Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Descoberta e Monitoramento de Blocos Modificados na Rede Bitcoin

Maycon V. S. Fabio

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. José Edil Guimarães de Medeiros

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Descoberta e Monitoramento de Blocos Modificados na Rede Bitcoin

Maycon V. S. Fabio

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. José Edil Guimarães de Medeiros (Orientador)
ENE/UnB

Prof. Eduardo Adilio Pelinson Alchier
CIC/UnB

Sr. Bruno Ely Reis Garcia
Vinteum

do Bacharelado em Ciência da Computação

Brasília, 6 de outubro de 2025

Dedicatória

À minha família – pelo apoio incondicional, pelo amor inabalável e por terem tornado possível que eu chegasse até aqui.

Que eu consiga, um dia, retribuir na mesma medida.

Agradecimentos

Agradeço primeiramente ao meu orientador, José Edil Guimarães de Medeiros, por ter me dado a oportunidade de trabalhar com ele e por ter me apresentado o universo técnico do Bitcoin. Pude aprender conceitos geniais que só foram possíveis com ele. Sem dúvida, um dos melhores professores que já tive — se não o melhor.

Agradeço também ao 0xB10C, autor do projeto *peer-observer*, por reservar parte de seu tempo para revisar nossas implementações, opinar sobre as ideias e contribuir diretamente para sua qualidade.

E claro, ao tal do Satoshi Nakamoto que sem ele não estaríamos aqui, seja lá que ser ou entidade ele for.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

A rede Bitcoin, como sistema distribuído descentralizado, enfrenta desafios persistentes na detecção de anomalias, como blocos mutated, que podem ser explorados em ataques de negação de serviço (DoS) e comprometer a propagação de transações e blocos. Este trabalho propõe uma extensão ao peer-observer, ferramenta open-source para monitoramento da rede P2P, com o desenvolvimento de um extrator de logs (*Log Extractor*) integrado via protobuf e NATS, capaz de identificar e classificar em tempo real todos os tipos de blocos *mutated* definidos no Bitcoin Core, utilizando simulações controladas em modo *regtest*. Perspectivas futuras incluem integração com a *mainnet* para análises estatísticas em escala.

Palavras-chave: Bitcoin, Monitoramento, Blocos modificados

Abstract

The Bitcoin network, as a decentralized distributed system, faces persistent challenges in detecting anomalies such as *mutated blocks*, which can be exploited in denial-of-service (DoS) attacks and impair the propagation of transactions and blocks across the network. This work proposes an extension to *peer-observer*, an open-source tool for monitoring the Bitcoin P2P network, through the development of a *Log Extractor* integrated via protobuf and NATS. The extractor is capable of identifying and classifying, in real time, all types of *mutated blocks* defined in Bitcoin Core, using controlled simulations in *regtest* mode. Future directions include integration with the Bitcoin *mainnet* to enable large-scale statistical analysis of these anomalies in production environments.

Keywords: Bitcoin, Monitoring, Mutated blocks

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Protocolo Bitcoin

O Bitcoin pode ser compreendido fundamentalmente como um protocolo de comunicação, ou protocolo de mensageria, que opera em uma rede descentralizada de computadores, conhecida como rede Bitcoin [1]. Essa rede, em funcionamento contínuo desde 2009, constitui um sistema distribuído que pode ser considerado o supercomputador mais poderoso do mundo, superando em ordens de magnitude infraestruturas como as do Google ou Facebook [2]. O protocolo define regras para a troca de mensagens entre nós (*nodes*), permitindo a coordenação de ações sem a necessidade de uma autoridade central, o que resolve problemas clássicos em sistemas distribuídos, como o consenso em ambientes não confiáveis.

As mensagens mais importantes nesse protocolo são as transações, que representam estruturas de dados essenciais para o funcionamento do sistema. Uma transação consiste em campos como versão, entradas (*inputs*), saídas (*outputs*) e *lock time*. As entradas referenciam saídas não gastas de transações anteriores (UTXOs, ou *Unspent Transaction Outputs*), representando os fundos que estão sendo gastos. As saídas, por sua vez, especificam os valores a serem transferidos e condições para seu gasto futuro, codificadas em um pequeno programa na linguagem *Script*, que define regras de validação baseadas em criptografia assimétrica. Essa semântica assemelha-se a uma entrada contábil, onde um valor é debitado de uma ou mais fontes e creditado a novos destinatários, garantindo a transferência de valor sem intermediários [1]. A Figura 1.1 ilustra oque é a UTXO e como ela é gasta em outra transação.

As mensagens, ou transações, passam por um processo de validação em duas etapas principais. Inicialmente, ao serem propagadas pela rede, cada nó verifica sua validade de acordo com as regras do protocolo, incluindo a autenticação de assinaturas digitais e a ausência de gasto duplo em relação ao estado atual da blockchain. Transações válidas são

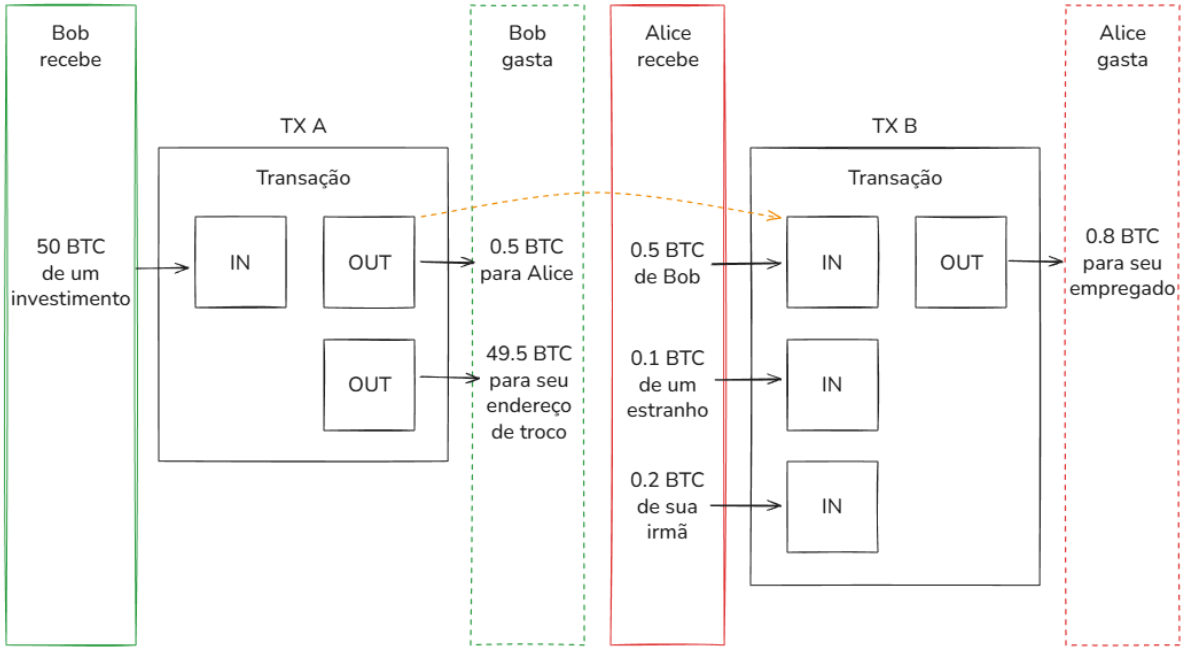

Figura 1.1: Dinâmica de gastos da UTXO's nas transações.

armazenadas temporariamente na *mempool* de cada nó, um *pool* de transações pendentes aguardando inclusão em um bloco. Na segunda etapa, mineradores selecionam transações da *mempool* para compor um novo bloco, resolvendo um problema computacionalmente intensivo conhecido como *proof-of-work*. Esse mecanismo não apenas confirma as transações, mas resolve o problema de gasto duplo ao impor uma ordem canônica às transações: a cadeia de blocos mais longa, que requer o maior esforço computacional acumulado, é considerada válida, tornando economicamente inviável reverter transações confirmadas [1]. Ferramentas como o *Mempool Space* permitem o monitoramento em tempo real dessa dinâmica [3].

1.2 Monitorando a rede do Bitcoin

Dado o caráter descentralizado da rede Bitcoin, o monitoramento de sua operação apresenta desafios únicos, especialmente na detecção de anomalias e ameaças à integridade do sistema. Problemas como o *flooding* de mensagens P2P, em que agentes maliciosos enviam grandes quantidades de dados para sobrecarregar a rede, ou a criação de nós falsos (*sybil attacks*), que tentam manipular o consenso ou obter informações sensíveis, representam riscos reais à operação estável da rede [4]. Esses ataques podem comprometer a eficiência da propagação de blocos e transações, impactando diretamente mineradores e outros participantes da rede.

Cada nó na rede possui apenas uma visão local, limitada às interações com um conjunto restrito de pares (*peers*), tipicamente 8 conexões de saída e até 125 de entrada. Essa perspectiva parcial impede que um único nó perceba o estado global da rede, tornando o monitoramento difícil e suscetível a limitações, como a incapacidade de detectar padrões de comportamento em escala ampla ou ataques que isolam visões locais, como *eclipse attacks* [4]. Para superar essas restrições, as abordagens principais envolvem a inserção de múltiplos nós de sondagem (*probe nodes* ou *honeynodes*) na rede, distribuídos geograficamente e configurados para coletar dados agregados sobre métricas como latência de mensagens, volume de tráfego e conexões ativas. Esses *probes* atuam como sensores passivos, permitindo uma visão mais abrangente ao correlacionar observações de diferentes pontos da rede.

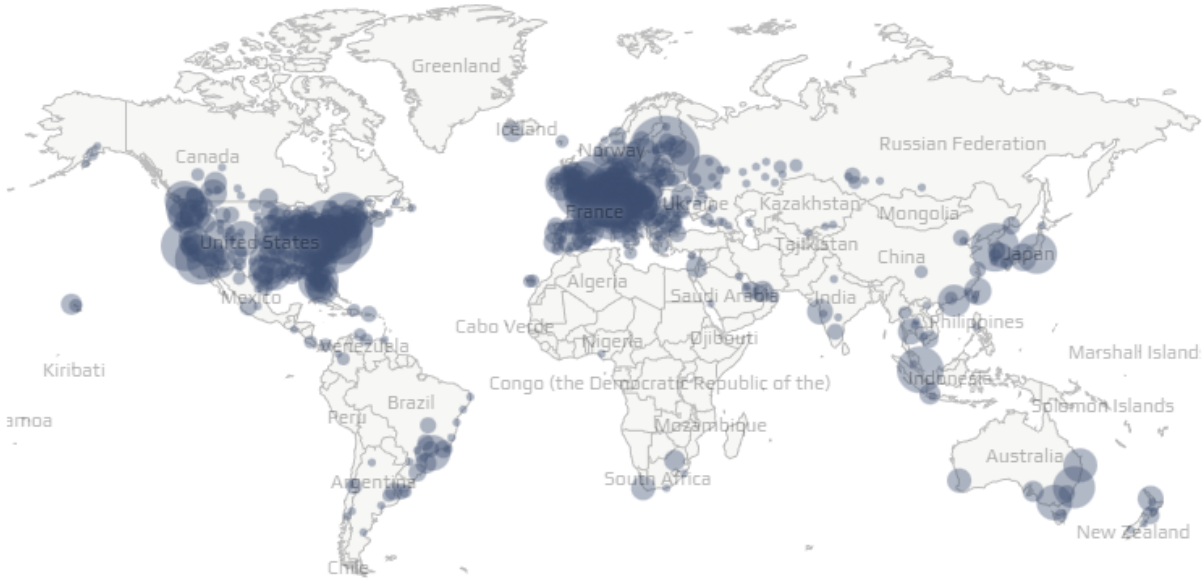

Figura 1.2: Distribuição geográfica da rede Bitcoin (Fonte: [5]).

É do interesse de mineradores e outros participantes, essa capacidade de monitoramento para identificar anomalias, como atrasos na propagação de blocos ou desconexões suspeitas, que podem indicar problemas técnicos ou ataques direcionados, minimizando impactos no desempenho e na segurança das operações. A detecção precoce dessas questões permite a adoção de contramedidas, como a reconfiguração de conexões ou o isolamento de nós maliciosos, minimizando perdas de desempenho e garantindo a continuidade da mineração [6].

1.3 Contribuições para o monitoramento

O monitoramento contínuo da rede *peer-to-peer* do Bitcoin tem atraído crescente atenção da comunidade acadêmica e de desenvolvedores, resultando em diversas iniciativas que ampliam a compreensão sobre topologia, propagação de blocos, detecção de anomalias e resiliência do protocolo. Dentre essas iniciativas destacam-se o Warnet, infraestrutura aberta para simulação e análise de comportamentos emergentes em redes Bitcoin [7]; o protocolo AToM, que realiza monitoramento ativo da topologia com precisão superior a 90% mesmo na presença de nós maliciosos [8]; o Pool Detective, desenvolvido pelo MIT Digital Currency Initiative, capaz de observar a propagação de blocos a partir de múltiplas localizações geográficas e detectar comportamentos anômalos de *pools* de mineração [9]; e o Bitcore, suite de ferramentas que oferece interface nativa à rede P2P para análise em tempo real [10].

Em particular, uma dessas soluções é o **peer-observer**, uma ferramenta *open-source* projetada para coletar e analisar dados da rede P2P do Bitcoin, permitindo a identificação de padrões anômalos e o acompanhamento em tempo real do comportamento dos nós [11]. Por meio da análise de métricas como latência de mensagens, volume de transações no *mempool* (o conjunto de transações não confirmadas) e conexões ativas, o peer-observer oferece uma visão detalhada da dinâmica da rede, sendo particularmente útil para seus participantes que buscam garantir a estabilidade e a segurança de suas operações.

Este trabalho **contribuiu** diretamente para o projeto *peer-observer* [11] por meio da implementação e integração de um novo módulo de extração de dados — o *Log Extractor* —, responsável por monitorar em tempo real os *logs* do Bitcoin Core. Essa funcionalidade foi incorporada ao repositório oficial via *pull request* #272 [12], aceito e mesclado em setembro de 2025. Discorreremos mais sobre o assunto no Capitulo 3.

O principal desenvolvedor do *peer-observer*, conhecido pelo pseudônimo 0xB10C, mantém um repositório público de ideias de projetos para monitoramento da rede Bitcoin. Nessa lista de desejos, aberta desde 2024 e ainda em desenvolvimento, o autor destaca explicitamente a necessidade de "*track the number of mutated blocks and understand why they are mutated*" na seção dedicada ao relay de blocos [13]. Essa sugestão reconhece a relevância persistente do fenômeno, e propõe a sua contabilização sistemática como uma métrica valiosa para a saúde da rede, assim como entender porque são propagados.

Essa lacuna foi reforçada por uma observação concreta publicada pelo mesmo autor em março de 2024. Em seu blog pessoal, 0xB10C relatou a transmissão, pela *pool* ViaBTC, de blocos modificados sem dados de witness, que provocavam erros de validação do tipo `bad-witness-nonce-size` em nós Bitcoin Core modernos [14]. Esses blocos, enviados de forma não solicitada via mensagens `block` por clientes `bitpeer` da ViaBTC, violavam as regras de compromisso de witness definidas no BIP-141 [15], sendo rejeitados durante a

verificação contextual, embora versões válidas fossem propagadas via *compact blocks*. O incidente, detectado em janeiro/fevereiro de 2024, evidenciou que variações contemporâneas de blocos modificados ainda ocorrem na rede real, consumindo recursos de validação e motivando melhorias posteriores no Bitcoin Core (pull request #29412, que introduz verificações precoces de mutação [16]).

Embora o interesse em monitorar *mutated blocks* tenha sido explicitamente manifestado tanto na *wishlist* quanto na análise do caso ViaBTC, nenhuma implementação específica foi incorporada ao peer-observer ou a outras ferramentas amplamente disponíveis até o momento. Neste trabalho, preenchemos exatamente essa lacuna identificada pela comunidade ao estender o peer-observer com funcionalidades dedicadas à detecção automática, contabilização e análise de blocos modificados.

1.4 Propagação de Blocos

Antes de falarmos sobre os blocos modificados, precisamos entender como a propagação de blocos na rede *peer-to-peer* (P2P) do Bitcoin funciona. Esse processo ocorre de forma descentralizada, onde um nó que minera ou recebe um novo bloco o anuncia e transmite para seus pares conectados. O protocolo básico, descrito na documentação oficial do Bitcoin [17], segue uma sequência de mensagens que permite o anúncio, a solicitação e o envio de dados, visando evitar sobrecargas na rede.

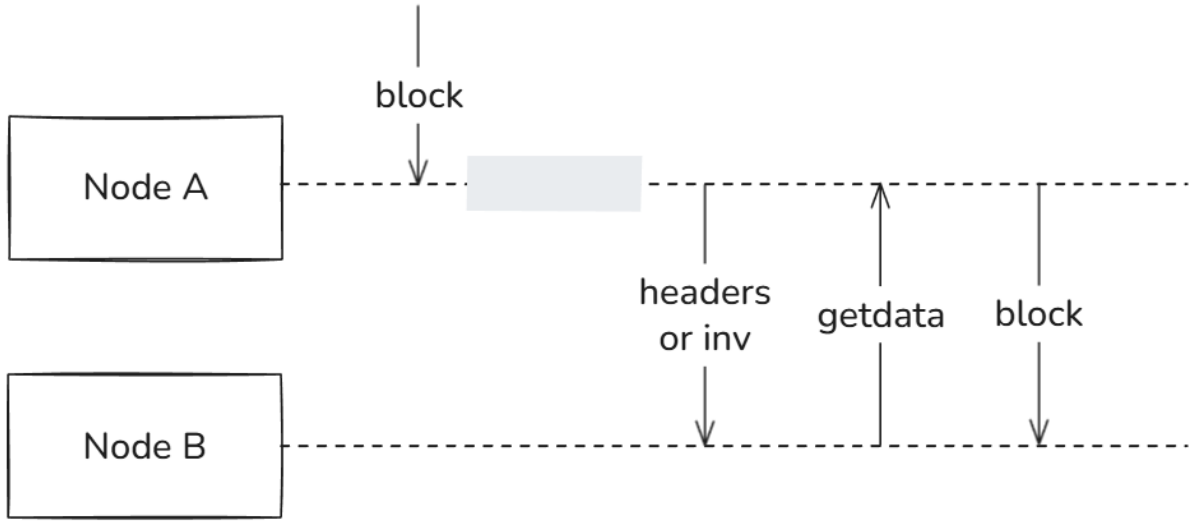

Figura 1.3: Fluxo da propagação de blocos.

O processo inicia com o anúncio do bloco por meio da mensagem `inv`, que contém o *hash* do bloco. Essa mensagem informa aos pares sobre a existência do novo bloco,

sem transmitir seu conteúdo completo imediatamente. O objetivo é permitir que os nós receptores avaliem a relevância do bloco, como verificar se já o tem ou se ele estende a cadeia principal, antes de solicitar mais dados [17].

Em seguida, o nó receptor, ao identificar o bloco como relevante, envia uma mensagem `getdata` especificando o tipo `MSG_BLOCK` e o *hash* correspondente. Essa solicitação requere o envio do bloco completo pelo nó anunciante, por meio da mensagem `block`. Essa mensagem inclui o *header* do bloco — composto por campos como versão, *hash* do bloco anterior, *Merkle root*, *timestamp*, *bits* e *nonce* — e a lista integral de transações. Após o recebimento, o nó valida o bloco e, se aprovado, o propaga para seus próprios pares [17].

Posteriormente, o BIP 152 implementou o *Compact Block Relay*. Esse mecanismo envia blocos compactos (`cmpctblock`) contendo o *header*, um *nonce* e IDs curtos de transações, permitindo reconstrução local com base na *mempool* e solicitação apenas de dados faltantes via `getblocktxn` e `blocktxn`. Como resultado, o tamanho das mensagens reduz-se significativamente, de megabytes para kilobytes na maioria dos casos, melhorando a velocidade de propagação e reduzindo o risco de congestionamento na rede [18].

1.5 Blocos *mutated*

No contexto da rede Bitcoin, blocos *mutated* referem-se a blocos cuja estrutura é alterada de forma maliciosa ou acidental após sua propagação inicial. A principal razão para a criação dessa classificação é mitigar ataques de negação de serviço (DoS), como o *eclipse attack* [4], nos quais um atacante pode explorar o cache de rejeição de blocos inválidos para impedir que um nó processe a versão válida do bloco, levando a isolamento da rede ou atrasos na propagação.

É importante destacar como blocos *mutated* e inválidos são relacionados. Um bloco inválido viola as regras de consenso da rede Bitcoin, levando à sua rejeição pelos nós participantes. Por exemplo, um bloco que tenta gastar uma UTXO já utilizada em uma transação anterior seria considerado inválido e descartado imediatamente [1]. Um bloco *mutated* é uma classificação que pode ser dada a um bloco inválido, partindo de um bloco válido modificamos campos não essenciais para o cálculo do **hash do cabeçalho** – como por exemplo, dados relacionados ao *witness* sem o *witness commitment* correspondente, introduzido pelo BIP-141 (*Segregated Witness*) [15]. Essas alterações não afetam a integridade do *hash* do bloco, mas podem causar inconsistências na validação, potencialmente exploradas em ataques de negação de serviço.

A classificação "*mutated*" foi reforçada para incluir verificações específicas de *witness*, como *bad-witness-nonce-size*, *bad-witness-merkle-match* e *unexpected-witness*, além de outras, garantindo que esses blocos sejam rejeitados cedo sem *caching* [16]. Uma carac-

terística interessante é que, no comando *getchaintips* da interface RPC que é responsável por mostrar todos blocos de ponta (fim da cadeia de blocos), incluindo os "órfãos", os blocos *mutated* não aparecem já que não são guardados.

1.6 Objetivos

O objetivo principal deste trabalho é expandir as capacidades do peer-observer de detectar blocos modificados (*mutated blocks*) na rede Bitcoin, focando na identificação de erros específicos que podem comprometer a integridade da rede. Especificamente, busca-se replicar e analisar cenários de falhas associadas a esses blocos, conforme descrito na literatura técnica [19, 20].

Para alcançar esse objetivo, adota-se uma abordagem que combina simulação controlada em ambiente de teste (*regtest mode* do Bitcoin Core) com análise detalhada de *logs*. Inicialmente, serão criados scripts para simular os tipos de erros identificados, permitindo a validação da detecção em condições controladas. A análise do código-fonte do Bitcoin Core, particularmente no arquivo `validation.cpp`, revelou que ferramentas existentes, como o *peer-observer* ainda não eram adequadas para essa detecção [11], tornando necessário estender a implementação da mesma com a extração de informações diretamente dos *logs* do nó.

Nesse contexto, desenvolveu-se um extrator de *logs* (*log extractor*) capaz de receber e processar esses dados gerados pelo nó, também com a possibilidade de lidar com categorias de depuração ativadas, como a flag *validation*. O escopo limita-se a essa prova de conceito, abrangendo exclusivamente:

- Implementação de uma ferramenta que simule uma rede com um nó malicioso, que gera os blocos modificados e permita um nó vítima recebê-los;
- O desenvolvimento e validação do extrator de *logs* para captura de dados;
- O desenvolvimento de um processador de *logs*, capaz de classifica-los de modo genérico ou específico, possibilitando a identificação dos *logs* alvo;
- Metrificação dos *logs* relacionados a blocos modificados;
- Uma visualização básica dos eventos coletados.

Não serão realizadas análises estatísticas sobre a frequência desses eventos na rede principal (*mainnet*), nem avaliações de impactos operacionais ou econômicos. Ao final, será disponibilizada uma ferramenta para análise de *logs* gerados pelo Bitcoin Core, com apresentação de dados coletados como evidência da eficácia da detecção.

Capítulo 2

Arquitetura do Peer-Observer

2.1 Honeynode

Um *honeynode*, no contexto da rede Bitcoin, refere-se a um nó configurado como uma armadilha (*honeypot*) para monitorar atividades anômalas ou maliciosas na rede P2P. Esses nós são projetados para se comportar de forma passiva e conforme as regras do protocolo, atraindo interações de pares potencialmente maliciosos sem necessariamente participar ativamente da validação ou propagação de transações e blocos. Essa abordagem permite a detecção de ataques, como tentativas de *flooding* ou *sybil attacks*, por meio da análise de conexões e mensagens recebidas.

2.2 Peer Observer

O Peer Observer é uma ferramenta desenvolvida na linguagem de programação Rust, projetada para monitorar eventos emitidos pelo Bitcoin Core, a implementação de referência do protocolo Bitcoin. A Figura 2.1 ilustra a arquitetura do Peer Observer, destacando o fluxo de dados desde o *honeynode* para os **extratores**, passando pelo servidor NATS, até o processamento pelas **ferramentas**. Exploraremos mais detalhadamente cada passo a seguir.

2.2.1 Extratores de dados

Há vários meios de coleta dados de diferentes interfaces e mecanismos, como chamadas RPC, *tracepoints* eBPF (*extended Berkeley Packet Filter*), entre outros. Cada um desses métodos de coleta é gerenciado por implementações classificadas como **extractors** (extratores), que vigiam eventos específicos da rede Bitcoin, como transações, blocos propagados ou alterações no mempool. Quando um evento de interesse é identificado, o extrator se-

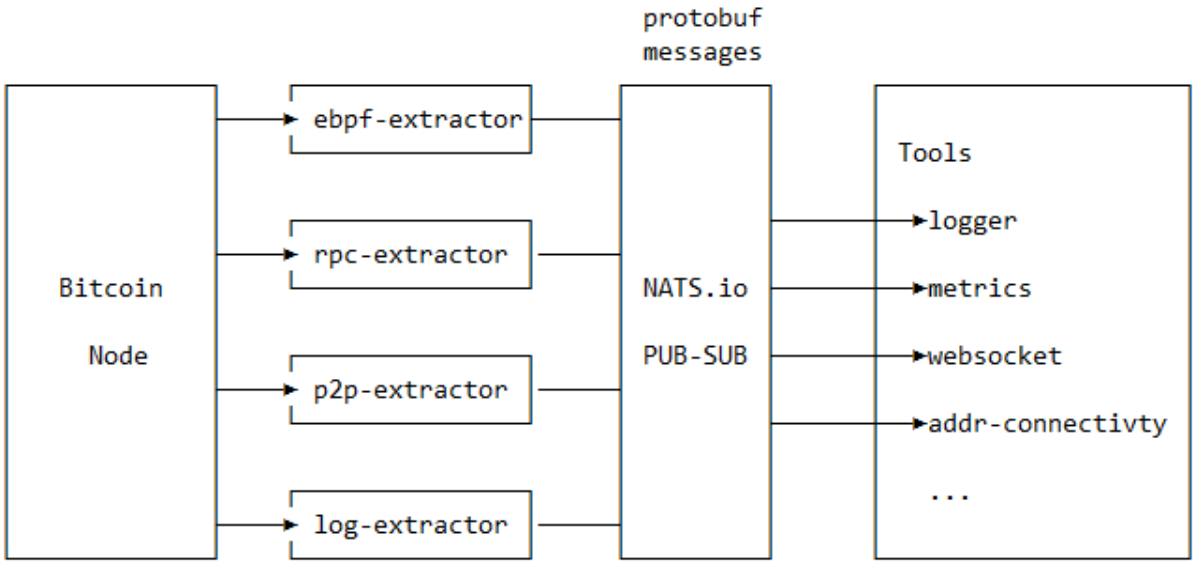

Figura 2.1: Diagrama da arquitetura do Peer Observer (Fonte: [11]).

rializa os dados no formato Protobuf. Esses dados serializados são então publicados em um servidor NATS, um sistema de mensageria. Abaixo estão os extratores atualmente implementados [11]:

- **Extrator eBPF**: Extrai eventos de um nó Bitcoin Core em tempo real utilizando *tracepoints*. Captura eventos como mensagens P2P recebidas e enviadas, conexões P2P abertas e fechadas, alterações no *mempool*, entre outros. Implementado com as capacidades USDT da biblioteca libbpf-rs, que requer privilégios elevados no sistema para se conectar aos *tracepoints* e só é suportado a partir da v29.0 do Bitcoin Core.
- **Extrator RPC**: Busca periodicamente dados da interface RPC do Bitcoin Core para extrair eventos. Consulta a interface RPC em intervalos regulares e publica os resultados como eventos RPC. Utiliza a biblioteca corepc para interagir com o servidor RPC do Bitcoin Core. Requer acesso à interface RPC do nó.
- **Extrator P2P**: Captura eventos P2P de conexões de entrada para um nó *honeypot*. Recebe conexões de entrada de nós Bitcoin e publica medições selecionadas de P2P como eventos. Projetado para monitorar o comportamento P2P em um ambiente controlado, exigindo que o nó aceite conexões de entrada.

Protobuf

O *Protocol Buffers* (Protobuf) é um mecanismo de serialização de dados estruturados desenvolvido pelo Google, utilizado para codificar informações de forma compacta e eficiente. Ele permite a definição de esquemas de dados em arquivos `.proto`, que descrevem a

estrutura das mensagens por meio de campos tipados, como inteiros, *strings* ou mensagens aninhadas. Esses esquemas são compilados para gerar código em diversas linguagens de programação, facilitando a interoperabilidade entre sistemas heterogêneos [21].

No funcionamento básico, uma mensagem Protobuf é serializada em um formato binário que ocupa menos espaço que representações textuais como JSON ou XML, preservando desempenho em cenários de alta taxa de transferência. A serialização segue uma codificação *varint* para campos numéricos e *length-delimited* para *strings*, garantindo eficiência tanto em banda quanto em processamento. A compatibilidade entre versões é mantida por meio de números de campo únicos, permitindo evoluções no esquema sem quebrar sistemas existentes.

2.2.2 Mecanismos de conexão - NATS

O NATS é um *middleware* orientado a mensagens que permite a troca de dados entre aplicações e serviços por meio de mensagens. Ele facilita a construção de aplicações distribuídas e escaláveis, permitindo o armazenamento e a distribuição de dados em tempo real em diferentes ambientes, linguagens de programação, provedores de nuvem e sistemas locais [22].

No funcionamento básico, aplicações cliente utilizam bibliotecas específicas para publicar, subscrever, solicitar e responder mensagens entre instâncias ou aplicações separadas. A infraestrutura de serviço é composta por um ou mais servidores NATS interconectados, que podem variar de um único processo leve (menos de 20 MB) em um dispositivo final a um super-cluster global abrangendo múltiplos provedores de nuvem e regiões.

As aplicações se conectam aos servidores NATS por meio de uma URL que especifica endereço IP, porta e tipo de conexão (como TCP simples, TLS ou WebSocket), além de detalhes de autenticação quando necessário.

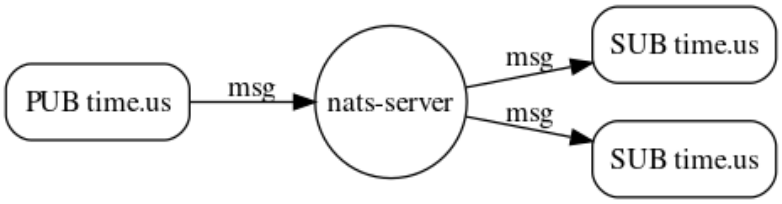

Figura 2.2: Fluxo de mensagens no NATS (Fonte: [22]).

O modelo principal de comunicação é o *publish-subscribe*, no qual mensagens são endereçadas por *strings* de *subjects*, independentemente da localização na rede. Um publicador envia uma mensagem com dados, que é recebida e processada por um ou mais assinan-

tes. Esse design promove o compartilhamento de código de manipulação de mensagens, isolamento de preocupações e escalabilidade com volumes crescentes de mensagens.

No contexto desse trabalho, os extratores publicam mensagens no NATS e as *tools* consomem essas mensagens, basicamente uma ponte.

2.2.3 Consumidores de dados

No fluxo de operação do Peer-Observer, após os extratores coletarem e publicarem eventos no servidor NATS, entram em cena os componentes classificados como **tools** (ferramentas), que atuam como consumidores de dados. Essas ferramentas se inscrevem em tópicos específicos no NATS e, ao receberem mensagens relevantes, deserializam do formato *protobuf* e executam processamentos personalizados para análise ou visualização dos dados. Abaixo estão as ferramentas atualmente implementadas [11]:

- **Ferramenta logger**: Recebe eventos e os imprime no *stdout* (com filtragem opcional por tipo). Útil para depuração e visualização rápida de eventos em tempo real.
- **Ferramenta metrics**: Consome eventos e produz métricas Prometheus. Agrega dados como desempenho da rede, permitindo monitoramento via Prometheus e visualização em Grafana.
- **Ferramenta websocket**: Recebe eventos e os publica como JSON via WebSocket. Facilita a integração com aplicações web ou clientes remotos para streaming de eventos.
- **Ferramenta connectivity-check**: Extrai endereços IP de mensagens addr(v2), tenta conexões e registra resultados de alcançabilidade. Útil para avaliar a conectividade na rede P2P.
- **Ferramenta record-getblocktxn-py**: Captura mensagens P2P getblocktxn enviadas e recebidas, e as registra. Implementada em Python, auxilia na análise de solicitações compactas de blocos.

Capítulo 3

Geração e detecção de blocos mutated

3.1 Os tipos de blocos *mutated*

Antes de abordar a geração de blocos modificados, é fundamental compreender os tipos de blocos *mutated*, assim como os mecanismos responsáveis por sua origem. Estenderemos oque foi introduzido na Seção 1.5, detalhando os 5 tipos de blocos com essa classificação.

3.1.1 bad-txnmrklroot

O erro `bad-txnmrklroot` acontece quando há uma inconsistência entre a *merkle root* calculada pelo nó que recebeu o bloco e o nó que enviou. Oque é uma *merkle root* e como ela é usada no bloco? Ela basicamente faz um SHA256 entre cada par de *transaction IDs*, sucessivamente como se fosse caminhando das folhas de uma árvore binária até a raíz, essa raíz é chamada de *merkle root*. Esse *hash* é armazenado no cabeçalho do bloco, onde é transmitido na rede para outros nós saberem se já possuem aquele conjunto de transações ou não, já que alterar qualquer uma delas no bloco resultaria em um *hash* completamente diferente.

Trecho de código onde essa validação ocorre:

```
uint256 merkle_root = BlockMerkleRoot(block, &mutated);
if (block.hashMerkleRoot != merkle_root) {
    return state.Invalid(
        /*result=*/BlockValidationResult::BLOCK_MUTATED,
        /*reject_reason=*/"bad-txnmrklroot",
        /*debug_message=*/"hashMerkleRoot mismatch");
}
```

3.1.2 bad-txns-duplicate

A vulnerabilidade associada ao erro `bad-txns-duplicate` refere-se a uma falha crítica na validação de blocos no Bitcoin Core, identificada na CVE-2012-2459 [20]. Essa vulnerabilidade permitia a construção de blocos *mutated*, nos quais transações duplicadas eram inseridas na *merkle three* sem alterar o hash raiz do bloco, possibilitando ataques de negação de serviço (DoS).

Como discutido na seção 3.1.1, são feitos vários *hashes* entre as transações para no final sobrar somente uma, oque fica bem simples e direto quando a árvore tem um número de transações que seja potência de 2. Temos que resolver os casos onde não há um par para a transação, isso foi solucionado duplicando essa transação "orfã".

Considere um conjunto de transações a, b, c. A construção da *merkle three* para três transações resulta na duplicação da última transação c no nível das folhas:

$$\text{merkle\_hash}([a, b, c]) = \text{hash}(\text{hash}(a||b), \text{hash}(c||c))$$

Se uma transação duplicada c for adicionada ao bloco, formando a, b, c, c, a *merkle three* mantém a mesma estrutura:

$$\text{merkle\_hash}([a, b, c, c]) = \text{hash}(\text{hash}(a||b), \text{hash}(c||c))$$

Ambas as configurações produzem o mesmo *merkle root*, pois o $\text{hash}(c||c)$ permanece inalterado. Assim, o cabeçalho do bloco – e consequentemente seu *hash* – não é modificado, permitindo que o bloco modificado passe pela validação inicial de nós honestos. A figura 3.1 ilustra um caso de 6 transações, onde a duplicação ocorre após um *hash* já ter acontecido.

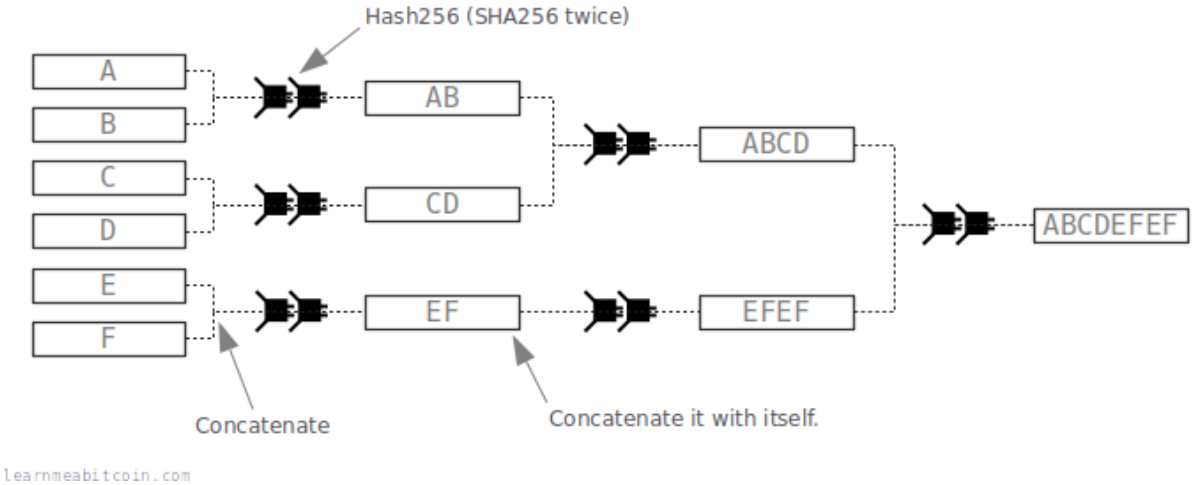

Figura 3.1: Funcionamento da merkle root (Fonte: [23]).

Como resultado, é possível criar facilmente dois blocos com o mesmo *hash* de bloco, embora um seja válido e o outro não, duplicando uma ou mais transações de forma que preserve o *hash* da raiz de Merkle [24]. A duplicação de qualquer transação torna o bloco inválido, pois viola a regra de gastos duplos (*double-spending*) de uma saída de transação passada.

Um nó Bitcoin não corrigido pode ser permanentemente travado em seu bloco mais alto atual utilizando esta vulnerabilidade, combinada com o fato de que o Bitcoin armazena blocos órfãos em um banco de dados persistente em disco. Para explorar esta falha, o atacante envia um bloco válido (que eventualmente será incorporado à *blockchain*) modificado pela duplicação de uma transação que preserva a *merkle root*. É importante destacar que o atacante não precisa minerar seu próprio bloco; em vez disso, pode escutar um bloco válido na rede, modifica-lo e propagá-lo para seus pares.

Quando o nó vítima recebe este bloco inválido, ele o armazena em disco, tenta processá-lo e o rejeita como inválido devido ao erro *bad-txns-duplicate*. No entanto, o Bitcoin considera que já possui o bloco (devido ao *hash* idêntico) e não tenta solicitá-lo novamente. Consequentemente, quando a *blockchain* avança além do bloco modificado, o Bitcoin exibe o aviso: "*WARNING: Displayed transactions may not be correct! You may need to upgrade, or other nodes may need to upgrade*" [24].

Este mecanismo de ataque de negação de serviço (DoS) permanente explora a combinação da validação incompleta de blocos mutados com a política de *cache* de blocos órfãos. A duplicação de transações, embora preserve o *hash* do cabeçalho do bloco, viola as regras de consenso do protocolo Bitcoin, resultando na rejeição permanente do nó afetado.

A correção desta vulnerabilidade, identificada como CVE-2012-2459, envolveu a implementação de validação estrita da árvore de Merkle, garantindo que mutações sejam detectadas independentemente da preservação do *hash* do cabeçalho [20].

Trecho de código onde essa validação ocorre:

```
// BlockMerkleRoot chama essa funcao
uint256 ComputeMerkleRoot(std::vector<uint256> hashes, bool* mutated) {
    ...
    if (hashes[pos] == hashes[pos + 1]) mutation = true;
    ...
}

...
uint256 merkle_root = BlockMerkleRoot(block, &mutated);
if (mutated) {
    return state.Invalid(
        /*result=*/BlockValidationResult::BLOCK_MUTATED,
```

```
        /*reject_reason=*/"bad-txns-duplicate",
        /*debug_message=*/"duplicate transaction");
}
```

3.1.3 bad-witness-nonce-size

O erro de `bad-witness-nonce-size` ocorre quando o campo *witness* da transação *coinbase* está ausente ou apresenta um tamanho diferente de 32 bytes, violando uma validação de formato estabelecida pela BIP 141 [15]. Esse valor, conforme definido pela proposta de *Segregated Witness* (SegWit), é tipicamente uma sequência de zeros (0x000...00), embora possa ser utilizado para compromissos adicionais de dados, como extensões futuras.

A BIP 141 introduziu o campo *witness* para segregar os dados de assinatura (como assinaturas ECDSA) do corpo principal das transações, otimizando o espaço nos blocos e mitigando problemas como a maleabilidade de transações. Na *coinbase*, o *witness* serve como um valor reservado para construir o compromisso de *witness* root, essencial para a validação de blocos contendo transações SegWit.

Trecho de código onde essa validação ocorre:

```
const auto& witness_stack{block.vtx[0]->vin[0].scriptWitness.stack};
if (witness_stack.size() != 1 || witness_stack[0].size() != 32) {
    return state.Invalid(
        /*result=*/BlockValidationResult::BLOCK_MUTATED,
        /*reject_reason=*/"bad-witness-nonce-size",
        /*debug_message=*/strprintf("%s : invalid witness reserved value size",
    ↪ __func__));
}
```

3.1.4 bad-witness-merkle-match

O erro `bad-witness-merkle-match` refere-se a uma falha de validação em blocos que implementam o SegWit, conforme definido na BIP 141 [15]. O erro ocorre quando há uma discrepância entre o *witness commitment* presente na transação *coinbase* e o *Merkle root* (explicado em 3.1.2) calculado a partir dos dados de *witness* das transações no bloco.

O *witness commitment* é um componente essencial do SegWit, projetado para garantir a integridade e a verificabilidade dos dados de testemunha sem exigir que todos os nós armazenem ou processem informações desnecessárias. Especificamente, ele consiste em um *hash* de 32 *bytes* inserido como uma saída (*output*) na transação *coinbase* de um bloco. Essa saída segue o formato OP_RETURN seguido pelo prefixo `aa21a9ed` e o *Merkle root* dos *witnesses*.

Trecho de código onde essa validação ocorre:

```
uint256 hash_witness = BlockWitnessMerkleRoot(block, /*mutated=*/nullptr);
CHash256().Write(hash_witness).Write(witness_stack[0]).Finalize(hash_witness);
if (memcmp(hash_witness.begin(), &block.vtx[0]->vout[commitpos].scriptPubKey[6], 32))
    ↪ {
    return state.Invalid(
        /*result=*/BlockValidationResult::BLOCK_MUTATED,
        /*reject_reason=*/"bad-witness-merkle-match",
        /*debug_message=*/strprintf("%s : witness merkle commitment mismatch",
    ↪ __func__));
}
```

3.1.5 unexpected-witness

O erro `unexpected-witness` ocorre quando uma transação contém dados de *witness* (usados no protocolo SegWit para separar assinaturas do corpo da transação) em um bloco que não possui um *witness commitment* na transação *coinbase*. Conforme indicado no código-fonte do Bitcoin Core, em `validation.cpp`, blocos sem esse compromisso não devem incluir transações com dados de *witness*, pois isso poderia permitir *spam* na rede, explorando espaço de bloco sem validação adequada [19].

Trecho de código onde essa validação ocorre:

```
if (expect_witness_commitment) {
    ...
    return true;
}
for (const auto& tx : block.vtx) {
    if (tx->HasWitness()) {
        return state.Invalid(
            /*result=*/BlockValidationResult::BLOCK_MUTATED,
            /*reject_reason=*/"unexpected-witness",
            /*debug_message=*/strprintf("%s : unexpected witness data found",
    ↪ __func__));
    }
}
```

3.2 Geração com BTC Traffic

O *BTC Traffic* foi desenvolvido com o objetivo primário de gerar tráfego de dados controlado entre múltiplos nós da rede Bitcoin, incluindo um *honeynode* dedicado ao moni-

toramento passivo. Essa ferramenta permite simular interações de propagação de blocos e transações em um ambiente isolado, facilitando a captura e análise de eventos anômalos na rede [25].

Para validar a capacidade do *honeynode* em detectar blocos *mutated*, adotou-se o repositório *BTC Traffic* como prova de conceito (POC). A estratégia inicial consistia em configurar nós gerenciados pelo programa para minerar blocos válidos periodicamente e, intencionalmente, produzir blocos *mutated* em intervalos específicos. Esses blocos anômalos seriam propagados aos pares conectados e, consequentemente, capturados pelo *honeynode* para registro e análise.

Um aspecto crítico na implementação refere-se ao processo de validação inerente ao *Bitcoin Core*. Antes de aceitar e propagar um novo bloco, cada nó executa verificações rigorosas definidas no arquivo `validation.cpp` [19]. Para possibilitar a geração de blocos *mutated*, foi necessária uma *build* personalizada do *Bitcoin Core*, na qual as rotinas de validação responsáveis por rejeitar tais anomalias foram comentadas ou desabilitadas. Essa modificação permitiu que os nós produtores criassem e anunciassem blocos inválidos sem interrupção imediata.

Os tipos de blocos *mutated* descritos na Seção 3.1 foram produzidos e propagados com sucesso, conforme esperado, com exceção do variante *bad-txns-duplicate* (Seção 3.1.2). Esse tipo específico envolve a inclusão de transações que realizam *double-spending* de um mesmo UTXO (*Unspent Transaction Output*), o que demandaria alterações extensivas no código-fonte, incluindo modificações no gerenciamento de *mempool* e na lógica de consenso para simular nós maliciosos de forma realista [1].

Diante dessa limitação, optou-se por uma simplificação na POC: em vez de configurar nós mal intencionados para gerar blocos anômalos automaticamente, adotou-se um procedimento manual para forçar a produção do *hash* do bloco *mutated*. Utilizando *hashes* de blocos válidos previamente capturados, modificou-se manualmente o conteúdo para introduzir a duplicidade de transações. Em seguida, esses blocos são submetidos à rede por meio do comando *submitblock* do *bitcoin-cli*, permitindo a propagação controlada e a captura pelo *honeynode*.

Essa abordagem, embora mais cansativa e menos automatizada, comprovou a eficácia do sistema de monitoramento em identificar e registrar eventos de blocos *mutated*, incluindo o tipo *bad-txns-duplicate*.

3.3 Log Extractor

Conforme mencionado na Seção 1.6, a detecção de blocos *mutated* no *honeynode* depende da identificação de eventos de validação inválidos registrados exclusivamente nos logs do

Bitcoin Core. A análise do código-fonte, em particular do módulo `validation.cpp` [19], revela que rejeições (ou aceitações) de blocos são reportadas por meio de mensagens de *log* com um objeto de estado (*state*), contendo o estado final da validação, como "*Valid*" ou "*bad-txns-duplicate*".

Ferramentas existentes no *peer-observer*, como os extratores EBPF (baseado em *tracepoints* do *kernel*), RPC (consulta periódica via interface remota) e P2P (monitoramento de conexões de entrada), não capturam essas mensagens de *log* [11]. Torna-se, portanto, necessária a implementação de um extrator dedicado aos *logs* do nó, responsável por consumir continuamente as saídas de `debug.log`, processá-las, classificá-las e publicá-las no sistema de mensageria NATS.

A arquitetura do *Log Extractor* adota um *Unix named pipe* (FIFO – *first-in/first-out*) como mecanismo de entrada de dados. Esse recurso permite redirecionar a saída de logs do *bitcoind* para um arquivo especial no sistema de arquivos, acessível como um fluxo de leitura contínua. A escolha pelo *named pipe* simplifica a configuração do ambiente, pois elimina a necessidade de redirecionamentos complexos no comando de inicialização do nó. Duas abordagens de integração foram consideradas:

1. **Redirecionamento direto na execução do *bitcoind***: o `stdout` ou `stderr` é vinculado ao *pipe* por meio de redirecionamento no *shell* (`bitcoind ...> /path/to/pipe`).
2. **Monitoramento do arquivo `debug.log` via `tail -f`**: o comando `tail -f /path/to/debug.log > /path/to/pipe` alimenta o *pipe* com novas linhas à medida que são gravadas no disco.

O extrator opera em um loop assíncrono, verificando continuamente a disponibilidade de novas linhas no *pipe*. Ao receber uma linha, realiza:

1. **Exemplo**: `2025-09-27T01:52:01Z [validation] Enqueuing BlockConnected: block hash=41109f...952f0b block height=437`
2. **Parsing inicial**: extração de campos comuns, incluindo *timestamp* (no formato `YYYY-MM-DDTHH:MM:SSZ`), categoria de *debug* (ex.: `[net]`, `[validation]`) se disponível, e o corpo da mensagem.
3. **Classificação via *log matchers*** (Seção 3.3.1): aplicação de expressões regulares para identificar padrões relevantes, como `BlockConnected` ou `BlockChecked`.
4. **Processamento estruturado**: para *logs* classificados, coleta de metadados específicos, como o *hash* do bloco, altura (*height*), motivo da rejeição, *peer* origem (quando disponível), etc.

5. **Publicação no NATS**: os dados estruturados são serializados com o Protobuf e enviados a um *subject* do Log Extractor. Logs não reconhecidos ainda assim são publicados em um formato mais genérico para fins de auditoria.

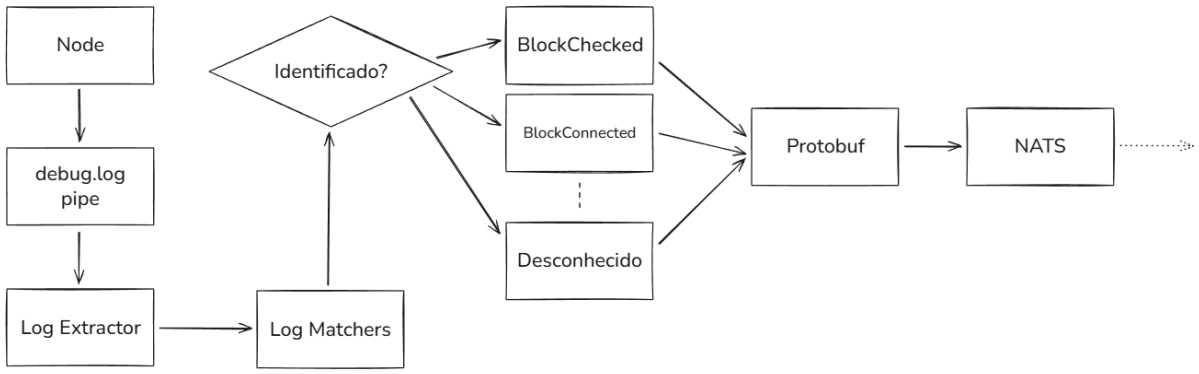

Figura 3.2: Fluxo dos *logs* no Log Extractor.

Essa abordagem garante baixa latência na detecção e robustez frente a picos de tráfego de *logs*. Testes em ambiente controlado com o *BTC Traffic* confirmaram a captura de todos os eventos de blocos *mutated* gerados (Seção 3.2).

Desenvolvido e mesclado no *pull request* #272 [12], a implementação do *Log Extractor* preenche uma lacuna no ecossistema do *peer-observer*, abrindo portas ao monitoramento em tempo real de métricas que são somente registradas nos *logs*.

3.3.1 Log Matchers

Os *Log Matchers* constituem um componente central do *Log Extractor*, responsável pela classificação e pelo processamento estruturado das linhas de *log* geradas pelo *Bitcoin Core*. Esse módulo opera como uma camada de abstração que transforma mensagens textuais em eventos tipados, compatíveis com o formato *Protocol Buffers* (*protobuf*), facilitando a serialização e a análise subsequente dos dados.

Interface e Funcionamento

A interface dos *Log Matchers* é definida de forma minimalista, composta por um único método:

```
fn parse_event(line: &str) -> Option<Event>;
```

O método recebe uma linha de *log* como `String` e retorna `Option<Event>`. Quando o valor retornado é `Some(Event)`, indica que o *matcher* reconheceu e *parseou* corretamente

a mensagem; caso contrário, `None` sinaliza que o padrão não corresponde, permitindo que a linha seja avaliada pelo próximo *matcher* na cadeia.

Essa abordagem segue o padrão *chain of responsibility*, promovendo extensibilidade e modularidade [26]. Novos classificadores podem ser implementados e inseridos em uma lista pré-existente de *Log Matchers* sem alterar o fluxo principal do extrator. Caso nenhum *matcher* corresponda à entrada, a linha é categorizada como "desconhecida" (*unknown*), preservando a integridade do pipeline e possibilitando análise posterior de eventos não mapeados.

Manutenibilidade e Fragilidade de Logs

Diferentemente de regras de negócio estáveis, os formatos de *log* do *Bitcoin Core* estão sujeitos a alterações frequentes entre versões do software. Atualizações no código-fonte podem modificar a ordem dos campos, introduzir novos tokens ou alterar a semântica de mensagens existentes, comprometendo a compatibilidade dos *Log Matchers*.

Em sistemas centralizados, onde o formato dos dados é imposto por terceiros, tais mudanças demandariam adaptações reativas. No contexto do *Bitcoin Core*, no entanto, o código é aberto e versionado. Uma estratégia inicial seria fixar o monitoramento a uma versão específica do nó (por exemplo, v29.0). Contudo, essa prática é contraproducente em um ambiente de pesquisa de longo prazo, pois limita a capacidade de acompanhar a evolução da rede.

Para mitigar esse risco, adota-se uma estratégia de testes automatizados em múltiplos níveis:

- **Testes unitários**: utilizam exemplos reais de linhas de *log* extraídas de execuções controladas do *Bitcoin Core*. Cada *matcher* é validado contra um conjunto de casos positivos e negativos, garantindo cobertura de variações conhecidas.
- **Testes de integração**: executam um nó em `regtest`, simulando cenários reais como recepção de blocos, propagação de transações e condições de *mempool*. Dessa forma, podemos testar versões diferentes e garantir que oque temos implementado continua funcionando.

Esses testes permitem a detecção precoce de quebras de compatibilidade durante atualizações do *Bitcoin Core*.

3.4 Detecção

Para ilustrar os padrões de *logs* gerados pelo Bitcoin Core ao rejeitar blocos *mutated*, apresentam-se a seguir exemplos extraídos de execuções controladas no modo *regtest*.

Esses *logs* foram obtidos ao ativar categorias de depuração como `validation`, permitindo a captura de mensagens detalhadas sobre os erros de validação. Cada exemplo corresponde a um tipo específico de erro discutido na Seção 3.1, destacando o *timestamp*, a categoria de *log*, o *hash* do bloco e o motivo da rejeição.

Entre as mensagens registradas, o *log* com o prefixo `BlockChecked` destaca-se por aparecer consistentemente em todos os casos de rejeição de blocos *mutated*, fornecendo informações mais completas e estruturadas em comparação com outras entradas, como `ProcessNewBlock`. Essa linha inclui o estado final da validação (`state=`), o *hash* do bloco e uma descrição textual do erro, tornando-a o alvo principal para detecção no Log Extractor. Ao focar nessa mensagem, garante-se maior robustez na identificação, uma vez que ela é emitida após a conclusão do processo de `ConnectBlock` no arquivo `validation.cpp` [19], independentemente do tipo específico de mutação.

Os padrões a seguir servem como base para a classificação via expressões regulares no Log Extractor (Seção 3.3).

bad-txnmrklroot (Seção 3.1.1) Esse erro indica uma inconsistência na *merkle root* calculada.

```
2025-10-28T02:18:37Z [validation] BlockChecked: block
    ↪ hash=3909cd2a5ff36b9a40368609f92945e5b7111bca3cb4d04b72c39964aeb5d156
    ↪ state=bad-txnmrklroot, hashMerkleRoot mismatch
2025-10-28T02:18:37Z [error] ProcessNewBlock: AcceptBlock FAILED
    ↪ (bad-txnmrklroot, hashMerkleRoot mismatch)
```

bad-txns-duplicate (Seção 3.1.2) Relacionado à vulnerabilidade CVE-2012-2459, esse *log* captura duplicações de transações que preservam o *hash* do bloco [20].

```
2025-10-28T02:20:12Z [validation] BlockChecked: block
    ↪ hash=1811952798ccd2ad1264b064c3da6313fba7497159b999ee045afb046c7d0232
    ↪ state=bad-txns-duplicate, duplicate transaction
2025-10-28T02:20:12Z [error] ProcessNewBlock: AcceptBlock FAILED
    ↪ (bad-txns-duplicate, duplicate transaction)
```

bad-witness-nonce-size (Seção 3.1.3) Esse padrão reflete violações na ausência ou tamanho do *witness* na *coinbase*, conforme BIP 141 [15].

```
2025-10-28T02:21:04Z [error] AcceptBlock: bad-witness-nonce-size,
    ↪ CheckWitnessMalleation : invalid witness reserved value size
2025-10-28T02:21:04Z [validation] BlockChecked: block
    ↪ hash=7e38f165598ceec2178db6f063e7745c2bdd4952d52bea44bfe069c686ea5a55
```

```
    ↪ state=bad-witness-nonce-size, CheckWitnessMalleation : invalid witness
    ↪ reserved value size
2025-10-28T02:21:04Z [error] ProcessNewBlock: AcceptBlock FAILED
    ↪ (bad-witness-nonce-size, CheckWitnessMalleation : invalid witness
    ↪ reserved value size)
```

bad-witness-merkle-match (Seção 3.1.4) Indica discrepância no *commitment* do *witness*, essencial para a integridade do SegWit [15].

```
2025-10-28T02:21:51Z [error] AcceptBlock: bad-witness-merkle-match,
    ↪ CheckWitnessMalleation : witness merkle commitment mismatch
2025-10-28T02:21:51Z [validation] BlockChecked: block
    ↪ hash=117dd69bd553742a13bd5fd094132c64b31dc33836711f72503bef22d714af02
    ↪ state=bad-witness-merkle-match, CheckWitnessMalleation : witness merkle
    ↪ commitment mismatch
2025-10-28T02:21:51Z [error] ProcessNewBlock: AcceptBlock FAILED
    ↪ (bad-witness-merkle-match, CheckWitnessMalleation : witness merkle
    ↪ commitment mismatch)
```

unexpected-witness (Seção 3.1.5) Esse erro surge quando dados de *witness* são encontrados em blocos sem o *commitment* apropriado, potencialmente explorando *spam* na rede.

```
2025-10-28T02:22:28Z [error] AcceptBlock: unexpected-witness,
    ↪ CheckWitnessMalleation : unexpected witness data found
2025-10-28T02:22:28Z [validation] BlockChecked: block
    ↪ hash=0ffc0b771bce4405645c1b1584bf04bb3c1a084cd4950784df9bc560d7e75a3b
    ↪ state=unexpected-witness, CheckWitnessMalleation : unexpected witness
    ↪ data found
2025-10-28T02:22:28Z [error] ProcessNewBlock: AcceptBlock FAILED
    ↪ (unexpected-witness, CheckWitnessMalleation : unexpected witness data
    ↪ found)
```

3.5 Metrificação

A metrificação dos eventos detectados pelo Log Extractor é realizada por meio da ferramenta *metrics* integrada ao Peer Observer, que se inscreve em todos os canais do servidor NATS para capturar e processar dados em tempo real [11]. Essa ferramenta consome as mensagens publicadas pelos extratores, incluindo o Log Extractor desenvolvido neste trabalho, e as decodifica utilizando o formato Protobuf.

Ao receber um evento, a ferramenta *metrics* realiza uma análise baseada no tipo de mensagem. No contexto deste estudo, o foco reside nos eventos do tipo *LogExtractorEvent*, com ênfase no subtipo *BlockCheckedLog*, que encapsula as informações sobre a validação de blocos, incluindo rejeições por mutações. Para quantificar esses eventos, implementou-se um contador no Prometheus, um sistema de monitoramento de séries temporais amplamente adotado em ambientes distribuídos [27]. Esse contador incrementa para cada ocorrência de *BlockCheckedLog*, permitindo o rastreamento geral da atividade de validação de blocos no nó monitorado.

Adicionalmente, para uma análise mais granular das anomalias, desenvolveu-se um contador específico para blocos *mutated*. Esse métrica utiliza *labels* para categorizar o tipo de erro identificado, como os tipos descritos na Seção 3.1. Essa abordagem facilita a identificação de padrões de ataques ou falhas, permitindo alertas configuráveis no Prometheus.

As métricas processadas são exportadas continuamente para o servidor Prometheus, que armazena os dados em formato de séries temporais. Essa integração possibilita a criação de painéis de visualização no Grafana, conforme será discutido na Seção 4.2. Testes em ambiente controlado, utilizando o BTC Traffic para simular blocos *mutated*, confirmaram a precisão da metrificação, com incrementos corretos nos contadores correspondentes a cada tipo de rejeição observada nos *logs* [25], abaixo exemplificamos os dados no formato disponibilizado pelo Prometheus via HTTP.

Listagem 3.1: Dados do Prometheus

```
# HELP peerobserver_log_mutated_blocks Number of mutated blocks detected by status.
# TYPE peerobserver_log_mutated_blocks counter
peerobserver_log_mutated_blocks{status="bad-txnmrklroot"} 1
peerobserver_log_mutated_blocks{status="bad-txns-duplicate"} 2
peerobserver_log_mutated_blocks{status="bad-witness-merkle-match"} 3
peerobserver_log_mutated_blocks{status="bad-witness-nonce-size"} 1
peerobserver_log_mutated_blocks{status="unexpected-witness"} 1
```

Capítulo 4

Resultados

4.1 Cenário de teste

Como mencionado no Capítulo 1 e com os desenvolvimentos detalhados no Capítulo 3, precisamos encaixar todas as peças e comprovar que a extensão que nos comprometemos a montar do peer-observer funciona.

Para realizar esse teste, é necessário configurar um ambiente básico seguindo a sequência de passos a seguir, garantindo a integração entre as ferramentas de monitoramento e visualização de dados:

- Inicie o *Log Extractor* conforme as instruções detalhadas na documentação do repositório peer-observer [11]. Neste momento, utilize a *branch* associada ao *pull request* #287, uma vez que as alterações relacionadas aos blocos *mutated* ainda não foram incorporadas à *branch* principal [28];
- Execute um nó do Bitcoin Core no modo `-regtest`, ativando a opção de depuração para validação (`-debug=validation`) e configurando o *pipe* apropriado para integração com o extrator de logs;
- Ative a ferramenta de métricas (*metrics tool*) do peer-observer, conforme orientado na respectiva documentação [11], para capturar e processar os dados;
- Inicie o servidor NATS, responsável pela comunicação assíncrona entre os componentes do sistema;
- Ative o servidor Prometheus, que atuará como banco de dados de séries temporais para armazenar as métricas coletadas;
- Inicie o Grafana, a plataforma de visualização que permitirá a análise gráfica dos dados;

- Importe os painéis pré-configurados na pasta `tools/metrics/dashboards` para o Grafana, facilitando a exibição e interpretação das informações monitoradas.

Com todo o ferramental executando, usamos o **BTC Traffic** para gerar os blocos com seus respectivos erros.

Listagem 4.1: Saídas retornadas pelo BTC Traffic

```
Block raw hex (bad-txnmrklroot):
    ↪ 0400000042fa9cce0002c01fbc1a44f1...915ef8a528388221f71352b300000000
Block raw hex (bad-txns-duplicate):
    ↪ 0400000042fa9cce0002c01fbc1a44f1...915ef8a528388221f71352b300000000
Block raw hex (bad-witness-nonce-size):
    ↪ 0400000042fa9cce0002c01fbc1a44f1...915ef8a528388221f71352b300000000
Block raw hex (bad-witness-merkle-match):
    ↪ 0400000042fa9cce0002c01fbc1a44f1...915ef8a528388221f71352b300000000
Block raw hex (unexpected-witness):
    ↪ 0400000042fa9cce0002c01fbc1a44f1...915ef8a528388221f71352b300000000

You can use 'bitcoin-cli -regtest submitblock <hex>' to submit it.
```

Após isso, basta executar o comando `submitblock` do CLI do bitcoin core usando os blocos em hexadecimal como mencionado na saída do BTC Traffic 4.1. Assim que submetemos o bloco, podemos checar a rota do servidor prometheus para confirmar se os erros foram contabilizados, o esperado é ver algo similar como na Listagem 3.1. E de maneira visual, podemos acessar o *dashboard* configurado no grafana para observar os resultados, as figuras 4.1 e 4.2 da próxima seção ilustram isso.

4.2 Visualização

A visualização dos dados coletados é essencial para a análise do comportamento da rede Bitcoin, especialmente no contexto da detecção de blocos *mutated*. Devido à raridade desses eventos, as ferramentas de visualização devem ser projetadas para capturar e apresentar informações de longo prazo, permitindo a identificação de padrões infrequentes. Como destacado na literatura, blocos *mutated* representam uma forma de ataque que explora vulnerabilidades na validação de blocos, como descrito na CVE-2012-2459 [20], onde a manipulação de transações do bloco que não modificam o cabeçalho do bloco pode levar a rejeições e desconexões na rede.

A raridade dos casos de blocos *mutated* decorre do mecanismo de defesa implementado nos nós da rede Bitcoin. Ao receber um bloco inválido, um nó adiciona o peer remetente a uma lista de bloqueio (*blacklist*) temporária, conforme implementado no código de

validação do Bitcoin Core [19]. Essa medida reduz a conectividade de nós atacantes, tornando mais difícil a propagação de blocos manipulados em larga escala.

Para a visualização, utilizou-se o Grafana, uma plataforma open-source para monitoramento e análise de métricas. Dado o caráter esporádico dos eventos, optou-se por dashboards que abrangem "toda a história" dos dados coletados, em vez de janelas temporais curtas, como na Figura 4.1.

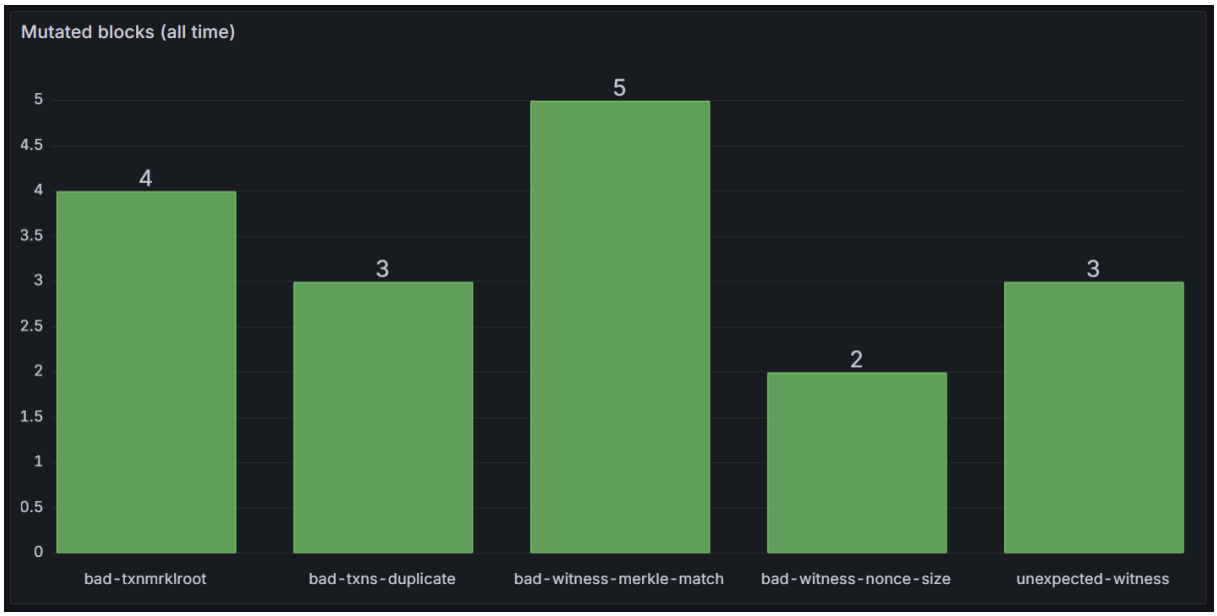

Figura 4.1: Blocos mutated capturados em toda história.

Essa abordagem permite a detecção de anomalias raras, integrando métricas como o número de rejeições por tipo de mutação (e.g., *bad-txnmrklroot*, *bad-txns-duplicate*). Os dados extraídos via Log Extractor e metrificados são exportados para um banco de dados temporal, como o Prometheus, facilitando queries históricas e alertas em tempo real, na Figura 4.2 exemplificamos como sabemos quando um desses eventos aconteceu, por mais que esse tipo de gráfico vá ficar a maior parte do tempo sem dados devido a característica desse tipo de evento.

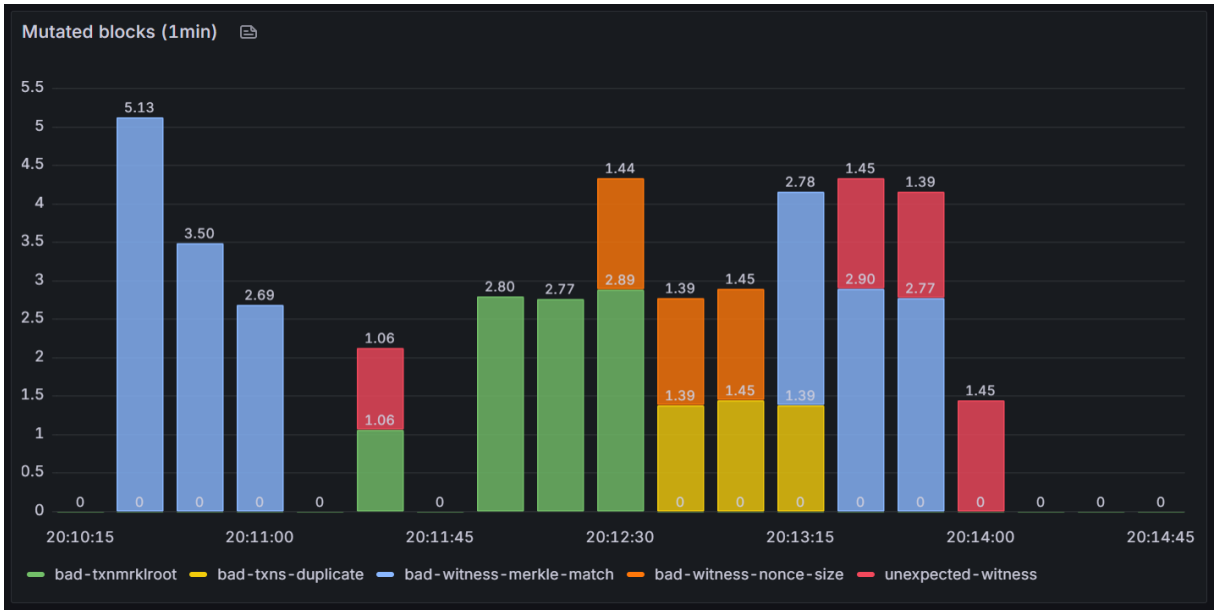

Figura 4.2: Blocos mutated capturados com intervalo de 1 minuto.

Capítulo 5

Conclusão

A rede Bitcoin foi projetada para garantir a imutabilidade dos blocos uma vez confirmados na *blockchain*. Contudo, o protocolo P2P permite que um nó anuncie um bloco cujo cabeçalho é válido, mas cujas transações foram alteradas de forma que o mesmo permaneça inalterado – anomalia conhecida como *mutated block* (Seção 1.5).

Esse comportamento, explorado na vulnerabilidade CVE-2012-2459 [20, 24], poderia ser usado para ataques de negação de serviço (DoS) contra versões antigas do Bitcoin Core – onde não havia verificações nesse sentido. O monitoramento e detecção desse tipo de evento é do interesse dos participantes da rede, uma vez que ajuda a garantir o desempenho adequado e a segurança das operações.

Como descritos na Seção 1.6 e trabalhado mais detalhadamente no Capítulo 3, este trabalho alcançou grande parte dos objetivos propostos.

Implementação de uma ferramenta que simule uma rede com um nó malicioso, que gera os blocos modificados e permita um nó vítima recebê-los. Este ponto foi parcialmente contemplado com a implementação do BTC Traffic [25], devido a uma dificuldade de personalização da implementação do bitcoin core, tivemos que simplifica-lo a uma geração de blocos modificados sem a simulação de uma rede com vários nós – sendo um deles malicioso. Essa limitação se deu devido ao erro `bad-txns-duplicate`, que embora seja possível gerarmos um bloco desse tipo, a submissão dele por um nó requereria mais modificações para ignorar erros no sentido de *double-spending*, conforme discutido na Seção 3.2.

O desenvolvimento e validação do extrator de logs para captura de dados. Este ponto foi concluído, com contribuições diretas via pull request #272 [12]. Esse componente faz essas capturas por meio de um *pipe unix* ligado aos *logs* publicados pelo nó, seja por *stdout* ou ao arquivo de *logs* (`debug.log`), como detalhado na Seção 3.3. Se ligando à arquitetura já estabelecida pelo projeto, onde os extratores mandam dados

serializados no formato *protobuf* ao sistema de mensageria NATS, para que possam ser consumidos pelas ferramentas (*tools*).

O desenvolvimento de um processador de logs, capaz de classifica-los de modo genérico ou específico, possibilitando a identificação dos logs alvo. Complementando ao *Log Extractor*, este ponto também foi alcançado por meio do que chamamos de *Log Matchers*. Essa interface em comum foi útil para termos várias implementações de *parsers* de *logs*, onde um deles foi para identificar blocos *mutated*. Usando de expressões regulares para identificar os *logs* desejados, assim como um *parsing* genérico para o restante não identificado, como descrito na Seção 3.3.1. Para reforçar a robustez da implementação, foi implementado tanto testes unitários como de integração que simulam nós do Bitcoin Core, garantindo que futuras alterações no formato dos *logs* não quebrem o monitoramento.

Metrificação dos logs relacionados a blocos modificados. Contemplado por meio da ferramenta *metrics* já implementada no arsenal do *peer-observer*. Nesse lugar, capturamos os eventos publicados no NATS relacionados ao *Log Extractor*, usamos um contador do *prometheus* para agregar os *logs* do tipo `BlockChecked`, juntamente de uma *label* que guarda a categoria do erro identificado. Mais detalhes na Seção 3.5.

Uma visualização básica dos eventos coletados. Também se aproveitando dos dados gerados pela ferramenta *metrics* que são publicados no servidor *prometheus*, contemplamos esse ponto ao usar o *grafana* que consome esses dados e montam os gráficos, demonstrados no Capítulo 4.

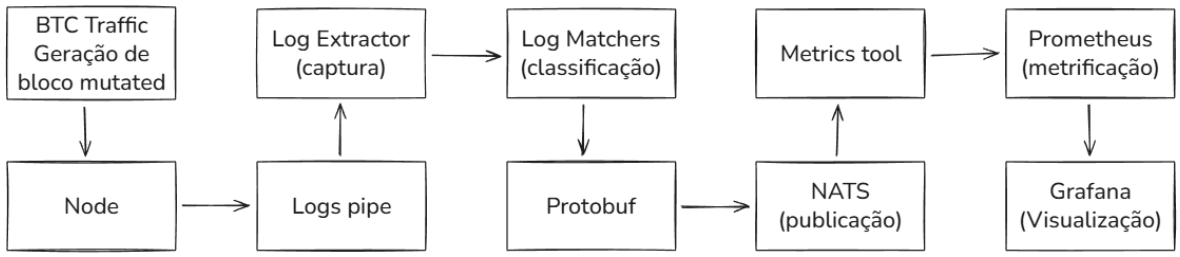

Figura 5.1: Fluxo geral dos *logs*.

Com isso, a extensão proposta transforma o *peer-observer* em uma ferramenta completa de monitoramento ativo de blocos modificados, capaz de detectar em tempo real tentativas de exploração da vulnerabilidade, fornecendo dados quantitativos que antes não estavam disponíveis publicamente.

5.1 Sugestões de trabalhos futuros

Embora o presente trabalho tenha alcançado os objetivos propostos — extensão da implementação do peer-observer para detectar e registrar blocos modificados (*mutated blocks*) propagados na rede Bitcoin, com identificação do tipo de erro, quando e que *hash* do bloco foi —, o projeto atual abre caminho principalmente para coleta de dados e análise.

O primeiro ponto consiste na **coleta de dados por um período prolongado** (pelo menos 6-12 meses contínuos). A observação atual, concentrada apenas em ambientes controlados (*regtest*), não permite afirmar com segurança a frequência real desses eventos nem capturar variações sazonais ou correlações com atualizações do protocolo. Um monitoramento de longo prazo possibilitaria a construção de séries temporais robustas e a identificação de padrões de ataque que só emergem em escalas de tempo maiores.

Outra limitação atual é a presença de uma **blacklist de peers** que, após o envio de blocos inválidos, são desconectados temporariamente – tipicamente 24 horas. Embora essa medida proteja o honeynode de consumo excessivo de recursos, ela reduz a capacidade de observar comportamentos repetitivos dos atacantes (por exemplo, tentativas reiteradas com blocos diferentes ou evolução das técnicas de mutação). A **remoção ou substituição da blacklist** por uma política menos incisiva sobre blocos inválidos permitiria coletar um volume significativamente maior de amostras e compreender melhor a persistência dos atores maliciosos.

Uma extensão particularmente interessante seria a implantação de **múltiplos honeynodes geograficamente distribuídos** (América do Sul, América do Norte, Europa, Ásia e África). Essa abordagem permitiria:

- verificar se os blocos modificados são propagados globalmente ou apenas para subredes específicas;
- avaliar a eficácia de mecanismos de defesa do Bitcoin Core em diferentes versões e configurações de rede, com um nó "normal" para referência.

Por fim, em questão de infraestrutura para esses honeynodes, caso tenha poucos recursos, diria para ativar no mínimo a flag de logs de depuração `validation` para pegar esses eventos, juntamente com a de `net`, a fim de diminuir o volume de logs, por mais interessante que seja ativar todos. Como são eventos raros, depois de um período coletando dados poderíamos vasculhar os logs focando nos eventos pegos pelo Log Extractor e ver que endereços enviaram esses blocos já que no log do erro em si não é especificado, a flag de `net` provavelmente nos diria. Lembre-se que configurar um *log-rotate* é necessário enquanto mantemos um nó que tem volume grande de logs por longos períodos, a fim de evitar que o arquivo fique muito grande.

Essas sugestões não apenas ampliariam o escopo científico do projeto, mas também poderiam contribuir para o desenvolvimento de mecanismos de detecção precoce de ataques do tipo DoS baseados em blocos modificados, beneficiando toda a comunidade Bitcoin.

Referências

[1] Nakamoto, Satoshi: *Bitcoin: A peer-to-peer electronic cash system*. Whitepaper, 2008. `https://bitcoin.org/bitcoin.pdf`, acesso em 2025-10-07. 1, 2, 6, 17

[2] IEEE Spectrum Staff: *The future of the web looks a lot like the bitcoin blockchain*. IEEE Spectrum, julho 2015. `https://spectrum.ieee.org/the-future-of-the-web-looks-a-lot-like-bitcoin`, acesso em 2025-11-25. 1

[3] Mempool Space Team: *Mempool space: Bitcoin network explorer and mempool visualizer*. `https://mempool.space`, 2025. Ferramenta open-source para monitoramento em tempo real da mempool e da rede Bitcoin. 2

[4] Heilman, Ethan, Alison Kendler, Aviv Zohar e Sharon Goldberg: *Eclipse attacks on bitcoin's peer-to-peer network*. Em *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, página 129–144, USA, 2015. USENIX Association, ISBN 9781931971232. 2, 3, 6

[5] Bitnodes Project: *Reachable bitcoin nodes - bitnodes*, novembro 2025. `https://bitnodes.io/`, acesso em 2025-11-28. 3

[6] Eyal, Ittay e Emin Gün Sirer: *Majority is not enough: bitcoin mining is vulnerable*. Commun. ACM, 61(7):95–102, junho 2018, ISSN 0001-0782. `https://doi.org/10.1145/3212998`. 3

[7] Bitcoin Dev Project: *Warnet: Monitor and analyze the emergent behaviors of bitcoin networks*. GitHub Repository, 2024. `https://github.com/bitcoin-dev-project/warnet`, acesso em 2025-11-20. 4

[8] Franzoni, Federico, Xavier Salleras e Vanesa Daza: *Atom: Active topology monitoring for the bitcoin peer-to-peer network*, 2021. `https://arxiv.org/abs/2107.12912`. 4

[9] MIT Digital Currency Initiative: *Pool detective: Mining pool monitoring*. GitHub Repository, 2020. `https://github.com/mit-dci/pooldetective`, acesso em 2025-11-20. 4

[10] BitPay: *Bitcore: Open source developer tools for the bitcoin network*. GitHub Repository, 2013. `https://github.com/bitpay/bitcore`, acesso em 2025-11-20. 4

[11] 0xB10C: *Peer-observer: A tool for monitoring the bitcoin p2p network*. GitHub Repository, 2022. `https://github.com/0xB10C/peer-observer`, acesso em 2025-10-07. 4, 7, 9, 11, 18, 22, 24

[12] Fabio, Maycon V. S.: *Implement log extractor*. Pull Request #272 in Peer-Observer, outubro 2025. https://github.com/0xB10C/peer-observer/pull/272, acesso em 2025-11-20. 4, 19, 28

[13] 0xB10C: *monitoring wishlist – issue #8*. GitHub. https://github.com/0xB10C/project-ideas/issues/8, acesso em 2025-11-20. 4

[14] 0xB10C: *Viabtc's mutated blocks without witness data*, março 2024. https://b10c.me/observations/10-viabtc-blocks-without-witness-data/, acesso em 2025-11-20. 4

[15] Wuille, Pieter, Eric Lombrozo e Johnson Wu: *Segregated witness (consensus layer)*, 2015. https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki, Bitcoin Improvement Proposal 141. 4, 6, 15, 21, 22

[16] dergoegge: *p2p: Don't process mutated blocks*. Pull Request #29412, Bitcoin Core repository, fevereiro 2024. https://github.com/bitcoin/bitcoin/pull/29412, acesso em 2025-11-20, Opened on February 8, 2024. Merged into Bitcoin Core v27.0 as defense-in-depth against mutated block processing. 5, 6

[17] Bitcoin Developer Documentation: *P2p network*. Bitcoin Developer Reference, 2025. https://developer.bitcoin.org/reference/p2p_networking.html, acesso em 2025-12-17. 5, 6

[18] Corallo, Matt: *Bip 152: Compact block relay*. Bitcoin Improvement Proposal 152, 2016. https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki, acesso em 2025-12-17. 6

[19] Bitcoin Core: *validation.cpp*. https://github.com/bitcoin/bitcoin/blob/d20f10affba83601f1855bc87d0f47e9dfd5caae/src/validation.cpp, acesso em 2025-10-22. 7, 16, 17, 18, 21, 26

[20] National Vulnerability Database: *Cve-2012-2459: Bitcoin: block validation vulnerability*, 2012. https://nvd.nist.gov/vuln/detail/CVE-2012-2459, acesso em 2025-10-16. 7, 13, 14, 21, 25, 28

[21] Google Developers: *Protocol buffers documentation: Overview*. Protobuf Documentation, 2008. https://protobuf.dev/overview/, acesso em 2025-11-10. 10

[22] Collison, Derek: *Nats: Connective technology for adaptive edge & distributed systems*. NATS Documentation, 2010. https://docs.nats.io/nats-concepts/what-is-nats, acesso em 2025-11-10. 10

[23] Learn Me A Bitcoin: *Merkle root: A fingerprint for the transactions in a block*, 2025. https://learnmeabitcoin.com/technical/block/merkle-root/, acesso em 2025-10-14. 13

[24] kjj2: *Cve-2012-2459: block validation vulnerability*, 2012. https://bitcointalk.org/index.php?topic=102395.0, acesso em 2025-10-16. 14, 28

[25] Clube Bitcoin UnB: *btc-traffic: Ferramenta para geração de tráfego na rede bitcoin*, 2023. `https://github.com/ClubeBitcoinUnB/btc-traffic`, acesso em 2025-11-04. 17, 23, 28

[26] Gamma, Erich, Richard Helm, Ralph Johnson e John Vlissides: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995, ISBN 0201633612. 20

[27] Prometheus Authors: *Prometheus: Monitoring system & time series database*. Prometheus Documentation, 2012. `https://prometheus.io/docs/introduction/overview/`, acesso em 2025-11-12. 23

[28] Fabio, Maycon V. S.: *Capture mutated blocks from logs*. Pull Request #287 in Peer-Observer, 2025. `https://github.com/peer-observer/peer-observer/pull/287`, acesso em 2025-11-27. 24