

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIAS DA COMPUTAÇÃO

Cristian Alexandre Alchini

Análise de Ameaças e Vulnerabilidades em Blockchains Permissionadas

Florianópolis
2025

Cristian Alexandre Alchini

Análise de Ameaças e Vulnerabilidades em Blockchains Permissionadas

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciências da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciências da Computação.

Orientador: Lucas Machado da Palma, Dr.

Coorientador: Prof. Jean Everson Martina, Dr.

Florianópolis

2025

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Alchini, Cristian Alexandre
Análise de Ameaças e Vulnerabilidades em Blockchains
Permissionadas / Cristian Alexandre Alchini ; orientador,
Lucas Machado da Palma, coorientador, Jean Everson
Martina, 2025.
97 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Ciências da Computação, Florianópolis, 2025.

Inclui referências.

1. Ciências da Computação. 2. Segurança. 3. Blockchain.
4. Contratos Inteligentes. I. Machado da Palma, Lucas.
II. Martina, Jean Everson. III. Universidade Federal de
Santa Catarina. Graduação em Ciências da Computação. IV.
Título.

Cristian Alexandre Alchini
Análise de Ameaças e Vulnerabilidades em Blockchains Permissionadas

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciências da Computação e aprovado em sua forma final pelo curso de Graduação em Ciências da Computação.

Florianópolis, 10 de novembro de 2025.

Prof^a Lúcia Helena Martins Pacheco, Dr
Coordenador do Curso

Banca Examinadora:

Lucas Machado da Palma, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Jean Everson Martina, Dr.
Avaliador
Universidade Federal de Santa Catarina

Larissa Gremelmaier Rosa
Avaliador
Universidade Federal de Santa Catarina

Arthur Gabriel Crippa Milanez
Avaliador
Universidade Federal de Santa Catarina

RESUMO

A tecnologia blockchain tem se mostrado uma ferramenta promissora para cenários que exigem alta confiabilidade, integridade dos dados e descentralização. Para além das blockchains públicas, as blockchains permissionadas oferecem vantagens como controle de acesso, maior eficiência e transparência, sendo especialmente adequadas para aplicações que lidam com dados sensíveis e exigem alto nível de segurança e desempenho. A medida que esses sistemas se tornam mais complexos, passam também a estar mais suscetíveis a vulnerabilidades e ataques, que podem comprometer sua segurança. Este trabalho tem como objetivo analisar as principais vulnerabilidades, ameaças e tipos de ataques em blockchains permissionadas. A partir de uma revisão sistemática da literatura são identificados as principais vulnerabilidades apontadas pela comunidade científica, e a partir desta análise, propõe-se experimentos práticos de detecção de vulnerabilidades em contratos inteligentes, direcionados a um caso de uso concreto, visando avaliar a performance dos mecanismos de detecção. Ao final, espera-se contribuir para o desenvolvimento de práticas mais robustas e seguras para aplicações sobre blockchains permissionadas.

Palavras-chave: Blockchain. Vulnerabilidades. Contratos Inteligentes. Ataques. Segurança.

ABSTRACT

Blockchain technology has proven to be a promising tool for scenarios that require high reliability, data integrity, and decentralization. Beyond public blockchains, permissioned blockchains offer advantages such as access control, greater efficiency, and transparency, being especially suitable for applications that handle sensitive data and require high levels of security and performance. As these systems become more complex, they also become more susceptible to vulnerabilities and attacks that can compromise their security. This work aims to analyze the main vulnerabilities, threats, and types of attacks in permissioned blockchains. Through a systematic literature review, the main vulnerabilities identified by the scientific community are determined, and based on this analysis, practical experiments for detecting vulnerabilities in smart contracts are proposed, focused on a concrete use case, aiming to evaluate the performance of detection mechanisms. Ultimately, this work is expected to contribute to the development of more robust and secure practices for applications on permissioned blockchains.

Keywords: Blockchain. Vulnerabilities. Smart Contracts. Attacks. Security.

SUMÁRIO

1	INTRODUÇÃO	9
1.1	ORGANIZAÇÃO DO TRABALHO	11
2	OBJETIVOS	12
2.1	OBJETIVO GERAL	12
2.2	OBJETIVOS ESPECÍFICOS	12
3	MÉTODO DE PESQUISA	13
4	CONCEITOS FUNDAMENTAIS	14
4.1	REDES PAR-A-PAR (P2P) E CHAMADA DE PROCEDIMENTO REMOTO (RPC)	14
4.2	TOKENS E CRIPTOMOEDAS	14
4.3	TRANSAÇÃO	14
4.4	CRIPTOGRAFIA	15
4.5	CONTRATOS INTELIGENTES	15
4.6	PROTOCOLO DE CONSENSO	15
4.7	TIPOS DE BLOCKCHAINS	16
4.7.1	Blockchains Públicas	16
4.7.2	Blockchains Permissionadas	16
4.8	PRINCIPAIS IMPLEMENTAÇÕES DE BLOCKCHAIN	16
4.8.1	Bitcoin	16
4.8.2	Ethereum	17
4.8.3	Hyperledger	17
4.8.4	Hyperledger Fabric	17
4.8.5	Hyperledger Besu	17
4.9	TIPOS DE PROTOCOLOS DE CONSENSO	18
4.9.1	Protocolos em blockchains públicas	18
4.9.1.1	<i>Proof of Work (PoW)</i>	18
4.9.1.2	<i>Proof of Stake (PoS)</i>	18
4.9.2	Protocolos em blockchains permissionadas	19
4.9.2.1	<i>Protocolos Tolerantes a Falhas</i>	19
4.9.2.2	<i>Raft</i>	19
4.9.2.3	<i>Protocolos Tolerantes a Falhas Bizantinas</i>	19
4.9.2.4	<i>Practical Byzantine Fault Tolerance (PBFT)</i>	20
4.10	INFRAESTRUTURA DE CHAVE PÚBLICA (PKI) E AUTORIDADES CERTIFICADORAS (CA)	20
4.11	CONCEITOS EM HYPERLEDGER FABRIC	21

4.11.1	Estrutura de uma rede Fabric	21
4.11.2	Membership Service Provider (MSP)	22
4.11.3	Private Data Collections (PDC)	22
4.12	ATAQUES	23
4.12.1	Ataque de Negação de Serviço Distribuído (DDoS)	23
4.12.2	Ataque Sybil	24
4.12.3	Ataques de 51 %	24
4.12.4	Ataques de Gasto Duplo	24
5	REVISÃO BIBLIOGRÁFICA	25
5.1	RESUMO DOS TRABALHOS	26
6	DESENVOLVIMENTO	37
6.1	VULNERABILIDADES E ATAQUES EM BLOCKCHAINS PERMISSIO- NADAS	37
6.1.1	Vulnerabilidades em componentes de infraestrutura e rede	37
6.1.2	Vulnerabilidades em Contratos Inteligentes	42
6.1.3	Medidas de prevenção em contratos inteligentes	46
6.2	EXPERIMENTOS	47
6.2.1	Análise de Contratos Inteligentes	47
6.2.2	Caso de uso: RRDES	48
6.2.3	Ferramentas de Análise Utilizadas	49
6.2.3.1	<i>PDChecker</i>	50
6.2.3.2	<i>HFCCT</i>	51
6.2.3.3	<i>Golangci-lint</i>	51
6.2.4	Metodologia de avaliação	52
6.2.4.1	<i>Chaincode experimental</i>	53
6.3	RESULTADOS	56
6.3.1	Chaincode experimental	56
6.3.2	Chaincodes RRDES	57
6.4	LIMITAÇÕES DAS FERRAMENTAS ANALISADAS E POSSÍVEIS ME- LHORIAS	61
7	CONCLUSÃO	66
	REFERÊNCIAS	68
	APÊNDICES	70
	APÊNDICE A – CÓDIGO FONTE	71

APÊNDICE B – ARTIGO EM FORMATO SBC	81
---	-----------

1 INTRODUÇÃO

O conceito de blockchain foi apresentado pela primeira vez por Satoshi Nakamoto (Nakamoto, 2008), descrevendo uma estrutura que funciona como um livro-razão. Essa estrutura é composta por blocos que armazenam dados e estão interligados de modo a formar uma cadeia. A ligação entre os blocos ocorre da seguinte maneira: cada bloco, exceto o primeiro, chamado de bloco gênese, possui um mecanismo criptográfico que referencia o conteúdo do bloco anterior, chamado de hash. Caso os dados de um bloco sejam alterados, o hash do bloco subsequente se torna inválido, comprometendo toda a cadeia. Essas características garantem a integridade e a imutabilidade da rede.

Além disso, a blockchain opera de maneira distribuída, onde cada nó da rede mantém uma cópia completa de toda a cadeia de blocos. Para garantir a consistência da cadeia entre os diferentes nós, é utilizado um protocolo de consenso que assegura que todos os participantes possuam o mesmo registro e concordem com a inserção de novas informações. Essa característica proporciona confiança aos participantes da rede, permitindo uma abordagem descentralizada e eliminando a necessidade de uma autoridade central (Vieira; Velasco; Carvalho, 2023).

As blockchains públicas, como por exemplo, a rede Bitcoin¹, não requerem autorização para participação, permitindo que qualquer pessoa possa realizar transações na rede. Além disso, essas redes são abertas e auditáveis. Já blockchains permissionadas, são redes restritas, nas quais o acesso é controlado por uma entidade ou grupo de entidades confiáveis, não sendo acessíveis a qualquer participante. Apenas usuários com permissão podem atuar na rede. Um exemplo de ferramenta no contexto destas blockchains é o Hyperledger Fabric, da Linux Foundation (Hyperledger Foundation, 2025b).

Blockchains permissionadas são utilizadas em contextos onde os participantes da rede são conhecidos e possuem identidades para participar da rede, mas onde não necessariamente há confiança entre as partes. Contextos onde é necessário que se saiba quais são as entidades responsáveis pelas operações realizadas no sistema, mas onde as informações não estejam disponíveis para toda a internet. Ou seja, estejam restritas ao grupo de participantes. Além disso, por possuírem um número limitado de membros, possuem a capacidade de adotar ferramentas que são mais eficientes, com maiores taxas de transação e menor latência.

Buscando essas características, as blockchains permissionadas estão sendo adotadas tanto por empresas globais (Li et al., 2022) quanto pelo setor público (BlockNews, 2024), desempenhando um papel fundamental em transações de alto impacto financeiro e social. As blockchains, contudo, mesmo com seus mecanismos de segurança distribuídos, podem conter vulnerabilidades que, se exploradas, permitem a ocorrência de ataques com potencial de comprometer dados, interromper serviços e gerar prejuízos significativos.

A história das blockchains já registra incidentes marcantes, como o ataque à DAO,

¹ bitcoin.org/en

ocorrido em junho de 2016, na rede Ethereum². Na ocasião, uma falha de implementação em um contrato inteligente - código executado de maneira distribuída sobre a blockchain - resultou em uma perda estimada de 70 milhões de dólares (Homoliak et al., 2021). Embora esse caso tenha ocorrido em uma blockchain pública, ele evidencia como vulnerabilidades podem ter consequências graves em sistemas baseados em tecnologias descentralizadas.

Vulnerabilidades, neste contexto, representam fraquezas ou falhas no sistema que podem ser exploradas para comprometer sua segurança. Os exemplos vão desde protocolos vulneráveis, causados por criptografia insuficiente, até problemas em componentes de armazenamento ou em dependências de software. Outro tipo de exposição são os contratos vulneráveis, nos quais as complexidades de desenvolvimento de contratos inteligentes podem levar a comportamentos inseguros.

Essas fragilidades aumentam a exposição do sistema a ameaças, permitindo que agentes internos ou externos possam comprometer a confidencialidade, integridade e disponibilidade da rede. As ameaças, por sua vez, são eventos, condições ou agentes capazes de explorar vulnerabilidades, resultando em danos ou acessos não autorizados. Quando essas ameaças se concretizam, surgem os ataques.

Um ataque é a materialização de uma ameaça, ou seja, uma ação deliberada para explorar uma vulnerabilidade e comprometer o sistema. Adversários externos podem tentar acessar a rede blockchain para ler ou, eventualmente, manipular dados, aproveitando-se das vulnerabilidades. Ataques bem-sucedidos podem comprometer o funcionamento, ou até paralisar o sistema. Podem subverter nós da rede, provedores de identidade, explorar falhas em códigos de contratos inteligentes, prejudicar o consenso e manipular transações.

Nesse contexto, este trabalho realiza uma revisão bibliográfica para mapear as principais vulnerabilidades, ameaças e ataques em blockchains permissionadas. O objetivo é obter informações que possibilitem tomar decisões baseadas no conhecimento atualizado sobre o estado da arte em segurança de blockchains permissionadas.

Após este mapeamento, é apresentada uma listagem das principais vulnerabilidades e ataques. A partir desta lista, são realizados experimentos em um caso de uso concreto, com o objetivo de avaliar a segurança da aplicação e o desempenho de mecanismos de detecção de vulnerabilidades em contratos inteligentes.

Com este trabalho, busca-se contribuir para a segurança de sistemas que utilizam blockchains permissionadas. Isto é realizado apresentando as principais vulnerabilidades e ataques desses sistemas e propondo experimentos que possam auxiliar o desenvolvimento de aplicações mais confiáveis, seguras e eficientes.

² ethereum.org/en

1.1 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte forma: no Capítulo 2 são apresentados os objetivos do trabalho; no Capítulo 3 é descrito o método de pesquisa; no Capítulo 4 são apresentados os conceitos fundamentais para a compreensão do tema; no Capítulo 5 são apresentados os resultados da revisão bibliográfica; e no Capítulo 6 é apresentado o desenvolvimento do trabalho com a descrição dos experimentos e seus resultados.

2 OBJETIVOS

2.1 OBJETIVO GERAL

Mapear e apresentar as principais vulnerabilidades e ataques presentes em blockchains permissionadas. Realizar experimentos buscando avaliar mecanismos de detecção de vulnerabilidades em contratos inteligentes, de modo a contribuir com a segurança dos sistemas que adotam esta tecnologia.

2.2 OBJETIVOS ESPECÍFICOS

- Apresentar as principais vulnerabilidades e ataques em blockchains permissionadas reportados pela comunidade científica.
- Propor medidas de prevenção e mitigação de vulnerabilidades e ataques.
- Avaliar a efetividade de ferramentas de detecção de vulnerabilidades.

3 MÉTODO DE PESQUISA

Neste trabalho será realizada uma revisão sistemática da literatura com o objetivo de identificar e analisar os estudos mais relevantes relacionados a vulnerabilidades, ameaças e ataques em blockchains permissionadas, com ênfase na tecnologia Hyperledger Fabric. Para isso, serão consultadas bases de dados científicas como IEEE Xplore e ACM Digital Library, utilizando palavras-chave específicas, como blockchain permissionada, Hyperledger Fabric, vulnerabilidades, ameaças e ataques. Esta etapa permitirá consolidar um panorama do estado da arte.

A partir desta revisão, será construída uma listagem contendo as principais vulnerabilidades, ataques e medidas de prevenção e mitigação associados a blockchains permissionadas, subdividindo-os às respectivas camadas da estrutura da blockchain. Com esta listagem, serão realizados experimentos práticos de modo a aprofundar o conhecimento de um subconjunto de elementos apresentados.

Os experimentos buscarão avaliar maneiras de encontrar vulnerabilidades com antecedência, buscando, dessa forma, reforçar a segurança de um caso de uso avaliado e validar as ferramentas e métodos utilizados. O ambiente experimental será documentado, incluindo as ferramentas, configurações e scripts utilizados, para garantir a reprodutibilidade.

4 CONCEITOS FUNDAMENTAIS

Esta seção apresenta uma revisão dos principais conceitos associados às tecnologias blockchain, abordando seus modelos, aplicações e características fundamentais. Também são discutidos os protocolos de consenso empregados nesses sistemas e os princípios de segurança que os sustentam. Esses conceitos estabelecem a base necessária para compreender, nos capítulos seguintes, o funcionamento das ferramentas analisadas e as razões pelas quais determinadas vulnerabilidades podem emergir em aplicações de blockchain.

4.1 REDES PAR-A-PAR (P2P) E CHAMADA DE PROCEDIMENTO REMOTO (RPC)

Blockchains, enquanto redes distribuídas de nós, operam utilizando a arquitetura *peer-to-peer* (ou par-a-par). Nesse tipo de arquitetura, cada nó atua simultaneamente como cliente e servidor, permitindo o compartilhamento de dados e serviços sem a necessidade de um servidor centralizado.

Para que aplicações cliente interajam com a rede blockchain, a comunicação é realizada a partir de uma *Remote Procedure Call* (RPC), ou chamada de procedimento remoto (Kolb et al., 2020). É um protocolo que permite que um programa de computador execute um procedimento (uma sub-rotina ou função) em um espaço de endereço diferente (geralmente em outro computador em uma rede) como se fosse uma chamada de procedimento local.

Em suma, P2P é a arquitetura utilizada na comunicação entre os nós da rede e RPC é o protocolo utilizado para comunicação entre uma aplicação cliente e os nós do sistema blockchain.

4.2 TOKENS E CRIPTOMOEDAS

No contexto de blockchain, tokens e criptomoedas são representações digitais de valor ou ativos que podem ser criados, transferidos e gerenciados por meio de redes distribuídas. As criptomoedas funcionam como meios de troca nativos de determinadas blockchains, sendo usadas para pagamentos, recompensas de validação de blocos ou como reserva de valor. Já os tokens podem ter funções mais amplas, representando desde direitos de governança e acesso a serviços até ativos do mundo real, como propriedades ou ações (Antonopoulos; Wood, 2018).

4.3 TRANSAÇÃO

As transações são entradas elementares na rede e podem incluir dados, pedidos para transferir criptomoedas ou outros tipos de *assets*, além de instruções em forma de código. As transações enviadas para a blockchain são validadas por todos os nós da rede, que mantêm uma réplica da cadeia de blocos (Homoliak et al., 2021).

4.4 CRIPTOGRAFIA

A criptografia é um recurso utilizado para evitar que terceiros tenham acesso ou consigam interpretar os dados de uma mensagem privada trocada entre dois ou mais participantes. No contexto de blockchain, a criptografia tem como principal objetivo assegurar que a transferência e o armazenamento de dados ocorram de forma segura. Ela permite o controle de permissões, a privacidade da rede, a proteção dos detalhes das transações contra acessos indevidos e a prevenção de fraudes.

As blockchains utilizam, de modo geral, dois tipos de algoritmos criptográficos: a criptografia de chave assimétrica e as funções de hash. A criptografia de chave assimétrica utiliza um sistema de autenticação com duas chaves: uma chave pública, gerada a partir de um algoritmo fixo, e uma chave privada, criada aleatoriamente. Esse tipo de criptografia é utilizado para assinaturas digitais, que asseguram a integridade dos dados transmitidos e comprovam que não houve alterações no conteúdo. Já as funções de hash servem para garantir a imutabilidade do livro-razão e permitem que os participantes verifiquem a integridade da cadeia de blocos. O hash funciona como uma impressão digital única para o conteúdo de cada bloco e oferece uma camada de segurança que confirma que os blocos se mantêm inalterados a medida que a blockchain cresce (Nasir; Hassan; Zaini, 2024).

4.5 CONTRATOS INTELIGENTES

Contratos inteligentes são programas de computador imutáveis, tipicamente instanciados em sistemas blockchain, que executam um conjunto de instruções. Eles servem como um componente fundamental em diversas aplicações descentralizadas, encapsulando os principais aspectos da lógica de negócio da aplicação (Hamdi; Fourati; Ayed, 2024).

4.6 PROTOCOLO DE CONSENSO

No contexto de blockchains, o protocolo de consenso é responsável por garantir que todos os participantes concordem com a mesma sequência de transações em uma ordem específica (Brotsis et al., 2020). O consenso é aplicado para alcançar um acordo sobre a validade dos dados nos nós da rede distribuída, sem depender de uma autoridade central, e fazendo com que a blockchain seja segura e confiável para realizar transações.

O mecanismo de consenso afeta algumas métricas de performance como latência para realização de transações, tolerância a falhas de nós, tolerância a nós maliciosos e vazão da rede. A implementação de um mecanismo de consenso apropriado é fundamental para garantir a segurança e o desempenho de aplicações baseadas em blockchain (Nasir; Hassan; Zaini, 2024).

4.7 TIPOS DE BLOCKCHAINS

Neste trabalho, a categorização entre os tipos de blockchain se dará como em (Cha et al., 2022), a partir da divisão entre blockchains públicas e permissionadas.

4.7.1 Blockchains Públicas

Blockchains públicas são um tipo de livro-razão distribuído que permite que qualquer usuário possa participar da rede. É tipicamente muito transparente, altamente descentralizada e aberta para acesso público. Todos os nós nesse tipo de rede possuem direitos iguais para acessar a blockchain, gerar novos blocos e verificar os dados desses blocos. Os nós da rede não precisam confiar nem conhecer uns aos outros. São blockchains não permissionadas por essência (Nasir; Hassan; Zaini, 2024). Blockchains não permissionadas são redes nas quais não há permissão para fazer parte do protocolo de consenso. Esse tipo de blockchain usualmente requer que os nós de consenso executem protocolos nos quais seja necessária a prova de recursos, de modo a evitar ações maliciosas no consenso da rede (Homoliak et al., 2021).

4.7.2 Blockchains Permissionadas

As blockchains permissionadas foram desenvolvidas com o propósito de executar uma rede blockchain entre um grupo onde os membros são conhecidos e identificáveis. Neste tipo de blockchain, os participantes devem solicitar autorização para o(s) administrador(es) para participar da rede (Nasir; Hassan; Zaini, 2024).

Nesse contexto, blockchains privadas são um tipo específico de blockchains permissionadas onde apenas uma entidade possui a prerrogativa de adicionar novos blocos à cadeia, mesmo que várias entidades possam ser capazes de ler o conteúdo da blockchain (Kolb et al., 2020)

4.8 PRINCIPAIS IMPLEMENTAÇÕES DE BLOCKCHAIN

Há diversas plataformas que oferecem serviços utilizando a tecnologia blockchain, desde os primeiros sistemas que oferecem apenas a rastreabilidade das transações até ferramentas mais complexas com máquinas virtuais e protocolos de consenso configuráveis. Nesse sentido, segue uma descrição das principais implementações.

4.8.1 Bitcoin

A rede Bitcoin é um sistema global descentralizado de transferências de valor, operado por milhares de nós independentes espalhados pelo mundo. É a rede proposta por (Nakamoto, 2008) ao introduzir o conceito de blockchain. Ela permite que qualquer pessoa envie e re-

ceba a sua moeda nativa, bitcoins, sem depender de intermediários, como bancos ou governos. Todas as transações são registradas de forma pública, imutável e auditável, buscando garantir transparência e segurança para os participantes da rede.

4.8.2 Ethereum

A rede Ethereum, lançada em 2015, é uma plataforma blockchain descentralizada que opera com criptomoedas. Sua moeda nativa é o Ether. Mas diferentemente da rede bitcoin, o Ethereum busca transcender o papel de apenas um ambiente para troca de valores. Ele oferece aos programadores a habilidade de desenvolver contratos inteligentes e construir aplicativos descentralizados (dApps). A linguagem utilizada para desenvolver os contratos inteligentes é Solidity, que é executada na *Ethereum Virtual Machine* (EVM). Cada nó da rede Ethereum, possui uma cópia da EVM (Hamdi; Fourati; Ayed, 2024).

4.8.3 Hyperledger

Hyperledger não é uma plataforma blockchain, e sim um projeto colaborativo de código aberto, iniciado pela Linux Foundation¹. Seu objetivo é reunir diferentes frameworks, ferramentas e bibliotecas para o desenvolvimento de tecnologias blockchain, focando principalmente em aplicações empresariais. As ferramentas são principalmente voltadas, mas não apenas, para blockchains permissionadas.

4.8.4 Hyperledger Fabric

Hyperledger Fabric é uma das plataformas mais conhecidas dentro do projeto Hyperledger. Ele é um framework de blockchain permissionado que permite a criação de redes altamente modulares e configuráveis, com suporte a contratos inteligentes (chamados de chain-codes) escritos em diferentes linguagens, como Golang, Java e NodeJS. O Fabric possui uma arquitetura de canais, que possibilita transações privadas entre subconjuntos de participantes na mesma rede, garantindo confidencialidade e escalabilidade (Hyperledger Foundation, 2025b).

4.8.5 Hyperledger Besu

Hyperledger Besu é outra plataforma dentro do ecossistema Hyperledger, mas voltada para blockchains compatíveis com o Ethereum. Ele é um cliente Ethereum de código aberto, projetado para atender tanto a redes públicas quanto a redes permissionadas. O Besu oferece suporte total à Ethereum Virtual Machine (EVM), possibilita o uso de contratos inteligentes Ethereum e permite escolher diferentes algoritmos de consenso, o que o torna flexível para

¹ linuxfoundation.org

cenários onde se queira aproveitar o ecossistema Ethereum de forma mais controlada (Hyperledger Foundation, 2025a).

4.9 TIPOS DE PROTOCOLOS DE CONSENSO

4.9.1 Protocolos em blockchains públicas

Os principais protocolos relacionados a blockchains públicas são a prova de trabalho (*Proof of Work* - PoW) e a prova de participação (*Proof of Stake* - PoS).

4.9.1.1 *Proof of Work (PoW)*

O protocolo de prova de trabalho é o mecanismo de consenso original do artigo apresentado por Nakamoto (Nakamoto, 2008) para a rede Bitcoin. A principal operação do PoW é a habilidade de provar a solução de um quebra-cabeça (*puzzle*) matemático, cuja dificuldade é ajustável, de modo a obter a autoridade de adicionar um novo bloco à cadeia (Nasir; Hassan; Zaini, 2024). Esse processo de resolver o problema matemático e adicionar novos blocos à cadeia é chamado de mineração (ou *mining*), enquanto os nós que o realizam são chamados de mineradores (ou *miners*).

PoW é um protocolo principalmente utilizado em blockchains públicas, que visa garantir a segurança e a descentralização da rede. Contudo, possui algumas desvantagens. Uma delas é o alto processamento computacional necessário para resolver o problema matemático. Outro ponto é que o processo de mineração dura em torno de 10 minutos, fazendo com que a verificação de blocos seja adiada e a velocidade das transações seja lenta. Portanto, não é o protocolo adequado para sistemas que demandam maior eficiência, com taxas de transações mais altas e menor latência (Nasir; Hassan; Zaini, 2024).

4.9.1.2 *Proof of Stake (PoS)*

Algoritmos de prova de participação, ou *proof of stake*, foram concebidos com a intenção de substituir os algoritmos de prova de trabalho em blockchains públicas. Eles também permitem a entrada e a saída de qualquer participante na rede, mas não envolvem o consumo de energia do PoW. Em um sistema de prova de participação, um nó reivindica a capacidade de adicionar novos blocos à cadeia e receber taxas de transação como recompensa ao submeter uma certa quantidade de tokens como um depósito de segurança (colocar em "stake"). Esse nó então se torna um validador, em oposição ao minerador da prova de trabalho. Implementações de prova de participação tipicamente assumem que dois terços dos tokens colocados em "stake" são mantidos por participantes honestos, o que seria uma premissa mais forte do que a exigência da prova de trabalho de que uma simples maioria dos participantes seja honesta (Kolb et al., 2020).

4.9.2 Protocolos em blockchains permissionadas

Com relação às blockchains permissionadas, os algoritmos de consenso tratam principalmente da gerência de dois problemas, a tolerância a falhas dos nós e a presença de nós maliciosos. Nesse sentido, os protocolos de consenso de blockchains permissionadas podem ser classificados em *Crash Fault Tolerant* (CFT), isto é, tolerantes a falhas, e *Byzantine Fault Tolerant* (BFT), tolerantes a nós bizantinos, ou nós maliciosos.

4.9.2.1 Protocolos Tolerantes a Falhas

Em protocolos tolerantes a falhas, o foco está em manter o consenso e o funcionamento do sistema mesmo que nós individuais estejam indisponíveis por um determinado tempo. Sistemas CFT assumem que não há participantes maliciosos na rede (Nasir; Hassan; Zaini, 2024). Nesses protocolos, a criação de blocos é rápida e a finalidade das transações, sequência de blocos que garante que uma transação está confirmada na cadeia, é garantida. Um dos principais algoritmos de consenso CFT é o Raft.

4.9.2.2 Raft

Raft é um algoritmo proposto para tolerar falhas. É baseado em um esquema de votação e mantém a consistência dos dados a partir da sincronização de logs. De acordo com o algoritmo, os nós desempenham diferentes papéis baseados nos seus diferentes estados em um sistema distribuído, como líderes, candidatos e seguidores.

O tempo no Raft é segmentado em termos, que são períodos de duração arbitrária. Em cada termo, o mesmo nó funciona continuamente como líder. O algoritmo Raft processa as requisições de transação dos clientes: ele as revisa, categoriza e organiza cronologicamente. As transações são então compiladas em blocos e disseminadas para os seguidores. A tarefa dos seguidores é confirmar a validade do bloco com o líder e replicá-lo. A confirmação de uma transação no Raft exige que mais de 50% dos nós completem a adição do bloco. É um algoritmo que possui uma alta vazão em comparação com implementações de algoritmos BFT, contudo não é capaz de controlar o comportamento de nós maliciosos. Consequentemente, uma vez que um líder for controlado maliciosamente, ele poderia ter um grande impacto no sistema de consenso (Nasir; Hassan; Zaini, 2024).

4.9.2.3 Protocolos Tolerantes a Falhas Bizantinas

A tolerância a falhas bizantinas é uma abordagem comumente presente em sistemas distribuídos para alcançar consenso mesmo que alguns nós da rede falhem em responder ou respondam com informações incorretas. Um nó que apresenta falhas pode se comportar de forma estranha, enviando mensagens diferentes para diferentes nós na rede. É difícil definir

se esse nó é malicioso ou defeituoso. Nesse sentido, para decidir sobre o nó com falha, os nós honestos do sistema chegam a um consenso, e um sistema que consegue chegar a um acordo sem ser afetado por um nó malicioso ou defeituoso pode ser caracterizado como um sistema BFT. Em vez de identificar o componente malicioso ou com defeito e corrigir o problema existente, o sistema BFT continua a funcionar como se nenhum componente do sistema estivesse com falhas. No entanto, isso pode resultar em menor *throughput* e eficiência (Nasir; Hassan; Zaini, 2024).

Sistemas de livro-razão distribuídos (*distributed ledger systems*) são classificados como BFT se conseguem resolver o problema dos generais bizantinos (Byzantine General's Problem - BGP). O BGP descreve o desafio de chegar a um acordo em um sistema distribuído com participantes potencialmente traidores. No caso da blockchain seria a possibilidade de existir nós maliciosos (Nasir; Hassan; Zaini, 2024).

4.9.2.4 *Practical Byzantine Fault Tolerance (PBFT)*

O protocolo de tolerância a falhas bizantinas prática, é uma implementação BFT capaz de suportar até $1/3$ dos nós maliciosos. Os nós no PBFT possuem basicamente dois papéis, nós primários e nós de backup. Nós primários são responsáveis por isolar as requisições de transações, transferindo-as para o nó de backup e gerando o bloco após o período de verificação.

Apesar de ser um algoritmo seguro, tolerante a comportamentos maliciosos, possui problemas de escalabilidade, assim como os protocolos BFT, no geral. São algoritmos desenvolvidos para sistemas com poucos nós, pois demandam um nível muito alto de comunicação entre os participantes para adquirir o consenso. Um crescimento exponencial no número de nós, junto ao alto volume de comunicação poderia congestionar o sistema blockchain. Nesse cenário, BFT-Smart é outra implementação parecida com o PBFT, visando maior performance e melhorando questões de modularização e confiabilidade (Nasir; Hassan; Zaini, 2024).

4.10 INFRAESTRUTURA DE CHAVE PÚBLICA (PKI) E AUTORIDADES CERTIFICADORAS (CA)

A Infraestrutura de Chave Pública (PKI) é um conjunto de tecnologias, políticas e procedimentos que viabiliza o gerenciamento seguro de identidades digitais. Ela se baseia na geração de pares de chaves assimétricas e na utilização de certificados digitais, que vinculam uma chave pública a uma identidade reconhecida. A verificação de identidade, integridade e autenticidade de mensagens ou transações torna-se possível porque as partes confiam em uma Autoridade Certificadora (CA) para emitir, validar, renovar ou revogar esses certificados.

Em blockchains permissionadas, como o Hyperledger Fabric, a PKI tem um papel central na organização da rede. Ao contrário de blockchains públicas, onde os participantes geralmente operam com identidades pseudo-anônimas, em redes permissionadas os nós, usuários e organizações precisam ser identificados de forma clara. Nesse sentido, a CA atua como uma en-

tidade confiável, responsável por garantir que cada participante tenha um certificado válido que comprove sua identidade e permita sua atuação na rede de acordo com regras pré-estabelecidas.

Essa abordagem possibilita a implementação de mecanismos adicionais de controle de acesso e governança, como o Membership Service Provider (MSP) no Hyperledger Fabric, que depende das chaves e certificados emitidos pelas CAs para definir as permissões, políticas e direitos de cada organização ou participante na rede. Essa estrutura de identidade robusta é fundamental para garantir a segurança, a rastreabilidade e a confiança em ambientes colaborativos onde os participantes são conhecidos e autorizados (Hyperledger Foundation, 2025b).

4.11 CONCEITOS EM HYPERLEDGER FABRIC

O Hyperledger Fabric se destaca como uma das principais tecnologias em blockchains permissionadas, e nesse sentido, é importante que determinados conceitos relacionados ao seu funcionamento e às suas características sejam melhor elaborados.

4.11.1 Estrutura de uma rede Fabric

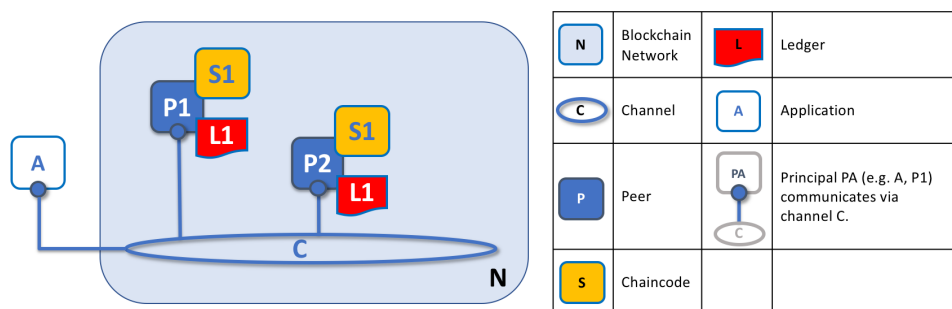


Figura 1 – Estrutura Hyperledger Fabric Simplificada

Fonte: Hyperledger Foundation

Em uma rede Fabric, os participantes são organizados em organizações reconhecidas na rede. Cada organização pode operar peers, que são nós responsáveis, conforme a Figura 1, por armazenar o ledger (livro-razão) e executar as chaincodes, que são os contratos inteligentes do Fabric. Os peers executam diferentes funções, recebem requisições de clientes, endossam transações, ou seja, simulam transações antes de serem ordenadas e validam o bloco antes de adicioná-lo à cadeia.

Um componente central nessa arquitetura é o orderer node (ou serviço de ordenação), que garante a ordenação determinística das transações em blocos, distribuindo-os para os peers participantes de um canal específico. Os canais são uma funcionalidade que permite a criação de redes lógicas isoladas dentro da mesma infraestrutura, possibilitando que subconjuntos de

organizações compartilhem dados de forma privada. Nesse sentido, as redes Fabric são formadas principalmente pela combinação de peers, orderers, canais, organizações e chaincodes para atender diferentes casos de uso que implementam blockchains (Hyperledger Foundation, 2025b).

4.11.2 Membership Service Provider (MSP)

O MSP é o mecanismo que possui a prerrogativa de tornar identidades verificáveis em papéis compatíveis com o funcionamento de uma rede Fabric. Ele permite que as organizações, canais e nós estabeleçam MSPs que determinem quem possui permissões para realizar as diferentes tarefas da rede. O MSP não é um provedor de identidade, esse é o papel da CA, mas sim um conjunto de configurações que irá associar papéis, privilégios e permissões dentro da rede blockchain (Hyperledger Foundation, 2025b). A Figura 2 ilustra a estrutura de uma rede Fabric com mais componentes, como as organizações, autoridades certificadoras, identidades e o MSP.

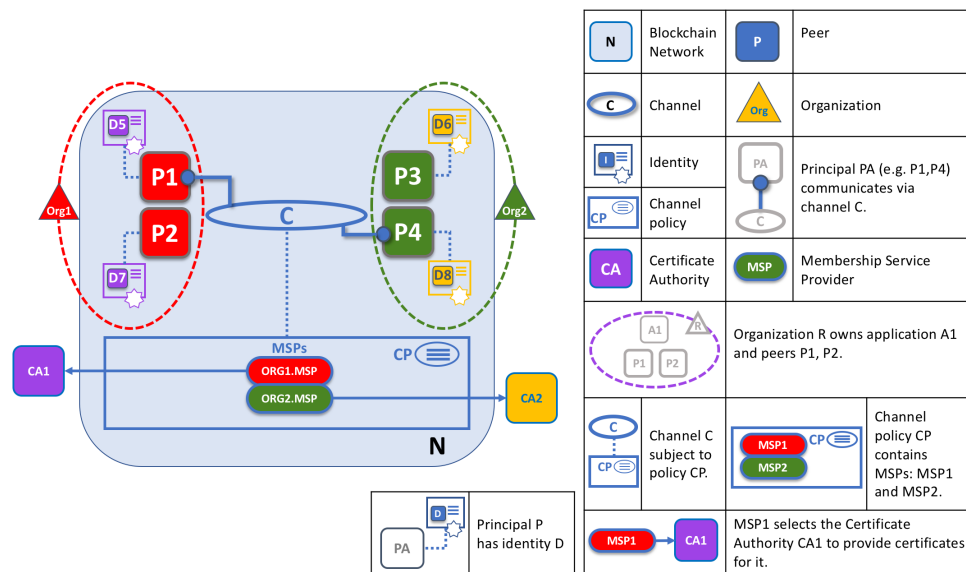


Figura 2 – Estrutura Hyperledger Fabric com MSP

Fonte: Hyperledger Foundation

4.11.3 Private Data Collections (PDC)

Em redes Fabric, uma forma de manter o acesso a determinados dados restrito a apenas um grupo de organizações é criar um canal que compreenda apenas esse conjunto de participantes. Desse modo, apenas essas organizações poderão acessar esses dados. Contudo, há cenários em que existe a necessidade de mesmo dentro de um canal, apenas um subconjunto de organizações terem acesso sobre determinadas informações, e é nesse contexto que o Fabric implementa

as Coleções de Dados Privados (*Private Data Collections* - PDC) (Hyperledger Foundation, 2025b).

PDCs permitem a um subconjunto de organizações dentro de um canal ler e escrever dados privados, sem a necessidade de criar um novo canal. As coleções podem ser definidas de forma explícita dentro das definições de uma chaincode. Além disso, cada chaincode possui um espaço privado implícito reservado para informações específicas de cada organização (Hyperledger Foundation, 2025b). A figura 3 ilustra o par da organização 'ORG1' com acesso aos dados privados, enquanto o par da organização 'ORG2' visualiza apenas os dados do canal compartilhado.

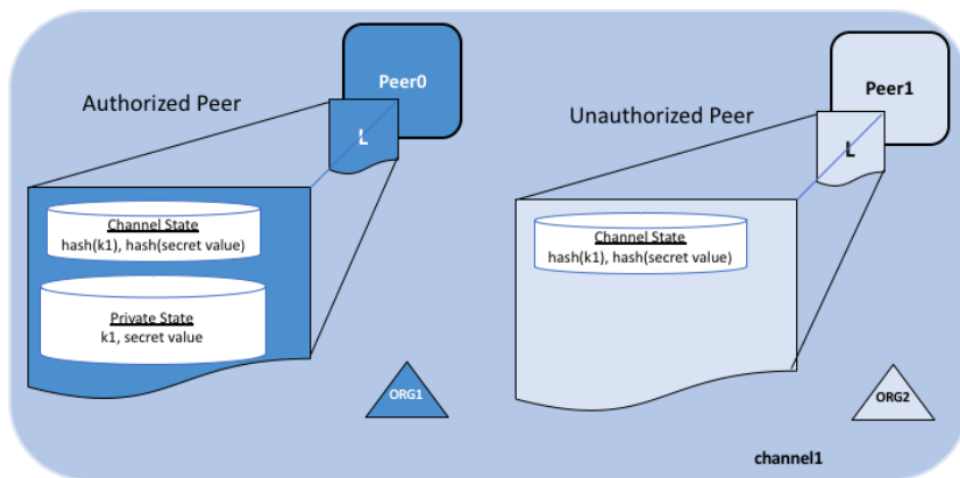


Figura 3 – Dados privados em Hyperledger Fabric

Fonte: Hyperledger Foundation

4.12 ATAQUES

Alguns ataques relacionados à blockchain possuem bastante relevância na literatura e por essa razão serão melhor elaborados.

4.12.1 Ataque de Negação de Serviço Distribuído (DDoS)

Nesse tipo de ataque, um atacante tenta fazer com que recursos de um sistema se tornem inacessíveis para os usuários através de um envio massivo de requisições, provenientes de diferentes fontes. Buscando, dessa forma, sobrecarregar o serviço e causar lentidão ou mesmo um "crash" do sistema. No contexto de aplicações blockchain, um agente malicioso envia uma quantidade significativa de transações de modo a interferir no funcionamento correto de algum nó da rede, tornando-o inacessível (Nasir; Hassan; Zaini, 2024).

4.12.2 Ataque Sybil

Um ataque Sybil ocorre quando uma entidade maliciosa cria e controla múltiplas identidades falsas dentro de uma rede, geralmente P2P ou blockchain. Essencialmente, o ataque visa ganhar influência desproporcional, criando uma massa de "personalidades" artificiais para enganar o sistema. Isso pode, por exemplo, permitir que o atacante controle uma maioria de nós em uma rede blockchain, comprometendo a segurança e a confiança de toda a rede. Ao se passar por múltiplos usuários legítimos, o atacante Sybil pode manipular sistemas de reputação, diminuindo a credibilidade de usuários genuínos, pode forjar consenso, utilizando as identidades falsas para dominar votações ou processos de decisão e, além disso, obter acesso fraudulento, assumindo perfis para realizar transações ou acessar recursos indevidamente (Nasir; Hassan; Zaini, 2024).

4.12.3 Ataques de 51%

Um ataque de 51% é uma vulnerabilidade na arquitetura de consenso de uma blockchain que permite a um invasor obter controle sobre toda a rede. Ele ocorre quando um agente malicioso consegue adquirir mais de 50% do poder computacional ou de voto da blockchain.

Esse tipo de ataque é bastante conhecido em blockchains públicas e foi utilizado inicialmente contra o Bitcoin. Contudo, mesmo que a probabilidade seja consideravelmente menor, um ataque de 51% também pode ocorrer em blockchains permissionadas. Se uma única entidade controlar mais de 51% dos nós ou do poder de voto em uma blockchain permissionada, ela poderá manipular transações, interromper o processo de consenso e, conseqüentemente, comprometer a integridade e a segurança da rede (Nasir; Hassan; Zaini, 2024).

4.12.4 Ataques de Gasto Duplo

O ataque de gasto duplo ocorre, em uma rede blockchain, quando um usuário tenta usar os mesmos recursos para diversas transações antes que a validação esteja completa. Esse ataque explora a vulnerabilidade que surge da capacidade de replicar uma transferência legítima que ainda não tenha sido registrada de forma imutável na blockchain (Nasir; Hassan; Zaini, 2024).

É um ataque que afeta principalmente protocolos de consenso com finalidade lenta, onde as transações precisam de um determinado número de blocos adicionados à rede para serem confirmadas. Dois ou mais blocos conflitantes podem existir na mesma altura, causando uma inconsistência chamada de *fork*. Nesse sentido, algum token pode ser temporariamente gasto em ambos os blocos conflitantes, enquanto apenas um único bloco será depois adicionado à cadeia honesta (Homoliak et al., 2021).

5 REVISÃO BIBLIOGRÁFICA

Este capítulo tem por objetivo analisar o que já foi encontrado pela comunidade científica de vulnerabilidades em blockchains, focando principalmente em blockchains permissionadas. Além disso, se buscará observar que experimentos já foram realizados e de que forma os autores tratam o tema. Dessa forma, estes trabalhos servirão como base para o desenvolvimento deste trabalho, fornecendo uma lista de vulnerabilidades e ataques conhecidos e as formas de explorá-los em um caso de uso específico.

Para o desenvolvimento dessa revisão, foi utilizado como base o método proposto por (Kitchenham et al., 2009). Além disso, as perguntas que nortearam a busca foram:

1. Quais são as principais vulnerabilidades e ameaças específicas das blockchains permissionadas?
2. Quais são os principais tipos de ataques direcionados às blockchains permissionadas e como esses ataques exploram essas vulnerabilidades?
3. Que tipos de experimentos têm sido propostos para verificar essas vulnerabilidades?
4. Quais medidas de prevenção, detecção e mitigação têm sido propostas ou implementadas para lidar com essas vulnerabilidades e ataques em blockchains permissionadas?

De modo a responder a estas perguntas, na tabela 1 foram definidas as palavras-chave e sinônimos para representar os filtros nas plataformas de busca.

Tabela 1 – Palavras-chave e seus sinônimos

Palavra-chave	Sinônimos
Permissioned Blockchain	Permissioned Ledger
Threats	Risks, Cyber Threats
Vulnerabilities	Weaknesses, Security Flaws
Attacks	Cyberattacks

A partir das palavras-chave e sinônimos, foram utilizados ACM Digital Library¹, IEEE Xplore², SpringerLink³, ScienceDirect⁴ e Google Scholar⁵ como plataformas para pesquisa e, então, definiu-se a seguinte *query* de busca:

```
("permissioned blockchain" OR "permissioned ledger" OR "private blockchain"
OR "private ledger") AND ("threats" OR "risks" OR "cyber threats")
AND ("vulnerabilities" OR "weaknesses") AND ("attacks" OR "cyberattacks")
```

¹ dl.acm.org

² ieeexplore.ieee.org

³ link.springer.com

⁴ sciencedirect.com

⁵ scholar.google.com

Para seleccionar os trabalhos, inicialmente foi feita uma seleção dos resultados pelo título, depois pelo resumo e então foi realizada a leituras dos textos. Como filtros, foram utilizados artigos publicados desde 2020. Em algumas bases de dados a *query* acabou se mostrando bastante ampla e resultou em vários trabalhos. Nestes casos, foram analisados apenas os 260 primeiros resultados.

Tabela 2 – Resultados da busca e seleção de artigos por base de dados

Database	Seleção 1 (título)	Seleção 2 (resumo)	Seleção 3 (completo)
ACM Digital Library	8	2	2
IEEE Xplore	17	4	3
ScienceDirect	11	2	2
SpringerLink	18	1	1
Total	54	9	8

5.1 RESUMO DOS TRABALHOS

Nesta seção, são apresentados os resumos dos trabalhos utilizados para a revisão bibliográfica. Estes trabalhos fornecem informações relevantes sobre vulnerabilidades e ataques em blockchains, principalmente em blockchains permissionadas. Além disso, em alguns casos, realizam experimentos que avaliam a detecção de vulnerabilidades e o desempenho em cenários de ataque.

Os resumos irão apresentar o problema tratado no artigo, as soluções apresentadas, experimentos realizados, resultados, trabalhos futuros e uma análise sobre a contribuição do trabalho.

The Security Reference Architecture for Blockchains: Toward a Standardized Model for Studying Vulnerabilities, Threats, and Defenses

O principal problema apontado por (Homoliak et al., 2021) é a ausência de uma padronização para a análise das ameaças à segurança em sistemas blockchain. Os autores argumentam que, embora existam trabalhos nessa área, eles geralmente se concentram em plataformas específicas ou abordam as blockchains de forma excessivamente genérica. Diferentemente desses estudos, segundo os autores, esse trabalho não se limita a uma plataforma específica, focando-se em conceitos gerais de blockchain. Por exemplo, ao tratar de protocolos de consenso, são discutidas ameaças associadas a diferentes abordagens, como prova de trabalho, prova de participação e protocolos tolerantes a falhas bizantinas, buscando, assim, contemplar uma ampla gama de decisões arquiteturais adotadas em sistemas blockchain.

Como proposta, os autores apresentam um modelo de análise de ameaças e riscos de segurança em blockchains, acompanhado de possíveis contramedidas. São introduzidos dois modelos: o *stacked model* e o *threat-risk model*. O *stacked model* organiza a arquitetura em quatro camadas: aplicação, replicação da máquina de estados, consenso e rede. Já o *threat-*

risk model contempla os principais componentes e agentes envolvidos — como proprietários, ativos, agentes de ameaça, ameaças, contramedidas e riscos —, além de sobrepor-se ao modelo anterior, proporcionando uma visão integrada.

Embora não sejam realizados experimentos empíricos no escopo do trabalho, os autores apresentam uma classificação abrangente de ameaças distribuídas entre as diferentes camadas da arquitetura blockchain. Além disso, discutem pontos positivos e negativos de diversas decisões arquitetônicas, não apenas sob o ponto de vista da segurança.

O trabalho também apresenta gráficos relacionando categorias específicas de vulnerabilidades, ameaças e defesas, bem como uma amostra de incidentes reais ocorridos em sistemas blockchain. Os autores observam que o número de incidentes efetivamente registrados é substancialmente menor do que a quantidade de ameaças teóricas identificadas, especialmente nas camadas de consenso e de aplicação. No caso da camada de aplicação, a maioria dos incidentes está associada à exploração de componentes centralizados por agentes internos ou externos. Já na camada de consenso, predominam incidentes relacionados à violação temporária dos pressupostos do protocolo, como os ataques de 51%.

Como trabalhos futuros, os autores sugerem diversas direções de pesquisa relacionadas às diferentes camadas analisadas. Entre elas, destacam-se investigações sobre o equilíbrio entre finalidade mais rápida, segurança e escalabilidade; estudos sobre protocolos de rede utilizados em blockchains e o impacto da criptografia na vazão da rede; pesquisas voltadas à mitigação de pontos únicos de falha em implementações com elementos ainda centralizados; e avaliações da segurança das aplicações que serão construídas sobre essas infraestruturas.

Trata-se de um artigo bastante completo, que oferece uma análise detalhada das decisões arquiteturais em sistemas blockchain e de seus impactos na segurança. A classificação das ameaças proposta é útil para compreender a diversidade de ataques possíveis e pode servir como base para categorizar novas ameaças identificadas em estudos futuros. Nesse sentido, o trabalho contribui para a sistematização do conhecimento sobre segurança em blockchain.

No entanto, observa-se que o foco do artigo recai majoritariamente sobre arquiteturas públicas e não permissionadas, havendo pouca ênfase em soluções permissionadas — que são o principal objeto de estudo deste trabalho de conclusão de curso. Apesar disso, os modelos propostos ainda oferecem uma base conceitual relevante, que pode ser adaptada ou estendida para o contexto de blockchains permissionadas.

On the Security and Privacy of Hyperledger Fabric: Challenges and Open Issues

Neste trabalho, (Brotsis et al., 2020) parte do pressuposto de que o Hyperledger Fabric (referido como Fabric ao longo do texto) é uma das ferramentas mais populares para a implementação de blockchains permissionadas, tendo causado um impacto significativo no mercado. No entanto, os autores observam que há poucos estudos que abordem de forma abrangente as ameaças à segurança, os riscos arquiteturais e os mecanismos de privacidade associados ao Fabric, bem como as respectivas contramedidas de mitigação.

Nesse contexto, os autores realizam uma análise da ferramenta sob a ótica da segu-

rança, dividindo sua investigação em quatro componentes, nos quais podem ocorrer ataques e vazamentos de informação: (i) o protocolo de consenso, (ii) os contratos inteligentes (chaincodes), (iii) a rede e (iv) os mecanismos de preservação da privacidade. Para cada um desses componentes, são identificados os riscos específicos e apresentadas contramedidas com o objetivo de fortalecer a segurança do sistema como um todo.

Como resultado da análise, os autores fornecem um mapeamento dos riscos associados a cada componente do sistema, juntamente com as medidas de mitigação correspondentes. Em relação ao protocolo de consenso, observam que os mecanismos implementados no Fabric são tolerantes a falhas em nós de ordenação (orderers), mas não são resilientes a comportamentos maliciosos nestes nós⁶. No que diz respeito à privacidade, os autores tratam de quatro aspectos principais: canais, coleções de dados privados, provas de conhecimento zero (zero-knowledge proofs) e o Membership Service Provider (MSP). Nesse sentido, os autores destacam que o Fabric ainda não oferece suporte a provas de conhecimento zero não interativas nem a assinaturas resistentes a ataques quânticos.

Em relação aos contratos inteligentes (chaincodes), são identificadas vulnerabilidades específicas principalmente relacionadas ao uso da linguagem Go⁷, a mais comum para o desenvolvimento das chaincodes, com ênfase em questões como concorrência e baixa entropia na geração de chaves aleatórias. Para esses casos, os autores propõem contramedidas práticas e métodos de verificação capazes de evitar a exploração dessas falhas. Por fim, no componente de rede, os riscos concentram-se principalmente na centralidade do MSP, responsável pelo gerenciamento de permissões e identidades, bem como na exposição dos validadores (endorsers) das transações, que podem ser alvos de ataques direcionados.

Como trabalhos futuros, os autores sugerem investigações sobre o uso de provas de teoremas, ferramentas de execução simbólica e modelagem formal com o objetivo de mitigar riscos associados aos contratos inteligentes. Além disso, tratam da aplicação de Ambientes de Execução Segura (Trusted Execution Environments – TEE) como forma de proteger o MSP contra possíveis ataques. TEEs são áreas seguras e isoladas dentro de um processador que garantem a confidencialidade e a integridade dos dados e do código que são executados dentro delas. Citam também a pseudo-anonimização dos validadores (endorsers) para dificultar ataques de negação de serviço direcionados e ataques do tipo wormhole, além de avanços no suporte a provas de conhecimento zero não interativas e a assinaturas com segurança pós-quântica.

O trabalho é relevante por tratar de forma específica o Hyperledger Fabric, uma das ferramentas mais populares para implementação de blockchains permissionadas. Além disso, ao dividir os riscos em quatro componentes distintos, os autores oferecem uma abordagem que permite uma compreensão mais precisa e direcionada das fragilidades do sistema.

Assessing the Performance of Ethereum and Hyperledger Fabric Under DDoS Attacks for

⁶ A partir da versão 3.0 do Fabric foi implementado o protocolo BFT - tolerante a nós maliciosos. https://hyperledger-fabric.readthedocs.io/en/latest/raft_bft_migration.html

⁷ <https://go.dev/>

Cyber-Physical Systems

O trabalho de (Jayadev; Moradpoor; Petrovski, 2024) parte do pressuposto de que, embora a tecnologia blockchain apresente diversas características positivas — como descentralização, transparência e imutabilidade —, os dados nela armazenados possuem alto valor e, portanto, eventuais ataques poderiam ocasionar perdas significativas, tanto financeiras quanto de reputação. Dentre os possíveis ataques, destaca-se o de negação de serviço distribuído (DDoS).

Nesse contexto, o principal objetivo desse artigo é realizar uma avaliação de desempenho de duas tecnologias blockchain: Ethereum e Hyperledger Fabric. A questão central que norteia o estudo é: "Quais são as características comparativas de desempenho do Ethereum e do Hyperledger Fabric em relação a ataques DDoS?" Para alcançar esse objetivo, os autores analisam o desempenho das plataformas mencionadas utilizando as métricas de latência e vazão. O objetivo da simulação é compreender o comportamento das tecnologias blockchain, com diferentes arquiteturas, mecanismos de consenso e protocolos criptográficos, em ataques DDoS, avaliando seu desempenho sob variados cenários.

Os experimentos foram conduzidos por meio da criação de duas redes: uma baseada no Hyperledger Fabric e outra no Hyperledger Besu, ambas executadas em um computador pessoal. A simulação do ataque DDoS foi realizada com o auxílio do Hyperledger Caliper, uma ferramenta de benchmark para blockchain que permite a avaliação de desempenho com base em casos de uso pré-definidos. Os cenários de teste incluíram:

1. Manter constante o número de usuários e variar a quantidade de transações;
2. Manter constante o número de transações e variar o número de usuários;
3. Variar gradualmente tanto o número de usuários quanto o de transações.

Os resultados obtidos indicam que, nos diferentes cenários testados, o Hyperledger Fabric apresentou maior resiliência frente ao ataque DDoS, alcançando melhores índices de latência e vazão, com diferentes números de nós, transações e usuários simulados.

Como propostas para trabalhos futuros, os autores pretendem avaliar o desempenho de outras plataformas blockchain e explorar novos cenários de teste, como por exemplo determinar o número máximo de transações que uma plataforma específica pode processar com eficiência ótima. Além disso, planejam realizar experimentos com um número maior de nós, a fim de investigar os impactos da escalabilidade nessas tecnologias.

O trabalho se destaca por ser objetivo e claro ao descrever os experimentos realizados, e por oferecer uma análise consistente do desempenho de duas plataformas blockchain diante de um cenário de ataque. Sua contribuição é relevante, especialmente por apresentar resultados comparativos que podem orientar escolhas tecnológicas que demandem robustez frente a ataques DDoS.

Joint Modeling of Hyperledger Fabric and Sybil attack: Petri Net approach

Neste trabalho, (La Salle et al., 2023) realçam a importância das blockchains permissionadas e, principalmente, do Hyperledger Fabric, enquanto alternativa em ascensão, por ser utilizada por grandes empresas pelo mundo. Nesse sentido, é uma ferramenta que possivelmente estará suscetível a ataques cibernéticos e destaca a importância de estudar a sua segurança. Os autores destacam a importância de tratar ameaças que podem emanar de atores internos da rede e, por essa razão, tratam de um ataque Sybil. O artigo relata uma ausência de trabalhos académicos que modelam formalmente o comportamento do Hyperledger Fabric sob ataque.

Nesse sentido, as principais contribuições deste artigo são apresentar uma modelagem do Hyperledger Fabric (HLF) com Redes de Petri, uma ferramenta matemática para modelar e analisar sistemas concorrentes e paralelos, sob a versão v2.2 do HLF. E introduzir um modelo conjunto do HLF sob um cenário de ataque Sybil, juntamente com simulações numéricas desse modelo, envolvendo variações de parâmetros e considerações sobre a estrutura da rede, a fim de quantificar o nível de impacto no desempenho que o ataque Sybil exerce sobre a rede do HLF. Para validar a modelagem, implementam o HLF v2.2 em Kubernetes⁸, utilizando Google Cloud Platform (GCP)⁹ e Hyperledger Caliper (HC)¹⁰, uma ferramenta para executar a carga de trabalho e obter as métricas de desempenho.

Os resultados obtidos indicam que o ataque Sybil é mais bem-sucedido quando há um número maior de pares de endosso maliciosos. Os experimentos comparam cenários nos quais o número de pares maliciosos é igual ao de pares regulares e outros em que os pares maliciosos são em maior número. Neste último caso, observa-se menor degradação de desempenho da rede. Além disso, uma maior taxa de propostas de transações por parte do cliente malicioso também resulta em menor impacto no desempenho e favorece o sucesso do ataque.

Dessa forma, no que se refere a estratégias para mitigar a efetividade do ataque Sybil, os resultados sugerem que a redução no número de pares de endosso regulares e a limitação da taxa de propostas de transações por cliente são medidas eficazes para dificultar o sucesso do ataque.

Como trabalhos futuros, os autores propõem a generalização da modelagem apresentada para aplicação em outros tipos de sistemas, bem como a adaptação para diferentes configurações e tipos de ataques ao HLF. Destacam, ainda, a possibilidade de explorar novas métricas a partir de diferentes variações estruturais do sistema.

De maneira geral, o trabalho descreve de forma sólida os experimentos realizados e apresenta os resultados com clareza. A abordagem de modelagem com Redes de Petri mostra-se promissora para investigações futuras. No entanto, a discussão dos resultados poderia ser mais aprofundada, especialmente no que se refere às implicações práticas para a segurança de redes baseadas em HLF.

Vulnerabilities and attacks assessments in blockchain 1.0, 2.0 and 3.0: tools, analysis and

⁸ <https://kubernetes.io>

⁹ <https://console.cloud.google.com/>

¹⁰ <https://github.com/hyperledger-caliper/caliper>

countermeasures

Este artigo, (Hamdi; Fourati; Ayed, 2024) destaca a crescente importância da tecnologia blockchain e sua aplicação em diversos setores da sociedade, como saúde, finanças e cadeia de suprimentos. No entanto, apesar de sua relevância, os autores relatam que aspectos relacionados à performance e à robustez frente a ataques e vulnerabilidades ainda se configuram como desafios em aberto. Além disso, demandam investigações mais aprofundadas. Adicionalmente, a análise de ferramentas voltadas à segurança dos contratos inteligentes é apontada como uma área que carece de maior desenvolvimento.

Nesse contexto, os autores apresentam um panorama evolutivo das gerações da tecnologia blockchain — da 1.0 à 4.0 — e discutem diferentes taxonomias relacionadas a ataques nesse ecossistema. A partir dessa análise, propõem uma taxonomia própria e classificam os ataques segundo as diferentes gerações da tecnologia. O trabalho também aborda métodos de avaliação de vulnerabilidades em contratos inteligentes, como análise estática, análise dinâmica, execução simbólica, interpretação abstrata e resolução de restrições. Dentro desse escopo, são avaliadas ferramentas que visam detectar riscos em contratos inteligentes das plataformas Ethereum e Hyperledger Fabric (chaincodes).

Com relação às chaincodes, são apresentadas quatro ferramentas de análise estática: ReviveCC¹¹, que realiza uma avaliação de segurança dos contratos, Chaincode Analyzer¹², que busca encontrar potenciais riscos em contratos escritos em Golang, como geração de valores aleatórios, acesso a recursos fora da rede blockchain e iterações em objetos do tipo map. Chaincode scanner¹³, uma ferramenta comercial disponível através de uma página web, que busca encontrar potenciais riscos e Go tools¹⁴, uma ferramenta de análise estática recomendada pela comunidade do Hyperledger para analisar programas escritos em Go.

Essas ferramentas foram testadas com a utilização de diferentes chaincodes que propositalmente continham vulnerabilidades, sendo estas: variáveis globais, gerador de números aleatórios, riscos em consultas por faixa (range query) e read-your-write. Dentre as ferramentas mais específicas de chaincode, a que obteve o melhor desempenho foi a Chaincode analyzer, encontrando as quatro vulnerabilidades. A ferramenta mais genérica (Go tools), voltada para qualquer programa escrito em Go, naturalmente, não apontou os riscos associados à blockchain.

Com relação a trabalhos futuros, os autores propõem a criação de ferramentas de análise open-source especificamente voltadas às chaincodes do Hyperledger Fabric e à sua segurança.

Trata-se de um trabalho bastante abrangente, que contribui para a área ao apresentar uma taxonomia própria de ataques, contextualizar as gerações de blockchain e avaliar ferramentas existentes para análise de contratos inteligentes. A análise comparativa das ferramentas de segurança, especialmente no contexto do Hyperledger Fabric, oferece uma contribuição prá-

¹¹ github.com/sivachokkapu/revive-cc

¹² lf-decentralized-trust-labs.github.io/labs/archived/chaincode-analyzer.html

¹³ chainsecurity.com/smart-contract-audit-reports

¹⁴ github.com/dominikh/go-tools

tica e relevante para pesquisadores e desenvolvedores interessados em fortalecer a segurança de aplicações baseadas em blockchain.

Understanding and Detecting Privacy Leakage Vulnerabilities in Hyperledger Fabric Chaincodes

O trabalho de (Chen et al., 2024) tem como foco o Hyperledger Fabric, por ser uma das principais plataformas utilizadas para a construção de blockchains permissionadas. Para lidar com dados sensíveis, o Fabric oferece um recurso chamado coleções de dados privados (Private Data Collections – PDCs), que permite restringir o acesso a determinados dados a apenas um subconjunto dos participantes de um canal. O estudo concentra-se em como os desenvolvedores utilizam essas coleções ao criarem contratos inteligentes (chaincodes) e nos riscos associados ao uso inadequado desse recurso, que podem resultar em vazamentos de informações privadas.

Os autores identificam que as PDCs são, muitas vezes, complexas demais para que os desenvolvedores compreendam seu funcionamento de forma plena, o que acaba levando à introdução de vulnerabilidades nos contratos. A partir de uma amostra empírica, foi constatado que 91,67% dos chaincodes analisados apresentavam falhas de segurança relacionadas ao vazamento de dados privados.

Para investigar o problema, os autores realizaram uma análise baseada em perguntas frequentes no Stack Overflow e na avaliação de repositórios públicos no GitHub, com o objetivo de mapear as principais causas dessas falhas. A partir disso, formularam definições formais das vulnerabilidades identificadas e desenvolveram uma ferramenta chamada PDChecker¹⁵, voltada à detecção automática de vazamentos de dados privados em contratos inteligentes desenvolvidos com o Hyperledger Fabric.

O trabalho mostrou que as principais origens dessas vulnerabilidades foram: uso de dados privados como argumentos na chamada de funções; utilização desses dados em instruções condicionais (como if e switch); e retorno de dados privados por essas funções.

Além disso, foram conduzidos dois experimentos. No primeiro, testou-se a capacidade do PDChecker de identificar as três causas mencionadas de falhas, tendo como resultado uma acurácia superior a 94%. No segundo experimento, o PDChecker foi comparado a outras ferramentas de análise de segurança de chaincodes (HFContract-Fuzzer, chaincode-scanner, HFCCT, chaincode-analyzer, CCDetector e LVTool), considerando não apenas vulnerabilidades de privacidade, mas também outros tipos, como não determinismo (interno e externo), falhas e inseguranças de projeto. Neste experimento, o PDChecker apresentou desempenho superior, sendo a ferramenta com os melhores resultados no conjunto de testes aplicados.

Como recomendações práticas, os autores sugerem que dados privados, ou utilizados para derivar informações privadas, sejam sempre enviados por meio da função `GetTransient()`, evitando-se passá-los diretamente como argumentos. Além disso, funções que consultam dados privados devem ser restritas a operações de leitura, utilizando-se o comando `peer chaincode query` em vez de `peer chaincode invoke`. Recomenda-se também evitar o uso de dados

¹⁵ github.com/zm-stack/PDChecker

privados como condições em estruturas de controle de fluxo, ou como valores de retorno em funções de atualização de estado.

Como trabalho futuro, os autores propõem a expansão da ferramenta PDChecker para também suportar contratos desenvolvidos em Java e Node.js, além da incorporação de novas técnicas que aprimorem a acurácia da detecção. De forma geral, o trabalho se mostra relevante por realizar uma análise fundamentada em dados reais, levantando as principais causas de vazamento de dados. Em particular, destaca-se o valor da ferramenta desenvolvida, que representa uma contribuição para a mitigação de riscos em soluções baseadas em blockchain. O tema abordado é especialmente sensível, principalmente quando relacionado à gestão de dados privados em setores como o educacional, onde a segurança da informação deve ser rigorosamente garantida.

Trustworthiness Evaluation for Permissioned Blockchain-Enabled Applications

O trabalho de (Cha et al., 2022) busca estabelecer diretrizes de avaliação e confiabilidade para sistemas de blockchains permissionadas. Os autores entendem que aplicações de blockchains públicas normalmente estão associadas às trocas de criptomoedas e, para outros casos de uso, blockchains permissionadas tendem a ser a melhor opção. Nesse sentido, para que estas aplicações possuam a aderência de usuários, elas precisam garantir que sejam confiáveis e seguras.

Os autores então propõem um framework de gerenciamento de riscos de segurança para aplicações em blockchains permissionadas. O estudo classifica as medidas de proteção em 6 categorias. Classificando práticas de segurança da informação já estabelecidas nestas categorias e incluindo controles específicos relacionados a blockchains permissionadas. As diretrizes de segurança citadas como base são: ISO/IEC (Organização Internacional de Normalização/-Comissão Eletrotécnica Internacional) 27001, ISO/IEC 27002, PCI DSS (Padrão de Segurança de Dados da Indústria de Cartões de Pagamento) e Controles CIS (Controles Críticos de Segurança).

Um resumo das categorias e dos controles de segurança propostos são:

Segurança de perímetro: Inclui segurança de perímetro lógica, como firewalls e controles de acesso à rede, e segurança de perímetro física, evitando que pessoas não autorizadas acessem áreas com recursos protegidos.

Segurança de Host: software antivírus, firewall de software, backups, gerenciamento de identidade, controle de acesso e etc., nos nós participantes e dispositivos associados. E gerenciamento seguro de chaves, como HSMs (Módulos de Segurança de Hardware).

Gerenciamento de Nós e Segurança de Rede: monitoramento da operação dos nós, administração adequada das permissões de cada nó, impedimento para que nós não autorizados entrem na rede, etc.

Segurança do mecanismo de consenso: garantia da execução correta do algoritmo, determinar de forma antecipada o objetivo de quantos nós bizantinos são tolerados na rede, identificação dos recursos mínimos para manter as operações da rede, tratar incidentes relaci-

onados a transações maliciosas, delegar o gerenciamento de nós a partes independentes e ser transparente na gestão de dados privados.

Segurança de aplicação: ter um modelo de ameaça da aplicação, definir requisitos de segurança antes do desenvolvimento da aplicação, fornecer guias de segurança aos usuários, revisão manual e automática de código de contratos inteligentes e aplicações de back-end e front-end que interagem com a blockchain. Testes unitários, de integração, de vulnerabilidades e de penetração. Separação dos estágios de desenvolvimento, teste e produção e estabelecer mecanismos de atualização apropriados da aplicação.

Segurança da organização: organizações envolvidas na rede devem estabelecer e manter uma documentação de procedimentos e informações de segurança, de modo a evitar ambiguidades, designar um responsável pela segurança da informação, estabelecer treinamentos e realizar verificações de vulnerabilidades, testes de penetração e mesmo de engenharia social de modo a encontrar deficiências nas partes envolvidas.

Como forma de validação do framework proposto, o artigo elabora um checklist de segurança e o aplica em uma organização específica, apresentando a taxa de conformidade de cada item avaliado. Além disso, são realizados quatro experimentos com o objetivo de analisar a integridade e a disponibilidade de uma rede que utiliza o algoritmo PBFT (Practical Byzantine Fault Tolerance). Nestes experimentos, mais de um terço dos nós são encerrados de forma inesperada. Os resultados indicam que redes compostas por nós independentes entre si — ou seja, em que a falha de um nó não compromete a operação dos demais — tendem a ser mais resilientes a falhas.

Entre as limitações do estudo, os autores destacam a ausência de uma aplicação prática do framework em um ambiente real de blockchain permissionada, o que poderia revelar pontos de melhoria. Além disso, ressaltam que o modelo proposto poderia ainda ser integrado com outros padrões de segurança como ISO/IEC 27001, ISO/IEC 22301, ISO/IEC 27017.

De modo geral, é um artigo relevante por trazer medidas de segurança para aplicações de blockchains permissionadas de um ponto de vista bastante completo, tratando desde a segurança do perímetro físico dos recursos dos nós até a análise de código dos contratos inteligentes. Nesse sentido, são medidas que podem ser úteis para garantir a segurança de aplicações reais de blockchains permissionadas. No entanto, seria desejável que os experimentos tivessem sido descritos com maior nível de detalhamento.

A Vulnerability Detection Framework for Hyperledger Fabric Smart Contracts Based on Dynamic and Static Analysis

O artigo de (Li et al., 2022) analisa o crescimento da blockchain e sua aplicação em diferentes domínios, como saúde, cadeia de suprimentos e finanças. E dentro desse contexto, contratos inteligentes são destacados como ferramentas essenciais para o desenvolvimento de aplicações descentralizadas. Contudo, relatam também que, conforme os contratos se tornam cada vez mais complexos, mais falhas de segurança tendem a surgir, o que pode acarretar prejuízos financeiros significativos.

Diante desse cenário, os autores abordam o problema da ausência de ferramentas específicas para a detecção de vulnerabilidades em contratos inteligentes desenvolvidos para a plataforma Hyperledger Fabric. Embora existam soluções consolidadas no ecossistema Ethereum, os autores argumentam que, para o Fabric, ainda há uma lacuna. Como solução, é proposta a ferramenta HFCCT¹⁶, voltada para a detecção automática de vulnerabilidades em chaincodes escritos na linguagem Go.

A ferramenta proposta possui duas funcionalidades principais: execução simbólica dinâmica, responsável por mapear cada caminho condicional possível de execução e testá-lo com valores simbólicos; e análise estática baseada em árvore sintática abstrata (Abstract Syntax Tree - AST), que aplica um conjunto de regras pré-definidas sobre as informações extraídas da AST para identificar potenciais vulnerabilidades.

Para avaliar a eficácia da ferramenta, os autores realizaram experimentos com quinze chaincodes disponíveis no GitHub. Após a remoção manual de falsos positivos, os resultados apontaram que 10% dos erros estavam relacionados a falhas sintáticas no uso da linguagem Go; 15% a comportamentos não determinísticos, como uso de números aleatórios, iteração sobre estruturas do tipo Map e utilização de timestamps do sistema; 30% a funções de consulta por intervalo, como `GetQueryResult()`; e 45% a erros não tratados.

Como trabalhos futuros, os autores sugerem aprimorar o conjunto de dados utilizados, refinar a categorização de vulnerabilidades em chaincodes e melhorar os métodos de detecção, com o objetivo de reduzir falsos positivos e aumentar a precisão da ferramenta.

De modo geral, o artigo trata de uma questão relevante, que é a detecção automática de vulnerabilidades em chaincodes. Uma área que, de fato, carece de soluções mais maduras. O método utilizado para desenvolver a ferramenta, utilizando execução simbólica e AST, mostra-se promissora. Por outro lado, ainda é uma ferramenta que necessita de mais avaliações e aperfeiçoamento para ter sua robustez comprovada em cenários reais.

Discussão

Estes trabalhos buscam fazer uma análise de aspectos de segurança em blockchains. Alguns deles tratam as blockchains de maneira mais geral e trazendo vários aspectos de blockchains públicas como (Homoliak et al., 2021) ou (Hamdi; Fourati; Ayed, 2024), mas ainda são relevantes por apresentarem um panorama mais amplo da segurança em blockchains e métodos de categorização e taxonomia para avaliar as vulnerabilidades e ataques. Estes trabalhos servem como uma importante referência para futuras categorizações.

Analisando-se os trabalhos voltados a blockchains permissionadas, percebe-se que, todos os autores utilizaram Hyperledger Fabric como referência. Mostrando que, nesse momento, essa é a principal ferramenta de aplicações deste tipo de blockchain. Nesse contexto, houve tanto trabalhos que tiveram um papel mais voltado para listagem de vulnerabilidades como (Brotsis et al., 2020), bem como trabalhos de análise de desempenho em situações de ataque como (La Salle et al., 2023) e (Jayadev; Moradpoor; Petrovski, 2024) e trabalhos que propuse-

¹⁶ github.com/bugfrog1024/HFCCT

ram ferramentas para detecção de vulnerabilidades como (Chen et al., 2024) e (Li et al., 2022). Nesse sentido, percebe-se que estes são pontos relevantes a serem aprofundados.

6 DESENVOLVIMENTO

6.1 VULNERABILIDADES E ATAQUES EM BLOCKCHAINS PERMISSIONADAS

Nesta seção, são apresentadas as vulnerabilidades, ataques e medidas de prevenção e mitigação associados a blockchains permissionadas. Esses elementos foram divididos em duas listas: a primeira contendo as vulnerabilidades relativas à estrutura do Fabric, à rede, ao protocolo de consenso e aos usuários; enquanto a segunda lista está voltada aos contratos inteligentes.

Com relação aos contratos, esta é uma "camada" das redes blockchains que recebeu bastante enfoque nos trabalhos analisados e, além disso, dispõe de mais ferramentas voltadas para análises de segurança. Portanto, preferiu-se separar seus elementos em uma nova lista, de modo a abordá-los de maneira mais qualificada.

6.1.1 Vulnerabilidades em componentes de infraestrutura e rede

- **Vulnerabilidade:** MSP (Membership Service Provider) Centralizado
 - **Comprometimento do MSP:** No Hyperledger Fabric, o MSP atua como a autoridade responsável pelo gerenciamento de identidades e permissões na rede. Assim, seu comprometimento representa um ponto crítico de falha, pois um MSP malicioso pode adicionar e remover identidades e controlar o nível de acesso de cada elemento da rede. Esse controle indevido pode comprometer a integridade da rede, permitindo a manipulação de transações e o enfraquecimento da confiança entre os participantes (Brotsis et al., 2020; Hamdi; Fourati; Ayed, 2024).
 - * **Prevenção e mitigação:** Ambiente de Execução Confiável (TEE) para assegurar o MSP. As capacidades oferecidas pelo TEE podem proteger o MSP (Membership Service Provider) em todas as fases de sua invocação, incluindo: o registro de cada nó, a assinatura de transações e sua verificação. Um MSP habilitado para TEE também pode mitigar diversos riscos de privacidade no Fabric e reforçar as defesas da rede contra possíveis ataques (Hamdi; Fourati; Ayed, 2024).
- **Vulnerabilidade:** PKI's vulneráveis
 - **Ataques relacionados à PKI:** A infraestrutura de chaves públicas é empregada em redes blockchain para autenticar entidades e garantir a integridade das informações. Entretanto, esse modelo tradicional apresenta riscos associados a pontos únicos de falha e possível exposição de dados sensíveis. Como a gestão de certificados depende de uma autoridade certificadora confiável, o comprometimento desse elemento pode permitir a emissão de credenciais fraudulentas, violar a confidencia-

lidade dos usuários e afetar a integridade da rede. Dessa forma, a CA torna-se um componente crítico cuja falha pode comprometer todo o ambiente permissionado.

– **Ataques Sybil:**

- * **Prevenção e mitigação:** O trabalho de (Hamdi; Fourati; Ayed, 2024) traz algumas soluções apresentadas por outros autores para ataques Sybil: uma solução em que cada nó analisa o comportamento dos demais. Ao identificar rapidamente esses comportamentos, colocar os nós maliciosos em uma lista não de nós não confiáveis e alertar os demais; e TrustChain, um sistema que facilita transações confiáveis entre partes desconhecidas.

• **Vulnerabilidade:** Serviço de Ordenação Malicioso

- **Comprometimento do Serviço de Ordenação:** O serviço de ordenação de uma rede Fabric é o componente exclusivamente responsável por atingir consenso sobre os blocos criados na blockchain. Diferentemente do Bitcoin e do Ethereum, o Fabric utiliza um protocolo de consenso determinístico, em vez de algoritmos de consenso probabilísticos. Essa escolha garante que os blocos gerados pelo OS sejam finais e corretos. Contudo, caso o OS seja comprometido, diversos ataques podem ocorrer, incluindo ataques de sabotagem, forks intencionais, manipulação do tamanho de blocos, manipulação do tempo de batch, retenção de blocos e reordenação de transações.

• **Vulnerabilidade:** Redes não confiáveis compartilhadas; Aspectos de DNS e dos protocolos de roteamento; Design de protocolos p2p.

- **Ataques Man-In-The-Middle (MITM):** Esse é um dos tipos mais comuns de vulnerabilidades em redes, em que um invasor atua como intermediário para interceptar o tráfego e obter acesso a informações pessoais ou sigilosas dos usuários. Uma vez que o atacante consegue interceptar os dados no canal de comunicação, ele pode explorá-los para outros fins maliciosos. O ataque MITM aproveita fragilidades nos mecanismos de troca de chaves e nos sistemas de armazenamento para obter chaves secretas.

- * **Prevenção e mitigação:** Utilizar um mecanismo de autenticação avançado que impeça os invasores de se infiltrarem no sistema.

- **Ataques de DNS:** surgem a partir de envenenamento de cache, que afeta principalmente blockchains que utilizam DNS centralizado para recuperar pares online a partir de uma lista fixa de servidores DNS (DNS seeders).

- * **Prevenção e mitigação:** Extensão de segurança do DNS, chamada DNSSEC, que fornece autenticação e integridade dos dados. Além do DNS padrão, a resolução de nomes também pode ser feita utilizando servidores DNS alternativos.

- **Ataque de particionamento:** em redes blockchain é uma técnica em que um invasor tenta dividir a rede peer-to-peer (P2P) em partes isoladas, impedindo que os nós se comuniquem entre si de maneira eficaz.
 - * **Prevenção e mitigação:** O trabalho de (Hamdi; Fourati; Ayed, 2024) cita uma pesquisa com base na análise do Hyperledger Fabric, onde os autores derivaram uma fórmula para configurar o consenso Proof-of-Authority (PoA) a fim de evitar a formação de verdades concorrentes quando a rede é acidentalmente particionada.
- **Ataques de roteamento:** são desvios de tráfego, sequestros de rotas ou ataques de negação de serviço (DoS). Mais suscetíveis em blockchains públicas, onde além da simples interceptação ou modificação de dados, esses ataques podem levar à partição da rede, o que, por sua vez, aumenta os riscos de ataques de 51% ou ataques de mineração egoísta.
 - * **Prevenção e mitigação:** As contramedidas incluem nós com múltiplas conexões (multi-homed) ou o uso de VPN para diversidade de rotas, escolha de pares adicionais cujas conexões não passem pelos mesmos sistemas autônomos, preferência por pares hospedados no mesmo sistema autônomo dentro do mesmo prefixo /24 (para reduzir o risco de partições), e obtenção do mesmo bloco a partir de múltiplos pares.
- **Ataques de sequestro BGP:** O invasor explora uma vulnerabilidade presente nos operadores de rede para interceptar e modificar o roteamento do tráfego entre gateways. Em sistemas blockchain, esse tipo de ataque divide o poder de mineração (ou o servidor de um pool de mineração) em vários grupos, causando atrasos na propagação de blocos na rede. Além disso, um ataque de roteamento BGP pode ser usado para dividir a rede em várias redes isoladas que não conseguem se comunicar entre si (Hamdi; Fourati; Ayed, 2024).
 - * **Prevenção e mitigação:** Segundo (Hamdi; Fourati; Ayed, 2024) o protocolo BGPsec, que impede que tráfego indesejado obtenha acesso ao sistema, é uma abordagem importante utilizada para combater ataques de sequestro BGP. Monitorar e validar o tráfego de rede em momentos pré-determinados também pode ser benéfico.
- **Ataques de eclipse:** têm como objetivo sequestrar todas as conexões de um nó com seus pares em uma rede blockchain. Como consequência, todo o tráfego recebido e enviado pelo nó fica sob total controle do atacante. Ataques de eclipse surgem a partir de ameaças relacionadas ao DNS e ao roteamento na rede, podendo ser resultado de vulnerabilidades nos protocolos peer-to-peer.

- * **Prevenção e mitigação:** A melhoria na aleatoriedade da escolha de pares. Uso de links de rede redundantes ou conexões "out-of-band" para verificar transações (por exemplo, por meio de um explorador de blockchain). Uso de redes de disseminação (gossip) "out-of-band" (por exemplo, servidores Web), que se comunicam com clientes leves para trocar suas visões sobre a blockchain (isto é, cabeçalhos de blocos) juntamente com o tráfego nativo da Web. Vale também destacar que as contramedidas para ataques de DNS e roteamento são aplicáveis nesse contexto.
- **Ataques DDoS à conectividade dos nós de consenso:** podem resultar na perda do poder de consenso, impedindo que esses nós sejam recompensados. Já em nós validadores, esse tipo de ataque causa a interrupção de alguns serviços dependentes da blockchain.
 - * **Prevenção e mitigação:** Uma forma de mitigação é emparelhar apenas com nós permitidos. Métodos para evitar DDoS volumétricos incluem: filtragem on-premise (usando um dispositivo de rede adicional); filtragem em nuvem (como o redirecionamento do tráfego para um serviço de mitigação de DDoS na nuvem quando um ataque é detectado); filtragem híbrida (combinação das anteriores).
- **Vulnerabilidade:** Violação de premissas do protocolo de consenso
 - **Centralização Adversarial do Poder de Consenso:** Nesses ataques, uma premissa fundamental do projeto - a distribuição descentralizada do poder de consenso - é violada. Exemplos incluem: ataques de 51% em protocolos PoW (Prova de Trabalho) e PoS (Prova de Participação); Ataques com 1/3 de nós bizantinos em protocolos BFT (Tolerância a Falhas Bizantinas) e suas combinações.
 - * **Prevenção e mitigação:** Incentivo à descentralização através de esquemas que: recompensam a participação honesta; desencorajam ou punem violações do protocolo. Análise estatística de anomalias súbitas na distribuição histórica do poder de consenso entre os nós.
 - **Violação de premissas do protocolo da rede:** Protocolos que assumem entrega de mensagens em redes síncronas ou parcialmente síncronas inevitavelmente falham quando essa premissa não se sustenta. Por exemplo: essa suposição pode ser violada em protocolos BFT por um network scheduler (agendador de rede) imprevisível, como demonstrado no protocolo PBFT.
 - * **Prevenção e mitigação:** Desenvolvimento de protocolos BFT assíncronos, que podem utilizar: criptografia baseada em limiares (threshold-based cryptography); broadcast confiável e consistente de mensagens.
- **Vulnerabilidade:** Particionamento dos nós de consenso

- **Ataques em "shards":** Sharding significa que os nós de consenso são distribuídos entre subgrupos (ou shards), de forma que cada nó valida apenas as transações do seu próprio grupo. Esta fragmentação pode tornar mais fácil para um invasor comprometer um único fragmento do que a rede inteira.
 - * **Prevenção e mitigação:** A principal técnica de mitigação é garantir uma distribuição verdadeiramente aleatória dos nós entre os fragmentos, minimizando assim a possibilidade de um adversário manipular a aleatoriedade usada na distribuição.
- **Vulnerabilidade:** Previsibilidade na eleição de um nó líder
 - **DoS contra um Líder:** Líderes de rodada em protocolos BFT podem sofrer um ataque DoS, já que sua liderança é conhecida antes do início da rodada, o que pode resultar na reinicialização da rodada.
 - * **Prevenção e mitigação:** Seleção privada de líder: Um nó pode determinar de forma privada se é um líder potencial usando uma Função Aleatória Verificável (VRF - Verifiable Random Function) e imediatamente publicar um candidato a bloco. Dessa forma, após a publicação desses dados, já é tarde demais para um ataque DoS contra o nó. Assinaturas de Limiar em Configuração sem Líder: Outra abordagem para lidar com esse ataque envolve a agregação de assinaturas de limiar (threshold signatures) em um ambiente sem líder.
- **Vulnerabilidade:** Ausência de checkpoints e ausência de rotação de chaves
 - **Corrupção Posterior:** este é um caso específico de violação das premissas do protocolo em que o poder adversarial de consenso atinge 2/3. Nesse cenário, o adversário precisa roubar chaves privadas de (potencialmente) 2/3 dos nós de consenso já "aposentados"(inativos) para então reexecutar o protocolo de consenso e reescrever o histórico da blockchain.
 - * **Prevenção e mitigação:** Criptografia com Evolução de Chaves (key-evolving cryptography) e assinaturas digitais forward-secure (ex.: árvores de certificados d-árias), que exigem: evolução periódica das chaves privadas; apagamento definitivo de chaves já utilizadas. Checkpoints irreversíveis após um número fixo de blocos ou transações contextuais, que incorporam o hash de um bloco recente válido na própria transação.
- **Vulnerabilidade:** Peers Endossantes Identificados
 - **Ataques de DoS contra endossantes selecionados:** podem ser realizados, seja para impedir que transações específicas sejam incluídas no ledger (livro-razão) ou para degradar o desempenho da rede.

* **Prevenção e mitigação:** Anonimizar ou pseudo-anonimizar a identidade dos peers de endosso.

- **Vulnerabilidade:** Criptografia Vulnerável

- **Ataques quânticos:** Os algoritmos criptográficos usados em blockchains atualmente se baseiam na dificuldade computacional de problemas como o logaritmo discreto em curvas elípticas ou a fatoração de grandes inteiros. No entanto, o avanço rápido dos algoritmos quânticos aceleram tarefas como geração de hashes e quebra de chaves criptográficas. Isso compromete tanto a criptografia de chave pública quanto as funções de hash.

- * **Prevenção e mitigação:** Diante desse cenário, surgiu a área de pesquisa chamada blockchain pós-quântica, que busca soluções para esses ataques.

- **Vulnerabilidade:** Usuários da Rede

- **Ataques de Phishing:** No contexto de ataques a blockchains, o phishing é uma forma de engenharia social que visa enganar indivíduos para obter acesso não autorizado a contas relacionadas à blockchain, chaves privadas, senhas ou outras informações confidenciais. São utilizadas diversas técnicas, como envio de e-mails enganosos, criação de sites ou aplicativos falsos, ou uso de redes sociais para induzir as vítimas a revelar suas informações sensíveis. Ao obter controle sobre os ativos de um usuário, os invasores podem roubar fundos, manipular transações ou comprometer a integridade da rede.

- * **Prevenção e mitigação:** Os usuários devem estar atentos à verificação da autenticidade de sites e aplicativos, conferir cuidadosamente a identidade dos remetentes de e-mails e evitar clicar em links suspeitos ou fornecer informações pessoais a fontes desconhecidas. A adoção de medidas de segurança robustas, como autenticação de dois fatores (2FA), uso de carteiras físicas para armazenar chaves privadas e atualização regular de softwares, pode reduzir significativamente o risco de se tornar vítima de um ataque de phishing.

- **Ataque Wormhole:** Se um peer comportar-se de forma maliciosa e coludir com um adversário externo ao canal, esse ataque pode resultar no vazamento de informações privadas de todos os peers de um canal específico. Esse problema evidencia que o mecanismo de controle de acesso do Fabric depende da confiança em cada peer dentro de um canal.

6.1.2 Vulnerabilidades em Contratos Inteligentes

A listagem apresentada nesta subseção sintetiza vulnerabilidades identificadas na literatura científica, organizando-as em uma classificação que reflete tanto características arquite-

turais específicas do Fabric, quanto problemas genéricos de programação, que adquirem criticidade particular no contexto de sistemas blockchain.

A classificação proposta compreende quatro dimensões: vulnerabilidades inerentes à arquitetura do Fabric (*phantom reads*, inconsistências temporais entre fases de transação); falhas no gerenciamento de confidencialidade via coleções de dados privados; bugs de implementação; e fontes de não determinismo que violam as premissas do fluxo das transações no Fabric.

Uma característica distintiva do Fabric é a adoção de linguagens de propósito geral (Go, Java, JavaScript) para desenvolvimento de chaincodes, em contraste com abordagens baseadas em linguagens específicas de domínio, como Solidity no Ethereum. Esta decisão arquitetural resulta em uma redução na barreira de entrada, na medida em que os desenvolvedores não precisam aprender uma linguagem nativa para construir as aplicações. Por outro lado, o uso incorreto da linguagem pode também causar problemas se mal utilizadas em um sistema distribuído, determinístico e imutável.

Como praticamente a totalidade dos trabalhos encontrados relacionados à segurança de contratos inteligentes em Fabric, tratam especificamente da utilização do Go enquanto linguagem de desenvolvimento, esta lista também levará isso em conta e, em alguns itens, tratará de questões específicas do Go, ao invés de especificidades de Java ou Javascript. Mesmo assim, acredita-se que de maneira geral, os fundamentos apontados por esses itens também podem ser aplicados para o desenvolvimento de contratos utilizando outras linguagens.

- **Vulnerabilidades decorrentes da arquitetura do Fabric:** se refere a vulnerabilidades decorrentes do uso inadequado de mecanismos específicos do modelo *execute-order-validate* do Fabric.
 - ***Phantom read:*** O Hyperledger Fabric fornece alguns métodos de consulta por intervalo para acessar os bancos de dados de estado, como `GetQueryResult()`, `GetPrivateDataQueryResult()` e `GetHistoryForKey()`. Esses métodos são executados durante a fase de endosso da transação, mas não são reexecutados durante a fase de validação. O que significa que uma eventual alteração nos dados entre a fase de endosso e a fase de validação não será detectada. Portanto, modificações no ledger baseadas nos resultados da query podem operar sobre dados obsoletos, causando phantom reads ou inconsistências lógicas.
 - ***Read your write:*** No Hyperledger Fabric, a operação de escrever os dados de uma transação no ledger somente é executada após a transação ser concluída e verificada. Portanto, se uma leitura é executada logo após uma escrita, ela não irá obter o valor atualizado esperado como retorno.
 - **Invocação de chaincode entre canais:** introduzem complexidade significativa e podem levar a violações dos princípios de isolamento e privacidade que os canais foram projetados para garantir.

- **Vazamento de dados Privados:** refere-se a exposição não autorizada de informações sensíveis devido ao uso inadequado de Private Data Collections (PDCs) do Fabric.
 - **Dados privados em argumentos de funções:** Parâmetros passados em invocações de chaincode são incluídos na proposta de transação e gravados no ledger público, tornando-se acessíveis a todos os peers da rede. Dados sensíveis devem ser transmitidos através de *transient data*, que não são persistidos no ledger.
 - **Dados privados em operações condicionais:** o fluxo de execução de um programa é afetado pelas operações condicionais. Se dados privados são utilizados nessas operações, um eventual atacante pode começar a inferir informações privadas a partir dos diferentes caminhos que um programa percorre. Por exemplo, se o saldo de uma conta é negativo e uma operação de saque não autorizada é informada em uma transação, a informação sobre a situação do saldo é vazada para todos os participantes.
 - **Dados privados no *payload* de retorno:** uma chaincode pode retornar dados em uma função dentro do *payload* de uma transação, que poderá ser acessado por todos os participantes. Nesse sentido, chaincodes não podem incluir dados privados que sejam incluídos no payload em retorno de funções.
- **Não determinismo interno:** Trata tanto de fontes de dados que podem ter valores diferentes nos diferentes pares - como variáveis globais e campos da chaincode - como processos de execução não determinística, onde a execução de uma mesma função em diferentes peers pode levar a resultados diferentes e incertos.
 - **Variáveis globais de estado e declaração de campo:** Variáveis globais e campos da struct da chaincode podem ser alterados localmente. Devido à política de endosso, cada nó par não executa necessariamente todas as transações. Portanto, os valores armazenados dessa maneira podem ficar inconsistentes entre os nós.
 - **Range map:** Devido aos detalhes internos da implementação da linguagem Go, a iteração sobre estruturas do tipo *map* com o uso de *range* é não-determinística, a ordem das chaves pode variar. Assim, é necessário atenção ao manipular dados escritos do *ledger* dessa maneira.
 - **Reificação de objetos (ponteiros):** No caso de manipulação de variáveis por ponteiros, o valor recuperado em um endereço específico muito provavelmente irá variar entre ambientes. Nesse sentido, deve haver cuidado para não armazenar o endereço da variável no ledger.
 - **Concorrência no programa:** Se programas concorrentes não forem tratados corretamente, é fácil causar condições de corrida que resultam em execução não determinística, resultando em falha da transação.

- **Timestamp do sistema:** É difícil garantir que os timestamps sejam executados de forma simultânea em todos os peers. Sua utilização de modo incorreto resultará na falha do processo de endosso.
- **Geração de números aleatórios:** Como a chaincode é simulada de forma independente por cada par na fase de endosso, os valores gerados aleatoriamente em cada par serão diferentes. Aplicando-se os mesmos efeitos citados anteriormente.
- **Não determinismo externo:** refere-se às chamadas ou execução de recursos externos à blockchain, onde não há um controle sobre o resultado retornado, podendo gerar inconsistência entre os pares.
 - **Acesso a serviços web, acesso a bibliotecas externas e arquivos do sistema e execução de comandos do sistema:** Chamadas externas não determinísticas podem ter diferentes lógicas de execução ou fontes de dados incertas em diferentes pares. Por exemplo, se as chaincodes fazem chamadas a serviços web, quando diferentes pares executam essas chaincodes para obter dados do site, esse site pode retornar dados diferentes para cada par, causando inconsistência.
- **Vulnerabilidades de implementação não específicas da plataforma:** classe de vulnerabilidades que existem independentemente da tecnologia blockchain, mas cujas consequências são amplificadas no contexto de smart contracts devido a características como imutabilidade, irreversibilidade de transações, e alto valor dos ativos gerenciados.
 - **Overflow matemático:** se uma chaincode soma valores de tokens ou ativos sem a validação adequada, um atacante poderia explorar isso para criar transações que, ao ultrapassarem o limite máximo do tipo numérico, resultem em valores negativos ou muito pequenos, permitindo, por exemplo, transferências fraudulentas ou manipulação de saldos.
 - **Overflow em conversão de tipos:** ocorre quando dados são convertidos entre tipos diferentes (como de string para inteiro, ou de int64 para int32) sem validação apropriada dos limites. Um atacante pode explorar isso enviando valores que são válidos no tipo original, mas que se tornam corrompidos após a conversão. Isso pode permitir, por exemplo, o bypass de validações de limites, a manipulação de quantidades em transferências de ativos ou a corrupção de dados no world state, comprometendo a integridade das transações.
 - **Erros não tratados:** Em Go, não há o mecanismo de exceção, fazendo com que os desenvolvedores devam tratar os erros retornados pelas funções de maneira apropriada.
 - **Parâmetros não checados:** comum em qualquer cenário de desenvolvimento. As entradas das funções devem ser verificadas de modo a evitar eventuais falhas durante a execução da função.

6.1.3 Medidas de prevenção em contratos inteligentes

A literatura apresenta um conjunto diversificado de estratégias que vão desde práticas básicas de implementação segura até métodos formais sofisticados de verificação. Estas medidas podem ser categorizadas em dois grupos principais: práticas de implementação segura, que estabelecem configurações e controles no ambiente de execução; e métodos de análise e verificação de código, que empregam diferentes técnicas para identificar vulnerabilidades antes da implantação dos contratos.

- Práticas de implementação segura:
 - Limitações de tempo na execução da chaincode podem ser aplicadas;
 - A execução da chaincode pode ser feita sem privilégios root;
 - Verificação dos argumentos de entrada da chaincode, incluindo caracteres de escape.
- Métodos de análise e verificação do código:
 - Métodos formais:
 - * Prova de teoremas: Um dos métodos mais comuns para formalizar contratos inteligentes. A lógica simbólica (com axiomas ou premissas) é usada na prova de teoremas para demonstrar as propriedades de segurança necessárias que aumentam a precisão dos contratos.
 - * Modelagem formal: Esse método formaliza os riscos dos contratos inteligentes por meio de declarações precisas que definem a relação entre os componentes dos contratos, resultando em comunicação clara e sem ambiguidades e resultados replicáveis/reproduzíveis.
 - Métodos de análise:
 - * Análise estática: envolve essencialmente a avaliação do código de um programa sem executá-lo. Examina a estrutura do código, a saída e os grafos de fluxo de controle para identificar problemas comuns de segurança. Ferramentas de análise estática podem ser dedicadas à detecção automática de bugs ou à verificação automática de propriedades genéricas.
 - * Análise dinâmica: é um método que verifica uma aplicação em tempo de execução. Ela detecta vulnerabilidades que podem não ser descobertas pela análise estática.
 - Técnicas avançadas de análise:
 - * Execução simbólica: é a execução de código usando símbolos em vez de valores concretos das variáveis. É um método para identificar entradas que fazem diferentes partes do programa serem executadas. A execução simbólica roda o

programa de forma abstrata e pode explorar, simultaneamente, múltiplos caminhos que o programa pode seguir com diferentes entradas.

- * **Interpretação abstrata:** é um método utilizado na avaliação de ataques, que consiste em abstrair o comportamento de um programa ou sistema para analisar suas propriedades e identificar vulnerabilidades potenciais. Fornece um arcabouço para analisar o comportamento de programas por meio da aproximação de suas possíveis execuções e da inferência de propriedades. No contexto de análise de ataques em blockchain, pode ser aplicada para analisar contratos inteligentes e identificar falhas de segurança, criando modelos abstratos do comportamento do contrato e utilizando esses modelos para identificar vulnerabilidades.
- * **Resolução de restrições (constraint solving):** é um método utilizado na análise de ataques que consiste na resolução de restrições para analisar e verificar o comportamento de um sistema. É especialmente útil na análise de propriedades de segurança e correção de contratos inteligentes implantados na blockchain. Pode detectar diversos tipos de vulnerabilidades, como estouro aritmético, reentrância e acesso não autorizado. Ao definir restrições que representam as condições sob as quais essas vulnerabilidades ocorrem, torna-se possível analisá-las sistematicamente.
- * **Correspondência de padrões (pattern matching):** consiste em verificar se uma determinada sequência de tokens contém os elementos de um padrão específico. Envolve a comparação entre um padrão teórico previsto e um padrão empírico observado. Esse processo é usado para encontrar vulnerabilidades potenciais e classificá-las por nível de risco. Padrões de código vulnerável são identificados com base em conhecimento estatístico obtido a partir de vulnerabilidades conhecidas em contratos inteligentes. Embora o método não garanta precisão, é mais leve e rápido em comparação com métodos formais, e sua precisão pode ser aprimorada com ajustes finos.

6.2 EXPERIMENTOS

6.2.1 Análise de Contratos Inteligentes

Diante dos itens apresentados, observa-se que o ecossistema de blockchain abrange uma ampla variedade de vulnerabilidades, ameaças e possíveis vetores de ataque. Essa diversidade decorre tanto da natureza distribuída e descentralizada das redes blockchain quanto da multiplicidade de camadas que as compõem, desde a infraestrutura de rede e mecanismos de consenso até o armazenamento de dados e, por fim, os contratos inteligentes que executam a lógica de negócio.

Nesse contexto, é fundamental compreender não apenas o funcionamento dessas camadas, mas também os riscos que cada uma delas apresenta, a fim de adotar mecanismos de mitigação capazes de evitar falhas de segurança e proteger a integridade das aplicações. Assim, embora o panorama levantado seja amplo, o escopo do presente trabalho concentra-se em um subconjunto dessas vulnerabilidades, selecionado com base na revisão da literatura e na relevância prática em aplicações de blockchains permissionadas.

A literatura analisada, incluindo os trabalhos de (Chen et al., 2024), (Li et al., 2022) e (Hamdi; Fourati; Ayed, 2024), destaca particularmente a criticidade dos contratos inteligentes no contexto de blockchains permissionadas. Essas peças de software são responsáveis pela implementação das regras que governam as operações na rede e, portanto, pela segurança e integridade das transações registradas. Vulnerabilidades nesses contratos podem comprometer o correto funcionamento do sistema, possibilitar ataques e até mesmo expor informações confidenciais.

Considerando essa relevância, este trabalho concentra sua investigação na camada de contratos inteligentes, aprofundando a análise sobre as vulnerabilidades associadas à sua implementação e avaliando ferramentas capazes de detectá-las de forma automatizada. Especificamente, será analisado um contrato inteligente vulnerável, de modo a validar as ferramentas, e então, serão analisadas chaincodes de um caso de uso real, com o objetivo de identificar potenciais falhas de segurança observados na literatura.

Para isso, será conduzido um estudo experimental utilizando ferramentas de análise estática de código e linters especializados, permitindo tanto avaliar o grau de segurança dos contratos empregados no caso de uso quanto verificar a eficácia e as limitações das ferramentas de análise na detecção de vulnerabilidades. Adicionalmente, serão discutidas propostas de melhoria para tais ferramentas, fundamentadas na aplicação prática e nas evidências coletadas durante os testes.

6.2.2 Caso de uso: RRDES

O caso de uso utilizado para experimentação no desenvolvimento deste trabalho será a rede blockchain do projeto RRDES. Este projeto surgiu como iniciativa do Laboratório de Segurança em Computação - LabSEC da Universidade Federal de Santa Catarina - UFSC. A ideia deste projeto é construir uma rede de rastreabilidade do ensino superior brasileiro, vinculando essa rede ao Diploma Digital¹, já em implantação pelo Ministério da Educação - MEC.

Um dos objetivos do Diploma Digital é evitar a falsificação de diplomas de graduação, garantindo que uma Instituição de Ensino Superior - IES, vinculada ao MEC, tenha, de fato, emitido o documento e que não tenha sido forjado por entidades não autorizadas². Contudo,

¹ portal.mec.gov.br/diplomadigital/

² PF deflagra operação contra rede de falsificação de diplomas e exercício ilegal de profissões. <https://www.gov.br/pf/pt-br/assuntos/noticias/2025/06/pf-deflagra-operacao-contra-rede-de-falsificacao-de-diplomas-e-exercicio-ilegal-de-profissoes>. Acesso em junho de 2025

por si só, o programa não é capaz de garantir a integridade da IES emissora, ou seja, que de fato o estudante portador do diploma tenha realizado todo o percurso exigido pelo MEC para obter aquele diploma.

É nesse contexto que entra a blockchain do RRDES. Seu objetivo é rastrear o percurso dos alunos de graduação do país. Ao receber como entrada documentos que garantam que o estudante seguiu o percurso proposto pelo seu curso e tenha sido aprovado, será possível afirmar que o diploma emitido para este aluno é íntegro e, de fato, atesta que ele está apto a receber o novo título.

Essa rede é construída utilizando-se o Hyperledger Fabric e, atualmente, recebe como entrada documentos no formato XML e no padrão do diploma digital. Para cada documento há uma chaincode específica responsável por verificar as informações ali contidas e por realizar a leitura e escrita da cadeia de blocos. Como exemplos de documentos inseridos estão: dados regulatórios de cursos e IES, currículos de curso, diplomas, históricos escolares e etc..

De modo geral, rastreabilidade é um contexto no qual a blockchain tem alto potencial de uso enquanto ferramenta de armazenamento de dados (Homoliak et al., 2021). Pois ela garante a preservação do histórico dos dados inseridos e a imutabilidade desses dados. Ideal para preservar o progresso dos estudantes durante a jornada de graduação. Tratando-se de implementação, o objetivo é que exista um conjunto de IES brasileiras que sejam os nós dessa rede blockchain e responsáveis por inserir os documentos e manter essa base de dados distribuída.

Nesse sentido, essa rede possuirá informações sensíveis ao país, pois será um banco de dados do ensino superior brasileiro. Além disso, terá informações confidenciais dos alunos, cuja proteção é essencial. O setor educacional, por sua vez, já foi vítima de vazamento de informações (Naísa; Carvalho, 2021; Petry, 2021). Portanto, é de extrema importância manter essa rede segura, garantindo que não possua vulnerabilidades e que preserve a confidencialidade, disponibilidade e a integridade das informações armazenadas.

6.2.3 Ferramentas de Análise Utilizadas

A partir da análise da literatura, verificou-se que diversas ferramentas foram propostas com o objetivo de auxiliar na detecção de vulnerabilidades em contratos inteligentes, buscando fazer com que potenciais falhas pudessem ser percebidas antes dos códigos entrarem em produção. Principalmente considerando que determinadas falhas em contratos podem significar a perda de ativos e, conseqüentemente, perda financeira.

Dentre as ferramentas propostas, alguns métodos foram empregados, como por exemplo, análise estática, análise dinâmica, *fuzz testing*, grafos de conhecimento e etc. Contudo, poucas destas propostas disponibilizaram o código fonte para que pudesse ser testado pela comunidade. Além disso, do pouco que foi aberto ao público, um número ainda mais restrito possui o código atualizado. Tendo essas considerações em vista, as ferramentas selecionadas foram PDChecker, HFCCT (Hyperledger Fabric Chaincode Testing Tool) e Golangci-lint³.

³ <https://golangci-lint.run/>

O PDChecker se mostrou uma opção pois trata de quinze vulnerabilidades de chaincodes, dentro do conjunto descrito anteriormente e, em especial, busca encontrar problemas relacionados ao vazamento de dados privados, o que é crucial para garantir a segurança das informações armazenadas. Além disso, dentre as ferramentas encontradas, é a mais atualizada, com *commits* realizados nos últimos 18 meses.

O HFCCT, por outro lado, não possui *commits* realizados dentro desse período. Contudo, é uma ferramenta de código aberto e que também se propõe a detectar um conjunto de vulnerabilidades em chaincodes, o que a torna importante por oferecer um contraponto ao PDChecker. Já o Golangci-lint foi selecionado pois em (Hamdi; Fourati; Ayed, 2024) também são utilizados linters da linguagem dos contratos para complementar a detecção de vulnerabilidades. Além disso, essa ferramenta é mantida ativamente pela comunidade do Go e é a mais abrangente na quantidade de potenciais problemas que pode encontrar, visto que é um agregador de diferentes linters.

Portanto, essas ferramentas foram escolhidas tanto por disponibilidade, quanto por contemplarem diferentes abordagens de detecção de falhas em chaincodes desenvolvidas em Go para a plataforma Hyperledger Fabric. A comparação dos resultados desses três analisadores poderá enriquecer a análise e contribuir para o avanço de melhorias na detecção prévia de vulnerabilidades em contratos inteligentes.

6.2.3.1 PDChecker

O PDChecker é uma ferramenta de análise estática desenvolvida em Go como uma extensão do linter Revive⁴, sobre o qual adiciona novas regras de verificação. Seu propósito é detectar vulnerabilidades e más práticas específicas de contratos inteligentes de Hyperledger Fabric escritos também em Go.

O conjunto de verificações do PDChecker abrange as seguintes vulnerabilidades:

- Importação de bibliotecas *blacklisted*
- Invocação de chaincode entre canais
- Variável global
- Phantom read
- Reificação de objetos (ponteiros)
- Dados privados em argumentos
- Dados privados em operações condicionais
- Dados privados no retorno de funções
- Range map
- Read after write

⁴ <https://github.com/mgechev/revive>

- Go routines (concorrência)
- Overflow de conversão
- Overflow matemático
- Erros não tratados
- Parâmetros não checados

O PDChecker realiza análise estática, sem executar o código, e foi projetado para ser leve e de fácil integração a pipelines de desenvolvimento. No entanto, apresenta limitações, como a incapacidade de realizar varreduras recursivas em diretórios, o que exige sua execução individual sobre cada arquivo do código fonte.

6.2.3.2 HFCCT

O HFCCT é uma ferramenta de análise desenvolvida em Python e Go. Seu funcionamento baseia-se em duas abordagens: análise estática e análise simbólica. No entanto, durante os testes realizados, apenas a análise estática pôde ser executada, devido a problemas de compatibilidade com as chaincodes de teste.

Nesse sentido, foi possível, com HFCCT explorar as seguintes vulnerabilidades:

- Importação de bibliotecas *blacklisted*
- Invocação de chaincode entre canais
- Declaração de campo
- Reificação de objetos (ponteiros)
- Range map
- Read after write
- Go routines
- Erros não tratados

Assim como o PDChecker, o HFCCT não realiza busca recursiva por arquivos, sendo necessário executá-lo manualmente em cada arquivo-fonte.

6.2.3.3 Golangci-lint

O Golangci-lint é uma ferramenta utilizada na comunidade Go para detecção de erros, más práticas e violações de estilo em código-fonte. É composto por um conjunto de diferentes ferramentas, como Staticcheck⁵, um dos principais linters de Go, e Gosec⁶, para análise de

⁵ <https://github.com/dominikh/go-tools>

⁶ <https://github.com/securego/gosec>

segurança. Embora não tenha sido projetado especificamente para chaincodes, o Golangci-lint pode auxiliar na identificação de problemas que podem impactar a segurança de contratos inteligentes.

Considerando o foco em contratos inteligentes e as vulnerabilidades citadas anteriormente, essa ferramenta foi executada na chaincode de testes com vistas a descobrir quais das vulnerabilidades apontadas anteriormente poderiam ser detectadas por ela. Sendo estas:

- Erros não tratados
- Variável global
- Overflow de conversão de tipos

Vulnerabilidade	PDChecker	HFCCT	Golangci-lint
Declaração Campo		X	
Range map	X	X	
Ponteiro	X	X	
Read after write	X	X	
Concorrência	X	X	
Importação <i>blacklisted</i>	X	X	
Invoc. canal ext.	X	X	
Erro não tratado	X	X	X
Variável global	X		X
Overflow conversão	X		X
Parâmetro não checado	X		
Privacidade - cond.	X		
Privacidade - ret.	X		
Phantom read	X		
Privacidade - arg	X		
Overflow matemático	X		

Tabela 3 – Vulnerabilidades cobertas pelas ferramentas

6.2.4 Metodologia de avaliação

Os testes foram realizados em um computador pessoal executando o sistema operacional Ubuntu 24.04, equipado com um processador Intel i7-1255U e 16 GB de memória RAM. Todas as ferramentas foram executadas localmente, sem o uso de ambientes em nuvem ou virtualização adicional. As ferramentas PDChecker e HFCCT foram obtidas diretamente de seus repositórios oficiais no GitHub.

O PDChecker foi compilado localmente utilizando os comandos `make install` e `make build`. Suas regras de execução são definidas previamente em um arquivo no formato TOML, que deve ser informado como parâmetro ao binário no momento da execução.

O HFCCT, desenvolvido em Python, teve suas dependências instaladas em um ambiente virtual (venv), e foi executado a partir do arquivo `hfcct.py`, ao qual foi acrescentado o caminho das chaincodes a serem analisadas. Tanto o PDChecker quanto o HFCCT não possuem busca recursiva de arquivos, sendo necessária a execução manual em cada arquivo de um módulo.

O `golangci-lint` foi instalado como dependência do Go por meio do comando `go get` e executado via linha de comando com `go run` sobre os módulos das chaincodes.

A metodologia de avaliação foi dividida em duas etapas. Na primeira, desenvolveu-se uma chaincode propositalmente vulnerável, contendo falhas como leitura após escrita, vazamento de dados privados, uso de variável global, entre outras. Essa chaincode foi utilizada para verificar a capacidade das ferramentas em identificar vulnerabilidades previamente conhecidas.

Na segunda etapa, as três ferramentas foram aplicadas a um conjunto de oito chaincodes pertencentes a um projeto real baseado em Hyperledger Fabric. Essas chaincodes apresentam estrutura modular, com arquivos distribuídos entre os diretórios `core/` e `fabric/`: enquanto os arquivos de `core/` realizam apenas verificações básicas de parâmetros, os arquivos em `fabric/` concentram a maior parte da lógica de negócio e das funções principais.

Dessa forma, para maximizar a relevância da análise, o PDChecker e o HFCCT foram executados exclusivamente nos arquivos localizados em `fabric/`, onde há maior probabilidade de ocorrência de vulnerabilidades. O `golangci-lint`, por sua vez, foi executado sobre todos os arquivos das chaincodes, dada sua capacidade de análise global dos módulos em Go.

6.2.4.1 Chaincode experimental

Para avaliar a efetividade das ferramentas, estas foram inicialmente executadas em uma chaincode com vulnerabilidades inseridas propositalmente no código. Nessa chaincode foram incluídas todas as vulnerabilidades que tanto o PDChecker quanto o HFCCT se propõem a detectar.

A chaincode é composta por dois arquivos, `main.go`, responsável por instanciar o contrato inteligente e `example_contract.go` onde está definida a lógica do contrato. Esta chaincode faz uso de duas bibliotecas do Hyperledger Fabric: `Fabric-contract-api-go`⁷, que oferece uma interface de alto nível para desenvolvimento de contratos em Fabric, e `Fabric-chaincode-go`⁸, que fornece a camada de baixo nível (shim) responsável pela comunicação com o peer e pelo acesso às variáveis de estado e contexto das transações.

Os contratos de Fabric são construídos a partir de uma *struct* que os define. Neste caso, a *struct* se chama `ExampleContract`. O contrato gerencia dois assets de exemplo, também representados por duas *structs*: `Asset` e `Owner`, sendo o `Owner`, proprietário de um `Asset`.

```
type ExampleContract struct {
    contractapi.Contract
```

⁷ <https://github.com/hyperledger/fabric-contract-api-go>

⁸ <https://github.com/hyperledger/fabric-chaincode-go>

```

stub shim.ChaincodeStubInterface

ownerCounter int // V: Field Declaration
}

type Owner struct {
    ID    string `json:"id"`
    Name  string `json:"name"`
    Age   uint64 `json:"age"`
}

type Asset struct {
    AssetType    string `json:"assetType"`
    ID           string `json:"id"`
    Description   string `json:"description"`
    Amount       int32  `json:"amount"`
    Owner        *Owner `json:"owner"`
    CreationTime string `json:"creationTime"`
}

```

Listing 6.1 – Structs da chaincode

As operações do contrato foram projetadas tanto para simular comportamentos comuns de chaincodes em produção quanto para forçar a introdução de vulnerabilidades, com duplo propósito: criar exemplos didáticos e viabilizar a detecção das falhas pelas ferramentas. Implementaram-se funções para criar, atualizar e consultar os assets, como `CreateAsset`, além de funções puramente didáticas, como `UnhandledError`. Justamente por esse fim didático, a chaincode pode apresentar erros no processo de deploy e execução.

```

func (sc *ExampleContract) CreateAsset(
    assetID string,
    description string,
    ownerID string) error {
    assetKey, err := sc.stub.CreateCompositeKey("asset", []string{assetID})
    if err != nil {
        return errors.New("unable to create composite key")
    }

    // V: Unhandled error
    existing, err := sc.stub.GetState(assetKey)

    if existing != nil {
        return fmt.Errorf("cannot create world state pair with key %s. Already
↪ exists", assetID)
    }

    // Read asset
    ownerBytes, err := sc.stub.GetState(ownerID)

```

```

if err != nil {
    return errors.New("unable to interact with world state")
}

var owner Owner
err = json.Unmarshal(ownerBytes, &owner)
if err != nil {
    return errors.New("unable to unmarshal")
}

var asset Asset
asset.Description = description
asset.ID = assetID
asset.Amount = 1
// V: Pointer.
asset.Owner = &owner
// V: Timestamp, Blacklisted import.
asset.CreationTime = time.Now().Format("Jan _2 15:04:05.000000")

// Serialize
assetBytes, err := json.Marshal(asset)
if err != nil {
    return errors.New("unable to marshal asset")
}

// Store asset
err = sc.stub.PutState(assetKey, []byte(assetBytes))
if err != nil {
    return errors.New("unable to interact with world state")
}

return nil
}

```

Listing 6.2 – Exemplo de função de ExampleContract com maior similaridade a uma implementação real

Conforme exibido em Listing 6.2 e Listing 6.3, foram inseridos comentários ao longo do código para identificar os pontos onde as vulnerabilidades foram introduzidas. Esses comentários começam com a letra “V:” seguida do nome da vulnerabilidade.

```

func (sc *ExampleContract) UnhandledError(assetID string) {
    // V: Unhandled Error
    sc.Read(assetID)
}

```

Listing 6.3 – Exemplo de função com fim mais didático para detecção de vulnerabilidade

6.3 RESULTADOS

6.3.1 Chaincode experimental

Em relação à chaincode experimental, conforme apresentado na tabela 4, o PDChecker foi capaz de detectar todas as vulnerabilidades às quais se propõe. Para a maioria delas, identificou apenas um caso, e, nos casos de utilização de ponteiros e de parâmetros não checados, apontou um número significativamente maior de ocorrências no código. Esse comportamento era esperado, visto que, de modo geral, apenas um elemento de cada tipo de vulnerabilidade havia sido inserido. Com relação aos erros não tratados: foram inseridos três casos distintos, com diferentes sintaxes, e o PDChecker foi capaz de detectar apenas um deles. No caso da declaração de campo em struct, essa é uma vulnerabilidade que a ferramenta não se propõe a identificar.

Vulnerabilidade	Qde.	PDChecker	HFCCT	Golangci-lint
Declaração Campo	1	-	0	-
Range map	1	1	1	-
Ponteiro	2	13(3)	2	-
Read after write	1	1	1	-
Concorrência	1	1	1	-
Importação <i>Blacklisted</i>	1	1	0	-
Invoc. canal ext.	1	1	0	-
Erro não tratado	3	1	4(1)	3
Variável global	1	1	-	1
Overflow conversão	2	2	-	2
Parâmetro não checado	20	21	-	-
Privacidade - cond.	1	1	-	-
Privacidade - ret.	1	1	-	-
Phantom read	1	1	-	-
Privacidade - arg	1	1	-	-
Overflow matemático	1	1	-	-

Tabela 4 – Vulnerabilidades inseridas (Qde.) e os resultados apontados por cada ferramenta. Os valores entre parênteses indicam a quantidade de *verdadeiros positivos*.

Os erros não tratados merecem atenção especial no contexto da linguagem Go, uma vez que esta não implementa o mecanismo de exceções. Nesse paradigma, os erros são retornados pelas funções e devem ser verificados explicitamente a cada chamada que possa gerar falhas. Observou-se que o PDChecker não identificou retornos de erro atribuídos à variável de descarte “_”, nem aqueles atribuídos a variáveis como “err” que não foram verificadas posteriormente. O único caso detectado foi o de uma chamada cuja função não teve o valor de retorno atribuído a nenhuma variável.

Dessa forma, apenas duas das 38 vulnerabilidades introduzidas não foram detectadas pela ferramenta, o que representa uma taxa de detecção de aproximadamente 95%. Considerando, contudo, a ocorrência de falsos positivos, especialmente relacionados ao uso de ponteiros e à ausência de verificação de parâmetros, observa-se que, dentre os 47 alertas emitidos, 77% correspondem a verdadeiros positivos (considerando um ponteiro adicional, classificado como verdadeiro) e 23% a falsos positivos.

No caso dos ponteiros, a ferramenta apontou 13 ocorrências, das quais apenas três configuram vulnerabilidades reais. Já em relação aos parâmetros não checados, trata-se de uma vulnerabilidade dependente da lógica de cada função, pois nem todos os parâmetros necessitam de validação. Ainda assim, considerando esse critério, 20 dos 21 alertas foram classificados corretamente como válidos. O único falso positivo correspondeu a um parâmetro pertencente à biblioteca *Contract-api*, o qual, a priori, não exige validações adicionais.

No que diz respeito ao HFCCT, a ferramenta mostrou-se precisa na detecção de vulnerabilidades do tipo *range map*, ponteiro, *read after write* e concorrência. Entretanto, não foi capaz de identificar a presença de declaração de campo, importação *blacklisted* e invocação de canal externo, vulnerabilidades que, em teoria, estão entre as que deveria detectar. No caso dos erros não tratados, apesar de a ferramenta ter indicado quatro ocorrências, apenas uma correspondia a um caso real, relacionada a um retorno de função ignorado (“_”).

Assim, das onze vulnerabilidades que o HFCCT se propõe a detectar, seis foram efetivamente identificadas, resultando em uma acurácia aproximada de 55%. Considerando as nove ocorrências detectadas no total, 67% foram classificadas como verdadeiros positivos e 33% como falsos positivos.

Por fim, no caso do Golangci-lint, o experimento teve como objetivo verificar quais das vulnerabilidades conhecidas em chaincodes poderiam ser apontadas pela ferramenta. Ao executá-la sobre a chaincode experimental, verificou-se que as vulnerabilidades de erro não tratado, variável global e overflow de conversão foram corretamente identificadas. Em todos esses casos, o número de ocorrências detectadas coincidiu com o número de inserções planejadas, resultando em 100% de acurácia e de verdadeiros positivos.

De modo geral, o PDChecker apresentou o melhor desempenho entre as ferramentas avaliadas, atingindo uma taxa de detecção de aproximadamente 95% e uma precisão de 77% em relação aos verdadeiros positivos. O HFCCT obteve resultados intermediários, com acurácia de cerca de 55% e precisão de 67%, demonstrando bom desempenho em determinados tipos de vulnerabilidade, mas com limitações na detecção de outras às quais se propõe. Já o Golangci-lint apresentou 100% de acurácia e precisão nos tipos de vulnerabilidade que é capaz de analisar, embora sua cobertura seja mais restrita.

6.3.2 Chaincodes RRDES

O projeto RRDES é composto, atualmente, por oito contratos inteligentes responsáveis pela interação com o ledger distribuído. Esses contratos estão, em sua maioria, relacionados à

Ferramenta	Taxa de detecção (%)	Precisão (%)	Cobertura das vulnerabilidades
PDChecker	95	77	Alta
HFCCT	55	67	Média
Golangci-lint	100	100	Baixa

Tabela 5 – Resumo comparativo entre as ferramentas de análise.

gestão e ao armazenamento de dados referentes a documentos do ensino superior brasileiro, possibilitando a rastreabilidade das informações acadêmicas, objetivo principal do projeto.

Os contratos e suas respectivas finalidades são descritos a seguir:

- **Histórico:** responsável pela gestão de históricos escolares.
- **Diploma:** voltado ao armazenamento e validação de diplomas de graduação.
- **Currículo:** utilizado para o gerenciamento de currículos de cursos de graduação.
- **Lato:** destinado ao controle de certificados de cursos de pós-graduação lato sensu.
- **Conselhos:** voltado à integração com conselhos profissionais, como o Conselho Federal de Medicina (CFM).
- **Dados Regulatórios:** responsável pelo registro de informações regulatórias das instituições de ensino superior e de seus cursos de graduação.
- **Lattes:** destinado ao armazenamento de informações provenientes de currículos Lattes.
- **XMLLog:** responsável pelo armazenamento de documentos no formato XML.

Vulnerabilidade	Histórico		Diploma		Currículo		Lato		Conselhos		Dados Reg.		Lattes		XMLLog		Total	
	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.
Parâmetro não checado	39	19	39	14	-	-	29	14	16	10	14	8	8	7	20	15	165	87
Ponteiro	7	0	7	0	32	0	10	0	2	0	4	0	2	0	6	0	70	0
Invoc. canal ext.	3	0	2	0	2	0	1	0	1	0	-	-	-	-	-	-	9	0
Importação <i>blacklisted</i>	-	-	2	0	-	-	1	0	-	-	-	-	-	-	-	-	3	0
Privacidade - cond.	1	0	-	-	-	-	1	0	-	-	-	-	-	-	-	-	2	0
Privacidade - ret.	-	-	-	-	-	-	1	1	-	-	-	-	-	-	-	-	1	1
Total	50	19	50	14	34	0	43	15	19	10	18	8	10	7	26	15	250	88

Tabela 6 – PDChecker: vulnerabilidades detectadas (Det.) e confirmadas (Ver.) em cada chaincode.

A Tabela 6 apresenta os resultados da execução do PDChecker sobre os arquivos das chaincodes do projeto RRDES. Das quinze vulnerabilidades que a ferramenta se propõe a detectar, apenas seis foram encontradas no conjunto analisado, sendo a maioria relacionada, assim como observado na chaincode experimental, ao uso de ponteiros e à ausência de verificação de parâmetros de entrada.

Os parâmetros não checados representam o principal problema identificado nas chaincodes. Apesar do número elevado de falsos positivos, a quantidade de casos confirmados é

significativa e, em sua maioria, está associada a funções de leitura. Em contrapartida, todos os alertas referentes ao uso de ponteiros foram classificados como falsos positivos. O mesmo se aplica às demais vulnerabilidades, que, embora tenham aparecido em menor quantidade, também se mostraram incorretas. A única exceção, além dos parâmetros não checados, foi um caso de vazamento de privacidade em retorno de função identificado na chaincode Lato, um achado pontual, porém relevante do ponto de vista de segurança.

No que se refere aos ponteiros, observou-se que, embora as chaincodes façam uso intenso desse recurso, os endereços não são empregados em operações potencialmente problemáticas, como o armazenamento de endereços no ledger, que poderiam resultar em comportamento não determinístico. Em relação à invocação de chaincodes em outros canais, verificou-se que as chamadas existentes ocorrem sempre dentro de um mesmo canal, não caracterizando vulnerabilidade.

Quanto à importação de bibliotecas *blacklisted*, o PDChecker indicou o uso da biblioteca nativa *time*. De fato, certas funções dessa biblioteca podem comprometer a determinicidade das transações, dependendo da forma como são utilizadas - como o uso de *timestamps*. Entretanto, nas chaincodes analisadas, a biblioteca é empregada apenas para formatação de datas de documentos, de modo determinístico, não representando risco. No caso dos vazamentos de dados em operações condicionais, todos os alertas envolveram apenas informações públicas, sem exposição de dados privados.

De modo geral, observa-se que, em comparação com a análise realizada sobre a chaincode experimental, o número de falsos positivos é consideravelmente maior. Dos 250 alertas emitidos pelo PDChecker, apenas 88 foram confirmados, resultando em uma taxa de verdadeiros positivos de aproximadamente 35% e de falsos positivos de 65%. Além disso, das seis vulnerabilidades detectadas pela ferramenta, apenas duas apresentaram resultados verificados como verdadeiros, enquanto as demais corresponderam a alertas incorretos.

Vulnerabilidade	Histórico		Diploma		Currículo		Lato		Conselhos		Dados Reg.		Lattes		XMLLog		Total	
	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.
Erro não tratado	7	1	6	0	17	0	2	0	5	1	0	0	3	1	2	0	42	3
Range risk	6	0	6	0	28	0	6	0	2	0	4	0	2	0	3	0	57	0
Ponteiro	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0	1	0
Total	13	1	12	0	45	0	8	0	7	1	4	0	5	1	6	0	100	3

Tabela 7 – HFCCT: vulnerabilidades detectadas (Det.) e confirmadas (Ver.) em cada chaincode.

A Tabela 7 apresenta os resultados obtidos com a execução do HFCCT sobre as chaincodes do projeto RRDES. Das oito vulnerabilidades que a ferramenta é capaz de detectar, apenas três foram identificadas no conjunto analisado: erro não tratado, *range risk* e ponteiro.

Entre os alertas gerados, apenas os casos classificados como erro não tratado se confirmaram como verdadeiros, totalizando três ocorrências, uma em cada uma das chaincodes Lato, Histórico e Conselhos. Todos os demais apontamentos correspondem a falsos positivos. Observou-se que muitos dos alertas de erro não tratado estão relacionados ao uso do caractere

“_” no código, o qual, embora possa indicar erros ignorados, também possui outras finalidades legítimas em Go, o que explica parte das detecções incorretas.

No caso do *range risk*, o HFCCT reportou diversas funções do Hyperledger Fabric que retornam iteradores. Entretanto, nenhuma delas é manipulada de forma não determinística, como ocorreria em iterações sobre mapas com *range map*, ou de maneira que possibilitasse *phantom reads*, como ao utilizar range queries do CouchDB. Assim, todos esses alertas foram considerados falsos positivos.

Quanto ao uso de ponteiros, embora o HFCCT tenha se mostrado bastante eficaz na detecção desse tipo de vulnerabilidade na chaincode experimental, o caso reportado na chaincode XMLLog refere-se apenas à declaração de uma variável, não apresentando qualquer risco à execução da rede.

De forma geral, a eficácia da ferramenta mostrou-se limitada: dos 100 alertas emitidos, apenas 3 foram confirmados, resultando em uma taxa de verdadeiros positivos de aproximadamente 3% e de falsos positivos de 97%.

Vulnerabilidade	Histórico		Diploma		Currículo		Lato		Conselhos		Dados Reg.		Lattes		XMLLog		Total	
	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.
Erro não tratado	3	3	3	3	3	3	5	5	2	2	3	3	2	2	3	3	24	24
Erro - atribuição inefetiva	-	-	-	-	-	-	-	-	-	-	-	-	1	1	-	-	1	1
Total	3	3	3	3	3	3	5	5	2	2	3	3	3	3	3	3	25	25

Tabela 8 – Golangci-lint: vulnerabilidades detectadas (Det.) e confirmadas (Ver.) em cada chaincode.

A Tabela 8 apresenta os resultados da execução do Golangci-lint sobre as chaincodes do projeto RRDES. A ferramenta identificou dois tipos de problemas, ambos relacionados à vulnerabilidade de erro não tratado. O caso denominado atribuição inefetiva, nestas chaincodes, está relacionado à variável *err*, a qual recebe o valor de um erro, mas nenhuma ação é tomada em relação a ela. Esse comportamento pode mascarar falhas de execução e dificultar o rastreamento de problemas.

Os 24 alertas de erro não tratado reportados pela ferramenta, ocorrem durante o fechamento de iteradores por meio da palavra-chave *defer*, como em instruções do tipo "defer iterator.Close()". Embora esse padrão não represente um risco crítico, a função Close() retorna um erro que não está sendo verificado, o que caracteriza os alertas apontados como verdadeiros.

Apesar do Golangci-lint possuir um escopo de análise mais restrito em relação a vulnerabilidades específicas de chaincodes, a ferramenta demonstrou elevada precisão, uma vez que todos os alertas gerados foram confirmados como verdadeiros positivos. Dessa forma, destaca-se sua utilidade como ferramenta complementar para detecção de más práticas de programação e falhas de tratamento de erros em contratos inteligentes desenvolvidos em Go.

Observa-se que as três ferramentas apresentam comportamentos distintos quanto à cobertura e precisão. Enquanto o PDChecker abrange um conjunto maior de vulnerabilidades, apresentou alta taxa de falsos positivos. O HFCCT, embora mais restrito, mostrou baixa eficácia nas chaincodes reais. Já o Golangci-lint, com escopo limitado a más práticas de código,

obteve o melhor desempenho em precisão, sem falsos positivos.

6.4 LIMITAÇÕES DAS FERRAMENTAS ANALISADAS E POSSÍVEIS MELHORIAS

A partir dos resultados obtidos e de uma análise aprofundada das ferramentas avaliadas, observa-se que ainda há espaço significativo para melhorias visando aumentar a eficácia e a precisão dos analisadores. Para que essas ferramentas cumpram seu papel no apoio ao desenvolvimento seguro de contratos inteligentes, é essencial que consigam auxiliar o desenvolvedor a identificar problemas de maneira antecipada, reduzindo esforço manual e a análise de falsos positivos.

No contexto do PDChecker e das vulnerabilidades que ele busca detectar, destaca-se a regra referente ao uso de ponteiros em chaincodes. A implementação atual dessa verificação se mostra limitada, pois qualquer ocorrência de ponteiros, seja por meio da declaração de variáveis do tipo ponteiro ou pelo uso do operador '&' para obtenção de endereços, gera um alerta. De forma similar, o HFCCT também pode ser aprimorado nesse aspecto, pois sua verificação se restringe apenas a identificar as operações "`= new()`", "`= *p`" e "`=&p`", sem correlacioná-las ao uso efetivo dos dados armazenados no ledger.

Essas abordagens excessivamente genéricas podem levar a uma experiência negativa para o desenvolvedor, que precisa verificar manualmente cada ocorrência para determinar se de fato existe uma vulnerabilidade. Em muitos casos, o uso de ponteiros em Go é legítimo e necessário, especialmente para obtenção de melhor performande nos contratos.

Essa regra deveria ser mais precisa, sinalizando apenas os usos que possam introduzir não determinismo e comprometer o processo de endosso das transações. Em particular, o risco real decorre da utilização de endereços de memória como parte de dados persistidos no ledger, como em atribuições do tipo `asset.owner = &owner` seguidas de operações de persistência, conforme 6.4

```
func (sc *ExampleContract) UpdateAssetOwnership() error {
    {...}

    asset.Owner = &owner // Assigning address.

    assetBytes, err = json.Marshal(asset)
    if err != nil {
        return errors.New("unable to marshal asset")
    }

    // Storing in the ledger
    err = sc.stub.PutState(assetKey, assetBytes)
    if err != nil {
        return errors.New("unable to store data")
    }
}
```

```

    return nil
}

```

Listing 6.4 – Uso incorreto de ponteiros

Nesses cenários, o endereço de memória pode variar entre peers, levando a divergência no estado e falhas de consenso. Portanto, uma melhoria relevante seria a capacidade da ferramenta de correlacionar a detecção de ponteiros com chamadas a métodos de gravação no ledger como PutState, aumentando a precisão e reduzindo falsos positivos.

Outro ponto relevante diz respeito à regra que identifica erros não tratados. A implementação atual do PDChecker limita-se a verificar se o valor de retorno do tipo error foi completamente ignorado, sinalizando apenas chamadas de funções cujo erro não é capturado. Dessa forma, atribuições como `err := f()` ou `_ = f()` não geram alerta, ainda que o erro não seja posteriormente analisado ou tratado.

```

func f() error {
    return error
}

func ErrorHandling() {
    err := f() // Not detected

    _ = f() // Not detected

    f() // Detected
}

```

Listing 6.5 – Detecção de erros não tratados - PDChecker

Embora essa lógica contribua para evitar situações em que o erro é simplesmente descartado, ela ainda é restrita, pois não abrange cenários em que o desenvolvedor armazena o erro, mas não realiza qualquer verificação ou ação subsequente. O HFCCT é também bastante limitado, na medida em que apenas aponta a existência de `'_'`, no código, sem diferenciá-los de outros usos que não envolvam o tratamento de erros.

Uma melhoria potencial para essa regra consistiria em estender a capacidade de análise, de modo a identificar casos nos quais o erro é atribuído, porém não utilizado em estruturas condicionais ou fluxos de tratamento, tais como `if err != nil {...}`. Essa evolução permitiria à ferramenta diferenciar entre a captura formal do erro e seu tratamento efetivo, reduzindo falsos negativos e aumentando a precisão da detecção.

Quanto à detecção de bibliotecas *blacklisted*, o PDChecker foi capaz de identificar o uso de uma dependência potencialmente problemática no código analisado, especificamente a biblioteca `time`. Contudo, essa capacidade ainda se mostra bastante limitada. A ferramenta mantém apenas uma pequena lista de bibliotecas que podem gerar alertas e, além disso, não realiza uma análise mais profunda sobre quais funções dessas dependências são efetivamente

utilizadas.

No caso da chaincode didática desenvolvida neste trabalho, na qual as vulnerabilidades são deliberadamente mais explícitas, a sinalização referente à importação de `time` foi pertinente, uma vez que seu uso estava associado a comportamentos não determinísticos. Entretanto, nas chaincodes reais do RRDES, essa biblioteca é utilizada exclusivamente para formatação de datas, operações determinísticas e que, portanto, não representam ameaça ao processo de endosso.

Sendo assim, uma ferramenta de análise mais robusta deveria ser capaz não apenas de identificar dependências potencialmente perigosas, mas também de avaliar o uso específico das funções importadas e rastrear o fluxo de dados resultante, verificando se há impacto sobre o determinismo, o armazenamento no ledger ou condições lógicas sensíveis dentro da execução da transação.

De forma semelhante, o HFCCT não identificou nenhuma importação considerada sensível, possivelmente porque sua lista de bibliotecas monitoradas é ainda mais restrita do que a do PDChecker, o que reforça a necessidade de aprimoramento na cobertura e profundidade dessa verificação.

Quanto a utilização de declaração de campos na struct da chaincode, essa é uma característica que o HFCCT se propõe a detectar e que é importante, considerando que possui os mesmos efeitos de utilização que o uso de variáveis globais. Contudo, na chaincode de exemplo, com essa vulnerabilidade exposta, a ferramenta não foi capaz de apontar o problema.

A implementação dessa regra no HFCCT assume um padrão específico de desenvolvimento baseado na chamada `shim.Start(new(<StructName>))`, a partir da qual extrai o nome da struct da chaincode e busca localizar sua definição no código. No entanto, as chaincodes utilizadas neste trabalho foram desenvolvidas com a biblioteca `contractapi`, que segue um padrão distinto de inicialização, conforme 6.6.

```
example := &chaincode.ExampleContract{}
cc, err := contractapi.NewChaincode(example)
if err := cc.Start(); err != nil {
    panic(err.Error())
}
```

Listing 6.6 – Inicialização da chaincode de exemplo

Nesse modelo, não há invocação de `shim.Start(new(...))`, o que impede o HFCCT de identificar corretamente o nome da estrutura e, consequentemente, de aplicar a regra. Como resultado, declarações de campos na struct, como em 6.1 não são detectadas pela ferramenta, apesar de representarem exatamente o cenário que a regra busca identificar.

Uma melhoria recomendada seria a adaptação da lógica de detecção para contemplar chaincodes que utilizam o padrão `contractapi`, permitindo a identificação do nome da struct a partir da passagem de referência ao construtor `contractapi.NewChaincode(<struct>)`. Essa modificação tornaria a regra mais abrangente e compatível com implementações modernas

de chaincode no Hyperledger Fabric, reduzindo o risco de falsos negativos e aumentando a utilidade da ferramenta. Além disso, esta mesma regra poderia ser adicionada ao PDChecker de modo a aumentar a cobertura de vulnerabilidades abordadas pela ferramenta.

Quanto à vulnerabilidade de *read after write*, embora ambas as ferramentas consigam identificá-la na chaincode de exemplo, elas apresentam limitações relevantes. Em particular, não detectam o fluxo interprocedural, ou seja, quando a operação de escrita e leitura ocorre por meio de chamadas a funções auxiliares, como ilustrado no trecho 6.7. Além disso, não contemplam todos os métodos de leitura disponíveis no Fabric.

```
// Read after write -> Not detected
func ReadAfterWrite(assetKey string) {
    StoreAsset(assetKey) // Calls PutState(assetKey, data)
    return ReadAsset(assetKey) // Calls GetState(assetKey)
}
```

Listing 6.7 – Read after write

O HFCCT detecta apenas chamadas diretas às funções de leitura e escrita `PutState` e `GetState`. O PDChecker, por sua vez, possui cobertura parcialmente superior ao incluir também `PutPrivateData` e `GetPrivateData`, que realizam operações sobre dados privados. Entretanto, ainda se observa uma lacuna na identificação de outras formas de leitura, como iterações com `GetStateByPartialCompositeKey`, entre outros métodos de consulta do Fabric.

Dessa forma, avanços são possíveis ao incorporar a análise de um conjunto mais amplo de funções de leitura e escrita, bem como ao considerar o fluxo de dados entre funções. Essa limitação também aparece em outras regras, como as relacionadas ao vazamento de privacidade, que igualmente não reconhecem fluxos interprocedurais.

No que se refere aos vazamentos de informações privadas, destaca-se uma limitação relevante na regra destinada a identificar a presença de dados sensíveis em parâmetros de funções, o que pode aumentar o risco de falsos negativos. Essa regra verifica se um parâmetro é utilizado em alguma chamada a `PutPrivateData`. No entanto, se no fluxo de execução, entre a declaração do parâmetro e a chamada a `PutPrivateData`, ocorrer uma chamada a `GetTransient`, mecanismo previsto pelo Fabric para transferência segura de dados privados, o PDChecker não sinaliza o possível vazamento proveniente dos argumentos.

```
// Private data in parameter -> Not detected
func StorePrivateData(assetKey string, privateData string) {

    // Retrieving another private data from transient source
    anotherPrivateData := GetTransient()

    // Storing privateData
    PutPrivateData("collection", assetKey, privateData) // Not detected
}
```

```
// Further processing with anotherPrivateData
}
```

Listing 6.8 – Dados privados em parâmetro de função

Essa limitação é significativa, pois um contrato pode utilizar corretamente `GetTransient` para receber dados privados, mas ainda assim introduzir informações sensíveis por meio de parâmetros de função, seja por desconhecimento do mecanismo ou por descuido no desenvolvimento. Portanto, é desejável que a regra seja aprimorada de forma a considerar esse cenário, distinguindo adequadamente quando um parâmetro representa uma potencial fonte de vazamento de dados privados.

No que se refere aos *phantom reads*, o `PDChecker` consegue identificar o uso de funções de consulta potencialmente suscetíveis a esse tipo de vulnerabilidade, como `GetQueryResult`, `GetHistoryForKey` e `GetPrivateDataQueryResult`. Contudo, a ferramenta não realiza a correlação entre essas operações de leitura e subsequentes operações de escrita que utilizem os dados retornados. Essa limitação pode resultar em falsos positivos, uma vez que a mera utilização dessas funções não implica, necessariamente, em um cenário de *phantom read*. Uma evolução desejável seria justamente a análise de fluxo entre leitura e escrita para confirmar a presença efetiva da vulnerabilidade no contrato inteligente.

O `HFCCT`, por sua vez, também se limita a detectar a presença dessas funções de leitura. Entretanto, a ferramenta amplia sua lista equivocadamente, incluindo chamadas como `GetStateByRange` e `GetStateByPartialCompositeKey`, as quais são executadas durante a fase de validação. Essa abordagem, além de igualmente suscetível à geração de falsos positivos, revela um equívoco conceitual sobre a arquitetura do `Fabric`, visto que é justamente a ausência de execução durante a fase de validação que implica na possibilidade de *phantom reads*.

Uma possível vulnerabilidade adicional que poderia ser contemplada pelo `PDChecker` é a detecção do uso de funções relacionadas à geração de valores aleatórios, como `rand.Int()`, passíveis de causar comportamento não determinístico e falha nas transações. Como aprimoramento, a ferramenta também poderia analisar se o valor aleatório influencia o estado da `blockchain` ou decisões de fluxo lógico, distinguindo casos benignos (sem persistência) de situações que efetivamente comprometem o determinismo da execução.

Em síntese, embora as ferramentas analisadas forneçam contribuições relevantes para a detecção de vulnerabilidades em `chaincodes`, ainda apresentam limitações importantes que podem comprometer a eficácia da análise. A incorporação de verificações mais granulares, sensibilidade ao fluxo de dados, suporte a chamadas interprocedurais e cobertura ampliada de funções do `Fabric` poderia aumentar a precisão e a utilidade prática das ferramentas. Nesse sentido, há um amplo espaço para evolução dessas abordagens.

7 CONCLUSÃO

A segurança em blockchains permissionadas abrange diversos aspectos, que envolvem diferentes níveis de abstração dentro da própria tecnologia, assim como aspectos variados a serem considerados durante sua análise. O foco pode estar na comunicação entre os nós, na determinação do consenso ou no armazenamento de chaves privadas, além de incluir medidas preventivas que vão desde a governança da rede até ferramentas de detecção executadas no pipeline de desenvolvimento.

A partir do desenvolvimento deste trabalho, verificou-se que, entre as diversas abordagens para tratar vulnerabilidades em blockchains permissionadas, um ponto fundamental é a segurança dos contratos inteligentes. Esses contratos, como mecanismos que aplicam a lógica de negócio sobre a blockchain, exigem um desenvolvimento seguro para garantir a proteção dos dados e a correta execução das transações do sistema.

Nesse contexto, diferentes estudos apontam vulnerabilidades presentes em contratos inteligentes e reforçam a necessidade de detectá-las previamente, minimizando a ocorrência de problemas futuros. Além disso, apresentam variadas ferramentas destinadas a essa detecção.

Considerando as vulnerabilidades presentes no código-fonte dos contratos, foram realizados experimentos para analisar as principais ferramentas disponíveis para detecção de vulnerabilidades em contratos do Hyperledger Fabric, avaliando sua eficácia e limitações para possível aplicação em projetos reais.

Após a análise dos resultados, constatou-se que as ferramentas analisadas representam um ponto de partida importante para os objetivos a que se propõem, oferecendo regras de análise de código capazes de detectar vulnerabilidades de forma antecipada.

Essas vulnerabilidades incluem tanto problemas comuns a todo o ecossistema de software, como overflow, parâmetros não checados e erros não tratados, quanto questões específicas da blockchain e do Fabric, como não determinismo e ausência de leituras durante a validação de transações. Dentre essas vulnerabilidades, destaca-se a detecção de vazamento de dados privados, ponto crucial na construção de contratos inteligentes.

Nesse sentido, o PDChecker revelou-se uma ferramenta promissora. É capaz de cobrir um conjunto relevante de vulnerabilidades, possui código aberto, é a ferramenta mais atual dentre as disponíveis e apresenta facilidade de integração a serviços de integração contínua.

Entretanto, algumas ressalvas são necessárias. Apesar das iniciativas importantes, as ferramentas apresentadas na literatura e analisadas nos experimentos ainda são protótipos. Sua capacidade de análise é limitada, com regras simples, o que gera quantidades significativas de falsos positivos e falsos negativos, dificultando sua adoção sem precauções.

No caso específico do HFCCT, um dos módulos da ferramenta não pôde ser executado, o que acabou limitando a análise. Contudo, tratando-se da análise estática observada, verifica-se que é uma ferramenta que não possui a capacidade de ser integrada ao fluxo de trabalho de desenvolvimento de projetos reais. Suas regras são descritas de maneira muito limitada, sua capacidade de detecção é baixa e possui pouca precisão nos alertas gerados.

Quanto ao Golangci-lint, trata-se de uma ferramenta consolidada no ecossistema Go, com contribuições ativas e atualizações constantes. Por isso, apresenta maior precisão nas vulnerabilidades que é capaz de detectar, embora, para questões específicas de blockchain, seus alertas sejam mais limitados. Ainda assim, é essencial para o desenvolvimento geral de projetos em Go.

Em suma, é de extrema importância que ferramentas de análise de código sejam utilizadas na construção de contratos inteligentes e detectem eventuais problemas de maneira antecipada. O PDChecker mostrou potencial para uso prático, embora seja importante considerar suas limitações. Este campo permanece aberto, com muitas oportunidades para futuras pesquisas e desenvolvimento, tanto na identificação de novas vulnerabilidades quanto na criação de ferramentas mais robustas e precisas para a detecção dessas falhas.

REFERÊNCIAS

- ANTONPOULOS, A. M.; WOOD, G. **Mastering Ethereum: Building Smart Contracts and DApps**. O'Reilly Media, 2018. Disponível em: <https://github.com/ethereumbook/ethereumbook>.
- BlockNews. **Linux lança grupo internacional que tem Banco Central do Brasil, Serpro e BNDES entre fundadores**. 2024. Acesso em: jun de 2025. Disponível em: <https://www.blocknews.com.br/financas-corporativo/linux-lanca-grupo-internacional-que-tem-banco-central-do-brasil-serpro-e-bndes-entre-fundadores/>.
- BROTSIS, S. et al. On the security and privacy of hyperledger fabric: Challenges and open issues. In: **2020 IEEE World Congress on Services (SERVICES)**. [S.l.: s.n.], 2020. p. 197–204.
- CHA, S.-C. et al. Trustworthiness evaluation for permissioned blockchain-enabled applications. **Computers, Materials and Continua**, v. 73, n. 2, p. 2731–2755, 2022. ISSN 1546-2218. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1546221822005598>.
- CHEN, Z. et al. Understanding and detecting privacy leakage vulnerabilities in hyperledger fabric chaincodes. In: **2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.: s.n.], 2024. p. 570–581.
- HAMDI, A.; FOURATI, L.; AYED, S. Vulnerabilities and attacks assessments in blockchain 1.0, 2.0 and 3.0: tools, analysis and countermeasures. **International Journal of Information Security**, v. 23, n. 2, p. 713–757, 2024. ISSN 1615-5270. Disponível em: <https://doi.org/10.1007/s10207-023-00765-0>.
- HOMOLIAK, I. et al. The security reference architecture for blockchains: Toward a standardized model for studying vulnerabilities, threats, and defenses. **IEEE Communications Surveys 'I&' Tutorials**, v. 23, n. 1, p. 341–390, 2021.
- Hyperledger Foundation. **Hyperledger Besu Documentation**. [S.l.], 2025. Disponível em: <https://besu.hyperledger.org/>.
- Hyperledger Foundation. **Hyperledger Fabric Documentation**. [S.l.], 2025. Disponível em: <https://hyperledger-fabric.readthedocs.io/>.
- JAYADEV, V.; MORADPOOR, N.; PETROVSKI, A. Assessing the performance of ethereum and hyperledger fabric under ddos attacks for cyber-physical systems. In: **Proceedings of the 19th International Conference on Availability, Reliability and Security**. New York, NY, USA: Association for Computing Machinery, 2024. (ARES '24). ISBN 9798400717185. Disponível em: <https://doi.org/10.1145/3664476.3670927>.
- KITCHENHAM, B. et al. Systematic literature reviews in software engineering – a systematic literature review. **Information and Software Technology**, Elsevier, v. 51, n. 1, p. 7–15, 2009.
- KOLB, J. et al. Core concepts, challenges, and future directions in blockchain: A centralized tutorial. Association for Computing Machinery, New York, NY, USA, v. 53, n. 1, fev. 2020. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3366370>.

La Salle, A. et al. Joint modeling of hyperledger fabric and sybil attack: Petri net approach. **Simulation Modelling Practice and Theory**, v. 122, p. 102674, 2023. ISSN 1569-190X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1569190X22001447>.

LI, P. et al. A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis. In: **Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2022. (EASE '22), p. 366–374. ISBN 9781450396134. Disponível em: <https://doi.org/10.1145/3530019.3531342>.

NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. 2008.

NASIR, N. M.; HASSAN, S.; ZAINI, K. M. Securing permissioned blockchain-based systems: An analysis on the significance of consensus mechanisms. **IEEE Access**, v. 12, p. 138211–138238, 2024.

NAÍSA, L.; CARVALHO, L. **Falha no sistema do Inep expõe dados de 5 milhões de estudantes**. 2021. Acesso em: junho de 2025. Disponível em: <https://www.uol.com.br/tilt/noticias/redacao/2021/09/10/falha-no-sistema-do-inep-vaza-dados-de-5-milhoes-de-estudantes.htm>.

PETRY, G. **Exclusivo: dados de quase 1,5 milhão de alunos são expostos por falha na Secretaria da Educação do DF**. 2021. Acesso em: junho de 2025. Disponível em: <https://thehack.com.br/exclusivo-dados-de-quase-1-5-milhao-de-alunos-sao-expostos-por-falha-na-secretaria-da-educacao-do-df/>.

VIEIRA, M.; VELASCO, G.; CARVALHO, S. A decentralized health data repository for remote patient monitoring using blockchain and fhir. In: **Anais do VI Workshop em Blockchain: Teoria, Tecnologias e Aplicações**. Porto Alegre, RS, Brasil: SBC, 2023. p. 85–98. ISSN 0000-0000. Disponível em: <https://sol.sbc.org.br/index.php/wblockchain/article/view/24623>.

Apêndices

APÊNDICE A – CÓDIGO FONTE

A seguir, é apresentado o código fonte da chaincode de exemplo, utilizada para avaliação de performance das ferramentas de detecção de vulnerabilidades.

```
package main

import (
    "example/chaincode"

    "github.com/hyperledger/fabric-contract-api-go/v2/contractapi"
)

func main() {
    example := &chaincode.ExampleContract{}
    example.BeforeTransaction = func(ctx contractapi.TransactionContextInterface) error {
        ↪ { return example.InitContract(ctx) }
    }
    example.Name = "example"

    cc, err := contractapi.NewChaincode(example)

    if err != nil {
        panic(err.Error())
    }

    if err := cc.Start(); err != nil {
        panic(err.Error())
    }
}
```

Listing A.1 – Arquivo main.go

```
package chaincode

import (
    "encoding/json"
    "errors"
    "fmt"
    "strconv"
    "sync"
    "time"

    "github.com/hyperledger/fabric-chaincode-go/v2/shim"
    "github.com/hyperledger/fabric-contract-api-go/v2/contractapi"
)

var totalCapacity uint64 // V: Global variable
```

```

type ExampleContract struct {
    contractapi.Contract
    stub shim.ChaincodeStubInterface

    ownerCounter int // V: Field Declaration
}

func (sc *ExampleContract) InitContract(ctx contractapi.TransactionContextInterface)
    ↪ error {
    sc.stub = ctx.GetStub()
    totalCapacity = 500

    return nil
}

type Owner struct {
    ID    string 'json:"id"'
    Name  string 'json:"name"'
    Age   uint64 'json:"age"'
}

type Asset struct {
    AssetType    string 'json:"assetType"'
    ID           string 'json:"id"'
    Description   string 'json:"description"'
    Amount       int32  'json:"amount"'
    Owner        *Owner 'json:"owner"'
    CreationTime string 'json:"creationTime"'
}

func (sc *ExampleContract) CreateAsset(assetID string, description string, ownerID
    ↪ string) error {
    assetKey, err := sc.stub.CreateCompositeKey("asset", []string{assetID})
    if err != nil {
        return errors.New("unable to create composite key")
    }

    // V: Unhandled error
    existing, err := sc.stub.GetState(assetKey)

    if existing != nil {
        return fmt.Errorf("cannot create world state pair with key %s. Already exists",
            ↪ assetID)
    }

    ownerBytes, err := sc.stub.GetState(ownerID)
    if err != nil {

```

```

    return errors.New("unable to interact with world state")
}

var owner Owner
err = json.Unmarshal(ownerBytes, &owner)
if err != nil {
    return errors.New("unable to unmarshal")
}

var asset Asset
asset.Description = description
asset.ID = assetID
asset.Amount = 1
asset.Owner = &owner // V: Pointer.
asset.CreationTime = time.Now().Format("Jan _2 15:04:05.000000") // V: Timestamp,
    ↪ Blacklisted import.

// Serialize
assetBytes, err := json.Marshal(asset)
if err != nil {
    return errors.New("unable to marshal asset")
}

// Store asset
err = sc.stub.PutState(assetKey, []byte(assetBytes))
if err != nil {
    return errors.New("unable to interact with world state")
}

return nil
}

// V: Privacy leakage from private data in arguments and in return statement
func (sc *ExampleContract) CreateOwner(name string, ageStr string, documentNumber
    ↪ string) (string, error) {
    age, err := strconv.ParseUint(ageStr, 10, 64)
    if err != nil {
        return "", errors.New("unable to parse string to uint")
    }

    var owner Owner
    owner.ID = documentNumber
    owner.Age = age
    owner.Name = name

    ownerBytes, err := json.Marshal(owner)
    if err != nil {
        return "", errors.New("unable to marshal asset")
    }

```

```

}

err = sc.stub.PutPrivateData("collectionID", owner.ID, ownerBytes)
if err != nil {
    return "", errors.New("unable to store private data")
}

jsonString := string(ownerBytes)
sc.ownerCounter++

// V: Privacy leakage when returning private data
return jsonString, nil
}

// V: Privacy leakage from branch condition
func (sc *ExampleContract) UpdateAssetOwnership(assetID string, ownerID string) error
    ⇨ {
    ownerBytes, err := sc.stub.GetPrivateData("collectionID", ownerID)
    if err != nil {
        return errors.New("unable to get private data")
    }
    if ownerBytes == nil {
        return fmt.Errorf("cannot update world state with key %s. Does not exist",
            ⇨ ownerID)
    }

    var owner Owner
    err = json.Unmarshal(ownerBytes, &owner)
    if err != nil {
        return errors.New("unable to unmarshal")
    }

    if owner.Age < 18 { // V: Privacy leakage: private data in branch statement
        return errors.New("owner must be at least 18 years old")
    }

    assetKey, err := sc.stub.CreateCompositeKey("asset", []string{assetID})
    if err != nil {
        return errors.New("unable to create composite key")
    }

    assetBytes, err := sc.stub.GetState(assetKey)
    if err != nil {
        return errors.New("unable to interact with world state")
    }
    if assetBytes == nil {
        return fmt.Errorf("cannot update world state with key %s. Does not exist",
            ⇨ assetID)
    }

```

```

}

var asset Asset
err = json.Unmarshal(assetBytes, &asset)
if err != nil {
    return errors.New("unable to unmarshal asset")
}

asset.Owner = &owner // V: Pointer.

err = sc.stub.PutState(assetKey, assetBytes)
if err != nil {
    return errors.New("unable to interact with world state")
}

return nil
}

// V: Math overflow
func sum(a int32, b int32) int32 {
    return a + b
}

// V: Non Determinism caused by concurrency (Go Routines)
func (sc *ExampleContract) UpdateAssetAmount(assetID string, args ...string) error {
    assetKey, err := sc.stub.CreateCompositeKey("asset", []string{assetID})
    if err != nil {
        return errors.New("unable to create composite key")
    }

    assetBytes, err := sc.stub.GetState(assetKey)
    if err != nil {
        return errors.New("unable to interact with world state")
    }
    if assetBytes == nil {
        return fmt.Errorf("cannot update world state pair with key %s. Does not exist",
            ↪ assetID)
    }

    var asset Asset
    err = json.Unmarshal(assetBytes, &asset)
    if err != nil {
        return errors.New("unable to unmarshal asset")
    }

    var wg sync.WaitGroup
    for _, valueStr := range args {
        wg.Add(1)

```

```

go func(incrementStr string) { // V: Go routines
    defer wg.Done()

    value, _ := strconv.ParseInt(incrementStr, 10, 64) // V: Unhandled error

    res := sum(asset.Amount, int32(value)) // V: Conversion overflow
    if res <= int32(totalCapacity) && res >= 0 { // V: Conversion overflow
        asset.Amount = res
    }
}(valueStr)
}
wg.Wait()

updatedAssetBytes, err := json.Marshal(asset)
if err != nil {
    return errors.New("unable to marshal asset")
}

err = sc.stub.PutState(assetKey, updatedAssetBytes)
if err != nil {
    return errors.New("unable to interact with world state")
}

return nil
}

func (sc *ExampleContract) Read(assetID string) (string, error) {
    assetKey, err := sc.stub.CreateCompositeKey("asset", []string{assetID})
    if err != nil {
        return "", errors.New("unable to create composite key")
    }

    existing, err := sc.stub.GetState(assetKey)
    if err != nil {
        return "", errors.New("unable to interact with world state")
    }

    if existing == nil {
        return "", fmt.Errorf("cannot read world state pair with key %s. Does not exist",
            ↪ assetKey)
    }

    return string(existing), nil
}

// V: Read After Write
func (sc *ExampleContract) UpdateAssetDescription(key string, description string)
    ↪ (*Asset, error) {

```

```

    assetKey, err := sc.stub.CreateCompositeKey("asset", []string{key})
    if err != nil {
        return nil, errors.New("unable to create composite key")
    }

    assetBytes, err := sc.stub.GetState(assetKey)
    if err != nil {
        return nil, errors.New("unable to interact with world state")
    }
    if assetBytes == nil {
        return nil, fmt.Errorf("cannot update world state pair with key %s. Does not
        ↪ exist", key)
    }

    var asset Asset
    err = json.Unmarshal(assetBytes, &asset)
    if err != nil {
        return nil, errors.New("unable to unmarshal asset")
    }

    asset.Description = description

    updatedAssetBytes, err := json.Marshal(asset)
    if err != nil {
        return nil, errors.New("unable to marshal asset")
    }

    err = sc.stub.PutState(assetKey, updatedAssetBytes)
    if err != nil {
        return nil, errors.New("unable to interact with world state")
    }

    // V: Read After Write
    assetBytes, err = sc.stub.GetState(assetKey)
    if err != nil {
        return nil, errors.New("unable to interact with world state")
    }

    err = json.Unmarshal(assetBytes, &asset)
    if err != nil {
        return nil, errors.New("unable to unmarshal asset")
    }

    return &asset, nil
}

func (sc *ExampleContract) ReadAllAssets(args ...string) ([]Asset, error) {
    if len(args) != 0 { // Workaround for PDChecker

```

```

        return nil, nil
    }

    iterator, err := sc.stub.GetStateByPartialCompositeKey("asset", []string{})
    if err != nil {
        return nil, errors.New("unable to interact with world state")
    }
    defer iterator.Close()

    var assetsMap = make(map[string]Asset)
    for iterator.HasNext() {
        queryResponse, err := iterator.Next()
        if err != nil {
            return nil, errors.New("unable to get next element")
        }

        _, cKeyParts, err := sc.stub.SplitCompositeKey(queryResponse.Key)
        if err != nil {
            return nil, errors.New("unable to split key")
        }

        var asset Asset
        err = json.Unmarshal(queryResponse.Value, &asset)
        if err != nil {
            return nil, errors.New("unable to unmarshal")
        }

        assetKey := cKeyParts[1]
        assetsMap[assetKey] = asset
    }

    var assets = make([]Asset, 0, 100)
    // V: Range over map.
    for _, asset := range assetsMap {
        assets = append(assets, asset)
    }

    return assets, nil
}

func (sc *ExampleContract) ChangeTotalCapacity(valueStr string) error {
    value, err := strconv.ParseUint(valueStr, 10, 64)
    if err != nil {
        return errors.New("unable to parse string to uint")
    }

    totalCapacity = value

```

```

    return nil
}

// V: cross-channel invocation
func (sc *ExampleContract) TransferAnotherAsset(ownerID string) error {
    response := sc.stub.InvokeChaincode("TransferChaincode",
        ↳ toChaincodeArgs("TransferAnotherAsset", ownerID), "another-channel")
    if response.GetStatus() != shim.OK {
        return errors.New("unable to invoke another chaincode")
    }

    return nil
}

// V: Phantom Read
func (sc *ExampleContract) UpdateAssetsByType(assetType string) error {
    queryString := fmt.Sprintf(`{"selector":{"amount":"%s"}}`, assetType)

    resultsIterator, err := sc.stub.GetQueryResult(queryString)
    if err != nil {
        return err
    }
    defer resultsIterator.Close()

    for resultsIterator.HasNext() {
        queryResult, err := resultsIterator.Next()
        if err != nil {
            return err
        }
        var asset Asset
        err = json.Unmarshal(queryResult.Value, &asset)
        if err != nil {
            return err
        }

        asset.Amount += 1

        updatedAssetBytes, err := json.Marshal(asset)
        if err != nil {
            return errors.New("unable to marshal asset")
        }

        err = sc.stub.PutState(queryResult.Key, updatedAssetBytes)
        if err != nil {
            return errors.New("unable to interact with world state")
        }
    }
}

```

```

    return nil
}

func (sc *ExampleContract) UnhandledError(assetID string) {
    // V: Unhandled Error
    sc.Read(assetID)
}

// toChaincodeArgs receives dynamic number of strings as parameters.
// It returns array byte of chaincode args.
func toChaincodeArgs(args ...string) [][]byte {
    bargs := make([][]byte, len(args))
    for i, arg := range args {
        bargs[i] = []byte(arg)
    }

    return bargs
}

```

Listing A.2 – Arquivo example_contract.go

APÊNDICE B – ARTIGO EM FORMATO SBC

Este apêndice apresenta o artigo científico formatado conforme normas da SBC.

Análise de Ameaças e Vulnerabilidades em Blockchains Permissionadas

Cristian Alexandre Alchini¹

¹Instituto de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

Abstract. *Blockchain technology has proven to be a promising tool for scenarios that require high reliability, data integrity, and decentralization. Beyond public blockchains, permissioned blockchains offer advantages such as access control, greater efficiency, and transparency, being especially suitable for applications that handle sensitive data and require high levels of security and performance. This work aims to analyze the main vulnerabilities and threats in permissioned blockchains. Through a systematic literature review, the main vulnerabilities identified by the scientific community are examined, and based on this analysis, practical experiments for detecting vulnerabilities in smart contracts are proposed, directed toward a concrete use case, with the goal of evaluating the performance of detection mechanisms. Ultimately, this work seeks to contribute to the development of more robust and secure practices for applications on permissioned blockchains.*

Resumo. *A tecnologia blockchain tem se mostrado uma ferramenta promissora para cenários que exigem alta confiabilidade, integridade dos dados e descentralização. Para além das blockchains públicas, as blockchains permissionadas oferecem vantagens como controle de acesso, maior eficiência e transparência, sendo especialmente adequadas para aplicações que lidam com dados sensíveis e exigem alto nível de segurança e desempenho. Este trabalho tem como objetivo analisar as principais vulnerabilidades e ameaças em blockchains permissionadas. A partir de uma revisão sistemática da literatura são identificados as principais vulnerabilidades apontadas pela comunidade científica, e a partir desta análise, propõe-se experimentos práticos de detecção de vulnerabilidades em contratos inteligentes, direcionados a um caso de uso concreto, visando avaliar a performance dos mecanismos de detecção. Ao final, espera-se contribuir para o desenvolvimento de práticas mais robustas e seguras para aplicações sobre blockchains permissionadas.*

1. Introdução

O conceito de blockchain foi apresentado pela primeira vez por Satoshi Nakamoto [Nakamoto 2008], descrevendo uma estrutura que funciona como um livro-razão. Essa estrutura é composta por blocos que armazenam dados e estão interligados de modo a formar uma cadeia. A ligação entre os blocos ocorre a partir de um hash criptográfico que referencia o conteúdo do bloco anterior (com exceção do primeiro bloco). Caso os dados de um bloco sejam alterados, o hash do bloco subsequente se torna inválido, comprometendo toda a cadeia. Essas características garantem a integridade e a imutabilidade da rede.

Além disso, a blockchain opera de maneira distribuída, onde cada nó da rede mantém uma cópia completa de toda a cadeia de blocos. Para garantir a consistência da cadeia entre os diferentes nós, é utilizado um protocolo de consenso que assegura que todos os participantes possuam o mesmo registro e concordem com a inserção de novas informações. Essa característica proporciona confiança aos participantes da rede, permitindo uma abordagem descentralizada e eliminando a necessidade de uma autoridade central [Vieira et al. 2023].

As blockchains públicas, como por exemplo, a rede Bitcoin¹, não requerem autorização para participação, permitindo que qualquer pessoa possa realizar transações na rede. Além disso, essas redes são abertas e auditáveis. Já blockchains permissionadas, são redes restritas, nas quais o acesso é controlado por uma entidade ou grupo de entidades confiáveis, não sendo acessíveis a qualquer participante. Apenas usuários com permissão podem atuar na rede. Um exemplo de ferramenta no contexto destas blockchains é o Hyperledger Fabric, da Linux Foundation [Hyperledger Foundation 2024].

As blockchains permissionadas estão sendo adotadas tanto por empresas globais [Li et al. 2022] quanto pelo setor público [BlockNews 2024], desempenhando um papel fundamental em transações de alto impacto financeiro e social. As blockchains, contudo, mesmo com seus mecanismos de segurança distribuídos, podem conter vulnerabilidades que, se exploradas, podem comprometer a consistência e a confidencialidade dos dados, bem como a disponibilidade dos serviços, gerando prejuízos significativos.

A história das blockchains já registra incidentes marcantes, como o ataque à DAO, ocorrido em junho de 2016, na rede Ethereum². Na ocasião, uma falha de implementação em um contrato inteligente - código executado de maneira distribuída sobre a blockchain - resultou em uma perda estimada de 70 milhões de dólares [Homoliak et al. 2021]. Embora esse caso tenha ocorrido em uma blockchain pública, ele evidencia como vulnerabilidades podem ter consequências graves em sistemas baseados em tecnologias descentralizadas.

Nesse contexto, este trabalho realiza uma revisão bibliográfica para mapear as principais vulnerabilidades, ameaças e ataques em blockchains permissionadas. Após este mapeamento, se buscará compreender quais são as principais ferramentas disponíveis para detecção dessas vulnerabilidades, com foco principal em contratos inteligentes. Nesse sentido, serão realizados experimentos com vistas a avaliar a eficácia destas ferramentas de detecção. Com este trabalho, busca-se contribuir para a segurança de sistemas que utilizam blockchains permissionadas.

2. Trabalhos relacionados

Em [Homoliak et al. 2021] os autores apresentam um modelo de análise de ameaças e riscos de segurança em blockchains, acompanhado de possíveis contramedidas. São introduzidos dois modelos: o *stacked model* e o *threat-risk model*. O *stacked model* organiza a arquitetura em quatro camadas: aplicação, replicação da máquina de estados, consenso e rede. Já o *threat-risk model* contempla os principais componentes e agentes envolvidos — como proprietários, ativos, agentes de ameaça, ameaças, contramedidas e riscos —, além de sobrepor-se ao modelo anterior, proporcionando uma visão integrada.

¹bitcoin.org/en

²ethereum.org/en

Também realizando uma classificação de vulnerabilidades, em [Brotsis et al. 2020] os autores realizam uma análise da ferramenta sob a ótica da segurança, dividindo sua investigação em quatro componentes, nos quais podem ocorrer ataques e vazamentos de informação: (i) o protocolo de consenso, (ii) os contratos inteligentes (chaincodes), (iii) a rede e (iv) os mecanismos de preservação da privacidade. Para cada um desses componentes, são identificados os riscos específicos e apresentadas contramedidas com o objetivo de fortalecer a segurança do sistema como um todo.

Com relação à ataques específicos, o trabalho de [Jayadev et al. 2024] realiza simulações de ataque DDoS em Hyperledger Fabric e Besu. Os resultados obtidos indicam que, nos diferentes cenários testados, o Hyperledger Fabric apresentou maior resiliência frente ao ataque DDoS, alcançando melhores índices de latência e vazão, com diferentes números de nós, transações e usuários simulados.

O trabalho de [La Salle et al. 2023] realiza uma modelagem do Hyperledger Fabric (HLF) com Redes de Petri, para simular ataques sybil. Os resultados obtidos indicam que este ataque é mais bem-sucedido quando há um número maior de pares de endosso maliciosos. Além disso, uma maior taxa de propostas de transações por parte do cliente malicioso também favorece o sucesso do ataque. Dessa forma, no que se refere a estratégias para mitigar a efetividade do ataque Sybil, os resultados sugerem que a redução no número de pares de endosso regulares e a limitação da taxa de propostas de transações por cliente são medidas eficazes para dificultar o sucesso do ataque.

O trabalho de [Hamdi et al. 2024] também classifica os ataques em blockchain em diferentes categorias, mas foca seus experimentos na segurança dos contratos inteligentes. Com relação ao Fabric, realiza um estudo comparativo com ferramentas de detecção de vulnerabilidades em chaincodes, verificando a capacidade destas em encontrar os riscos: variáveis globais, gerador de números aleatórios, riscos em consultas por faixa (range query) e read-your-write. Os resultados mostraram que as ferramentas estudadas são capazes de encontrar a maioria das vulnerabilidades propostas, alcançando taxas de pelo menos 75% de cobertura na detecção.

Com relação à ferramentas de detecção de vulnerabilidades em contratos inteligentes de Fabric, o trabalho de [Chen et al. 2024] apresenta uma ferramenta chamada PDChecker³, voltada à detecção automática de vazamentos de dados privados, bem como de outros riscos associados às chaincodes. Os resultados mostraram que, para detecção de vazamentos de dados privados, a acurácia foi superior à 94% e, com relação à outros conjuntos de vulnerabilidades, obteve desempenho superior comparado a outras ferramentas de análise.

Também neste domínio, o artigo de [Li et al. 2022] propõe uma solução chamada de HFCCT⁴, voltada para a detecção automática de vulnerabilidades em chaincodes. Como resultados de experimentos com a ferramenta, os autores apontaram que 10% dos erros estavam relacionados a falhas sintáticas no uso da linguagem Go; 15% a comportamentos não determinísticos, como uso de números aleatórios, iteração sobre estruturas do tipo Map e utilização de timestamps do sistema; 30% a funções de consulta por intervalo, como GetQueryResult(); e 45% a erros não tratados.

³github.com/zm-stack/PDChecker

⁴github.com/bugfrog1024/HFCCT

Por fim, considerando em um escopo mais abrangente de segurança, o trabalho de [Cha et al. 2022] busca estabelecer diretrizes de avaliação e confiabilidade para sistemas de blockchains permissionadas. É então proposto um framework de gerenciamento de riscos de segurança para aplicações em blockchains permissionadas. O estudo classifica as medidas de proteção em seis categorias. Classificando práticas de segurança da informação já estabelecidas e incluindo controles específicos relacionados a blockchains permissionadas.

3. Vulnerabilidades em Contratos Inteligentes

A listagem apresentada nesta seção sintetiza as vulnerabilidades identificadas na literatura científica. Como os trabalhos focam principalmente em Hyperledger Fabric como plataforma de implementação de blockchains permissionadas e Go como linguagem de implementação dos contratos, essa lista estará também relacionada a estas ferramentas.

- **Vulnerabilidades decorrentes da arquitetura do Fabric:** se refere a vulnerabilidades decorrentes do uso inadequado de mecanismos específicos do modelo *execute-order-validate* do Fabric.
 - **Phantom read:** O Hyperledger Fabric fornece alguns métodos de consulta por intervalo para acessar os bancos de dados de estado, como `GetQueryResult()`, `GetPrivateDataQueryResult()` e `GetHistoryForKey()`. Esses métodos são executados durante a fase de endosso da transação, mas não são reexecutados durante a fase de validação. O que significa que uma eventual alteração nos dados entre a fase de endosso e a fase de validação não será detectada. Portanto, modificações no ledger baseadas nos resultados da query podem operar sobre dados obsoletos, causando phantom reads ou inconsistências lógicas.
 - **Read your write:** No Hyperledger Fabric, a operação de escrever os dados de uma transação no ledger somente é executada após a transação ser concluída e verificada. Portanto, se uma leitura é executada logo após uma escrita, ela não irá obter o valor atualizado esperado como retorno.
 - **Invocação de chaincode entre canais:** introduzem complexidade significativa e podem levar a violações dos princípios de isolamento e privacidade que os canais foram projetados para garantir.
- **Vazamento de dados Privados:** refere-se a exposição não autorizada de informações sensíveis devido ao uso inadequado de Private Data Collections (PDCs) do Fabric.
 - **Dados privados em argumentos de funções:** Parâmetros passados em invocações de chaincode são incluídos na proposta de transação e gravados no ledger público, tornando-se acessíveis a todos os peers da rede. Dados sensíveis devem ser transmitidos através de *transient data*, que não são persistidos no ledger.
 - **Dados privados em operações condicionais:** o fluxo de execução de um programa é afetado pelas operações condicionais. Se dados privados são utilizados nessas operações, um eventual atacante pode começar a inferir informações privadas a partir dos diferentes caminhos que um programa percorre. Por exemplo, se o saldo de uma conta é negativo e uma operação de saque não autorizada é informada em uma transação, a informação sobre a situação do saldo é vazada para todos os participantes.

- **Dados privados no *payload* de retorno:** uma chaincode pode retornar dados em uma função dentro do *payload* de uma transação, que poderá ser acessado por todos os participantes. Nesse sentido, chaincodes não podem incluir dados privados que sejam incluídos no payload em retorno de funções.
- **Não determinismo interno:** Trata tanto de fontes de dados que podem ter valores diferentes nos diferentes pares - como variáveis globais e campos da chaincode - como processos de execução não determinística, onde a execução de uma mesma função em diferentes peers pode levar a resultados diferentes e incertos.
 - **Variáveis globais de estado e declaração de campo:** Variáveis globais e campos da struct da chaincode podem ser alterados localmente. Devido à política de endosso, cada nó par não executa necessariamente todas as transações. Portanto, os valores armazenados dessa maneira podem ficar inconsistentes entre os nós.
 - **Range map:** Devido aos detalhes internos da implementação da linguagem Go, a iteração sobre estruturas do tipo *map* com o uso de *range* é não-determinística, a ordem das chaves pode variar. Assim, é necessário atenção ao manipular dados escritos do *ledger* dessa maneira.
 - **Reificação de objetos (ponteiros):** No caso de manipulação de variáveis por ponteiros, o valor recuperado em um endereço específico muito provavelmente irá variar entre ambientes. Nesse sentido, deve haver cuidado para não armazenar o endereço da variável no ledger.
 - **Concorrência no programa:** Se programas concorrentes não forem tratados corretamente, é fácil causar condições de corrida que resultam em execução não determinística, resultando em falha da transação.
 - **Timestamp do sistema:** É difícil garantir que os timestamps sejam executados de forma simultânea em todos os peers. Sua utilização de modo incorreto resultará na falha do processo de endosso.
 - **Geração de números aleatórios:** Como a chaincode é simulada de forma independente por cada par na fase de endosso, os valores gerados aleatoriamente em cada par serão diferentes. Aplicando-se os mesmos efeitos citados anteriormente.
- **Não determinismo externo:** refere-se às chamadas ou execução de recursos externos à blockchain, onde não há um controle sobre o resultado retornado, podendo gerar inconsistência entre os pares.
 - **Acesso a serviços web, acesso a bibliotecas externas e arquivos do sistema e execução de comandos do sistema:** Chamadas externas não determinísticas podem ter diferentes lógicas de execução ou fontes de dados incertas em diferentes pares. Por exemplo, se as chaincodes fazem chamadas a serviços web, quando diferentes pares executam essas chaincodes para obter dados do site, esse site pode retornar dados diferentes para cada par, causando inconsistência.
- **Vulnerabilidades de implementação não específicas da plataforma:** classe de vulnerabilidades que existem independentemente da tecnologia blockchain, mas cujas consequências são amplificadas no contexto de smart contracts devido a características como imutabilidade, irreversibilidade de transações, e alto valor dos ativos gerenciados.

- **Overflow matemático:** se uma chaincode soma valores de tokens ou ativos sem a validação adequada, um atacante poderia explorar isso para criar transações que, ao ultrapassarem o limite máximo do tipo numérico, resultem em valores negativos ou muito pequenos, permitindo, por exemplo, transferências fraudulentas ou manipulação de saldos.
- **Overflow em conversão de tipos:** ocorre quando dados são convertidos entre tipos diferentes (como de string para inteiro, ou de int64 para int32) sem validação apropriada dos limites. Um atacante pode explorar isso enviando valores que são válidos no tipo original, mas que se tornam corrompidos após a conversão. Isso pode permitir, por exemplo, o bypass de validações de limites, a manipulação de quantidades em transferências de ativos ou a corrupção de dados no world state, comprometendo a integridade das transações.
- **Erros não tratados:** Em Go, não há o mecanismo de exceção, fazendo com que os desenvolvedores devam tratar os erros retornados pelas funções de maneira apropriada.
- **Parâmetros não checados:** comum em qualquer cenário de desenvolvimento. As entradas das funções devem ser verificadas de modo a evitar eventuais falhas durante a execução da função.

4. Experimentos

Será conduzido um estudo experimental utilizando ferramentas de análise estática de código e linters especializados. Especificamente, será analisado tanto um contrato inteligente vulnerável, de modo a validar as ferramentas testadas, quanto chaincodes de um caso de uso real, com o objetivo de identificar potenciais falhas de segurança observados na literatura. Adicionalmente, serão discutidas propostas de melhoria para tais ferramentas, fundamentadas na aplicação prática e nas evidências coletadas durante os testes.

4.1. Ferramentas de Análise Utilizadas

As ferramentas selecionadas foram PDChecker, HFCCT (Hyperledger Fabric Chaincode Testing Tool) e Golangci-lint⁵.

4.1.1. PDChecker

O PDChecker é uma ferramenta de análise estática desenvolvida em Go como uma extensão do linter Revive⁶, sobre o qual adiciona novas regras de verificação. Seu propósito é detectar vulnerabilidades e más práticas específicas de contratos inteligentes de Hyperledger Fabric escritos também em Go.

4.1.2. HFCCT

O HFCCT é uma ferramenta de análise desenvolvida em Python e Go. Seu funcionamento baseia-se em duas abordagens: análise estática e análise simbólica. No entanto, durante

⁵<https://golangci-lint.run/>

⁶<https://github.com/mgechev/revive>

os testes realizados, apenas a análise estática pôde ser executada, devido a problemas de compatibilidade com as chaincodes de teste.

4.1.3. Golangci-lint

O Golangci-lint é uma ferramenta utilizada na comunidade Go para detecção de erros, más práticas e violações de estilo em código-fonte. É composto por um conjunto de diferentes ferramentas, como Staticcheck⁷, um dos principais linters de Go, e Gosec⁸, para análise de segurança. Embora não tenha sido projetado especificamente para chaincodes, o Golangci-lint pode auxiliar na identificação de problemas que podem impactar a segurança de contratos inteligentes.

Considerando o foco em contratos inteligentes e as vulnerabilidades citadas anteriormente, essa ferramenta foi executada na chaincode de testes com vistas a descobrir quais vulnerabilidades poderiam ser detectadas por ela.

Vulnerabilidade	PDChecker	HFCCT	Golangci-lint
Declaração Campo		x	
Range map	x	x	
Ponteiro	x	x	
Read after write	x	x	
Concorrência	x	x	
Importação <i>blacklisted</i>	x	x	
Invoc. canal ext.	x	x	
Erro não tratado	x	x	x
Variável global	x		x
Overflow conversão	x		x
Parâmetro não checado	x		
Privacidade - cond.	x		
Privacidade - ret.	x		
Phantom read	x		
Privacidade - arg	x		
Overflow matemático	x		

Tabela 1. Vulnerabilidades cobertas pelas ferramentas

4.2. Chaincode Vulnerável

Para avaliar a efetividade das ferramentas, estas foram inicialmente executadas em uma chaincode, desenvolvida neste trabalho, com vulnerabilidades inseridas propositalmente no código. Nessa chaincode foram incluídas todas as vulnerabilidades que tanto o PD-Checker quanto o HFCCT se propõem a detectar. Esta chaincode foi desenvolvida em Go, para Hyperledger Fabric, e realiza operações de leitura e escrita sobre assets definidos no próprio arquivo do contrato.

⁷<https://github.com/dominikh/go-tools>

⁸<https://github.com/securego/gosec>

4.3. Caso de uso

O caso de uso utilizado para experimentação é o projeto RRDES. Este projeto é uma iniciativa do Laboratório de Segurança em Computação - LabSEC da Universidade Federal de Santa Catarina - UFSC. Seu objetivo é construir uma rede de rastreabilidade do ensino superior brasileiro, vinculando essa rede ao Diploma Digital⁹, já em implantação pelo Ministério da Educação - MEC. Essa rede é construída com Hyperledger Fabric e, atualmente, recebe como entrada documentos no formato XML e no padrão do diploma digital. Para cada documento há uma chaincode específica. No momento, há um total de oito chaincodes.

4.4. Ambiente de experimentação

Os testes foram realizados em um computador pessoal executando o sistema operacional Ubuntu 24.04, equipado com um processador Intel i7-1255U e 16 GB de memória RAM. Todas as ferramentas foram executadas localmente, sem o uso de ambientes em nuvem ou virtualização adicional. As ferramentas PDChecker e HFCCT foram obtidas diretamente de seus repositórios oficiais no GitHub, e o Golangci-lint foi instalado via Go.

5. Resultados

5.1. Chaincode experimental

Vulnerabilidade	Qde.	PDChecker	HFCCT	Golangci-lint
Declaração Campo	1	-	0	-
Range map	1	1	1	-
Ponteiro	2	13(3)	2	-
Read after write	1	1	1	-
Concorrência	1	1	1	-
Importação <i>Blacklisted</i>	1	1	0	-
Invoc. canal ext.	1	1	0	-
Erro não tratado	3	1	4(1)	3
Variável global	1	1	-	1
Overflow conversão	2	2	-	2
Parâmetro não checado	20	21	-	-
Privacidade - cond.	1	1	-	-
Privacidade - ret.	1	1	-	-
Phantom read	1	1	-	-
Privacidade - arg	1	1	-	-
Overflow matemático	1	1	-	-

Tabela 2. Vulnerabilidades inseridas (Qde.) e os resultados apontados por cada ferramenta. Os valores entre parênteses indicam a quantidade de *verdadeiros positivos*.

Em relação à chaincode experimental, conforme apresentado na tabela 2, o PDChecker detectou cerca de 95% das vulnerabilidades às quais se propõe a detectar, enquanto o HFCCT obteve uma taxa de 55%. O Golangci-lint, após execução na chaincode,

⁹portal.mec.gov.br/diplomadigital/

verificou-se que é capaz de detectar 3 vulnerabilidades. Considerando, a ocorrência de falsos positivos, o PDChecker apontou 23% de falsos positivos e o HFCCT 33%. Enquanto o Golangci-lint obteve 100% de verdadeiros positivos.

Ferramenta	Taxa de detecção (%)	Precisão (%)	Cobertura das vulnerabilidades
PDChecker	95	77	Alta
HFCCT	55	67	Média
Golangci-lint	100	100	Baixa

Tabela 3. Resumo comparativo entre as ferramentas de análise.

5.2. Chaincodes RRDES

Vulnerabilidade	Histórico		Diploma		Currículo		Lato		Conselhos		Dados Reg.		Lattes		XMLLog		Total	
	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.
Parâmetro não checado	39	19	39	14	-	-	29	14	16	10	14	8	8	7	20	15	165	87
Ponteiro	7	0	7	0	32	0	10	0	2	0	4	0	2	0	6	0	70	0
Invoc. canal ext.	3	0	2	0	2	0	1	0	1	0	-	-	-	-	-	-	9	0
Importação <i>blacklisted</i>	-	-	2	0	-	-	1	0	-	-	-	-	-	-	-	-	3	0
Privacidade - cond.	1	0	-	-	-	-	1	0	-	-	-	-	-	-	-	-	2	0
Privacidade - ret.	-	-	-	-	-	-	1	1	-	-	-	-	-	-	-	-	1	1
Total	50	19	50	14	34	0	43	15	19	10	18	8	10	7	26	15	250	88

Tabela 4. PDChecker: vulnerabilidades detectadas (Det.) e confirmadas (Ver.) em cada chaincode.

A Tabela 4 apresenta os resultados da execução do PDChecker sobre os arquivos das chaincodes do projeto RRDES. Das quinze vulnerabilidades que a ferramenta se propõe a detectar, apenas seis foram encontradas no conjunto analisado. De modo geral, observa-se que, em comparação com a análise realizada sobre a chaincode experimental, o número de falsos positivos é consideravelmente maior. Dos 250 alertas emitidos pelo PDChecker, apenas 88 foram confirmados, resultando em uma taxa de verdadeiros positivos de aproximadamente 35% e de falsos positivos de 65%. Além disso, das seis vulnerabilidades detectadas pela ferramenta, apenas duas apresentaram resultados verificados como verdadeiros, enquanto as demais corresponderam a alertas incorretos.

Vulnerabilidade	Histórico		Diploma		Currículo		Lato		Conselhos		Dados Reg.		Lattes		XMLLog		Total	
	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.
Erro não tratado	7	1	6	0	17	0	2	0	5	1	0	0	3	1	2	0	42	3
Range risk	6	0	6	0	28	0	6	0	2	0	4	0	2	0	3	0	57	0
Ponteiro	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	0	1	0
Total	13	1	12	0	45	0	8	0	7	1	4	0	5	1	6	0	100	3

Tabela 5. HFCCT: vulnerabilidades detectadas (Det.) e confirmadas (Ver.) em cada chaincode.

A Tabela 5 apresenta os resultados obtidos com a execução do HFCCT sobre as chaincodes do projeto RRDES. Entre os alertas gerados, apenas os casos classificados como erro não tratado se confirmaram como verdadeiros, totalizando três ocorrências, uma em cada uma das chaincodes Lato, Histórico e Conselhos. Todos os demais apontamentos correspondem a falsos positivos. De forma geral, a eficácia da ferramenta

Vulnerabilidade	Histórico		Diploma		Currículo		Lato		Conselhos		Dados Reg.		Lattes		XMLLog		Total	
	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.	Det.	Ver.
Erro não tratado	3	3	3	3	3	3	5	5	2	2	3	3	2	2	3	3	24	24
Erro - atribuição inefetiva	-	-	-	-	-	-	-	-	-	-	-	-	1	1	-	-	1	1
Total	3	3	3	3	3	3	5	5	2	2	3	3	3	3	3	3	25	25

Tabela 6. Golangci-lint: vulnerabilidades detectadas (Det.) e confirmadas (Ver.) em cada chaincode.

mostrou-se limitada: dos 100 alertas emitidos, apenas 3 foram confirmados, resultando em uma taxa de verdadeiros positivos de 3% e de falsos positivos de 97%.

A Tabela 6 apresenta os resultados da execução do Golangci-lint sobre as chaincodes do projeto RRDES. A ferramenta identificou dois tipos de problemas, ambos relacionados à vulnerabilidade de erro não tratado. Apesar do Golangci-lint possuir um escopo de análise mais restrito em relação a vulnerabilidades específicas de chaincodes, a ferramenta demonstrou elevada precisão, uma vez que todos os alertas gerados foram confirmados como verdadeiros positivos.

Observa-se que as três ferramentas apresentam comportamentos distintos quanto à cobertura e precisão. Enquanto o PDChecker abrange um conjunto maior de vulnerabilidades, apresentou alta taxa de falsos positivos. O HFCCT, embora mais restrito, mostrou baixa eficácia nas chaincodes reais. Já o Golangci-lint, com escopo limitado a más práticas de código, obteve o melhor desempenho em precisão, sem falsos positivos.

6. Limitações das Ferramentas Analisadas e Possíveis Melhorias

A partir dos resultados obtidos e de uma análise aprofundada das ferramentas avaliadas, observa-se que ainda há espaço significativo para melhorias visando aumentar a eficácia e a precisão dos analisadores. Para que essas ferramentas cumpram seu papel no apoio ao desenvolvimento seguro de contratos inteligentes, é essencial que consigam auxiliar o desenvolvedor a identificar problemas de maneira antecipada, reduzindo esforço manual e a análise de falsos positivos.

No contexto do PDChecker e das vulnerabilidades que ele busca detectar, destaca-se a regra referente ao uso de ponteiros em chaincodes. A implementação atual dessa verificação se mostra limitada, pois qualquer ocorrência de ponteiros, seja por meio da declaração de variáveis do tipo ponteiro ou pelo uso do operador '&' para obtenção de endereços, gera um alerta. De forma similar, o HFCCT também pode ser aprimorado nesse aspecto, pois sua verificação se restringe apenas a identificar as operações "new()", "= *p" e "= &p", sem correlacioná-las ao uso efetivo dos dados armazenados no ledger.

Essa regra deveria ser mais precisa, sinalizando apenas os usos que possam introduzir não determinismo e comprometer o processo de endosso das transações. Em particular, o risco real decorre da utilização de endereços de memória como parte de dados persistidos no ledger, como em atribuições do tipo `asset.owner = &owner` seguidas de operações de persistência, conforme 1

```
func (sc *ExampleContract) UpdateAssetOwnership() error {
    {...}
}
```

```

    asset.Owner = &owner // Assigning address.

    assetBytes, err = json.Marshal(asset)
    if err != nil {
        return errors.New("unable to marshal asset")
    }

    // Storing in the ledger
    err = sc.stub.PutState(assetKey, assetBytes)
    if err != nil {
        return errors.New("unable to store data")
    }

    return nil
}

```

Listing 1. Uso incorreto de ponteiros

Nesses cenários, o endereço de memória pode variar entre peers, levando a divergência no estado e falhas de consenso. Portanto, uma melhoria relevante seria a capacidade da ferramenta de correlacionar a detecção de ponteiros com chamadas a métodos de gravação no ledger como `PutState`, aumentando a precisão e reduzindo falsos positivos.

Outro ponto relevante diz respeito à regra que identifica erros não tratados. A implementação atual do PDChecker limita-se a verificar se o valor de retorno do tipo `error` foi completamente ignorado, sinalizando apenas chamadas de funções cujo erro não é capturado. Dessa forma, atribuições como `err := f()` ou `_ = f()` não geram alerta, ainda que o erro não seja posteriormente analisado ou tratado.

```

func f() error {
    return error
}

func ErrorHandling() {
    err := f() // Not detected

    _ = f() // Not detected

    f() // Detected
}

```

Listing 2. Detecção de erros não tratados - PDChecker

Embora essa lógica contribua para evitar situações em que o erro é simplesmente descartado, ela ainda é restrita, pois não abrange cenários em que o desenvolvedor armazena o erro, mas não realiza qualquer verificação ou ação subsequente. O HFCCT é também bastante limitado, na medida em que apenas aponta a existência de '_', no código, sem diferenciá-los de outros usos que não envolvam o tratamento de erros.

Uma melhoria potencial para essa regra consistiria em estender a capacidade de análise, de modo a identificar casos nos quais o erro é atribuído, porém não utilizado em

estruturas condicionais ou fluxos de tratamento, tais como `if err != nil {...}`.

Quanto à detecção de bibliotecas *blacklisted*, o PDChecker foi capaz de identificar o uso de uma dependência potencialmente problemática no código analisado, especificamente a biblioteca `time`. Contudo, essa capacidade ainda se mostra bastante limitada. A ferramenta mantém apenas uma pequena lista de bibliotecas que podem gerar alertas e, além disso, não realiza uma análise mais profunda sobre quais funções dessas dependências são efetivamente utilizadas.

Sendo assim, uma ferramenta de análise mais robusta deveria ser capaz não apenas de identificar dependências potencialmente perigosas, mas também de avaliar o uso específico das funções importadas e rastrear o fluxo de dados resultante, verificando se há impacto sobre o determinismo, o armazenamento no ledger ou condições lógicas sensíveis dentro da execução da transação.

Quanto a utilização de declaração de campos na struct da chaincode, essa é uma característica que o HFCCT se propõe a detectar e que é importante, considerando que possui os mesmos efeitos de utilização que o uso de variáveis globais. Contudo, na chaincode de exemplo, com essa vulnerabilidade exposta, a ferramenta não foi capaz de apontar o problema.

A implementação dessa regra no HFCCT assume um padrão específico de desenvolvimento baseado na chamada `shim.Start(new(<StructName>))`, a partir da qual extrai o nome da struct da chaincode e busca localizar sua definição no código. No entanto, as chaincodes utilizadas neste trabalho foram desenvolvidas com a biblioteca `contractapi`, que segue um padrão distinto de inicialização, conforme 3.

```
example := &chaincode.ExampleContract{}
cc, err := contractapi.NewChaincode(example)
if err := cc.Start(); err != nil {
    panic(err.Error())
}
```

Listing 3. Inicialização da chaincode de exemplo

Nesse modelo, não há invocação de `shim.Start(new(...))`, o que impede o HFCCT de identificar corretamente o nome da estrutura e, consequentemente, de aplicar a regra. Como resultado, declarações de campos na struct, como em ?? não são detectadas pela ferramenta, apesar de representarem exatamente o cenário que a regra busca identificar.

Uma melhoria recomendada seria a adaptação da lógica de detecção para contemplar chaincodes que utilizam o padrão `contractapi`, permitindo a identificação do nome da struct a partir da passagem de referência ao construtor `contractapi.NewChaincode(<struct>)`. Além disso, esta mesma regra poderia ser adicionada ao PDChecker de modo a aumentar a cobertura de vulnerabilidades abordadas pela ferramenta.

Quanto à vulnerabilidade de *read after write*, embora ambas as ferramentas consigam identificá-la na chaincode de exemplo, elas apresentam limitações relevantes. Em particular, não detectam o fluxo interprocedural, ou seja, quando a operação de escrita e leitura ocorre por meio de chamadas a funções auxiliares, como ilustrado no trecho 4.

Além disso, não contemplam todos os métodos de leitura disponíveis no Fabric.

```
// Read after write -> Not detected
func ReadAfterWrite(assetKey string) {
    StoreAsset(assetKey) // Calls PutState(assetKey, data)
    return ReadAsset(assetKey) // Calls GetState(assetKey)
}
```

Listing 4. Read after write

O HFCCT detecta apenas chamadas diretas às funções de leitura e escrita `PutState` e `GetState`. O `PDChecker`, por sua vez, possui cobertura parcialmente superior ao incluir também `PutPrivateData` e `GetPrivateData`, que realizam operações sobre dados privados. Entretanto, ainda se observa uma lacuna na identificação de outras formas de leitura, como iterações com `GetStateByPartialCompositeKey`, entre outros métodos de consulta do Fabric.

Dessa forma, avanços são possíveis ao incorporar a análise de um conjunto mais amplo de funções de leitura e escrita, bem como ao considerar o fluxo de dados entre funções. Essa limitação também aparece em outras regras, como as relacionadas ao vazamento de privacidade, que igualmente não reconhecem fluxos interprocedurais.

No que se refere aos vazamentos de informações privadas, destaca-se uma limitação relevante na regra destinada a identificar a presença de dados sensíveis em parâmetros de funções, o que pode aumentar o risco de falsos negativos. Essa regra verifica se um parâmetro é utilizado em alguma chamada a `PutPrivateData`. No entanto, se no fluxo de execução, entre a declaração do parâmetro e a chamada a `PutPrivateData`, ocorrer uma chamada a `GetTransient`, mecanismo previsto pelo Fabric para transferência segura de dados privados, o `PDChecker` não sinaliza o possível vazamento proveniente dos argumentos.

```
// Private data in parameter -> Not detected
func StorePrivateData(assetKey string, privateData string) {

    // Retrieving another private data from transient source
    anotherPrivateData := GetTransient()

    // Storing privateData
    PutPrivateData("collection", assetKey, privateData) // Not detected

    // Further processing with anotherPrivateData
}
```

Listing 5. Dados privados em parâmetro de função

Essa limitação é significativa, pois um contrato pode utilizar corretamente `GetTransient` para receber dados privados, mas ainda assim introduzir informações sensíveis por meio de parâmetros de função, seja por desconhecimento do mecanismo ou por descuido no desenvolvimento. Portanto, é desejável que a regra seja aprimorada de forma a considerar esse cenário, distinguindo adequadamente quando um parâmetro representa uma potencial fonte de vazamento de dados privados.

No que se refere aos *phantom reads*, o PDChecker consegue identificar o uso de funções de consulta potencialmente suscetíveis a esse tipo de vulnerabilidade, como `GetQueryResult`, `GetHistoryForKey` e `GetPrivateDataQueryResult`. Contudo, a ferramenta não realiza a correlação entre essas operações de leitura e subsequentes operações de escrita que utilizem os dados retornados. Essa limitação pode resultar em falsos positivos, uma vez que a mera utilização dessas funções não implica, necessariamente, em um cenário de *phantom read*. Uma evolução desejável seria justamente a análise de fluxo entre leitura e escrita para confirmar a presença efetiva da vulnerabilidade no contrato inteligente.

O HFCCT, por sua vez, também se limita a detectar a presença dessas funções de leitura. Entretanto, a ferramenta amplia sua lista equivocadamente, incluindo chamadas como `GetStateByRange` e `GetStateByPartialCompositeKey`, as quais são executadas durante a fase de validação. Essa abordagem, além de igualmente suscetível à geração de falsos positivos, revela um equívoco conceitual sobre a arquitetura do Fabric, visto que é justamente a ausência de execução durante a fase de validação que implica na possibilidade de *phantom reads*.

Uma possível vulnerabilidade adicional que poderia ser contemplada pelo PDChecker é a detecção do uso de funções relacionadas à geração de valores aleatórios, como `rand.Int()`, passíveis de causar comportamento não determinístico e falha nas transações. Como aprimoramento, a ferramenta também poderia analisar se o valor aleatório influencia o estado da blockchain ou decisões de fluxo lógico, distinguindo casos benignos (sem persistência) de situações que efetivamente comprometem o determinismo da execução.

Em síntese, embora as ferramentas analisadas forneçam contribuições relevantes para a detecção de vulnerabilidades em chaincodes, ainda apresentam limitações importantes que podem comprometer a eficácia da análise. A incorporação de verificações mais granulares, sensibilidade ao fluxo de dados, suporte a chamadas interprocedurais e cobertura ampliada de funções do Fabric poderia aumentar a precisão e a utilidade prática das ferramentas. Nesse sentido, há um amplo espaço para evolução dessas abordagens.

7. Conclusão

A segurança em blockchains permissionadas abrange diversos aspectos, que envolvem diferentes níveis de abstração dentro da própria tecnologia, assim como aspectos variados a serem considerados durante sua análise. Dentre essas abordagens, um ponto fundamental é a segurança dos contratos inteligentes.

Os contratos inteligentes, como mecanismos que aplicam a lógica de negócio sobre a blockchain, exigem um desenvolvimento seguro para garantir a proteção dos dados e a correta execução das transações do sistema. Nesse contexto, diferentes estudos apontam vulnerabilidades presentes em contratos inteligentes e reforçam a necessidade de detectá-las previamente, minimizando a ocorrência de problemas futuros. Além disso, apresentam variadas ferramentas destinadas a essa detecção.

Considerando as vulnerabilidades presentes no código-fonte dos contratos, foram realizados experimentos para analisar as principais ferramentas disponíveis para detecção de vulnerabilidades em contratos do Hyperledger Fabric, avaliando sua eficácia e limitações para possível aplicação em projetos reais.

Após a análise dos resultados, constatou-se que as ferramentas analisadas representam um ponto de partida importante para os objetivos a que se propõem, oferecendo regras de análise de código capazes de detectar vulnerabilidades de forma antecipada. Nesse sentido, o PDChecker revelou-se uma ferramenta promissora. É capaz de cobrir um conjunto relevante de vulnerabilidades, possui código aberto, é a ferramenta mais atual dentre as disponíveis e apresenta facilidade de integração a serviços de integração contínua.

Entretanto, algumas ressalvas são necessárias. Apesar das iniciativas importantes, as ferramentas apresentadas na literatura e analisadas nos experimentos ainda são protótipos. Sua capacidade de análise é limitada, com regras simples, o que gera quantidades significativas de falsos positivos e falsos negativos, dificultando sua adoção sem precauções.

No caso específico do HFCCT, um dos módulos da ferramenta não pôde ser executado, o que acabou limitando a análise. Contudo, tratando-se da análise estática observada, verifica-se que é uma ferramenta que não possui a capacidade de ser integrada ao fluxo de trabalho de desenvolvimento de projetos reais. Suas regras são descritas de maneira muito limitada, sua capacidade de detecção é baixa e possui pouca precisão nos alertas gerados.

Quanto ao Golangci-lint, trata-se de uma ferramenta consolidada no ecossistema Go, com contribuições ativas e atualizações constantes. Por isso, apresenta maior precisão nas vulnerabilidades que é capaz de detectar, embora, para questões específicas de blockchain, seus alertas sejam mais limitados. Ainda assim, é essencial para o desenvolvimento geral de projetos em Go.

Em suma, é de extrema importância que ferramentas de análise de código sejam utilizadas na construção de contratos inteligentes e detectem eventuais problemas de maneira antecipada. O PDChecker mostrou potencial para uso prático, embora seja importante considerar suas limitações. Este campo permanece aberto, com muitas oportunidades para futuras pesquisas e desenvolvimento, tanto na identificação de novas vulnerabilidades quanto na criação de ferramentas mais robustas e precisas para a detecção dessas falhas.

Referências

- BlockNews (2024). Linux lança grupo internacional que tem banco central do brasil, serpro e bndes entre fundadores. Acesso em: jun de 2025.
- Brotsis, S., Kolokotronis, N., Limniotis, K., Bendiab, G., and Shiaeles, S. (2020). On the security and privacy of hyperledger fabric: Challenges and open issues. In *2020 IEEE World Congress on Services (SERVICES)*, pages 197–204.
- Cha, S.-C., Shiung, C.-M., Li, W.-W., Peng, C.-N., Hung, Y.-H., and Yeh, K.-H. (2022). Trustworthiness evaluation for permissioned blockchain-enabled applications. *Computers, Materials and Continua*, 73(2):2731–2755.
- Chen, Z., Li, Y., Gao, J., Zhang, J., Wang, K., Hu, J., Guan, Z., and Chen, Z. (2024). Understanding and detecting privacy leakage vulnerabilities in hyperledger fabric chaincodes. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 570–581.

- Hamdi, A., Fourati, L., and Ayed, S. (2024). Vulnerabilities and attacks assessments in blockchain 1.0, 2.0 and 3.0: tools, analysis and countermeasures. *International Journal of Information Security*, 23(2):713–757.
- Homoliak, I., Venugopalan, S., Reijsbergen, D., Hum, Q., Schumi, R., and Szalachowski, P. (2021). The security reference architecture for blockchains: Toward a standardized model for studying vulnerabilities, threats, and defenses. *IEEE Communications Surveys 'I&' Tutorials*, 23(1):341–390.
- Hyperledger Foundation (2024). *Hyperledger Fabric Documentation*. Linux Foundation.
- Jayadev, V., Moradpoor, N., and Petrovski, A. (2024). Assessing the performance of ethereum and hyperledger fabric under ddos attacks for cyber-physical systems. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ARES '24, New York, NY, USA. Association for Computing Machinery.
- La Salle, A., Kumar, A., Jevtić, P., and Boscovic, D. (2023). Joint modeling of hyperledger fabric and sybil attack: Petri net approach. *Simulation Modelling Practice and Theory*, 122:102674.
- Li, P., Li, S., Ding, M., Yu, J., Zhang, H., Zhou, X., and Li, J. (2022). A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, EASE '22, page 366–374, New York, NY, USA. Association for Computing Machinery.
- Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.
- Vieira, M., Velasco, G., and Carvalho, S. (2023). A decentralized health data repository for remote patient monitoring using blockchain and fhir. In *Anais do VI Workshop em Blockchain: Teoria, Tecnologias e Aplicações*, pages 85–98, Porto Alegre, RS, Brasil. SBC.