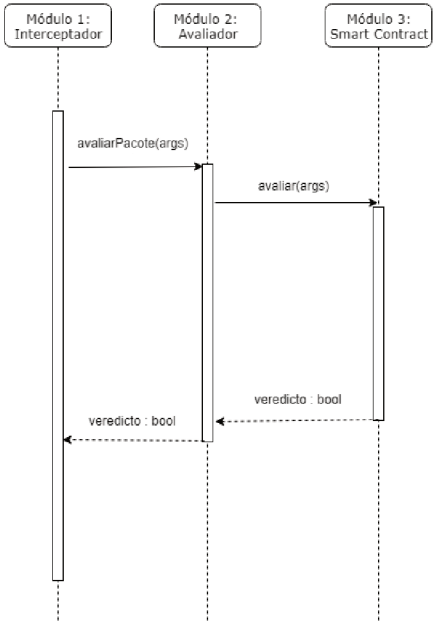
A proposta deste trabalho é apresentar um modelo de filtro de pacotes que permita o usuário repassar a decisão de entrada de cada pacote a um *smart contract*. Com isso, nós desejamos expandir o filtro de pacotes tradicional e incluir um fator econômico às regras e decisões de filtragem. Ao utilizar um contrato, o usuário possui maior liberdade e informações(como histórico de acessos, demanda e horário) para elaborar suas regras do que o filtro tradicional possibilita. Como trabalhamos nas camadas inferiores à camada de aplicação, nossa proposta pode ser utilizada de forma independente às diferentes aplicações mantidas pelo usuário. Sendo assim, nosso principal objetivo com essa proposta é permitir que o usuário possa efetuar a cobrança por acesso à sua rede, computador ou serviços de forma independente de aplicações. Adicionalmente, propomos uma forma do filtro comunicar o remetente sobre a existência do contrato em caso de rejeição de um pacote.

4.2 ARQUITETURA

O modelo de filtro é dividido em três módulos para permitir maior adaptação ao usuário. Primeiramente, criamos um módulo fixo, que é a base do funcionamento do sistema e é independente das necessidades do usuário, portanto, não é necessário que o usuário configure-o ou modifique sua implementação. Este é o módulo responsável por lidar diretamente com os pacote capturados e obter as informações de seus cabeçalhos, comunicar o *kernel* sobre a decisão tomada pelo contrato, e alertar o remetente em caso de erro. Os outros dois módulos seguintes seriam implementados pelo usuário conforme seus desejos e necessidades, mas que devem seguir um padrão determinado para que a comunicação com o módulo fixo seja possível. No caso, o segundo módulo é uma aplicação local responsável por intermediar a comunicação

do primeiro módulo com o terceiro, o *smart contract* do usuário.

Figura 9 Diagrama de sequência entre módulos



Fonte: Autoria própria.

A Fig.9 mostra a sequência de chamadas entre os módulos para que seja feito o veredicto. O Módulo 1(chamaremos de Interceptador) obtém as informações do pacote, que então as envia para o Módulo 2 solicitando qual deve ser o veredicto. O Módulo 2(Avaliador), implementado pelo usuário, é capaz de se comunicar com o *smart contract* empregado e redireciona a solicitação para o contrato(Módulo 3). Com posse das informações do pacote, o contrato é capaz de decidir se ele deve ser aceito ou não, e então o caminho inverso é feito com a decisão. O primeiro módulo então comunica o *kernel* se o pacote deve passar ou não.

4.2.1 Módulo 1 - Interceptador de Pacotes

Como dito, este módulo é a base para a funcionamento do nosso modelo e não seria modificado por diferentes usuários, a não ser configurar que pacotes serão interceptados. Ele é responsável por interceptar os pacotes presentes na fila do filtro de pacotes padrão, delegar a decisão aos módulos do usuário, informar o *kernel* a decisão tomada e emitir um alerta ao remetente em caso de negação.

4.2.1.1 Interceptação dos Pacotes

Utilizando um filtro de pacotes padrão(no caso do Linux, o *iptables*), o usuário deve definir o escopo do nosso filtro proposto. Para isso, ele deve definir uma ou mais regras do seu filtro de pacotes para redirecionar os pacotes que estarão sujeitos à decisão do seu contrato. Portanto, nosso filtro pode ser visto como um *target*(ação, veredicto) do filtro padrão. Para exemplificar, digamos que o usuário deseja as seguintes regras para os pacotes entrantes em sua rede: Todo pacote UDP em porta baixa deve ser aceito, mas os em portas altas devem ser avaliadas pelo contrato; e todo pacote TCP devem ser negados, menos os na porta 80(*web*), que devem ser aceitos, e os na porta 3000, que devem ser avaliados no contrato. A Tab.5 indica como seria o conjunto de regras desse usuário.

Tabela 4 – Exemplo de filtro de pacotes com *target* contrato.

Protocolo	Porta Destino	Ação
UDP	Porta Baixa	Permitir
UDP	Porta Alta	Contrato
TCP	80	Permitir
TCP	3000	Contrato
TCP	Qualquer	Negar

Fonte: Autoria Própria

Portanto, a primeira função deste módulo é capturar os pacotes filtrados que possuírem como ação definida a avaliação do contrato. Feito isso, o módulo tem acesso total ao pacote e é responsável por emitir a decisão sobre sua entrada. No entanto, nossa proposta é delegar essa decisão ao contrato, então obtemos as informações dos cabeçalhos do pacote e as enviamos ao contrato esperando por uma decisão, no

seguinte modelo:

Algoritmo 1 Veredicto com chamada de contrato

```
Entrada: Pacote
Saída: Veredicto
início
    obter cabeçalhos;
    if avaliador.verifique(endereçoDestino,
    endereçoFonte, portaDestino, portaFonte, protocolo)
    then
        aceite;
    else
        rejeite;
    end
fim
```

Nota-se que aqui criamos uma chamada padronizada para o módulo seguinte, que necessariamente deve possuir uma resposta *booleana* e receber os parâmetros indicados. Isso é importante pois a função *verifique()* deve ser implementada pelo usuário no avaliador, e esta deve ser feita utilizando o mesmo padrão.

4.2.1.2 Veredicto e Alerta ao Remetente

Como visto, após receber a resposta do contrato, este módulo emite o veredicto final. Caso aceito, o pacote passa normalmente pelo filtro e é entregue ao seu destino(ou enviado à uma próxima etapa de *firewall*). Se for rejeitado, o pacote é descartado. No entanto, nossa proposta é permitir incluir um fator econômico na decisão, portanto no caso de um serviço pago, por exemplo, não é interessante para o usuário que um cliente desavisado não possua conhecimento do contrato que deve ser pago. Por isso é importante indicar o motivo da rejeição do pacote.

Com tal objetivo em vista, após o descarte do pacote, enviamos um pacote ICMP ao cliente, indicando o pacote rejeitado e como o cliente pode obter acesso ao *smart contract* relacionado. Para obter a informação do contrato, definimos uma outra função padronizada em que o avaliador é chamado para retornar uma *string* contendo a mensagem sobre o contrato indicada pelo usuário.

4.2.2 Módulo 2 - Avaliador de Pacotes

Apesar de não ser aqui que realmente o pacote é avaliado, chamamos esse módulo de avaliador pois é assim que ele é visto pelo primeiro módulo. É neste módulo que a função de verificação chamada no módulo anterior deve ser implementada. Além disso, este módulo deve garantir a comunicação com o *smart contract* associado. Portanto, este módulo trabalha em conjunto com o contrato, pois é o canal de comunicação entre ele e o interceptador de pacotes, sendo também responsável por decidir que funções do contrato(ou qual contrato) será aplicado ao pacote, além de ser também capaz de dar alguns veredictos, se assim desejado.

4.2.2.1 Comunicação com o Smart Contract

Uma das principais ideias dessa proposta é que ela funcione para qualquer *smart contract*, independente de tecnologia ou plataforma. Portanto, neste modelo o interceptador não possui nenhum contato com o contrato e deixamos essa função para este módulo, que deve se adaptar conforme for a tecnologia utilizada para o *smart contract*, além de levar em consideração que o *smart contract* pode estar sendo executado localmente ou em outro local(uma rede ou outro computador).

4.2.2.2 Padronização de chamadas

A principal função deste módulo é responder às chamadas efetuadas pelo primeiro módulo. Portanto, as funções chamadas demonstradas nas Figuras 9 e 10 devem ser implementadas possuindo as mesmas assinaturas. Assim, garantimos que, para diferentes usuários, diferentes avaliadores possam ser criados sem a necessidade de alterar o interceptador. A justificativa para isso é dar transparência ao usuário e facilitar o uso deste modelo, diminuindo o nível de complexidade com que o usuário deve lidar. Por exemplo, neste etapa o usuário já possui acesso aos dados do pacote de forma simples, e o veredicto é dado somente com um valor *boolean*.

Nota-se que além destes dois padrões definidos, o usuário possui total liberdade para tratar os dados como desejar ou criar novas funções, mas que no entanto só são executadas a partir das duas funções citadas, pois há uma relação mestre-escravo entre o interceptador e o analisa-

dor. Aqui o usuário pode definir quais informações serão passadas ao contrato, analisar qual clausula(função) do contrato será utilizada para dar o veredicto ao pacote. Também é possível que o mesmo avaliador possua mais de um contrato associado, que podem ser consultados em casos diferentes. Tanto a decisão de função quanto qual *smart contract* seria consultado é transparente para o primeiro módulo, sendo que essa decisão seria realizada por regras escritas por quem configurou o avaliador. Essas regras levariam em conta também as informações de cada pacote, e por meio delas, decidir onde será realizada a decisão.

Além disso, este módulo também pode ser usado para otimizar a consulta, por exemplo, com um funcionamento semelhante ao de um filtro com base em estados, onde o contrato só seria consultado em alguns momentos, e o analisador daria o veredicto para a maioria dos pacotes, de modo que o *smart contract* seria consultado somente no início de uma conexão. Todas essas decisões partem de como e para que fins o usuário deseja utilizar este filtro.

4.2.3 Módulo 3 - Smart Contract

Por fim, temos o último módulo, o *smart contract*, que é onde a decisão do filtro realmente ocorre. Assim, no contexto da arquitetura, sua função é primariamente dar o veredicto sobre o pacote quando consultado pelo módulo avaliador. Sob a questão de implementação não propomos uma maneira específica de realizar essa tarefa, já que ela depende de como o usuário determina a relação do contrato com o avaliador, podendo, por exemplo, haver uma ou mais funções de verificação contendo diferentes assinaturas.

No entanto, a função do contrato no modelo vai além do fluxo que apresentamos nessa seção, porque é nele que é realizado o acordo entre as partes envolvidas na comunicação. Em termos simples, usando como exemplo um serviço prestado na rede protegida por nosso filtro, é utilizando o *smart contract* que o usuário desse serviço paga ao usuário do filtro(o servidor) por sua execução. Então, deve haver pelo menos uma função que permita esse pagamento, de modo que haja uma maneira para alguém que deseja se comunicar com a rede poder se "inscrever"e ter seus pacotes aceitos. Assim, o *smart contract* armazena os usuários que terão seus pacotes aceitos, e quando consultado, é capaz de responder ao avaliador. Podemos entender as funções de pagamento como **clausulas do contrato**, e também como **regras do filtro**, condições que o contratante deve garantir para que o acordo seja feito e

seus pacotes passem pelo filtro.

É possível criar diferentes regras para o contrato, de forma semelhante aos filtros de pacotes tradicionais, utilizando diferentes características dos pacotes, como portas e protocolos. Essas características podem ser unidas a um preço, adicionando assim o fator econômico que desejamos na proposta. Deste modo, quando um pacote chega em nosso filtro, ele além de ter que condizer com umas das regras de aceitação do filtro[1], ele deve possuir um status positivo quanto a um pagamento.

Tabela 5 – Exemplo de regras incluindo preço.

Protocolo	Porta Destino	Preço
UDP	Porta Alta	10
TCP	3000	20

Fonte: Autoria Própria

Na Tab.6 apresentamos um pequeno exemplo em que o contrato possuiria duas regras diferentes. O usuário que deseja se comunicar por TCP, utilizando a porta 3000 do destino deve pagar 20 unidades. Quando consultado sobre um pacote pelo avaliador, o contrato checa o protocolo, a porta destino, e verifica internamente se o preço correspondente foi pago. Em caso positivo, o contrato retornaria aceitação, já em caso negativo, rejeição. Observamos aqui que o *smart contract* emite decisão somente para os pacotes que condizem exatamente com essas regras(UDP em porta alta e TCP em porta 3000), e não haveria qualquer verificação de pagamento em qualquer pacote que não possua alguma dessas características. Como lidar com essas situações seriam decisão do administrador do filtro, podendo ser um veredicto padrão(aceitar todos ou rejeitar todos), ou garantir por meio da configuração de suas regras tradicionais de filtro(Módulo 1) e no Módulo 2 que somente pacotes que fazem parte do escopo de decisão do contrato sejam verificados por ele.

Nesse exemplo apresentamos o preço como um valor arbitrário, pois ele pode ser feito de diferentes maneiras. O preço pode ser por pacote, cenário em que o contratante compraria um número de pacotes que deseja enviar, como também o preço pode indicar uma unidade de tempo, em que o contratante compraria uma janela de tempo para efetuar sua transmissão, ou até mesmo indicar o preço por uma conexão TCP [2]. Além disso, o preço pode ser variável, já que o usuário pode

[1] Em referência à lista de regras apresentada na Seção 2.5.1,

[2] Seção 2.4.2.1

utilizar o uso atual de sua banda, o horário do dia, ou mesmo o histórico do contratante para calcular a base de seu preço.

Por fim, é utilizando a capacidade de adaptação dos *smart contracts* a diferentes problemas e demandas é que conseguimos expandir a capacidade do filtro de pacotes tradicional para incluir um fator econômico à sua decisão, de forma independente a qualquer aplicação.

5 DESENVOLVIMENTO

Neste capítulo apresentaremos as implementações realizadas em função de nossa proposta. Primeiramente, veremos como foi desenvolvido o Módulo 1 para executar em um sistema Linux e realizando todas as suas tarefas propostas. Por fim, iremos apresentar um exemplo dos Módulos 2 e 3, utilizados para testar o modelo como um todo. Definimos que o Módulo 2 seria implementado em Python devido à facilidade de comunicar com diferentes *blockchains*, e escolhemos a *Ethereum* como ferramenta para desenvolver e executar nosso *smart contract*.

Decidimos por utilizar as seguintes informações: **endereço de origem**, **endereço de destino**, **porta de origem**, **porta de destino**, **protocolo de transporte**. Os protocolos suportados são UDP e TCP, qualquer pacote com outro protocolo de transporte quer for avaliado por nosso filtro é automaticamente aceito.

5.1 IMPLEMENTAÇÃO DO MÓDULO 1 - INTERCEPTADOR DE PACOTES

O protótipo do interceptador de pacotes foi desenvolvido para ser executado em um sistema Linux, pois utilizamos o *software* de filtragem Netfilter. O módulo consiste de uma aplicação em espaço de usuário implementada em C que se comunica com o filtro de pacotes do *kernel* para informar as decisões. Nesta seção explicaremos como isso ocorre, como é realizado a comunicação com o segundo módulo e como é realizado o envio do alerta ao remetente.

5.1.1 Configuração Inicial: iptables e NFQUEUE

No sistema Linux, a forma padrão de configurar o filtro de pacotes é utilizando a aplicação iptables[1]. Portanto, é com ela que criamos as regras que redirecionam os pacotes desejados ao filtro com *smart contract*, utilizando o veredicto NFQUEUE, presente no iptables. Com este *target*, os pacotes são colocados em uma fila para decisão, que é dada por um programa que possua acesso à fila. Um exemplo de regra seria

```
$ iptables -A INPUT [ESPECIFICAÇÕES] -j NFQUEUE -queue-num 0
```

[1] Seção 2.5.5

onde `--queue-num 0` indica a fila para qual o pacote será direcionado, e as especificações seriam as características dos pacotes, como IP fonte e portas.

5.1.2 Interceptador

O primeiro pré requisito de nossa aplicação é que ela seja capaz de acessar a fila para qual os pacotes são enviados. A principal forma de se fazer isso é com a *lib* `libnetfilter_queue`, que realiza a comunicação entre o espaço de *kernel* e de usuário para este caso, acessando cada pacote e entregando o veredicto dado pela aplicação ao *kernel*. O protocolo usado pela *lib* para comunicação *kernel-usuário* é chamado *nfnetlink*, que utiliza um *socket* para a troca de mensagens sobre o pacote e seu veredicto.

Para cada novo pacote redirecionado à fila, uma função de *callback* é chamada em nossa aplicação esperando por um veredicto. Nela, temos acesso ao pacote e podemos extrair seu *payload*(utilizando a função `nfq_get_payload` definida na `libnetfilter_queue`) para ter acesso às suas informações. Inicialmente, obtemos acesso ao seu cabeçalho da camada de rede, e com isso os endereços de IP fonte e destino:

```
1  struct iphdr *ipHeader;
2  unsigned char *payloadData;
3  nfq_get_payload(packet, &payloadData);
4  ipHeader = (struct iphdr *)payloadData;
5
6  uint32_t sourceIP = be32toh(ipHeader->saddr);
7  uint32_t destIP = be32toh(ipHeader->daddr);
```

Utilizando o cabeçalho de IP também temos acesso à informação de qual é o protocolo de transporte do pacote. Com isso também podemos ter acesso ao cabeçalho da Camada de Transporte referente ao pacote. Para esse trabalho, as informações retiradas do cabeçalho de transporte são somente as portas fonte e destino(presente tanto no TCP quanto UDP), no entanto, futuramente podem ser utilizadas outras informações(flag TCP, por exemplo) para aumentar a capacidade do modelo. Com as portas obtidas, temos todas as informações necessárias para encaminhar a decisão ao avaliador.

5.1.2.1 Comunicação com o Módulo 2 e Veredicto

Como no caso do nosso protótipo o segundo módulo deve ser feito em Python(para que haja compatibildiade com as chamadas feitas), foi necessário encontrar uma forma de chamar funções de uma aplicação nessa linguagem pela nossa aplicação em C. Para tal, utilizamos a API padrão do Python para comunicação Python/C[2]. Na API, informamos os nomes definidos para o arquivo referente ao segundo módulo e as funções a serem chamadas, onde *pythonModule* é o nome obrigatório do arquivo do módulo 2, e *verify* e *getAddress* são os nomes obrigatórios das duas funções chamadas.

```
pName = PyUnicode_FromString("pythonModule");
  pModule = PyImport_Import(pName);
  pDict = PyModule_GetDict(pModule);
  pVerify = PyDict_GetItemString(pDict, "verify");
  pAddress = PyDict_GetItemString(pDict, "getAddress");
```

De volta à função de *callback*, nós chamamos a função de verificação após obter todos os parâmetros necessários. Com a resposta, é dado o veredicto ao *kernel*. Em caso de negação, a função responsável por montar e enviar o pacote ICMP à fonte é executada. Aqui, mostramos o fluxo para pacotes TCP, no entanto ele é praticamento o mesmo para pacotes UDP, por isso o omitimos. Optamos por separar os fluxos para tornar mais simples futuras implementações que tratem de peculiaridades para os diferentes protocolos da Camada de Transporte.

```
if(ipHeader->protocol == IPPROTO_TCP){
    struct tcphdr *tcpHeader = (struct tcphdr *)(payloadData
     + (ipHeader->ihl<<2));
    unsigned int sourcePort = ntohs(tcpHeader->source);
    unsigned int destPort = ntohs(tcpHeader->dest);

    PyObject *argList = Py_BuildValue("llIIs", sourceIP,
    destIP, sourcePort, destPort, "TCP");
    if(PyCallable_Check(pVerify)){
      pValueVerify = PyObject_CallObject(pVerify, argList);
    } else {
      PyErr_Print();
    }

    Py_DECREF(argList);

    if(PyObject_IsTrue(pValueVerify) == 1){
      return nfq_set_verdict(qh, id, NF_ACCEPT, 0, NULL);
    } else {
```

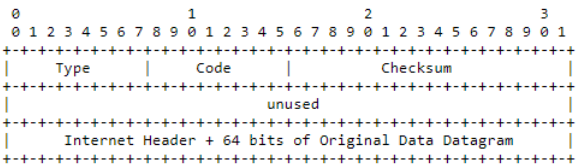
[2]https://docs.python.org/2/c-api/

```
18          set_icmp(ipStringSource, payloadData);
19          return nfq_set_verdict(qh, id, NF_DROP, 0, NULL);
20      }
21    }
22  }
```

5.1.2.2 Formato do Pacote ICMP

Normalmente, um REJECT do iptables descarta o pacote e envia um pacote ICMP de tipo 3 - Destino Inalcançável e código 3 - Porta Inalcançável à fonte. No entanto, além da *lib* `libnetfilter_queue` não suportar o veredicto REJECT, esse tipo de pacote ICMP não é suficiente para nós. Como vemos na Fig.10, o conteúdo do pacote de tipo 3 é composto do cabeçalho da camada de rede e mais 64 bits do conteúdo do pacote original(o que causou o erro), usados para identificar qual sobre qual pacote o ICMP se refere. Os 8 primeiros *bytes* do cabeçalho de transporte são suficientes para identificar as portas de origem e destino do pacote, assim como no caso de uma transmissão TCP, obter o número de sequencia do pacote. Essas informações são suficientes para o remetente identificar o pacote. Segundo o RFC792 (POSTEL, 1981a), não há espaço para informações adicionais, o que impossibilitaria a inclusão de informação sobre o *smart contract*, como desejamos.

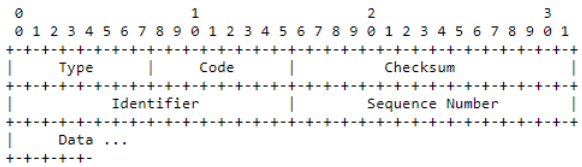
Figura 10 – Mensagem ICMP - Tipo 3



Fonte: RFC792.

Por outro lado, poderíamos utilizar o pacote de tipo 8 ou 0 (*ping* ou resposta de *ping*), que possuem um campo de informação de tamanho aberto, e assim poderíamos incluir a informação que desejamos, como podemos ver na Fig.11. No entanto, estaríamos extrapolando o propósito desse tipo de pacote, o que pode causar confusão no *host* que recebê-lo, além de conter informações desnecessárias no cabeçalho como Identificador(*Identifier*) e Número de Sequência(*Sequence Number*).

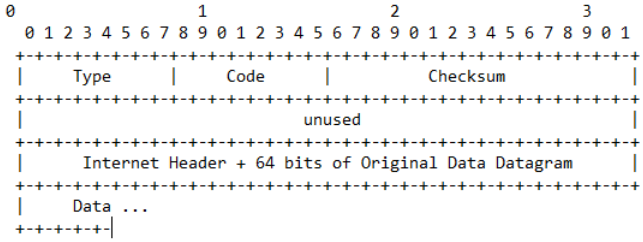
Figura 11 – Mensagem ICMP - Tipo 0 e 8



Fonte: RFC792.

Portanto, propomos utilizar um novo tipo de pacote ICMP de forma experimental, usando um dos valores 44 a 252 no campo Tipo que são reservados para implementações futuras (Internet Assigned Numbers Autority, 2018). De forma arbitrária, escolhemos o número 50 para determinar o nosso novo tipo. Utilizaremos um cabeçalho muito semelhante ao de tipo 3, contendo os bits necessários para identificar o pacote original, mas também contendo um espaço aberto para informações adicionais(*Data*). Assim, garantimos que o *host* que receber nosso aviso poderá identificar o pacote que foi rejeitado e também receber a informação de como acessar o *smart contract* necessário para concluir a comunicação. A Fig.12 ilustra o novo pacote. Para o campo de código, usaremos os mesmos definidos no tipo 3.

Figura 12 – Mensagem ICMP - Tipo Proposto



Fonte: Autoria Própria.

5.1.2.3 Construção do Pacote ICMP

A função `set_icmp` é responsável por montar e enviar o pacote ICMP que definimos. Nela, montamos o cabeçalho IP que transmitirá nosso pacote, informando o IP destino(obtido pelo pacote original, que causou o erro), e montamos o cabeçalho ICMP, informando o tipo 50, código 3 e calculamos o *checksum*. Além disso, copiamos o cabeçalho IP e os primeiros 8 *bytes* do cabeçalho de transporte do pacote original. Então, mais uma vez utilizando a API Python/C, fazemos uma chamada para o Módulo 2 requisitando o endereço do *smart contract*, que é copiado para o campo Data do nosso pacote. Com o pacote pronto, ele é enviado utilizando um *socket*.

Considerando, que o tamanho de um cabeçalho IP varia entre 20 e 60 *bytes*, e um endereço de contrato Ethereum pode ser representado em 20 *bytes*, o maior tamanho que nosso pacote ICMP pode assumir, contando com seu cabeçalho ICMP, é de 96 *bytes*, já o menor seria 56 *bytes*(cabeçalho ICMP + cabeçalho IP e 8 bytes da mensagem original + endereço do contrato). No entanto, este tamanho pode variar para outras plataformas de *smart contract* ou caso o usuário deseje enviar uma mensagem diferente.

5.2 MÓDULOS 2 E 3 - AVALIADOR E SMART CONTRACT

Apresentaremos aqui um exemplo dos dois módulos implementados pelo usuário, seguindo as regras definidas no projeto e na implementação do Módulo 1. Como utilizamos a API C/Python para realizar as chamadas no Módulo 1, o avaliador deve ser obrigatoriamente uma aplicação em Python. Quanto ao *smart contract*, utilizaremos como exemplo um *smart contract* Ethereum, por ser uma das formas mais utilizadas atualmente para trabalhar com *smart contracts*, possuindo então ampla documentação, e também por familiaridade com a linguagem Solidity, utilizada para implementar contratos Ethereum.

5.2.1 Implementação do Smart Contract

Para exemplificar, criamos o *smart contract* mais básico para o modelo, em que sua única função é verificar se existe um pagamento válido referente a uma informação que recebe. Um pagamento válido significa uma janela de tempo em que os pacotes serão aceitos, sendo

essa janela definida pela quantia paga pelo contratante. Definimos também a unidade de tempo como um bloco da *blockchain* Ethereum. O tempo de bloco é o tempo médio que leva para cada novo bloco aparecer na *blockchain*, que costuma ser constante. Durante a execução deste trabalho, o tempo de bloco aproximado da rede Ethereum era de 13 segundos[3]. Portanto, para cada unidade monetária(wei, a menor unidade do Ethereum) que o usuário paga, ele compra um bloco de tempo de transmissão. Utilizamos a estrutura *Registry* para definir a janela de tempo, onde é registrado o número do bloco que inicia a janela e o bloco que a encerra.

```
1    struct Registry{
2        uint256 endTime;
3        uint256 startTime;
4        bool registred;
5    }
```

No entanto ainda é necessário identificar o pacote e vinculá-lo com o registro. Para isso, utilizamos um *hash* das informações retiradas do cabeçalho de cada pacote(endereços, portas e protocolo), de modo que todo pacote com as mesmas credenciais geram o mesmo *hash*. Utilizando esse *hash*, podemos vincular essas credenciais a cada registro com um *hash map*.

```
1    mapping(bytes32 => Registry) public packet;
```

Para realizar um pagamento, o contratante informa o *hash* das credenciais dos pacotes que deseja transmitir, um número usado para calcular qual será o número do bloco inicial da janela, e o valor da transação. A função `pay` é usada para o pagamento, criando e vinculando um registro. Nela calculamos o tempo permitido de transmissão com base no valor do pagamento(representado por `msg.value`).

```
1    function pay(bytes32 hash, uint256 blocktime) payable
     public {
2        require(packet[hash].registred == false);
3        packet[hash].registred = true;
4        packet[hash].startTime = blocktime + block.number;
5        packet[hash].endTime = packet[hash].startTime + msg.
     value;
6        owner.transfer(msg.value);
7    }
```

Por fim, temos a função `verify`, que é a função chamada pelo avaliador para dar o veredicto sobre um pacote. Sempre que chamada, recebendo um *hash* como parâmetro, é verificado se há um registro

[3] <etherscan.io/chart/blocktime>. Acesso em 17/05/2019.

associado ao *hash* recebido e se o bloco mais recente da *blockchain* está dentro da janela de tempo. Em caso positivo, é retornado um valor verdadeiro para o veredicto do avaliador, já em caso negativo retorna-se um valor falso, pois ou nunca houve um pagamento para essas credenciais, ou não estamos na janela de tempo permitida(ou ela ainda não começou, ou já se expirou). É importante notar que a palavra *view* na assinatura da função faz com que não haja custo para a execução desta função, pois ela é somente usada para consulta. Isso possibilita que tal função seja chamada várias vezes pelo avaliador livremente.

```
1         function verify(bytes32 hash) view public returns (
      bool){
2         return (packet[hash].registred
3         && block.number >= packet[hash].startTime
4         && block.number <= packet[hash].endTime)
5     }
```

5.2.1.1 Deploy do Contrato

Com o contrato pronto nós podemos incluí-lo em uma *blockchain*. A partir desse momento, ele se torna acessível ao público, recebendo um endereço. O endereço é seu identificador na *blockchain* e é necessário para qualquer um que deseje acessá-lo(por exemplo no avaliador, mostraremos a respeito na próxima sub-seção). No entanto, ele se torna imutável e para realizar qualquer modificação é necessário republicá-lo, gerando um outro endereço. Outra informação gerada que precisamos para realizar chamadas do contrato no avaliador é a ABI(*Applicaiton Binary Interface*) do contrato. A ABI funciona como um manual de instruções para nossa aplicação, informando como deve ser realizada as chamadas de função e como as respostas devem ser interpretadas.

5.2.2 Implementação do Avaliador

Assim como o *smart contract*, implementamos o avaliador mais básico necessário para o modelo, executando somente as sua função obrigatória de permitir a comunicação entre o primeiro e último módulo. Utilizamos a *lib* Web3 para realizar a comunicação com a rede Ethereum, utilizando-a para interagir com as funções do nosso *smart contract*. Com isso, definimos o endereço do nodo *Ethereum* que será consultado, e também incluímos o endereço de nosso *smart contract*.

```
from web3 import Web3, HTTPProvider

w3 = Web3(HTTPProvider('http://192.168.66.244:8545'))

smartAddress = "0x4ac023e56952099a806b591b8722cc4e49805766"
```

Definimos também as duas funções necessárias para o funcionamento do primeiro módulo. A função `getAddress`, quando chamada só retorna o endereço do contrato(`smartAddress`). Já a função `verify`, que é chamada para todo pacote, é responsável por receber as credenciais dos pacotes e concatená-los para então aplicar a função *hash*. Com o *hash* obtido e preparado, basta chamar a função `verify` do contrato e retornar a resposta ao Módulo 1. Para realizar a chamada, utilizamos o endereço e a ABI do contrato. Observa-se que respeitamos a assinatura de função necessária definida no Módulo 1. Observamos também a tarefa deste módulo de preparar os dados recebidos do primeiro módulo para haver compatibilidade com o contrato, tanto por suas clausulas(identificação pelo hash), quanto por sua tecnologia(implementado sob a Ethereum).

```
def verify(sourceIP, destIP, sourcePort, destPort, proto):

    m = hashlib.sha256()
    srcipBytes = struct.pack(">I", sourceIP)
    dstipBytes = struct.pack(">I", destIP)
    srcportBytes = struct.pack(">I", sourcePort)
    dstportBytes = struct.pack(">I", destPort)
    prtBytes = proto.encode("utf8")

    m.update(srcipBytes + dstipBytes + srcportBytes +
    dstportBytes + prtBytes)
    h = '0x' + m.hexdigest()

    with open('contract.abi', 'r') as abi_definition:
      abi = json.load(abi_definition)
    address = w3.toChecksumAddress(smartAddress)
    contract = w3.eth.contract(address=address, abi=abi)
    return(contract.functions.verify(h).call())
```

5.3 IMPLANTAÇÃO DO PROTÓTIPO E TESTES

O filtro foi instalado e testado em um computador utilizando o sistema Ubuntu(Linux) como um *firewall* local, sendo ele o *host* destino. O objetivo do experimento foi avaliar a corretude dos veredictos e também avaliar o impacto sobre a eficiência da decisão.

Os testes foram realizados com o envio de pacotes TCP e UDP

de um computador fonte pra o computador destino com o filtro instalado. O nodo Ethereum consultado para comunicar com o *smart contract* estava em um terceiro computador na mesma rede local que o computador destino.

Por definição, utilizaremos nessa seção a palavra *destino*(em itálico) para se referir ao computador que utiliza nosso filtro e é o destino das mensagens, e utilizaremos *fonte*(em itálico) para se referir ao computador utilizado para enviar mensagens rumo ao *destino*.

5.3.1 Rede Ethereum

Decidimos por utilizar uma rede privada Ethereum, cuja seus nodos não estão conectados com a rede Ethereum principal, para publicar nosso *smart contract*, pois assim obtemos maior controle sobre o cenário de teste, como velocidade de mineração, quantidade de *tokens*(para simular o valor pago em cada transação) e acesso ao *smart contract*. Utilizamos também um nodo completo, responsável por minerar as transações e executar nosso contrato, sendo este nodo o consultado pela Módulo 2 a respeito do *smart contract*. Utilizamos o método detalhado por Montezano (2018) para criar a rede e definir o nodo.

5.3.2 Teste do Protótipo

Aqui iremos definir um cenário de teste para ilustrar o funcionamento de nosso filtro. Demonstraremos por meio de capturas de imagens a execução do filtro no *destino* e a captura do pacote ICMP emitido à *fonte*.

5.3.2.1 Cenário de Teste

Utilizamos a aplicação `Socat` para ser nossa aplicação exemplo. O Socat é um programa de linha de comando que permite o usuário criar conexões ponto-a-ponto com configurações próprias. Definimos que o *destino* utilizará a porta 4001 para receber pacotes UDP contendo mensagens de texto. Assim, qualquer computador pode enviar um pacote UDP para a porta 4001 do *destino* com uma mensagem de texto, e a mensagem aparecerá no terminal do *destino*. No lado da *fonte*, utilizaremos a porta 4000 para enviar nossas mensagens, direcionadas ao endereço do *destino*. Com esse cenário, simulamos o uso

de uma aplicação qualquer e podemos verificar o funcionamento de nosso filtro. O nodo Ethereum consultado pelo avaliador é um terceiro computador presente na rede, portanto o *smart contract* é executado neste computador. Abaixo, sumarizamos a forma que a *fonte* irá se comunicar:

- IP Destino:150.162.56.139
- IP Fonte: 150.162.56.165
- Porta Destino: 4001
- Porta Fonte: 4000
- Protocolo Usado: UDP

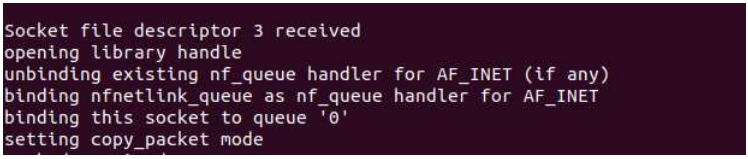
5.3.2.2 Execução

A princípio, sem nenhuma regra configurada com o iptables, qualquer pacote seria aceito. Com o filtro instalado no computador destino, redirecionamos todos os pacotes recebidos do computador *fonte* ao nosso filtro com o seguinte comando:

```
$ iptables -A INPUT 150.162.56.165 -j NFQUEUE -queue-num 0
```

Com isso feito, executamos nossa aplicação, que então fica no aguardo de novos pacotes, estado mostrado na Fig.13. As mensagens mostradas são *health checks* realizados na inicialização do sistema, obtidos do arquivo de exemplo de uso da *lib* `libnetfilter_queue`.

Figura 13 – Execução do Filtro - Inicio



Fonte: Autoria Própria.

A partir desse momento, qualquer pacote recebido proveniente do IP Fonte será rejeitado, independente se siga o padrão que definimos ou não. Agora, somente com a liberação no *smart contract* os pacotes desse IP serão aceitos. Ao enviarmos uma mensagem na *fonte*, o filtro indicará no terminal *destino* a rejeição(*Drop*), como mostrado na

Fig.14. Podemos ver também o *hash*[4] obtido pelos campos do cabeçalho que são verificados(IP e Porta fonte/destino e protocolo). Esse é o *hash* utilizado para consultar o *smart contract*. É importante notar que, diferente das outras informações mostradas no terminal, o *print* do *hash* é realizado pelo Módulo 2, e não pelo Módulo 1. A mensagem *Packet sent to: [...]* indica o envio da mensagem ICMP. No caso dessa imagem, é ilustrado uma rejeição de um pacote com o padrão definido no nosso cenário.

Figura 14 Execução do Filtro - Negado

```
Socket file descriptor 3 received
opening library handle
unbinding existing nf_queue handler for AF_INET (if any)
binding nfnetlink_queue as nf_queue handler for AF_INET
binding this socket to queue '0'
setting copy_packet mode
Packed Received
Entering callback
Packet Hash:
0x860e373d5df1516c756dd1c21d53ada8f74b18fb509cf55e0509e05564b43f7f
150.162.56.165
Socket set to TTL..
Packet sent to: (h: 150.162.56.165)(150.162.56.165)
Drop from:150.162.56.165 to:150.162.56.139 From port 4000 to port 4001
```

Fonte: Autoria Própria.

Para que um pacote igual seja aceito, é necessário realizar uma transação no *smart contract* informando esse mesmo *hash*. Utilizando o Remix[5], abrimos o *smart contract* por meio de seu endereço. Pela interface do Remix, usamos a função `pay` para registrar o *hash* que indica o padrão dos pacotes que definimos para esse cenário. Transferimos 25 *wei* para comprar 25 blocos(como citado, nossa unidade de tempo para esse *smart contract*) e definimos o tempo inicial da janela como o primeiro bloco em que essa transação for publicada(*blocktime 0*).

Com o pagamento realizado e o bloco da transação constando na *blockchain*, os pacotes UDP enviados pela porta 4000 da *fonte*, com destino à porta 4001 do *destino* serão aceitos. Na Fig.15 mostramos o comportamento do filtro ao aceitar o pacote. Todos os pacotes que sigam o padrão definido no cenário são aceitos, desde que a janela de tempo paga ainda esteja válida. Pacotes que possuam outro padrão(e consequentemente geram um *hash* diferente) continuam sendo

[4] A função *hash* utilizada foi a sha-256.

[5] Remix é uma IDE para implementação de *smart contracts* em Solidity. Nela podemos carregar contratos já existentes para acessar e executar suas funções.

Figura 15 – Execução do Filtro - Aceito

```
labsec@labsec-System-Product-Name:~/Área de Trabalho/Gava/TCC$ sudo ./test.o

Socket file descriptor 3 received
opening library handle
unbinding existing nf_queue handler for AF_INET (if any)
binding nfnetlink_queue as nf_queue handler for AF_INET
binding this socket to queue '0'
setting copy_packet mode
Packed Received
Entering callback
Packet Hash:
0x860e373d5df1516c756dd1c21d53ada8f74b18fb509cf55e0509e05564b43f7f
Accept from:150.162.56.165 to:150.162.56.139 From port 4000 to port 4001
```

Fonte: Autoria Própria.

rejeitados. Na Fig. 16 mostramos a mensagem enviada("Mensagem de Teste") sendo recebida pelo socat executando no *destino*. Após 25 blocos minerados por nosso nodo a *fonte* para de aceitar mais pacotes.

Figura 16 – Mensagem recebida na aplicação destino

```
labsec@labsec-System-Product-Name:~$ socat UDP-listen:4001 -
Mensagem de Teste
```

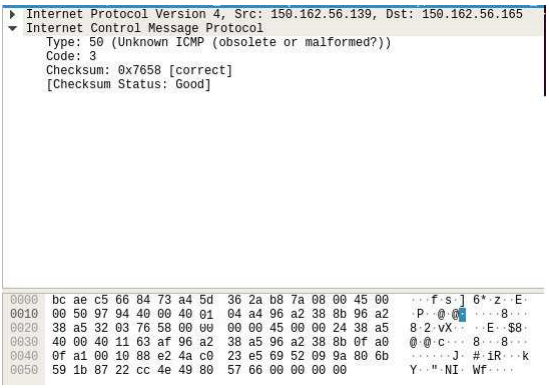
Fonte: Autoria Própria.

5.3.2.3 Verificando envio de ICMP

Para testar o envio e recebimento do pacote ICMP em caso de negação utilizamos a ferramenta Wireshark[6] na *fonte*. A Fig.17 mostra a captura de um pacote ICMP emitido pelo *destino* quando um pacote é rejeitado. Na parte de cima, em texto, podemos ver que o Wireshark identifica o pacote como ICMP e indica seu tipo e código(Como o 50, que usamos, não é definido em nenhum RFC o Wireshark é incapaz de identificá-lo, por isso a mensagem "*obsolete or malformed?*"). Em

[6]Wireshark é uma aplicação de captura de pacotes, que nos permite observar quaisquer pacotes que passem pelo computador. Com ela, podemos observar os campos desejados nos cabeçalhos, como também analisar o *payload* dos pacotes

Figura 17 – Recebimento pacote ICMP - Wireshark



Fonte: Autoria Própria.

baixo, em notação hexadecimal, é mostrado todo o pacote. As linhas 0000 e 0010 são os bytes do cabeçalho IP do pacote. A partir do terceiro byte da linha 0020 em diante são representados os campos que definimos na seção 5.1.2.2. Nas linhas 0040 e 0050 podemos ver o endereço de nosso *smart contract*(4ac023...5766).

5.3.3 Teste de Desempenho

5.3.3.1 Método de medição de desempenho

Para avaliar o desempenho do filtro utilizamos um cenário semelhante ao da seção anterior, mas com uma alteração, definimos o socat na fonte para não só receber uma mensagem mas também enviar uma resposta, simulando um servidor. Deste modo, mediremos o efeito sobre o desempenho pela perspectiva do cliente.

A maneira escolhida para realizar a medição do tempo de requisição e resposta foi a utilização do Wireshark na *fonte*, que captura a mensagem de solicitação e a resposta recebida, marcando-os com um *timestamp* em segundos. Por subtrair o *timestamp* do pacote de solicitação do *timestamp* da resposta obtemos o tempo de resposta do servidor. Para fins de comparação, utilizamos o tempo de resposta em uma caso que nosso filtro não é empregado, e comparamos com os casos de pacotes aceitos por nosso filtro e também de pacotes negados(nos ca-

sos de negação, utilizamos a resposta ICMP para obter o *timestamp*). É importante relembrar que o nodo consultado para a execução do *smart contract* não está no mesmo computador que os módulos 1 e 2, portanto é necessário uma consulta pela rede, o que pode gerar latência. Como esse teste está sujeito a outras variáveis obtemos dez amostras para cada caso e calculamos a média simples como resultado final.

5.3.3.2 Resultados do teste de desempenho

A Tab. 7 mostra os resultados das medições para cada amostra, obtida em segundos. O valor mostrado para cada amostra é o tempo de resposta obtido. A última linha indica a média simples obtida das dez amostras. Para fins de representação neste trabalho, arredondamos os números após seis algarismos à direita. Como podemos ver, utilizando somente o filtro padrão(iptables) o tempo de resposta foi de, em média, 2,6 milissegundos. Já utilizando o nosso protótipo, o tempo de resposta passou para 27 milissegundos. Essa mudança de desempenho pouco afetaria aplicações voltadas ao usuário, como *web pages*, aplicativos de trocas de mensagens e emails. No entanto, aplicações que dependam de eficiência no tempo de resposta ou realizam grandes trocas de pacotes em pouco tempo podem vir a ser afetados, portanto, cabe o usuário do filtro identificar se o uso do *smart contract* é benéfico.

Tabela 6 – Resultados do experimento de eficiência

	Filtro Padrão	Protótipo - Aceito	Protótipo - Negado
Amostra 1	0,003508	0,038735	0,041807
Amostra 2	0,002804	0,025444	0,050624
Amostra 3	0,003360	0,023083	0,041115
Amostra 4	0,002317	0,025512	0,038362
Amostra 5	0,003050	0,023174	0,048127
Amostra 6	0,003647	0,027236	0,025305
Amostra 7	0,001468	0,026877	0,035811
Amostra 8	0,002718	0,028792	0,037091
Amostra 9	0,001451	0,027475	0,039402
Amostra 10	0,001871	0,026291	0,026965
Média	0,002619	0,027262	0,038461

Fonte: Autoria Própria

Para o mesmo interceptador(Módulo 1) tais resultados estão su-

jeitos a alterações para diferentes decisões durante a implementação do Módulo 2 e escolha de *smart contract*, como também à fatores como a localização do nodo consultado quando utilizado um *smart contract* em *blockchain*. Como vimos, o nosso avaliador consulta o *smart contract* para todo pacote que recebe e não possui nenhuma tentativa de otimização. Por outro lado, consultamos um nodo dedicado presente na mesma rede que nosso filtro, o que nos fornece um tempo de resposta melhor, já que o efeito da latência é menor.

6 CONSIDERAÇÕES FINAIS

Neste trabalho propomos e prototipamos um modelo de filtro de pacotes que utiliza um *smart contract* para tomar suas decisões. O principal objetivo para essa proposta foi incluir um fator econômico, um pagamento por exemplo, ao processo de decisão dos filtros de pacotes tradicionais. O protótipo foi criado para um sistema Linux e apresentou bons resultados, realizando a filtragem corretamente e com um efeito sobre o desempenho tolerável para a maior parte das aplicações. Discutiremos nesse capítulo os resultados obtidos em relação aos objetivos propostos, faremos uma reflexão a respeito do modelo, e por fim indicaremos trabalhos futuros.

6.1 RESULTADOS GERAIS

Todos os objetivos propostos foram cumpridos com sucesso. Primeiramente, definimos um modelo de filtro de pacotes que permita a inclusão de um ou mais *smart contracts* como realizadores de decisão de entrada de pacotes. Com a inclusão do *smart contract* se torna possível utilizar um pagamento como regra adicional no filtro de pacotes. Utilizando o protótipo implementado para testes, o filtro proposto foi capaz de emitir veredictos corretamente com base no escopo de tipo de pacotes definido para ele.

Propusemos também uma forma de comunicar o usuário que teve seu pacote negado por não realizar o pagamento necessário sobre o *smart contract* relacionado ao filtro. Para tal, foi proposto um novo tipo de pacote ICMP capaz de informar qual pacote foi rejeitado e carregar uma mensagem obtida no filtro. O protótipo apresentado também foi capaz de realizar essa tarefa.

Utilizando o protótipo junto um *smart contract* exemplo, realizamos um teste de impacto sobre a eficiência da tomada de decisão. O resultado obtido foi considerado satisfatório, causando um impacto de aproximadamente 25 microssegundos, o que é imperceptível para o usuário humano.

6.2 DISCUSSÃO DO PROJETO

O modelo proposto possibilita a expansão das capacidades do filtro de pacote tradicional. Com ele, o usuário pode incluir a condição de um pagamento como mais um tipo regra na configuração de suas regras de filtragem. O principal objetivo por trás dessa ideia é possibilitar a cobrança de pagamento independente de quais aplicações estão envolvidas, já que o filtro de pacotes não depende da Camada de Aplicação.

Ao utilizar o modelo que foi proposto, o usuário ganha em liberdade para definir como serão suas regras, que informações do pacote serão avaliadas, como será feito o pagamento e também o que será levado em conta para calcular o preço. O usuário deve tomar a decisão de utilizar, e como utilizar, este modelo com base na sua necessidade e objetivos. Além disso, mesmo tendo seu funcionamento independente de aplicações, o usuário deve considerar quais aplicações estarão protegidas por trás do filtro, pois isso deve estar relacionado aos objetivos traçados pelo usuário, como também pode haver a interferência no funcionamento da aplicação, principalmente em relação à eficiência.

6.3 TRABALHOS FUTUROS

Este trabalho foi sobretudo propositivo, tratando da proposta de um modelo novo de filtro de pacotes, portanto, surgem novas questões a respeito. Aqui, iremos propor algumas ideias de extensão ao projeto que surgiram durante o desenvolvimento, como também alguns questionamentos que não foram possíveis elucidar neste trabalho.

Primeiramente, o protótipo implementado possui baixa complexidade, utilizando poucas informações dos pacotes e estando limitado somente aos protocolos UDP e TCP. Ao expandir as capacidades do módulo implementado, aumenta-se a capacidade de uso do filtro, o que pode trazer novos casos de uso não explorados neste trabalho. Além disso, o *smart contract* utilizado para o teste é também simples, usado somente para verificar o comportamento do sistema como um todo. É interessante verificar o comportamento do modelo utilizando diferentes *smart contracts* com complexidades diversas, como também implementar *smart contracts* em diferentes plataformas, encorpando a proposta atual.

Somado ao ponto citado, uma validação mais completa a respeito do efeito na eficiência seria de grande contribuição. Como citado

no trabalho, o teste realizado utilizou um nodo Ethereum presente na mesma rede que o hospedeiro do filtro, e a localização do nodo consultado pode causar efeitos diferentes na eficiência do modelo. Além disso, diferentes tecnologias de *smart contract* podem possuir resultados diferentes. Como houve impacto sobre a eficiência de decisão, um ensaio sobre a resistência a ataques de negação de serviço também seria proveitoso para levantar a suscetibilidade e resistência do filtro a este tipo de ataque.

REFERĘNCIAS

ALHARBY, M.; MOORSEL, A. van. Blockchain-based smart contracts: A systematic mapping study. 2017.

ARCARI, J. *Smart Contract Basics—A Legal Contract Perspective*. 2018. <https://medium.com/fordham-business-law-association/smart-contract-basics-a-legal-contract-perspective-8da04b022973>. Acessado em 09/04/2019.

BISHOP, M. What is computer security. *IEEE Security Privacy*, v. 1, p. 67–69, 2003.

BUTERIN, V. *A Next Generation Smart Contract Decentralized Application Platform*. 2014. <https://www.weusecoins.com/assets/pdf/library/Ethereum$_{w}hite_{p}aper-a_{n}ext_{g}eneration_{s}mart_{c}ontract_{a}nd_{d}ecentralized_{a}pplication_{p}latform-vitalik-buterin.pdf$>.

ELOVICI, Y.; BEN-SHIMOL, Y.; SHABTAI, A. Per-packet pricing scheme for ip networks. 2003.

FALKNER, M.; DEVETSIKIOTIS, M.; LAMBADARIS, I. An overview of pricing concepts for broadband ip networks. 2000.

International Telecommunications Union. *New ITU statistics show more than half the world is now using the Internet*. 2018. <https://news.itu.int/itu-statistics-leaving-no-one-offline/>.

Internet Assigned Numbers Autority. *Internet Control Message Protocol (ICMP) Parameters*. 2018. <http://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml>.

KUROSE, J. F.; ROSS, K. W. *Redes de Computadores e a Internet: Uma Abordagem Top Down*. [S.l.]: Pearson, 2010. 614 p.

MACKIE-MASON, H. V. J. Pricing congestible network resources. 1995.

MEHAR, M. I. et al. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Journal of Cases on Information Technology*, v. 21, 2019.

MONTEZANO, P. R. Sistema digital para notificações baseado em blockchain. 2018.

NAKAMURA, E. T.; GEUS, P. L. de. *Segurança de Redes em Ambientes Corporativos*. [S.l.]: Novatec Editora, 2017. 481 p.

PETERSSON, D. *How Smart Contracts Started And Where They Are Heading*. 2018. <https://www.forbes.com/sites/davidpetersson/2018/10/24/how-smart-contracts-started-and-where-they-are-heading/5a947fd637b6>. Acessado em 12/07/2019.

POPEJOY, S. Confidentiality in private blockchain. 2016.

POSTEL, J. *Internet Control Message Protocol*. [S.l.], September 1981. http://www.rfc-editor.org/rfc/rfc792.txt. <http://www.rfc-editor.org/rfc/rfc792.txt>.

POSTEL, J. *Internet Protocol*. [S.l.], September 1981. http://www.rfc-editor.org/rfc/rfc791.txt. <http://www.rfc-editor.org/rfc/rfc791.txt>.

RASKIN, M. The law and legality of smart contracts. 2017.

REICHL, P.; HAMMER, F. Charging for quality-of-experience: A new paradigm for pricing ip-based service. 2006.

ROBERTS, J. W. Internet traffic, qos, and pricing. 2004.

SCHNEIER, B. *Applied Cryptography - Protocols, Algorithms and Source code in C*. [S.l.]: Wiley, 1996. 758 p.

SILVA, G. M. da. *Segurança em Sistemas Linux*. [S.l.]: Editora Ciência Moderna, 2008. 222 p.

STINSON, D. R. *Cryptography: Theory and Practice*. [S.l.]: Chapman and Hall/CRC; 3 edition, 2006. 616 p.

SZABO, N. Smart contracts: Building blocks for digital markets. 1996.

TANENBAUM, A. S. *Redes de Computadores*. [S.l.]: Elsevier, 2003. 945 p.

WANG, S. et al. An overview of smart contract: Architecture, applications, and future trends. 2018.

APÊNDICE A – Artigo

Filtro de pacotes com auxílio de Smart Contracts

Gustavo Garcia Gava

[1]Departamento de Informática e Estatística - Universidade Federal de Santa Catarina

Resumo. *Este trabalho apresenta a proposta de um modelo de filtro de pacotes que utiliza um ou mais smart contracts para a realização do veredicto. Um filtro de pacotes é um tipo de firewall que funciona nas camadas de Rede e Transporte do modelo TCP/IP, atuando de forma independente às aplicações. Suas regras são estáticas e utilizam informações dos cabeçalhos de cada pacote para verificar a aderência à cada regra, portanto, questões que vão além destas informações devem ser tratadas na camada de aplicação. Ao acoplar um smart contract ao filtro de pacotes, desejamos dar a capacidade ao usuário de realizar acordos sobre o acesso à sua rede ou computador independentemente de suas aplicações, expandindo a capacidade do filtro padrão. Portanto, apresentamos um modelo de filtro dividido em três módulos, de modo a permitir total controle e personalização para o usuário. Mostramos também a implementação de um protótipo funcional capaz de realizar os objetivos propostos, demonstrando o funcionamento do modelo e seus resultados frente a um filtro de pacotes padrão.*

1. Introdução

A Internet está se tornando cada vez mais presente na vida da população. Hoje, estima-se que aproximadamente metade da população mundial tenha acesso à internet, número que chega a 81% em países desenvolvidos segundo a *International Telecommunications Union*(2018). Além disso, para as pessoas que possuem acesso, a internet vem tomando espaço em diferentes aspectos da rotina humana, seja para entretenimento, trabalho, comunicação, questões burocráticas, armazenamento de dados, etc. Portanto, quanto mais a internet se torna presente em nossa rotina, maior se torna a importância da segurança em nossas redes, questão presente em diferentes formas e níveis em toda a comunicação pela rede, seja pela privacidade dos nossos dados, garantia da qualidade de serviço ou até mesmo para nos livrarmos de empecilhos como o *spam*.

Nesse contexto, adotamos algumas ferramentas e práticas em conjunto para proteger nossos dados e comunicações. Aqui citamos os protocolos de comunicação com autenticação, criptografia, *softwares* antivírus e o próprio design seguro de cada aplicação[Bishop 2003]. Além destes citados, também utilizamos os *firewalls*, que são como barreiras posicionadas entre duas redes, servindo para controlar o que entra e sai da rede que protege.

No entanto, os *firewalls* a nível de rede, os filtros de pacote, não possuem conhecimento sobre o conteúdo das mensagens, eles só são capazes de enxergar os aspectos técnicos da conexão, mas não analisar o contexto desta. Assim, só temos um conjunto limitado de opções para decidir que dados podem entrar e sair. Então, se o administrador de uma rede desejar verificar algo mais, por exemplo um pagamento, para aceitar uma mensagem ou iniciar uma conexão? A princípio ele recorre às suas aplicações para verificar essa informação.

Como exemplo, vamos imaginar um servidor que presta um serviço pago, em que o cliente deve efetuar um pagamento para ter acesso ao serviço. Quando o servidor recebe uma requisição de serviço, esta passa pelo filtro de pacotes independentemente se o serviço foi pago ou não, e então essa avaliação é feita a nível de aplicação. Para cada diferente serviço adicionado para funcionar nesse servidor, cada uma dessas aplicações vão necessariamente ter que realizar essa checagem de pagamento.

Portanto, nesse trabalho propomos um modelo de filtro de pacotes que permita o usuário expandir a capacidade de decisão de seu filtro. No modelo proposto, a decisão de entrada e saída de pacotes é realizada por um *smart contract* escrito pelo administrador da rede. Ao utilizar um *smart contract* o usuário é capaz de definir quaisquer acordos que desejar para reger o acesso à sua rede ou computador, como por exemplo incluir a necessidade de um pagamento às suas regras. Assim, qualquer pessoa que desejar se comunicar com esse usuário deve cumprir as regras de acordo definidas pelo *smart contract*. Deste modo, possibilitamos a criação de uma conexão regida por um contrato, que estabelece regras e custos definidas pelo contratado.

O usuário pode desejar utilizar este modelo para diferentes tarefas, como a cobrança de serviços(independente de quais sejam as aplicações) ou recebimento de mensagens, como email. Seguindo no exemplo de um servidor, mas agora pensando em um serviço a princípio gratuito, mas limitado por uso, em que os usuários podem requisitar o serviço gratuitamente, desde que sua demanda não ultrapasse um dado limite. A partir do momento em que um usuário passa a demandar além do limite, seus pacotes podem ser direcionados à regra de consulta ao *smart contract*, que demandará um pagamento pelo uso. Com o *smart contract* é possível aplicar diferentes modelos de precificação para as requisições. Como citado, uma abordagem semelhante pode ser tomada para o recebimento de mensagens, com o exemplo de emails, aplicando um modelo ”me pague para lhe ouvir”.

Podemos citar também a possibilidade de vários usuários utilizarem o mesmo *smart contract*, criando assim uma sub-rede regida por contrato entre eles, em que somente os pagantes podem entrar e se comunicar nessa rede. As mensagens enviadas por membros dessa rede ainda seriam acessíveis a membros externos, mas diferentes formas de tornar essas mensagens privadas podem ser aplicadas, mas que não são o escopo deste trabalho.

2. Conceitos

2.1. Smart Contracts

Smart contract é uma forma de executar e validar um contrato digitalmente, por meio de um protocolo de transação. Um contrato é um conjunto de acordos entre duas ou mais partes mutuamente interessadas, que ditam direitos e obrigações das partes envolvidas. Um contrato possui diversos usos, como compra, venda e doação de bens, reger as regras sob uma prestação de serviço ou até mesmo reger a relação entre pessoas. Os contratos geralmente são validados pela lei, que garante sua execução e a aplicação de suas regras previstas em caso de quebra contratual.

O termo *smart contract* foi proposto por Nick Szabo [Szabo 1996] como uma forma de expandir a ideia e funcionamento dos contratos convencionais. Sua proposta

é por as clausulas de um contrato em *software* ou *hardware*, automatizando o processo de execução do contrato, e tornando-o seguro computacionalmente, de forma que seja custoso(ou impossível) para uma das partes quebrar o acordo. Deste modo, ele deve ser capaz de resistir tanto quebras não intencionais, como também quebras intencionais, de alguém que possua recursos e conhecimento para buscar alguma falha no protocolo.

Smart contracts funcionam por meio de respostas ao ambiente em que estão inseridos. Isso significa que quando uma situação prevista acontece, a sua programação emite uma resposta. O exemplo mais simples para isso é um pagamento, que quando realizado, pode gerar diferentes respostas do contrato(dependendo do valor, horário, ou qualquer outra especificação feita). Um evento que causa uma mudança no estado do contrato pode ser chamado de *gatilho*.

A princípio, podemos enxergar a ideia básica de um *smart contract* em qualquer sistema automático de vendas, como uma máquina de refrigerantes ou um sistema de compra online, por exemplo. No entanto, tais sistemas geralmente não englobam questões contratuais, obrigações e direitos das partes envolvidas.

2.2. Filtros de Pacotes

2.3. Filtros de Pacote

Os filtros de pacote verificam cada pacote e tomam sua decisão com base nas informações presentes nos cabeçalhos. Mais precisamente, o filtro funciona nas **camadas de Rede e de Transporte** do protocolo TCP/IP, levando em conta **endereços de origem e destino**, **protocolo de transporte**(TCP, UDP, ICMP, etc.), **portas de origem e destino**, além de outras informações específicas dos cabeçalhos dos diferentes protocolos de transporte[Nakamura and de Geus 2017]. Esse tipo de *firewall* também é chamado de *static packet filtering*, já que suas decisões são estáticas e não se adaptam sem interferência humana.

O filtro de pacotes pode ser configurado de diferentes maneiras dependendo da necessidade do administrador da rede. Por exemplo, uma companhia pode desejar que os computadores de sua rede não tenham acesso a *Voice over Internet Protocol* - VoIP e outros serviços semelhantes, o filtro pode ser configurado para rejeitar todo pacote UDP que não seja de DNS. Em outro caso, se o administrador desejar que não haja contato HTTP com a *web*, ele pode bloquear todo pacote TCP na porta 80. É possível também que seja indesejável receber *ping* na rede, portanto pode ser bloqueado pacotes ICMP do tipo *Echo*. Enfim, utilizando o filtro de pacotes é possível criar diferentes tipos de regras para diferentes motivos, baseando-se sempre na combinação de informações do cabeçalho dos pacotes, mas sem levar em consideração o conteúdo do pacote.

As regras do filtro são mantidas em uma lista(ou tabela), de modo que cada pacote tem seus dados do cabeçalho checados na lista, de forma sequencial. Assim, a primeira regra que engloba as informações do pacote é aplicada. A Tab.1 demonstra uma simplificação de uma lista de regras para os dois primeiros exemplos indicados anteriormente. Todo pacote que passar pelo filtro será comparado sequencialmente pelas quatro regras, e o filtro aplicará a ação da primeira regra em que o pacote se encaixar. Nota-se que a sequência das regras é de suma importância, pois se a ultima regra estivesse no topo, todo pacote seria permitido e as demais regras jamais seriam aplicadas.

Tabela 1. Exemplo de lista de regras do filtro de pacotes

IP:Porta Destino	Protocolo	Ação
IP rede:53	UDP	Permitir
IP rede:Todas	UDP	Negar
IP rede:80	TCP	Negar
IP rede:Todas	Todos	Permitir

Fonte: Autoria própria.

3. Arquitetura

O modelo de filtro é dividido em três módulos para permitir maior adaptação ao usuário. Primeiramente, criamos um módulo fixo, que é a base do funcionamento do sistema e é independente das necessidades do usuário, portanto, não é necessário que o usuário configure-o ou modifique sua implementação. Este é o módulo responsável por lidar diretamente com os pacote capturados e obter as informações de seus cabeçalhos, comunicar o *kernel* sobre a decisão tomada pelo contrato, e alertar o remetente em caso de erro. Os outros dois módulos seguintes seriam implementados pelo usuário conforme seus desejos e necessidades, mas que devem seguir um padrão determinado para que a comunicação com o módulo fixo seja possível. No caso, o segundo módulo é uma aplicação local responsável por intermediar a comunicação do primeiro módulo com o terceiro, o *smart contract* do usuário.

A Fig.1 mostra a sequência de chamadas entre os módulos para que seja feito o veredicto. O Módulo 1(chamaremos de Interceptador) obtém as informações do pacote, que então as envia para o Módulo 2 solicitando qual deve ser o veredicto. O Módulo 2(Avaliador), implementado pelo usuário, é capaz de se comunicar com o *smart contract* empregado e redireciona a solicitação para o contrato(Módulo 3). Com posse das informações do pacote, o contrato é capaz de decidir se ele deve ser aceito ou não, e então o caminho inverso é feito com a decisão. O primeiro módulo então comunica o *kernel* se o pacote deve passar ou não.

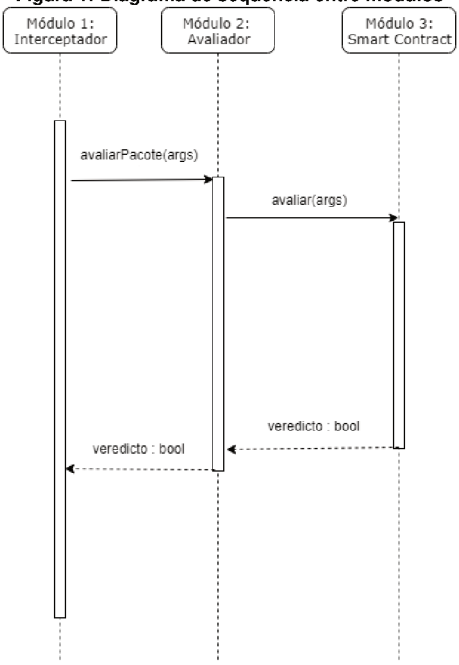
3.1. Módulo 1 - Interceptador de Pacotes

Como dito, este módulo é a base para a funcionamento do nosso modelo e não seria modificado por diferentes usuários, a não ser configurar que pacotes serão interceptados. Ele é responsável por interceptar os pacotes presentes na fila do filtro de pacotes padrão, delegar a decisão aos módulos do usuário, informar o *kernel* a decisão tomada e emitir um alerta ao remetente em caso de negação.

3.1.1. Interceptação dos Pacotes

Utilizando um filtro de pacotes padrão(no caso do Linux, o *iptables*), o usuário deve definir o escopo do nosso filtro proposto. Para isso, ele deve definir uma ou mais regras do seu filtro de pacotes para redirecionar os pacotes que estarão sujeitos à decisão do seu contrato. Portanto, nosso filtro pode ser visto como um *target*(ação, veredicto) do filtro padrão. Para exemplificar, digamos que o usuário deseja as seguintes regras para os pacotes entrantes em sua rede: Todo pacote UDP em porta baixa deve ser aceito, mas os

Figura 1. Diagrama de sequência entre módulos



Fonte: Autoria própria.

em portas altas devem ser avaliadas pelo contrato; e todo pacote TCP devem ser negados, menos os na porta 80(*web*), que devem ser aceitos, e os na porta 3000, que devem ser avaliados no contrato. A Tab.2 indica como seria o conjunto de regras desse usuário.

Tabela 2. Exemplo de filtro de pacotes com *target* contrato.

Protocolo	Porta Destino	Ação
UDP	Porta Baixa	Permitir
UDP	Porta Alta	Contrato
TCP	80	Permitir
TCP	3000	Contrato
TCP	Qualquer	Negar

Fonte: Autoria Própria

Portanto, a primeira função deste módulo é capturar os pacotes filtrados que possuírem como ação definida a avaliação do contrato. Feito isso, o módulo tem acesso total ao pacote e é responsável por emitir a decisão sobre sua entrada. No entanto, nossa proposta é delegar essa decisão ao contrato, então obtemos as informações dos cabeçalhos do pacote e as enviamos ao contrato esperando por uma decisão, no seguinte modelo:

Algoritmo 1: Veredicto com chamada de contrato

```
Entrada: Pacote
Saída: Veredicto
início
    obter cabeçalhos;
    if avaliador.verifique(endereçoDestino, endereçoFonte, portaDestino,
    portaFonte, protocolo) then
        aceite;
    else
        rejeite;
    end
fim
```

Nota-se que aqui criamos uma chamada padronizada para o módulo seguinte, que necessariamente deve possuir uma resposta *booleana* e receber os parâmetros indicados. Isso é importante pois a função *verifique()* deve ser implementada pelo usuário no avaliador, e esta deve ser feita utilizando o mesmo padrão.

3.1.2. Veredicto e Alerta ao Remetente

Como visto, após receber a resposta do contrato, este módulo emite o veredicto final. Caso aceito, o pacote passa normalmente pelo filtro e é entregue ao seu destino(ou enviado à uma próxima etapa de *firewall*). Se for rejeitado, o pacote é descartado. No entanto, nossa proposta é permitir incluir um fator econômico na decisão, portanto no caso de um serviço pago, por exemplo, não é interessante para o usuário que um cliente desavisado não possua conhecimento do contrato que deve ser pago. Por isso é importante indicar o motivo da rejeição do pacote.

Com tal objetivo em vista, após o descarte do pacote, enviamos um pacote ICMP ao cliente, indicando o pacote rejeitado e como o cliente pode obter acesso ao *smart contract* relacionado. Para obter a informação do contrato, definimos uma outra função padronizada em que o avaliador é chamado para retornar uma *string* contendo a mensagem sobre o contrato indicada pelo usuário.

3.2. Módulo 2 - Avaliador de Pacotes

Apesar de não ser aqui que realmente o pacote é avaliado, chamamos esse módulo de avaliador pois é assim que ele é visto pelo primeiro módulo. É neste módulo que a função de verificação chamada no módulo anterior deve ser implementada. Além disso, este módulo deve garantir a comunicação com o *smart contract* associado. Portanto, este módulo trabalha em conjunto com o contrato, pois é o canal de comunicação entre ele e o interceptador de pacotes, sendo também responsável por decidir que funções do contrato(ou qual contrato) será aplicado ao pacote, além de ser também capaz de dar alguns veredictos, se assim desejado.

3.2.1. Comunicação com o Smart Contract

Uma das principais ideias dessa proposta é que ela funcione para qualquer *smart contract*, independente de tecnologia ou plataforma. Portanto, neste modelo o interceptador não possui nenhum contato com o contrato e deixamos essa função para este módulo, que deve se adaptar conforme for a tecnologia utilizada para o *smart contract*, além de levar em consideração que o *smart contract* pode estar sendo executado localmente ou em outro local(uma rede ou outro computador).

3.2.2. Padronização de chamadas

A principal função deste módulo é responder às chamadas efetuadas pelo primeiro módulo. Portanto, as funções chamadas demonstradas nas Figuras 9 e 10 devem ser implementadas possuindo as mesmas assinaturas. Assim, garantimos que, para diferentes usuários, diferentes avaliadores possam ser criados sem a necessidade de alterar o interceptador. A justificativa para isso é dar transparência ao usuário e facilitar o uso deste modelo, diminuindo o nível de complexidade com que o usuário deve lidar. Por exemplo, neste etapa o usuário já possui acesso aos dados do pacote de forma simples, e o veredicto é dado somente com um valor *boolean*.

Nota-se que além destes dois padrões definidos, o usuário possui total liberdade para tratar os dados como desejar ou criar novas funções, mas que no entanto só são executadas a partir das duas funções citadas, pois há uma relação mestre-escravo entre o interceptador e o analisador. Aqui o usuário pode definir quais informações serão passadas ao contrato, analisar qual clausula(função) do contrato será utilizada para dar o veredicto ao pacote. Também é possível que o mesmo avaliador possua mais de um contrato associado, que podem ser consultados em casos diferentes. Tanto a decisão de função quanto qual *smart contract* seria consultado é transparente para o primeiro módulo, sendo que essa decisão seria realizada por regras escritas por quem configurou o avaliador. Essas regras levariam em conta também as informações de cada pacote, e por meio delas, decidir onde será realizada a decisão.

Além disso, este módulo também pode ser usado para otimizar a consulta, por exemplo, com um funcionamento semelhante ao de um filtro com base em estados, onde o contrato só seria consultado em alguns momentos, e o analisador daria o veredicto para a maioria dos pacotes, de modo que o *smart contract* seria consultado somente no início de uma conexão. Todas essas decisões partem de como e para que fins o usuário deseja utilizar este filtro.

3.3. Módulo 3 - Smart Contract

Por fim, temos o último módulo, o *smart contract*, que é onde a decisão do filtro realmente ocorre. Assim, no contexto da arquitetura, sua função é primariamente dar o veredicto sobre o pacote quando consultado pelo módulo avaliador. Sob a questão de implementação não propomos uma maneira específica de realizar essa tarefa, já que ela depende de como o usuário determina a relação do contrato com o avaliador, podendo, por exemplo, haver uma ou mais funções de verificação contendo diferentes assinaturas.

No entanto, a função do contrato no modelo vai além do fluxo que apresentamos nessa seção, porque é nele que é realizado o acordo entre as partes envolvidas na comunicação. Em termos simples, usando como exemplo um serviço prestado na rede protegida por nosso filtro, é utilizando o *smart contract* que o usuário desse serviço paga ao usuário do filtro(o servidor) por sua execução. Então, deve haver pelo menos uma função que permita esse pagamento, de modo que haja uma maneira para alguém que deseja se comunicar com a rede poder se "inscrever"e ter seus pacotes aceitos. Assim, o *smart contract* armazena os usuários que terão seus pacotes aceitos, e quando consultado, é capaz de responder ao avaliador. Podemos entender as funções de pagamento como **clausulas do contrato**, e também como **regras do filtro**, condições que o contratante deve garantir para que o acordo seja feito e seus pacotes passem pelo filtro.

É possível criar diferentes regras para o contrato, de forma semelhante aos filtros de pacotes tradicionais, utilizando diferentes características dos pacotes, como portas e protocolos. Essas características podem ser unidas a um preço, adicionando assim o fator econômico que desejamos na proposta. Deste modo, quando um pacote chega em nosso filtro, ele além de ter que condizer com umas das regras de aceitação do filtro, ele deve possuir um status positivo quanto a um pagamento.

Tabela 3. Exemplo de regras incluindo preço.

Protocolo	Porta Destino	Preço
UDP	Porta Alta	10
TCP	3000	20

Fonte: Autoria Própria

Na Tab.3 apresentamos um pequeno exemplo em que o contrato possuiria duas regras diferentes. O usuário que deseja se comunicar por TCP, utilizando a porta 3000 do destino deve pagar 20 unidades. Quando consultado sobre um pacote pelo avaliador, o contrato checa o protocolo, a porta destino, e verifica internamente se o preço correspondente foi pago. Em caso positivo, o contrato retornaria aceitação, já em caso negativo, rejeição. Observamos aqui que o *smart contract* emite decisão somente para os pacotes que condizem exatamente com essas regras(UDP em porta alta e TCP em porta 3000), e não haveria qualquer verificação de pagamento em qualquer pacote que não possua alguma dessas características. Como lidar com essas situações seriam decisão do administrador do filtro, podendo ser um veredicto padrão(aceitar todos ou rejeitar todos), ou garantir por meio da configuração de suas regras tradicionais de filtro(Módulo 1) e no Módulo 2 que somente pacotes que fazem parte do escopo de decisão do contrato sejam verificados por ele.

Nesse exemplo apresentamos o preço como um valor arbitrário, pois ele pode ser feito de diferentes maneiras. O preço pode ser por pacote, cenário em que o contratante compraria um número de pacotes que deseja enviar, como também o preço pode indicar uma unidade de tempo, em que o contratante compraria uma janela de tempo para efetuar sua transmissão, ou até mesmo indicar o preço por uma conexão TCP. Além disso, o preço pode ser variável, já que o usuário pode utilizar o uso atual de sua banda, o horário do dia, ou mesmo o histórico do contratante para calcular a base de seu preço.

Por fim, é utilizando a capacidade de adaptação dos *smart contracts* a diferentes

problemas e demandas é que conseguimos expandir a capacidade do filtro de pacotes tradicional para incluir um fator econômico à sua decisão, de forma independente a qualquer aplicação.

4. Desenvolvimento

Neste seção apresentaremos as implementações realizadas em função de nossa proposta. Primeiramente, veremos como foi desenvolvido o Módulo 1 para executar em um sistema Linux e realizando todas as suas tarefas propostas. Por fim, iremos apresentar um exemplo dos Módulos 2 e 3, utilizados para testar o modelo como um todo. Definimos que o Módulo 2 seria implementado em Python devido à facilidade de comunicar com diferentes *blockchains*, e escolhemos a *Ethereum* como ferramenta para desenvolver e executar nosso *smart contract*.

Decidimos por utilizar as seguintes informações: **endereço de origem**, **endereço de destino**, **porta de origem**, **porta de destino**, **protocolo de transporte**. Os protocolos suportados são UDP e TCP, qualquer pacote com outro protocolo de transporte quer for avaliado por nosso filtro é automaticamente aceito.

4.1. Implementação do Módulo 1 - Interceptador de Pacotes

O protótipo do interceptador de pacotes foi desenvolvido para ser executado em um sistema Linux, pois utilizamos o *software* de filtragem Netfilter. O módulo consiste de uma aplicação em espaço de usuário implementada em C que se comunica com o filtro de pacotes do *kernel* para informar as decisões. Nesta seção explicaremos como isso ocorre, como é realizado a comunicação com o segundo módulo e como é realizado o envio do alerta ao remetente.

4.1.1. Configuração Inicial: iptables e NFQUEUE

No sistema Linux, a forma padrão de configurar o filtro de pacotes é utilizando a aplicação iptables. Portanto, é com ela que criamos as regras que redirecionam os pacotes desejados ao filtro com *smart contract*, utilizando o veredicto NFQUEUE, presente no iptables. Com este *target*, os pacotes são colocados em uma fila para decisão, que é dada por um programa que possua acesso à fila. Um exemplo de regra seria iptables -A INPUT [ESPECIFICAÇÕES] -j NFQUEUE –queue-num 0 onde `--queue-num 0` indica a fila para qual o pacote será direcionado, e as especificações seriam as características dos pacotes, como IP fonte e portas.

4.1.2. Interceptador

O primeiro pré requisito de nossa aplicação é que ela seja capaz de acessar a fila para qual os pacotes são enviados. A principal forma de se fazer isso é com a *lib* `libnetfilter_queue`, que realiza a comunicação entre o espaço de *kernel* e de usuário para este caso, acessando cada pacote e entregando o veredicto dado pela aplicação ao *kernel*. O protocolo usado pela *lib* para comunicação *kernel-usuário* é chamado *nfnetlink*, que utiliza um *socket* para a troca de mensagens sobre o pacote e seu veredicto.

Para cada novo pacote redirecionado à fila, uma função de *callback* é chamada em nossa aplicação esperando por um veredicto. Nela, temos acesso ao pacote e podemos extrair seu *payload*(utilizando a função `nfq_get_payload` definida na `libnetfilter_queue`) para ter acesso às suas informações. Inicialmente, obtemos acesso ao seu cabeçalho da camada de rede, e com isso os endereços de IP fonte e destino:

```
struct iphdr *ipHeader;
unsigned char *payloadData;
nfq_get_payload(packet, &payloadData);
ipHeader = (struct iphdr *)payloadData;

uint32_t sourceIP = be32toh(ipHeader->saddr);
uint32_t destIP = be32toh(ipHeader->daddr);
```

Utilizando o cabeçalho de IP também temos acesso à informação de qual é o protocolo de transporte do pacote. Com isso também podemos ter acesso ao cabeçalho da Camada de Transporte referente ao pacote. Para esse trabalho, as informações retiradas do cabeçalho de transporte são somente as portas fonte e destino(presente tanto no TCP quanto UDP), no entanto, futuramente podem ser utilizadas outras informações(flag TCP, por exemplo) para aumentar a capacidade do modelo. Com as portas obtidas, temos todas as informações necessárias para encaminhar a decisão ao avaliador.

4.1.3. Comunicação com o Módulo 2 e Veredicto

Como no caso do nosso protótipo o segundo módulo deve ser feito em Python(para que haja compatibildiade com as chamadas feitas), foi necessário encontrar uma forma de chamar funções de uma aplicação nessa linguagem pela nossa aplicação em C. Para tal, utilizamos a API padrão do Python para comunicação Python/C[1]. Na API, informamos os nomes definidos para o arquivo referente ao segundo módulo e as funções a serem chamadas, onde *pythonModule* é o nome obrigatório do arquivo do módulo 2, e *verify* e *getAddress* são os nomes obrigatórios das duas funções chamadas.

```
pName = PyUnicode_FromString("pythonModule");
pModule = PyImport_Import(pName);
pDict = PyModule_GetDict(pModule);
pVerify = PyDict_GetItemString(pDict, "verify");
pAddress = PyDict_GetItemString(pDict, "getAddress");
```

De volta à função de *callback*, nós chamamos a função de verificação após obter todos os parâmetros necessários. Com a resposta, é dado o veredicto ao *kernel*. Em caso de negação, a função responsável por montar e enviar o pacote ICMP à fonte é executada. Aqui, mostramos o fluxo para pacotes TCP, no entanto ele é praticamento o mesmo para pacotes UDP, por isso o omitimos. Optamos por separar os fluxos para tornar mais simples futuras implementações que tratem de peculiaridades para os diferentes protocolos da Camada de Transporte.

```
if(ipHeader->protocol == IPPROTO_TCP){
    struct tcphdr *tcpHeader = (struct tcphdr *)(payloadData + (
   ipHeader->ihl<<2));
```

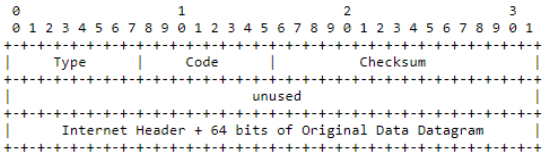
[1]https://docs.python.org/2/c-api/

```
    unsigned int sourcePort = ntohs(tcpHeader->source);
    unsigned int destPort = ntohs(tcpHeader->dest);

    PyObject *argList = Py_BuildValue("llIIs", sourceIP, destIP,
    sourcePort, destPort, "TCP");
    if(PyCallable_Check(pVerify)){
      pValueVerify = PyObject_CallObject(pVerify, argList);
    } else {
      PyErr_Print();
    }

    Py_DECREF(argList);

    if(PyObject_IsTrue(pValueVerify) == 1){
      return nfq_set_verdict(qh, id, NF_ACCEPT, 0, NULL);
    } else {
      set_icmp(ipStringSource, payloadData);
      return nfq_set_verdict(qh, id, NF_DROP, 0, NULL);
    }
  }
}
```

4.1.4. Formato do Pacote ICMP

Normalmente, um REJECT do iptables descarta o pacote e envia um pacote ICMP de tipo 3 - Destino Inalcançável e código 3 - Porta Inalcançável à fonte. No entanto, além da *lib* `libnetfilter_queue` não suportar o veredicto REJECT, esse tipo de pacote ICMP não é suficiente para nós. Como vemos na Fig.2, o conteúdo do pacote de tipo 3 é composto do cabeçalho da camada de rede e mais 64 bits do conteúdo do pacote original(o que causou o erro), usados para identificar qual sobre qual pacote o ICMP se refere. Os 8 primeiros *bytes* do cabeçalho de transporte são suficientes para identificar as portas de origem e destino do pacote, assim como no caso de uma transmissão TCP, obter o número de sequencia do pacote. Essas informações são suficientes para o remetente identificar o pacote. Segundo o RFC792 [Postel 1981], não há espaço para informações adicionais, o que impossibilitaria a inclusão de informação sobre o *smart contract*, como desejamos.

Figura 2. Mensagem ICMP - Tipo 3

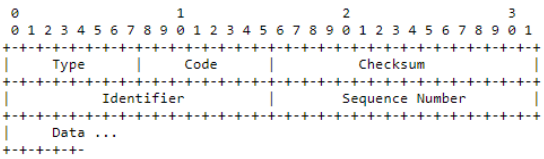


Fonte: RFC792.

Por outro lado, poderíamos utilizar o pacote de tipo 8 ou 0 (*ping* ou resposta de *ping*), que possuem um campo de informação de tamanho aberto, e assim poderíamos incluir a informação que desejamos, como podemos ver na Fig.3. No entanto, estaríamos extrapolando o propósito desse tipo de pacote, o que pode causar confusão no *host*

que recebê-lo, além de conter informações desnecessárias no cabeçalho como Identificador(*Identifier*) e Número de Sequência(*Sequence Number*).

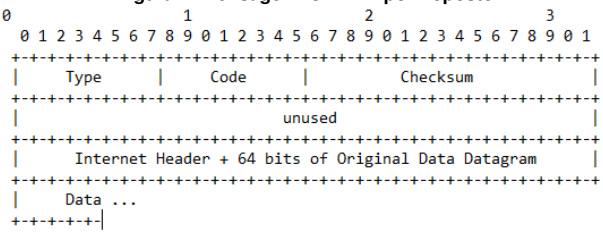
Figura 3. Mensagem ICMP - Tipo 0 e 8



Fonte: RFC792.

Portanto, propomos utilizar um novo tipo de pacote ICMP de forma experimental, usando um dos valores 44 a 252 no campo Tipo que são reservados para implementações futuras [Internet Assigned Numbers Autority 2018]. De forma arbitrária, escolhemos o número 50 para determinar o nosso novo tipo. Utilizaremos um cabeçalho muito semelhante ao de tipo 3, contendo os bits necessários para identificar o pacote original, mas também contendo um espaço aberto para informações adicionais(*Data*). Assim, garantimos que o *host* que receber nosso aviso poderá identificar o pacote que foi rejeitado e também receber a informação de como acessar o *smart contract* necessário para concluir a comunicação. A Fig.4 ilustra o novo pacote. Para o campo de código, usaremos os mesmos definidos no tipo 3.

Figura 4. Mensagem ICMP - Tipo Proposto



Fonte: Autoria Própria.

4.1.5. Construção do Pacote ICMP

A função `set_icmp` é responsável por montar e enviar o pacote ICMP que definimos. Nela, montamos o cabeçalho IP que transmitirá nosso pacote, informando o IP destino(obtido pelo pacote original, que causou o erro), e montamos o cabeçalho ICMP, informando o tipo 50, código 3 e calculamos o *checksum*. Além disso, copiamos o cabeçalho IP e os primeiros 8 *bytes* do cabeçalho de transporte do pacote original. Então, mais uma vez utilizando a API Python/C, fazemos uma chamada para o Módulo 2 requisitando o endereço do *smart contract*, que é copiado para o campo Data do nosso pacote. Com o pacote pronto, ele é enviado utilizando um *socket*.

Considerando, que o tamanho de um cabeçalho IP varia entre 20 e 60 *bytes*, e um endereço de contrato Ethereum pode ser representado em 20 *bytes*, o maior tamanho que nosso pacote ICMP pode assumir, contando com seu cabeçalho ICMP, é de 96 *bytes*, já o menor seria 56 *bytes*(cabeçalho ICMP + cabeçalho IP e 8 bytes da mensagem original + endereço do contrato). No entanto, este tamanho pode variar para outras plataformas de *smart contract* ou caso o usuário deseje enviar uma mensagem diferente.

4.2. Módulos 2 e 3 - Avaliador e Smart Contract

Apresentaremos aqui um exemplo dos dois módulos implementados pelo usuário, seguindo as regras definidas no projeto e na implementação do Módulo 1. Como utilizamos a API C/Python para realizar as chamadas no Módulo 1, o avaliador deve ser obrigatoriamente uma aplicação em Python. Quanto ao *smart contract*, utilizaremos como exemplo um *smart contract* Ethereum, por ser uma das formas mais utilizadas atualmente para trabalhar com *smart contracts*, possuindo então ampla documentação, e também por familiaridade com a linguagem Solidity, utilizada para implementar contratos Ethereum.

4.2.1. Implementação do Smart Contract

Para exemplificar, criamos o *smart contract* mais básico para o modelo, em que sua única função é verificar se existe um pagamento válido referente a uma informação que recebe. Um pagamento válido significa uma janela de tempo em que os pacotes serão aceitos, sendo essa janela definida pela quantia paga pelo contratante. Definimos também a unidade de tempo como um bloco da *blockchain* Ethereum. O tempo de bloco é o tempo médio que leva para cada novo bloco aparecer na *blockchain*, que costuma ser constante. Durante a execução deste trabalho, o tempo de bloco aproximado da rede Ethereum era de 13 segundos[2]. Portanto, para cada unidade monetária(wei, a menor unidade do Ethereum) que o usuário paga, ele compra um bloco de tempo de transmissão. Utilizamos a estrutura *Registry* para definir a janela de tempo, onde é registrado o número do bloco que inicia a janela e o bloco que a encerra.

```
1    struct Registry{
2        uint256 endTime;
3        uint256 startTime;
4        bool registred;
5    }
```

No entanto ainda é necessário identificar o pacote e vinculá-lo com o registro. Para isso, utilizamos um *hash* das informações retiradas do cabeçalho de cada pacote(endereços, portas e protocolo), de modo que todo pacote com as mesmas credenciais geram o mesmo *hash*. Utilizando esse *hash*, podemos vincular essas credenciais a cada registro com um *hash map*.

```
1    mapping(bytes32 => Registry) public packet;
```

Para realizar um pagamento, o contratante informa o *hash* das credenciais dos pacotes que deseja transmitir, um número usado para calcular qual será o número do bloco inicial da janela, e o valor da transação. A função `pay` é usada para o pagamento,

[2] ¡etherscan.io/chart/blocktime¿. Acesso em 17/05/2019.

criando e vinculando um registro. Nela calculamos o tempo permitido de transmissão com base no valor do pagamento(representado por `msg.value`).

```
    function pay(bytes32 hash, uint256 blocktime) payable public {
        require(packet[hash].registred == false);
        packet[hash].registred = true;
        packet[hash].startTime = blocktime + block.number;
        packet[hash].endTime = packet[hash].startTime + msg.value;
        owner.transfer(msg.value);
    }
```

Por fim, temos a função `verify`, que é a função chamada pelo avaliador para dar o veredicto sobre um pacote. Sempre que chamada, recebendo um *hash* como parâmetro, é verificado se há um registro associado ao *hash* recebido e se o bloco mais recente da *blockchain* está dentro da janela de tempo. Em caso positivo, é retornado um valor verdadeiro para o veredicto do avaliador, já em caso negativo retorna-se um valor falso, pois ou nunca houve um pagamento para essas credenciais, ou não estamos na janela de tempo permitida(ou ela ainda não começou, ou já se expirou). É importante notar que a palavra *view* na assinatura da função faz com que não haja custo para a execução desta função, pois ela é somente usada para consulta. Isso possibilita que tal função seja chamada várias vezes pelo avaliador livremente.

```
        function verify(bytes32 hash) view public returns (bool){
        return (packet[hash].registred
        && block.number >= packet[hash].startTime
        && block.number <= packet[hash].endTime)
    }
```

4.2.2. Deploy do Contrato

Com o contrato pronto nós podemos incluí-lo em uma *blockchain*. A partir desse momento, ele se torna acessível ao público, recebendo um endereço. O endereço é seu identificador na *blockchain* e é necessário para qualquer um que deseje acessá-lo(por exemplo no avaliador, mostraremos a respeito na próxima sub-seção). No entanto, ele se torna imutável e para realizar qualquer modificação é necessário republicá-lo, gerando um outro endereço. Outra informação gerada que precisamos para realizar chamadas do contrato no avaliador é a ABI(*Applicaiton Binary Interface*) do contrato. A ABI funciona como um manual de instruções para nossa aplicação, informando como deve ser realizada as chamadas de função e como as respostas devem ser interpretadas.

4.3. Implementação do Avaliador

Assim como o *smart contract*, implementamos o avaliador mais básico necessário para o modelo, executando somente as sua função obrigatória de permitir a comunicação entre o primeiro e último módulo. Utilizamos a *lib* Web3 para realizar a comunicação com a rede Ethereum, utilizando-a para interagir com as funções do nosso *smart contract*. Com isso, definimos o endereço do nodo *Ethereum* que será consultado, e também incluímos o endereço de nosso *smart contract*.

```
from web3 import Web3, HTTPProvider

```

```
w3 = Web3(HTTPProvider('http://192.168.66.244:8545'))

smartAddress = "0x4ac023e56952099a806b591b8722cc4e49805766"
```

Definimos também as duas funções necessárias para o funcionamento do primeiro módulo. A função `getAddress`, quando chamada só retorna o endereço do contrato(`smartAddress`). Já a função `verify`, que é chamada para todo pacote, é responsável por receber as credenciais dos pacotes e concatená-los para então aplicar a função *hash*. Com o *hash* obtido e preparado, basta chamar a função `verify` do contrato e retornar a resposta ao Módulo 1. Para realizar a chamada, utilizamos o endereço e a ABI do contrato. Observa-se que respeitamos a assinatura de função necessária definida no Módulo 1. Observamos também a tarefa deste módulo de preparar os dados recebidos do primeiro módulo para haver compatibilidade com o contrato, tanto por suas clausulas(identificação pelo hash), quanto por sua tecnologia(implementado sob a Ethereum).

```
def verify(sourceIP, destIP, sourcePort, destPort, proto):

    m = hashlib.sha256()
    srcipBytes = struct.pack(">I", sourceIP)
    dstipBytes = struct.pack(">I", destIP)
    srcportBytes = struct.pack(">I", sourcePort)
    dstportBytes = struct.pack(">I", destPort)
    prtBytes = proto.encode("utf8")

    m.update(srcipBytes + dstipBytes + srcportBytes + dstportBytes +
    prtBytes)
    h = '0x' + m.hexdigest()

    with open('contract.abi', 'r') as abi_definition:
      abi = json.load(abi_definition)
    address = w3.toChecksumAddress(smartAddress)
    contract = w3.eth.contract(address=address, abi=abi)
    return(contract.functions.verify(h).call())
```

4.4. Implantação do Protótipo e Testes

O filtro foi instalado e testado em um computador utilizando o sistema Ubuntu(Linux) como um *firewall* local, sendo ele o *host* destino. O objetivo do experimento foi avaliar a corretude dos veredictos e também avaliar o impacto sobre a eficiência da decisão.

Os testes foram realizados com o envio de pacotes TCP e UDP de um computador fonte pra o computador destino com o filtro instalado. O nodo Ethereum consultado para comunicar com o *smart contract* estava em um terceiro computador na mesma rede local que o computador destino.

Por definição, utilizaremos nessa seção a palavra *destino*(em itálico) para se referir ao computador que utiliza nosso filtro e é o destino das mensagens, e utilizaremos *fonte*(em itálico) para se referir ao computador utilizado para enviar mensagens rumo ao *destino*.

4.4.1. Rede Ethereum

Decidimos por utilizar uma rede privada Ethereum, cuja seus nodos não estão conectados com a rede Ethereum principal, para publicar nosso *smart contract*, pois assim obtemos maior controle sobre o cenário de teste, como velocidade de mineração, quantidade de *tokens*(para simular o valor pago em cada transação) e acesso ao *smart contract*. Utilizamos também um nodo completo, responsável por minerar as transações e executar nosso contrato, sendo este nodo o consultado pela Módulo 2 a respeito do *smart contract*. Utilizamos o método detalhado por Montazano(2018) para criar a rede e definir o nodo.

4.4.2. Teste do Protótipo

Aqui iremos definir um cenário de teste para ilustrar o funcionamento de nosso filtro. Demonstraremos por meio de capturas de imagens a execução do filtro no *destino* e a captura do pacote ICMP emitido à *fonte*.

4.4.3. Cenário de Teste

Utilizamos a aplicação `Socat` para ser nossa aplicação exemplo. O Socat é um programa de linha de comando que permite o usuário criar conexões ponto-a-ponto com configurações próprias. Definimos que o *destino* utilizará a porta 4001 para receber pacotes UDP contendo mensagens de texto. Assim, qualquer computador pode enviar um pacote UDP para a porta 4001 do *destino* com uma mensagem de texto, e a mensagem aparecerá no terminal do *destino*. No lado da *fonte*, utilizaremos a porta 4000 para enviar nossas mensagens, direcionadas ao endereço do *destino*. Com esse cenário, simulamos o uso de uma aplicação qualquer e podemos verificar o funcionamento de nosso filtro. O nodo Ethereum consultado pelo avaliador é um terceiro computador presente na rede, portanto o *smart contract* é executado neste computador. Abaixo, sumarizamos a forma que a *fonte* irá se comunicar:

- IP Destino:150.162.56.139
- IP Fonte: 150.162.56.165
- Porta Destino: 4001
- Porta Fonte: 4000
- Protocolo Usado: UDP

4.4.4. Execução

A princípio, sem nenhuma regra configurada com o iptables, qualquer pacote seria aceito. Com o filtro instalado no computador destino, redirecionamos todos os pacotes recebidos do computador *fonte* ao nosso filtro com o seguinte comando: iptables -A INPUT 150.162.56.165 -j NFQUEUE –queue-num 0 Com isso feito, executamos nossa aplicação, que então fica no aguardo de novos pacotes, estado mostrado na Fig.5. As mensagens mostradas são *health checks* realizados na inicialização do sistema, obtidos do arquivo de exemplo de uso da *lib* `libnetfilter_queue`.

Figura 5. Execução do Filtro - Inicio

Fonte: Autoria Própria.

A partir desse momento, qualquer pacote recebido proveniente do IP Fonte será rejeitado, independente se siga o padrão que definimos ou não. Agora, somente com a liberação no *smart contract* os pacotes desse IP serão aceitos. Ao enviarmos uma mensagem na *fonte*, o filtro indicará no terminal *destino* a rejeição(*Drop*), como mostrado na Fig.6. Podemos ver também o *hash*[3] obtido pelos campos do cabeçalho que são verificados(IP e Porta fonte/destino e protocolo). Esse é o *hash* utilizado para consultar o *smart contract*. É importante notar que, diferente das outras informações mostradas no terminal, o *print* do *hash* é realizado pelo Módulo 2, e não pelo Módulo 1. A mensagem *Packet sent to: [...]* indica o envio da mensagem ICMP. No caso dessa imagem, é ilustrado uma rejeição de um pacote com o padrão definido no nosso cenário.

Figura 6. Execução do Filtro - Negado

Fonte: Autoria Própria.

Para que um pacote igual seja aceito, é necessário realizar uma transação no *smart contract* informando esse mesmo *hash*. Utilizando o Remix[4], abrimos o *smart contract* por meio de seu endereço. Pela interface do Remix, usamos a função `pay` para registrar o *hash* que indica o padrão dos pacotes que definimos para esse cenário. Transferimos 25 *wei* para comprar 25 blocos(como citado, nossa unidade de tempo para esse *smart contract*) e definimos o tempo inicial da janela como o primeiro bloco em que essa transação for publicada(*blocktime 0*).

Com o pagamento realizado e o bloco da transação constando na *blockchain*, os pacotes UDP enviados pela porta 4000 da *fonte*, com destino à porta 4001 do *destino* serão aceitos. Na Fig.7 mostramos o comportamento do filtro ao aceitar o pacote. Todos os pacotes que sigam o padrão definido no cenário são aceitos, desde que a janela de

[3] A função *hash* utilizada foi a sha-256.

[4] Remix é uma IDE para implementação de *smart contracts* em Solidity. Nela podemos carregar contratos já existentes para acessar e executar suas funções.

Figura 7. Execução do Filtro - Aceito

```
labsec@labsec-System-Product-Name:~/Área de Trabalho/Gava/TCC$ sudo ./test.o

Socket file descriptor 3 received
opening library handle
unbinding existing nf_queue handler for AF_INET (if any)
binding nfnetlink_queue as nf_queue handler for AF_INET
binding this socket to queue '0'
setting copy_packet mode
Packed Received
Entering callback
Packet Hash:
0x860e373d5df1516c756dd1c21d53ada8f74b18fb509cf55e0509e05564b43f7f
Accept from:150.162.56.165 to:150.162.56.139 From port 4000 to port 4001
```

Fonte: Autoria Própria.

tempo paga ainda esteja válida. Pacotes que possuam outro padrão(e consequentemente geram um *hash* diferente) continuam sendo rejeitados. Na Fig. 8 mostramos a mensagem enviada(”Mensagem de Teste”) sendo recebida pelo socat executando no *destino*. Após 25 blocos minerados por nosso nodo a *fonte* para de aceitar mais pacotes.

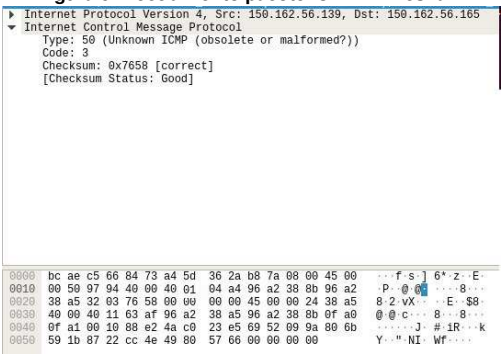
Figura 8. Mensagem recebida na aplicação destino

```
labsec@labsec-System-Product-Name:~$ socat UDP-listen:4001 -
Mensagem de Teste
```

Fonte: Autoria Própria.

4.4.5. Verificando envio de ICMP

Figura 9. Recebimento pacote ICMP - Wireshark



Fonte: Autoria Própria.

Para testar o envio e recebimento do pacote ICMP em caso de negação utilizamos

a ferramenta Wireshark[5] na *fonte*. A Fig.9 mostra a captura de um pacote ICMP emitido pelo *destino* quando um pacote é rejeitado. Na parte de cima, em texto, podemos ver que o Wireshark identifica o pacote como ICMP e indica seu tipo e código(Como o 50, que usamos, não é definido em nenhum RFC o Wireshark é incapaz de identificá-lo, por isso a mensagem "*obsolete or malformed?*"). Em baixo, em notação hexadecimal, é mostrado todo o pacote. As linhas 0000 e 0010 são os bytes do cabeçalho IP do pacote. A partir do terceiro byte da linha 0020 em diante são representados os campos que definimos na seção 5.1.2.2. Nas linhas 0040 e 0050 podemos ver o endereço de nosso *smart contract*(4ac023...5766).

4.5. Teste de Desempenho

4.5.1. Método de medição de desempenho

Para avaliar o desempenho do filtro utilizamos um cenário semelhante ao da seção anterior, mas com uma alteração, definimos o socat na fonte para não só receber uma mensagem mas também enviar uma resposta, simulando um servidor. Deste modo, mediremos o efeito sobre o desempenho pela perspectiva do cliente.

A maneira escolhida para realizar a medição do tempo de requisição e resposta foi a utilização do Wireshark na *fonte*, que captura a mensagem de solicitação e a resposta recebida, marcando-os com um *timestamp* em segundos. Por subtrair o *timestamp* do pacote de solicitação do *timestamp* da resposta obtemos o tempo de resposta do servidor. Para fins de comparação, utilizamos o tempo de resposta em uma caso que nosso filtro não é empregado, e comparamos com os casos de pacotes aceitos por nosso filtro e também de pacotes negados(nos casos de negação, utilizamos a resposta ICMP para obter o *timestamp*). É importante relembrar que o nodo consultado para a execução do *smart contract* não está no mesmo computador que os módulos 1 e 2, portanto é necessário uma consulta pela rede, o que pode gerar latência. Como esse teste está sujeito a outras variáveis obtemos dez amostras para cada caso e calculamos a média simples como resultado final.

4.5.2. Resultados do teste de desempenho

A Tab. 4 mostra os resultados das medições para cada amostra, obtida em segundos. O valor mostrado para cada amostra é o tempo de resposta obtido. A última linha indica a média simples obtida das dez amostras. Para fins de representação neste trabalho, arredondamos os números após seis algarismos à direita. Como podemos ver, utilizando somente o filtro padrão(iptables) o tempo de resposta foi de, em média, 2,6 milissegundos. Já utilizando o nosso protótipo, o tempo de resposta passou para 27 milissegundos. Essa mudança de desempenho pouco afetaria aplicações voltadas ao usuário, como *web pages*, aplicativos de trocas de mensagens e emails. No entanto, aplicações que dependam de eficiência no tempo de resposta ou realizam grandes trocas de pacotes em pouco tempo podem vir a ser afetados, portanto, cabe o usuário do filtro identificar se o uso do *smart contract* é benéfico.

[5]Wireshark é uma aplicação de captura de pacotes, que nos permite observar quaisquer pacotes que passem pelo computador. Com ela, podemos observar os campos desejados nos cabeçalhos, como também analisar o *payload* dos pacotes

Tabela 4. Resultados do experimento de eficiência

	Filtro Padrão	Protótipo - Aceito	Protótipo - Negado
Amostra 1	0,003508	0,038735	0,041807
Amostra 2	0,002804	0,025444	0,050624
Amostra 3	0,003360	0,023083	0,041115
Amostra 4	0,002317	0,025512	0,038362
Amostra 5	0,003050	0,023174	0,048127
Amostra 6	0,003647	0,027236	0,025305
Amostra 7	0,001468	0,026877	0,035811
Amostra 8	0,002718	0,028792	0,037091
Amostra 9	0,001451	0,027475	0,039402
Amostra 10	0,001871	0,026291	0,026965
Média	0,002619	0,027262	0,038461

Fonte: Autoria Própria

Para o mesmo interceptador(Módulo 1) tais resultados estão sujeitos a alterações para diferentes decisões durante a implementação do Módulo 2 e escolha de *smart contract*, como também à fatores como a localização do nodo consultado quando utilizado um *smart contract* em *blockchain*. Como vimos, o nosso avaliador consulta o *smart contract* para todo pacote que recebe e não possui nenhuma tentativa de otimização. Por outro lado, consultamos um nodo dedicado presente na mesma rede que nosso filtro, o que nos fornece um tempo de resposta melhor, já que o efeito da latência é menor.

5. Considerações Finais

Neste trabalho propomos e prototipamos um modelo de filtro de pacotes que utiliza um *smart contract* para tomar suas decisões. O principal objetivo para essa proposta foi incluir um fator econômico, um pagamento por exemplo, ao processo de decisão dos filtros de pacotes tradicionais. O protótipo foi criado para um sistema Linux e apresentou bons resultados, realizando a filtragem corretamente e com um efeito sobre o desempenho tolerável para a maior parte das aplicações. Discutiremos nesta seção os resultados obtidos em relação aos objetivos propostos, faremos uma reflexão a respeito do modelo, e por fim indicaremos trabalhos futuros.

5.1. Resultados Gerais

Todos os objetivos propostos foram cumpridos com sucesso. Primeiramente, definimos um modelo de filtro de pacotes que permita a inclusão de um ou mais *smart contracts* como realizadores de decisão de entrada de pacotes. Com a inclusão do *smart contract* se torna possível utilizar um pagamento como regra adicional no filtro de pacotes. Utilizando o protótipo implementado para testes, o filtro proposto foi capaz de emitir veredictos corretamente com base no escopo de tipo de pacotes definido para ele.

Propusemos também uma forma de comunicar o usuário que teve seu pacote negado por não realizar o pagamento necessário sobre o *smart contract* relacionado ao filtro. Para tal, foi proposto um novo tipo de pacote ICMP capaz de informar qual pacote foi rejeitado e carregar uma mensagem obtida no filtro. O protótipo apresentado também foi capaz de realizar essa tarefa.

Utilizando o protótipo junto um *smart contract* exemplo, realizamos um teste de impacto sobre a eficiência da tomada de decisão. O resultado obtido foi considerado satisfatório, causando um impacto de aproximadamente 25 microssegundos, o que é imperceptível para o usuário humano.

5.2. Discussão do Projeto

O modelo proposto possibilita a expansão das capacidades do filtro de pacote tradicional. Com ele, o usuário pode incluir a condição de um pagamento como mais um tipo regra na configuração de suas regras de filtragem. O principal objetivo por trás dessa ideia é possibilitar a cobrança de pagamento independente de quais aplicações estão envolvidas, já que o filtro de pacotes não depende da Camada de Aplicação.

Ao utilizar o modelo que foi proposto, o usuário ganha em liberdade para definir como serão suas regras, que informações do pacote serão avaliadas, como será feito o pagamento e também o que será levado em conta para calcular o preço. O usuário deve tomar a decisão de utilizar, e como utilizar, este modelo com base na sua necessidade e objetivos. Além disso, mesmo tendo seu funcionamento independente de aplicações, o usuário deve considerar quais aplicações estarão protegidas por trás do filtro, pois isso deve estar relacionado aos objetivos traçados pelo usuário, como também pode haver a interferência no funcionamento da aplicação, principalmente em relação à eficiência.

5.3. Trabalhos Futuros

Este trabalho foi sobretudo propositivo, tratando da proposta de um modelo novo de filtro de pacotes, portanto, surgem novas questões a respeito. Aqui, iremos propor algumas ideias de extensão ao projeto que surgiram durante o desenvolvimento, como também alguns questionamentos que não foram possíveis elucidar neste trabalho.

Primeiramente, o protótipo implementado possui baixa complexidade, utilizando poucas informações dos pacotes e estando limitado somente aos protocolos UDP e TCP. Ao expandir as capacidades do módulo implementado, aumenta-se a capacidade de uso do filtro, o que pode trazer novos casos de uso não explorados neste trabalho. Além disso, o *smart contract* utilizado para o teste é também simples, usado somente para verificar o comportamento do sistema como um todo. É interessante verificar o comportamento do modelo utilizando diferentes *smart contracts* com complexidades diversas, como também implementar *smart contracts* em diferentes plataformas, encorpando a proposta atual.

Somado ao ponto citado, uma validação mais completa a respeito do efeito na eficiência seria de grande contribuição. Como citado no trabalho, o teste realizado utilizou um nodo Ethereum presente na mesma rede que o hospedeiro do filtro, e a localização do nodo consultado pode causar efeitos diferentes na eficiência do modelo. Além disso, diferentes tecnologias de *smart contract* podem possuir resultados diferentes. Como houve impacto sobre a eficiência de decisão, um ensaio sobre a resistência a ataques de negação de serviço também seria proveitoso para levantar a suscetibilidade e resistência do filtro a este tipo de ataque.

Referências

Bishop, M. (2003). What is computer security. *IEEE Security Privacy*, 1:67–69.

International Telecommunications Union (2018). New itu statistics show more than half the world is now using the internet.

Internet Assigned Numbers Autority (2018). Internet control message protocol (icmp) parameters.

Montezano, P. R. (2018). Sistema digital para notificações baseado em blockchain.

Nakamura, E. T. and de Geus, P. L. (2017). *Segurança de Redes em Ambientes Corporativos*. Novatec Editora.

Postel, J. (1981). Internet control message protocol. STD 5, RFC Editor. `http://www.rfc-editor.org/rfc/rfc792.txt`.

Szabo, N. (1996). Smart contracts: Building blocks for digital markets.

ANEXO A – Código

Módulo 1 - Interceptador:

```
1  // COMPILE WITH gcc -Wall -o test.o nfqnl_test.c -
   lnfnetlink -lnetfilter_queue -I/usr/include/python3.6m -
   lpython3.6m -w
2
3  //sudo iptables -A INPUT -s [IP] -j NFQUEUE --queue-num
   0
4
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9  #include <netinet/in.h>
10 #include <linux/types.h>
11 #include <linux/netfilter.h>
12 #include <linux/ip.h>
13 #include <linux/tcp.h>
14 #include <linux/udp.h>
15 #include <endian.h>
16 #include <libnetfilter_queue/libnetfilter_queue.h>
17 #include <linux/icmp.h>
18 #include <arpa/inet.h>
19 #include <netdb.h>
20 #include <time.h>
21
22 #include <python3.6/Python.h>
23
24 PyObject *pName, *pModule, *pDict, *pVerify, *pAddress,*
   pValueAddress, *pValueVerify;
25
26 #define PING_PKT_S 60
27 #define PORT_NO 0
28
29 int sockfd;
30
31 struct ping_pkt
32 {
33     struct icmphdr hdr;
34     char msg[PING_PKT_S-sizeof(struct icmphdr)];
35 };
36
37 static u_int32_t print_pkt (struct nfq_data *tb)
38 {
39
40   int id = 0;
41   struct nfqnl_msg_packet_hdr *ph;
42   struct nfqnl_msg_packet_hw *hwph;
43   u_int32_t mark,ifi;
44   int ret;
45   char *data;
46
```

```
47        ph = nfq_get_msg_packet_hdr(tb);
48        if (ph) {
49          id = ntohl(ph->packet_id);
50        }
51        return id;
52      }
53
54      unsigned short checksum(void *b, int len)
55      {     unsigned short *buf = b;
56          unsigned int sum=0;
57          unsigned short result;
58
59          for ( sum = 0; len > 1; len -= 2 )
60              sum += *buf++;
61          if ( len == 1 )
62              sum += *(unsigned char*)buf;
63          sum = (sum >> 16) + (sum & 0xFFFF);
64          sum += (sum >> 16);
65          result = ~sum;
66          return result;
67      }
68
69      char *dns_lookup(char *addr_host, struct sockaddr_in *
        addr_con)
70      {
71          struct hostent *host_entity;
72          char *ip=(char*)malloc(NI_MAXHOST*sizeof(char));
73          int i;
74
75          if ((host_entity = gethostbyname(addr_host)) == NULL
        )
76          {
77              // No ip found for hostname
78              return NULL;
79          }
80          strcpy(ip, inet_ntoa(*(struct in_addr *)
81                                host_entity->h_addr));
82
83          (*addr_con).sin_family = host_entity->h_addrtype;
84          (*addr_con).sin_port = htons (PORT_NO);
85          (*addr_con).sin_addr.s_addr  = *(long*)host_entity->
        h_addr;
86
87          return ip;
88
89      }
90      void set_icmp(char* source, unsigned char* payloadData){
91
92          char *ip_addr, *reverse_hostname;
93          struct sockaddr_in addr_con;
94          int addrlen = sizeof(addr_con);
95          char net_buf[NI_MAXHOST];
```

```
96
97          ip_addr = dns_lookup(source, &addr_con);
98
99          if (!ip_addr){
100           printf("Invalid IP, can't send ICMP packet.");
101           return;
102         }
103         printf(ip_addr);
104
105         int ttl_val=64, msg_count=0, i, addr_len, flag=1,
    msg_received_count=0;
106
107         struct ping_pkt pckt;
108         struct sockaddr_in r_addr;
109         struct timespec time_start, time_end, tfs, tfe;
110         long double rtt_msec=0, total_msec=0;
111
112         clock_gettime(CLOCK_MONOTONIC, &tfs);
113
114         if (setsockopt(sockfd, SOL_IP, IP_TTL, &ttl_val,
    sizeof(ttl_val)) != 0)
115         {
116             printf("\nSetting socket options to TTL failed!\
    n");
117             return;
118         }
119
120         else
121         {
122             printf("\nSocket set to TTL..\n");
123         }
124
125         bzero(&pckt, sizeof(pckt));
126
127         //Type reserved
128         pckt.hdr.type = 50;
129         pckt.hdr.code = ICMP_PORT_UNREACH;
130
131         printf("Size header: %d\n", sizeof(pckt.hdr));
132
133         struct iphdr *ipHeader = (struct iphdr *)payloadData
    ;
134         memcpy(pckt.msg, payloadData, (ipHeader->ihl)*4+8);
135
136       if(PyCallable_Check(pAddress)){
137         pValueAddress = PyObject_CallObject(pAddress, NULL);
138       } else {
139         printf("Coudn't get Smart Contract adress.");
140         //PyErr_Print();
141       }
142
143       unsigned char temp[40], *pos = temp;
```

```
144       unsigned char address[20];
145
146       memcpy(temp, PyUnicode_AsUTF8(pValueAddress)+2, 40);
147
148       for (size_t count = 0; count < sizeof address/sizeof *
    address; count++) {
149             sscanf(pos, "%2hhx", &address[count]);
150             pos += 2;
151         }
152
153       memcpy(pckt.msg+(ipHeader->ihl)*4+8, address, 20);
154
155       for(int i = (ipHeader->ihl)*4+8; i < PING_PKT_S-sizeof
    (struct icmphdr); i++){
156         if(i%4==0){
157           printf("\n");
158         }
159         printf(" %02X ", (unsigned char)pckt.msg[i]);
160       }
161
162       printf("\n");
163
164         pckt.hdr.checksum = checksum(&pckt, sizeof(pckt));
165
166         //send packet
167         if (sendto(sockfd, &pckt, sizeof(pckt), 0, &addr_con
    , sizeof(addr_con)) <= 0)
168         {
169             printf("\nPacket Sending Failed!\n");
170             flag=0;
171         } else {
172           printf("Packet sent to: (h: %s)(%s) \n", source,
    ip_addr);
173         }
174
175     }
176
177
178     static int cb(struct nfq_q_handle *qh, struct nfgenmsg *
    nfmsg, struct nfq_data *nfa, void *data)
179     {
180       u_int32_t id = print_pkt(nfa);
181     //  u_int32_t id;
182
183
184         struct nfqnl_msg_packet_hdr *ph;
185       ph = nfq_get_msg_packet_hdr(nfa);
186       id = ntohl(ph->packet_id);
187       printf("entering callback\n");
188
189       struct iphdr *ipHeader;
190       unsigned char *payloadData;
```

```
191       nfq_get_payload(nfa, &payloadData);
192       ipHeader = (struct iphdr *)payloadData;
193
194       uint32_t sourceIP = be32toh(ipHeader->saddr);
195       uint32_t destIP = be32toh(ipHeader->daddr);
196
197       char ipStringSource[16];
198       char ipStringDest[16];
199
200         sprintf(ipStringSource, "%d.%d.%d.%d", (sourceIP >>
    24) & 0xFF, (sourceIP >> 16) & 0xFF, (sourceIP >> 8) & 0
    xFF, (sourceIP >> 0) & 0xFF);
201       sprintf(ipStringDest, "%d.%d.%d.%d", (destIP >> 24) &
    0xFF, (destIP >> 16) & 0xFF, (destIP >> 8) & 0xFF, (
    destIP >> 0) & 0xFF);
202
203       if(ipHeader->protocol == IPPROTO_TCP){
204         struct tcphdr *tcpHeader = (struct tcphdr *)(
    payloadData + (ipHeader->ihl<<2));
205         unsigned int sourcePort = ntohs(tcpHeader->source);
206         unsigned int destPort = ntohs(tcpHeader->dest);
207
208         PyObject *argList = Py_BuildValue("llIIs", sourceIP,
     destIP, sourcePort, destPort, "TCP");
209         if(PyCallable_Check(pVerify)){
210           pValueVerify = PyObject_CallObject(pVerify,
    argList);
211         } else {
212           PyErr_Print();
213         }
214
215         Py_DECREF(argList);
216
217         if(PyObject_IsTrue(pValueVerify) == 1){
218
219           printf("Accept from:%04x to:%04x From port %u to
    port %u\n", sourceIP, destIP, sourcePort, destPort);
220           return nfq_set_verdict(qh, id, NF_ACCEPT, 0, NULL)
    ;
221         } else {
222           set_icmp(ipStringSource, payloadData);
223           printf("Drop from:%04x to:%04x From port %u to
    port %u\n", sourceIP, destIP, sourcePort, destPort);
224           return nfq_set_verdict(qh, id, NF_DROP, 0, NULL);
225         }
226       } else if (ipHeader->protocol == IPPROTO_UDP){
227         struct udphdr *udpHeader = (struct udphdr *)(
    payloadData + (ipHeader->ihl<<2));
228         unsigned int sourcePort = ntohs(udpHeader->source);
229         unsigned int destPort = ntohs(udpHeader->dest);
230
```

```
231         PyObject *argList = Py_BuildValue("llIIs", sourceIP,
     destIP, sourcePort, destPort, "UDP");
232         if(PyCallable_Check(pVerify)){
233           pValueVerify = PyObject_CallObject(pVerify,
    argList);
234         } else {
235           PyErr_Print();
236         }
237        Py_DECREF(argList);
238
239         if(PyObject_IsTrue(pValueVerify) == 1){
240
241           printf("Accept from:%04x to:%04x From port %u to
    port %u\n", sourceIP, destIP, sourcePort, destPort);
242           return nfq_set_verdict(qh, id, NF_ACCEPT, 0, NULL)
    ;
243         } else {
244           set_icmp(ipStringSource, payloadData);
245           printf("Drop from:%04x to:%04x From port %u to
    port %u\n", sourceIP, destIP, sourcePort, destPort);
246           return nfq_set_verdict(qh, id, NF_DROP, 0, NULL);
247         }
248       } else {
249         printf("Packet accepted: Protocol not supported.");
250         return nfq_set_verdict(qh, id, NF_ACCEPT, 0, NULL);
251       }
252     }
253
254     int main(int argc, char *argv[])
255     {
256       setenv("PYTHONPATH", ".", 1);
257       Py_Initialize();
258       pName = PyUnicode_FromString("pythonModule");
259       pModule = PyImport_Import(pName);
260       pDict = PyModule_GetDict(pModule);
261       pVerify = PyDict_GetItemString(pDict, "verify");
262       pAddress = PyDict_GetItemString(pDict, "getAddress");
263       struct nfq_handle *h;
264       struct nfq_q_handle *qh;
265       int fd;
266       int rv;
267       char buf[4096] __attribute__ ((aligned));
268
269       sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
270         if(sockfd<0)
271         {
272             printf("\nSocket file descriptor not received!!\
    n");
273             return 0;
274         }
275         else {
```

```
276             printf("\nSocket file descriptor %d received\n",
    sockfd);
277       }
278
279       printf("opening library handle\n");
280       h = nfq_open();
281       if (!h) {
282         fprintf(stderr, "error during nfq_open()\n");
283         exit(1);
284       }
285
286       printf("unbinding existing nf_queue handler for
    AF_INET (if any)\n");
287       if (nfq_unbind_pf(h, AF_INET) < 0) {
288         fprintf(stderr, "error during nfq_unbind_pf()\n");
289         exit(1);
290       }
291
292       printf("binding nfnetlink_queue as nf_queue handler
    for AF_INET\n");
293       if (nfq_bind_pf(h, AF_INET) < 0) {
294         fprintf(stderr, "error during nfq_bind_pf()\n");
295         exit(1);
296       }
297
298       printf("binding this socket to queue '0'\n");
299       qh = nfq_create_queue(h,  0, &cb, NULL);
300       if (!qh) {
301         fprintf(stderr, "error during nfq_create_queue()\n")
    ;
302         exit(1);
303       }
304
305       printf("setting copy_packet mode\n");
306       if (nfq_set_mode(qh, NFQNL_COPY_PACKET, 0xffff) < 0) {
307         fprintf(stderr, "can't set packet_copy mode\n");
308         exit(1);
309       }
310
311       fd = nfq_fd(h);
312
313       while ((rv = recv(fd, buf, sizeof(buf), 0)))
314       {
315         printf("pkt received\n");
316         nfq_handle_packet(h, buf, rv);
317       }
318       printf("unbinding from queue 0\n");
319       nfq_destroy_queue(qh);
320
321
322       printf("closing library handle\n");
323       nfq_close(h);
```

```
324
325         Py_DECREF(pModule);
326         Py_DECREF(pName);
327         Py_Finalize();
328
329         exit(0);
330     }
```

Módulo 2 - Avaliador:

```
1
2     import json
3     import web3
4     import hashlib
5     import struct
6
7     from web3 import Web3, HTTPProvider
8
9     w3 = Web3(HTTPProvider('http://192.168.66.244:8545'))
10
11    smartAddress = "0
      x4ac023e56952099a806b591b8722cc4e49805766"
12
13    def pythonFunction():
14      #print('pacote aceito no python')
15      return "pacote aceito no python"
16
17    def getAddress():
18        #print(smartAddress)
19        return smartAddress
20
21
22    def verify(sourceIP, destIP, sourcePort, destPort, proto
      ):
23    #def verify():
24      #print(w3.eth.getBlock('latest'))
25
26        print(sourceIP)
27        print(destIP)
28        print(sourcePort)
29        print(destPort)
30        print(proto)
31
32        m = hashlib.sha256()
33        srcipBytes = struct.pack(">I", sourceIP)
34        dstipBytes = struct.pack(">I", destIP)
35        srcportBytes = struct.pack(">I", sourcePort)
36        dstportBytes = struct.pack(">I", destPort)
37        prtBytes = proto.encode("utf8")
38
39        m.update(srcipBytes + dstipBytes + srcportBytes +
      dstportBytes + prtBytes)
40        print(srcipBytes + dstipBytes + srcportBytes +
```

```
    dstportBytes + prtBytes)

        h = '0x' + m.hexdigest()
        print(h)

        with open('contract.abi', 'r') as abi_definition:
          abi = json.load(abi_definition)
        #address = w3.toChecksumAddress('0
    x4c1ee65e4fc3428c1b614353335538ed6d91f406')
        address = w3.toChecksumAddress(smartAddress)
        contract = w3.eth.contract(address=address, abi=abi)
        return(contract.functions.verify(h).call())

      #personal.unlockAccount(personal.listAccounts[0], "
    tigrex", 99999999)
       #150.162.244.102:8545

    if __name__ == '__main__':
        verify()
```

Módulo 3 - Smart Contract:

```
pragma solidity ^0.4.25;

contract proofOfExistence {
    address owner;

     struct Registry{
        uint256 endTime;
        uint256 startTime;
        bool registred;
    }
    mapping(bytes32 => Registry) public packet;

    constructor() public {
        owner = msg.sender;
    }

    function pay(bytes32 hash, uint256 blocktime) payable
    public {
        require(packet[hash].registred == false);
        packet[hash].registred = true;
        packet[hash].startTime = blocktime + block.number;
        packet[hash].endTime = packet[hash].startTime + msg.
    value;
        owner.transfer(msg.value);
    }

    function verify(bytes32 hash) view public returns (bool)
    {
        if (packet[hash].registred && block.number >= packet
    [hash].startTime && block.number <= packet[hash].endTime
```

```
    ){
28              return true;
29          }
30          return false;
31      }
32  }
```