**Blockchain: verificarea de smart contracts în Prolog**

Contents

[Contents 1](#_Toc7279804)

[INTRODUCERE 2](#_Toc7279805)

[CAPITOLUL 1. Rețelele Blockchain 3](#_Toc7279806)

[1.1. Clasificarea rețelelor blockchain. 5](#_Toc7279807)

[1.2. Caracteristici criptografice 7](#_Toc7279808)

[*1.2.1. Proprietatea 1: Rezistența la coliziune.* 8](#_Toc7279809)

[*1.2.2. Proprietatea 2: Neinversabilă* 9](#_Toc7279810)

[*1.2.3. Proprietatea 3: Rezistență la coliziuni parțiale* 9](#_Toc7279811)

[1.3. Pointeri hash 10](#_Toc7279812)

[1.4. Arbori Merkle. 11](#_Toc7279813)

[1.5. Semnături digitale 12](#_Toc7279814)

[CAPITOLUL 2. Contracte Smart 13](#_Toc7279815)

[2.1. Vulnerabilități ale contractelor smart în Ethereum. 14](#_Toc7279816)

[*2.1.1. Apeluri reperate ale aceleiași funcții.* 14](#_Toc7279817)

[*2.1.2. Apeluri intersectate între funcții*. 16](#_Toc7279818)

[*2.1.3. Dependența tranzacțiilor de ordinea de execuție.* 17](#_Toc7279819)

[*2.1.4. Dependența tranzacțiilor de un server de timp valid.* 17](#_Toc7279820)

[*2.1.5. Depășirea intervalului pentru numere întregi.* 17](#_Toc7279821)

[*2.1.6. DoS cu retragerea tranzacției*. 18](#_Toc7279822)

[*2.1.7. Atacuri DoS folosind limita de combustibil*. 19](#_Toc7279823)

[*2.1.8. Trimiterea forțată de monede către un contract* 20](#_Toc7279824)

[Bigchaindb 20](#_Toc7279825)

[Concepte generale ale bazei de date 21](#_Toc7279826)

[Componentele unui Asset 22](#_Toc7279827)

[Componentele unei tranzacții 22](#_Toc7279828)

[Caracteristicile BigchainDB 23](#_Toc7279829)

[Elemente criptografice ale BigchainDB 24](#_Toc7279830)

[Tendermint 25](#_Toc7279831)

# INTRODUCERE

# CAPITOLUL 1. Rețelele Blockchain

Tranzacțiile electronice sunt tot mai numeroase ca urmare a dezvoltării comerțului electronic și creșterii nivelului de încredere în instituțiile financiare care garantează validitatea acestor tranzacții. Acest sistem de plăți electronice se bazează în mod tradițional pe o a treia parte, o instituție financiară care servește ca parte terță de încredere (TTP) în procesarea plăților electronice. În timp ce sistemul funcționează destul de bine pentru cele mai multe dintre cazuri, acesta suferă încă de slăbiciunile inerente ale modelului bazat pe o a treia parte.

Într-un sistem clasic de plată electronică, cele două entități între care se face transferul (cumpărărtor și vânzător) trebuie să se bazeze pe o a treia entitate care va garanta transferul. Funcționarea acesteia este supusă erorilor neintenționate saudiferitelor tipuri de atacuri. Existența unui singur punct de eșec face ca aceste autorități să devină ținte foarte atractive pentru atacuri distribuite de blocare a serviciilor (DDoS). (Thesis-F-Sch) Se poate realiza protecția împotriva atacurilor, dar acest lucru înseamnă creșterea costurilor pentru menținerea serviciilor. O altă problemă a acestui sistem de plată este dată de faptul că un consumator trebuie să se înscrie pe fiecare platformă și să furnizeze datele personale proprietarului platformei, iar acest lucru ridică probleme de confidențialitate. De asemenea, metodele posibile de plată sunt limitate de către proprietarul fiecărei platforme. Consumatorii sunt forțați să utilizeze o anumită platformă de plată și uneori trebuie să plătească taxe de tranzacție ridicate.

Într-un articol semnat cu pseudonimul Satoshi Nakamoto în 2009 (bitcoin.pdf) este descrisă moneda electronică Bitcoin și structura sistemului de plată de la egal la egal - peer-to-peer - ca alternativă la sistemul clasic. Motivația acestui articol era de a defini un sistem electronic de plată bazat pe dovezi criptografice în loc de autorități de încredere. Acest sistem permite oricăror două părți dispuse să efectueze tranzacții directe între ele fără a a necesita intervenția unei a treia instituție. (bitcoin.pdf) Pentru evitarea situațiilor de plată repetată cu aceeași monedă către două conturi diferite, în lucrare este descrisă introducerea unui server de timp (Network Time Protocol – NTP) și a aplicării unui algoritm prin care fiecare tranzacție generează o dovadă computațională a ordinii sale cronologice.

Sistemul este sigur atâta timp cât nodurile cinstite controlează în mod colectiv mai multă putere de procesare decât oricare altul grup de noduri care ar putea funcționa eronat sau rău intenționat. Documentul lui Nakamoto conține planul pe care îl urmează în linii mari cele mai moderne scheme pentru generarea și transferul monedelor electronice.

Ideile de bază din spatele tehnologiei blockchain au apărut cu mult înaintea articolului lui Satoshi Nakamoto despre Bitcoin. În 1989, Leslie Lamport a elaborat protocolul Paxos, mecanismul cheie prin carese permite menținerea coerenței datelor stocare într-un sistem distribuit. În anul următor, Lamport a prezentat ideile care au adus contribuții majore în modelarea sistemele distribuite într-un articol intitulat ,,Parlamentul part-time”(lamport-paxos.pdf) în revista ACM Transactions on Computer Systems.

În acest articol, Leslie Lamport descrie un model de comunicare folosit pentru a se ajunge la un acord comun asupra datelor distribuite într-o rețea de calculatoare în care computerele sau rețeaua în sine ar putea fi nesigure. În 1991, Haber și Stornetta au propus prima schemă de aplicare a tehnologiei blockchain în crearea unui registru electronic pentru semnarea digitală a documentelor într-un mod care ar putea arăta cu ușurință că nici unul dintre documentele semnate din colecție nu a fost schimbat (princeton\_bit).

Așa cum sugerează și numele, blockchain poate fi văzut ca un lanț de blocuri de date, o bază de date distribuită de înregistrări sau un lanț de registre publice al tuturor tranzacțiilor care au fost executate și distribuite între părțile participante. [BlockchainPaper.pdf] Blocurile care compun registrele digitale distribuite de tranzacții sunt semnate criptografic. Fiecare bloc este legat de cel precedent, ceea ce conferă imutabilitatea datelor după validare prin decizia majorității. Pe măsură ce se adaugă blocuri noi, blocurile vechi devin mai greu de modificat (creând rezistență la manipulare). Blocurile de date sunt replicate în rețea prin diferite copii registrelor și orice conflict este rezolvat automat folosind reguli stabilite.(NIST.pdf)

Sistemul de obținere al unui consens între participanți este cel definit de Lamport prin protocolul Paxos cu mici diferențe ale etapelor de decizie impuse prin detaliile de implementare. Fiecare tranzacție din registrul public este verificată prin consultarea registrului public obținut prin consensul majorității participanților. Informațiile odată introduse în registrul public, nu pot fi șterse sau modificate. Cea mai largă utilizare a tehnologiei blockchain este pentru înregistrarea tranzacțiilor cu monezi electronice. Tehnologia blockchain poate fi aplicată însă într-o gamă largă de activități (scenarii) atât în lumea financiară, cât și în cea non-financiară. Caracterul consistent și imutabil al datelor poate fi valorificat pentru crearea unor baze de date în care să se stocheze orice tip de informație (bigchaindb), nu doar referitoare la conturi bancare și tranzacții dintre acestea.

Instituțiile financiare și băncile nu văd tehnologia blockchain ca pe o amenințare pentru modelul afacerilor tradiționale. Cele mai mari bănci din lume caută oportunități în acest domeniu, făcând cercetări petru aplicații inovatoare de tip blockchain. Consiliul European de Plăți a publicat în 2016 un anunț în care evaluează că blockchain va crește siguranța și usurința administrării sistemului de plăți, dar prezintă posiblități de dezvoltare și în afara domeniului financiar. (https://www.europeanpaymentscouncil.eu/news-insights/news/epc-poll-how-may-blockchain-technology-impact-european-payments-landscape-2025). Consiliul European intenționează să ofere5 premii intitulate ,,Blockains for Social Good” în valoare de 1 milion de euro pentru dezvolarea de soluții bazate pe tehnologia registrelor distribuite (Distributed Ledger Technology - DLT). Printre domeniile de aplicabilitate propuse se numără:

* *demonstrarea originii materiilor prime sau a produselor pentru asigurarea comerțului echitabil,*
* *monetizarea corectă a forței de muncă permițând o mai mare vizibilitate a cheltuielilor publice și o mai mare transparență a proceselor administrative și de producție,*
* *participarea la luarea deciziilor democratice, prin acordarea de responsabilitate, recompensarea participării și/sau anonimatului care să permită dezvoltarea unor rețele sociale sau a unor platforme descentralizate pentru economia colaborativă,*
* *gestionarea proprietății,*
* *registrul funciar sau alte înregistrări publice.*

(https://ec.europa.eu/research/eic/index.cfm?pg=prizes\_block chains)

## 1.1. Clasificarea rețelelor blockchain.

Rețelele blockchain pot fi clasificate pe baza modelului de permisiuni, care determină cine le poate publica noi blocuri de date. Dacă oricine poate face acest lucru, rețeaua blockain este publică (permissionless). (NIST.pdf) Dacă numai anumiți utilizatori pot publica blocuri, este privată (permissioned). Întru-un articol publicat de NIST, (NIST.IR.8202.pdf) rețelele private sunt comparate cu un intranet al unei corporații care este controlat de o autoritate, în timp ce o rețea publică poate fi văzută ca internetul public, unde oricine poate adăuga date.

***Rețelele blockchain publice*** sunt platforme descentralizate, deschise tuturor celor care publică blocuri, fără a avea nevoie de permisiunea unei autorități. Platformele blockchain publice sunt adesea publicate ca software open source. Cea mai populară platformă din această categorie este Ethereum pe care oricine poate publica blocuri de date și realiza tranzacții. Dezvoltarea de software pentru acest model este încurajată de diverse companii care au construit chiar și un magazin virtual de aplicații (https://market.modex.tech).

Deoarece oricine are dreptul să publice blocuri, acest lucru are ca rezultat faptul că oricine poate citi registrele publice și poate emite tranzacții (prin includerea acelor tranzacții în blocurile publicate).Deoarece rețelele de blocuri publice sunt deschise tuturor pentru a participa prin adăugarea de conținut, utilizatorii rău intenționați pot încerca să publice blocuri într-un mod care să afecteze sistemul (de exemplu prin încercarea de a acapara majoritatea nodurilor). Pentru a preveni acest lucru, rețelele blockchain publice utilizează adesea un acord multipartit sau un sistem de "consens" prin care se cere utilizatorilor să utilizeze sau să mențină resurse computaționale atunci când aceștia încearcă să publice blocuri. Acest lucru împiedică utilizatorii rău intenționați să perturbe ușor sistemul. Exemple de astfel de modele de consens includ introducerea unor dovezi ale efortului (Proof of Work) și dovezi ale mizei (Proof of Stake). (NIST.pdf)

Într-o rețea blockchain publică, numărul de noduri este de așteptat să fie mare, iar faptul că aceste noduri sunt anonime scade nivelul de încredere în acestea, deoarece oricărui nod îi este permis să se alăture rețelei. Mecanismele de consens pentru o astfel de configurare trebuie să ia în considerare comportamentele rău intenționate ca de exemplu în special atacurile Sybil. Acest tip de atacuri constau în faptul că o astfel de rețea blockchain permite unui singur utilizator să își genereze mai multe identități pentru a influența și a manipula procesul de obținere a unui consens. Bitcoin rezolvă această problemă prin conceperea rundelor de obținere a consensului astfel încăt acestea să fie computațional dificile.

Nodurile trebuie să dovedească că ele au consumat o cantitate semnificativă de energie ca dovată a muncii (Proof-of-Work - PoW) spre rezolvarea unei probleme criptografice dificile. Primele platforme blockchain libere au fost proiectate astfel încât să utilizeze direct PoW ca mecanism de obținere al unui consens, de exemplu Bitcoin, sau au folosit variații ale acestui mecanism, de exemplu: NameCoin, LiteCoin, DogeCoin și Monero. (369492812.pdf)

***Rețelele blockchain private*** sunt cele în care publicarea blocurilor de către utilizatori trebuie validată de o anumită autoritate (centralizată sau descentralizată). Întrucât numai utilizatorii autorizați mențin registrele publice, este posibilă restricționarea accesului la citire și emiterea de tranzacții. Rețelele blockchain private pot fi instanțiate și întreținute fie prin intermediul unor componente software de tip ***open source*** sau proprietare.

Acest tip de rețele folosesc aceleași mecanisme de trasabilitate a tranzacțiilor înregistrare pe măsură ce sunt adăugate lanțului de blocuri, precum și același sistem de stocare a datelor distribuit, rezilient și redundant ca și rețele blockchain libere. De asemenea, ele utilizează modele de obținere a unui consens pentru publicarea unor noi blocuri, însă aceste metode nu necesită cheltuieli pentru întreținerea resurselor (cum este cazul rețelelor blockhain libere). Acest lucru se datorează faptului că este necesară stabilirea unei identități pentru a fi numit membru al rețelei blockchain private; cei care mențin registrele au un nivel de încredere unul cu celălalt, deoarece toți sunt autorizați să publice blocuri și deoarece autorizația lor poate fi revocată dacă acțiunile lor sunt eronate. Modelele de consens în rețelele blockchain private sunt de obicei mai rapide și necesită mai puține resurse computaționale.

Totuși, dacă o singură entitate controlează cine poate publica blocuri, utilizatorii blockchain vor trebui să aibă încredere în acea entitate. Rețelele bockchain private pot fi, de asemenea, folosite de organizații care doresc să lucreze împreună, dar nu pot avea încredere completă între ele. Acestea pot stabili o rețea blcokchain autorizată și pot invita partenerii de afaceri să-și înregistreze tranzacțiile intr-un registru distribuit partajat. În funcție de nivelul de încredere reciprocă, aceste organizații pot determina modelul de obținere al consensului.

Unele rețele blockchain private permit posibilitatea de a dezvălui selectiv informațiile despre tranzacții pe baza identității sau a acreditărilor utilizatorilor. Aceasă caracteristică oferă un anumit grad de confidențialitate al tranzacțiilor. De exemplu, în registrul public este adăugată o înregistrare a unei tranzacții între doi utilizatori, dar conținutul real al tranzacției este accesibil doar părților implicate. (NIST.pdf)

Platformele blockchain private sunt de așteptat să fie mai reduse și, prin urmare, să utilizeze mecanisme alternative de obținere a unui consens. Realizarea unui consens într-un sistem distribuit are diverse soluții cunoscute în literatura de specialitate, de exemplu algoritmul propus de Laport, Paxos, RAFT sau alți algoritmi care asigură toleranța la eșecurile bizantine în care rețeaua este considerată nesigură, mesajele dintre noduri pot fi modificate sau pierdute și orice nod poate eșua în orice moment.(369492812.pdf)

## 1.2. Caracteristici criptografice

Orice sistem de plăți are nevoie de o modalitate de a controla furnizarea monedelor și de a impune diverse reguli de securitate pentru a preveni infracțiunile. În cazul sistemelor tradiționale, băncile sunt cele care controlează rezervele naționale de bani și adaugă detalii monedelor și bancnotelor astfel încât să nu poată fi contrafăcute. Aceste detalii fac dificilă falsificarea banilor, însă nu și imposibilă. În ultimă instanță, aplicarea legii este necesară pentru oprirea oamenilor de a încălca regulile sistemului.

De asemenea, monedele virtuale trebuie să aibă măsuri de securitate care împiedică oamenii să manipuleze sistemului. Dar spre deosebire de monedele obișnuite, regulile de securitate pentru criptarea lichidităților trebuie să fie puse în aplicare exclusiv din punct de vedere tehnologic fără ajutorul unei autorități centrale.

Bitcoin se bazează pe o submulțime de construcții criptografice relativ simple și bine cunoscute (funcții hash și semnături electronice). O funcție hash este o funcție matematică care se caracterizează prin trei proprietăți:

* *Poate primi ca parametru un șir de orice dimensiune.*
* *Produce un șir de dimensiune fixă.*
* *Este eficient de calculat.*

Aceste proprietăți definesc o funcție hash în sensul general. Pentru ca o funcție hash să fie sigură din punct de vedere criptografic, aceasta trebuie să mai prezinte încă trei proprietăți:

* *(1) rezistență la coliziuni,*
* *(2) neinversabilă,*
* *(3) rezistență la coliziuni parțiale. (princeton)*

### *1.2.1. Proprietatea 1: Rezistența la coliziune.*

O coliziuneare are loc atunci când pentru două șiruri diferite, funcția hash produce același rezultat. O funcție hash H(.) este rezistentă la coliziuni dacă este dificil de găsit o coliziune, adică pentru două valori x și y, având x≠y, nu putem găsi H(x)=H(y). Coliziunile există prin faptul că spațiul infinit al parametrilor de intrare este proiectat prin acestă funcție în spațiul finit a valorilor de ieșire. Putem spune că o funcție este rezistentă la coliziuni atunci când acestea sunt dificil de găsit.

Totuși, există metode care garantează obținerea unei coliziuni. Cel mai simplu exemplu, dar costisitor din punct de vedere computațional, este să luăm pentru o funcție hash care returnează un rezultat pe 256 de biți, 2^256+1 valori diferite de intrare. În acest caz epuizăm toate variantele diferite de ieșire și cu siguranță cel puțin o valoare va fi comună pentru două intrări. De fapt, dacă vom alege intrări aleatorii și vom calcula rezultatul funcției hash pentru acestea, vom găsi o coliziune cu probabilitate mare cu mult înainte de a examina 2^256+1 de intrări. Mai exact, dacă alegem aleatoriu doar 2^130+1 intrări, vom avea o șansă de 99,8% ca cel puțin două dintre valorile de intrare să aibă o coliziune. Faptul că putem găsi o coliziune doar examinând aproximativ radical din numărul de ieșiri posibile rezultă dintr-un fenomen cunoscut sub numele de paradoxul zilei de nastere.(curs cripto + princepton.pdf) În practică, acest lucru este imposibil deocamdată, deoarece chiar dacă un calculator ar putea obține 10.000 de rezultate hash pe secundă, ar trebui 10^54 de ani ca acesta să calculeze toate valorile hash posibile. Chiar dacă această metodă se dovedește a fi impracticabilă, nu înseamnă că pentru anumite funcții hash nu pot fi obținute coliziuni prin calcul matematic. De exemplu, funcția ***message-digest-5 (MD5)*** care a fost considerată pentru o lungă perioadă rezistentă la coliziuni, până când o astfel de metodă a fost găsită.

### *1.2.2. Proprietatea 2: Neinversabilă*

A doua proprietate care definește funcțiile hash este aceea că acestea trebuie să nu fie inversabile. Acest lucru implică și faptul că dacă obținem rezultatul hash al unor date y=H(x), nu există o metodă prin care să putem determina datele de intrare. Mai mult decât atât, trebuie să fie imposibilă obținerea unor informații referitoare la parametrii. Acest lucru se obține concatenând la șirul de intrare o valoare aleatoare de distribuție uniformă numită ***”nonce”***. Odată cu datele trimise, se va trimite și această valoare pentru ca funcția hash să returneze același rezultat la verificarea datelor.

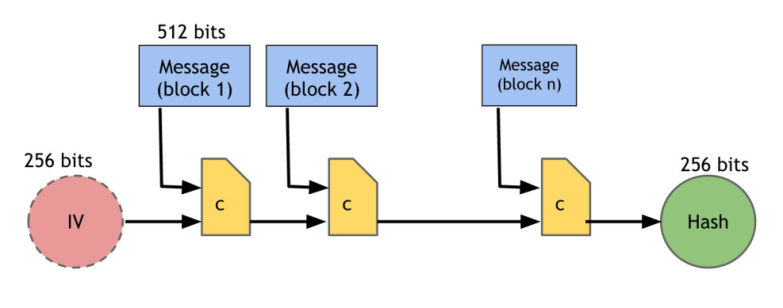
### *1.2.3. Proprietatea 3: Rezistență la coliziuni parțiale*

O funcție hash trebuie să fie formulată astfel încât pentru toate valorile de ieșire posibile de lungime n, dacă se alege o valoare k aleator, este imposibil de găsit o altă valoare x în mai puțin de 2^n pași, astfel încât funcția hash aplicată concatenării celor două valori să returneze y, H(k||x)=y.

Această problemă este folosită și pentru crearea acelor dovezi ale muncii (PoW Proof of Work). Pentru un șir primit (în mod obișnuit acesta este hashul tranzacției precedente), un miner trebuie să gasească o valoare aleatoare care, concatenată la șirul primit să rezulte un element al cărui rezultat hash respectă anumite condiții. Pentru acest tip de problemă nu există o altă metodă de rezolvare decât simpla căutare a spațiului de soluții.

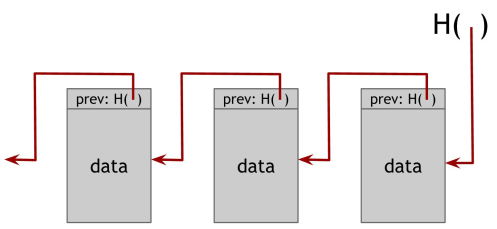
Cea mai folosită funcție hash care înglobează aceste caracteristici și este folosită în modelul Bitcoin este funcția SHA-256. În construirea acestei funcții, s-a început prin a defini o funcție hash care respecă aceste condiții în cazul particular al unui șir de lungime fixă. Trecerea de la o funcție pe șiruri de lungime fixă la una care funcționează pe șiruri de orice lungime s-a realizat prn transformarea Merkle-Damgard. SHA-256 este una dintre numeroasele funcții hash care utilizează această metodă. Funcția hash este construită prin adăugarea unor transformări peste o funcție de bază numită funcție de compresie. Aceasta este cea care are toate cele trei proprietăți și, prin urmare, modificările asupra rezultatului acesteia păstrează proprietățile.

Transformarea Merkle-Damgard. Presupunând că funcția de compresie are intrări de lungime ***m*** și produce o ieșire de o lungime mai mică ***n***, trebuie să găsim o metodă prin care să aplicăm funcția hash pe șiruri de orice dimensiune. Metoda presupune împărțirea șirului de intrare de lungime aleatoare în blocuri de lungime ***m-n***, apoi fiecare bloc este concatenat cu valoarea de ieșire a blocului anterior și este folosit ca paramentru pentru funcția de compresie. Lungimea șirului de intrare va fi în acest caz ***(m-n)+n=m***, care este lungimea parametrului de intrare pentru funcția de compresie. Astfel primul bloc, pentru care nu există nici o valoare hash anterioară, folosește un vector de inițializare (IV). Acest vector este reutilizat pentru fiecare apel al funcției hash și în general este menționat în standardul care definește funcția hash. SHA-256 utilizează o funcție de compresie care are o intrare de 768 de biți și produce ieșiri de 256 biți. Dimensiunea blocului este de 512 biți.

 [todo sursa imaginii sau de refacut la nivelul implementarii mele]

## 1.3. Pointeri hash

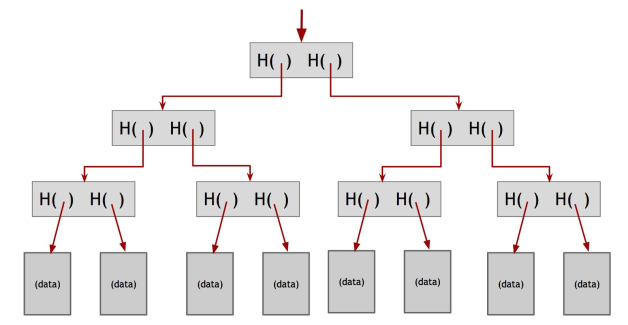
Un pointer hash este un pointer care indică locația unei structuri de date compusă din datele propriu-zise și rezumatul hash al acestora. Diferența față de pointerii obișnuiți constă în faptul că oferă nu doar calea de a obține o informație, ci și o cale de a verifica acea informație. Tehnologia blockchain se bazează pe acest tip de pointeri construind o listă înlănțuită în care fiecare nod are atașat un hash pointer către nodul precedent. La finalul listei din blockchain se află un hash pointer către ultimul nod adăugat și valoarea hash a acestuia.



Din modalitatea de construire a acestei liste rezultă caracterul imutabil al datelor din blockchain. De exemplu, dacă cineva modifică o parte din datele care au fost adăugate și înregistrate, această modificare va altera rezultatul hash a blocului curent. Modificarea hash-ului unui bloc duce la modificarea tuturor blocurilor următoare pentru că noile valori trebuie să ia în consideare și hashul precedent. Chiar și în cazul în care se reușește recalcularea tuturor blocurilor, ultimul hash din listă va trebui și el modificat, iar acest lucru indică foarte clar că lista este incorectă.

## 1.4. Arbori Merkle.

O altă structură de date pe care o putem crea folosind pointeri hash este arborele binar. Un arbore binar construit cu pointeri hash se numește arbore Merkle după numele inventatorului său Ralph Merkle. Frunzele arborelui sunt populate cu date, fiecare grupare de două frunze este unită printr-un nod care conține doi pointeri hash pentru fiecare dintre cele două frunze asociate. Nodurile formte din perechi de pointeri hash sunt mai departe reunite prin alte noduri similare care indică locața poinerilor de sub aceștia. Nodul rădăcină va avea același rol ca în cazul pointerului hash de la finalul listei. Dacă se execută modificări asupra datelor din frunze, acestea vor fi oglindite până la nivelul rădăcinii arborelui.

[todo sursa imaginii sau de refacut la nivelul implementarii mele]

O altă proprietate foarte utilă a arborilor Merkle, este aceea că, față de lista cu pointeri hash, putem determina în ***log(n)*** pași dacă un bloc de date aparține unui arbore Merkle cu ***n*** noduri. În cazul listelor, este necesară parcurgerea întregii structuri în cel mai nefavorabil caz. Pentru rețele blockchain de dimensiuni mari, această diferență de timpi este foarte importantă deoarece dorim să verificăm dacă un bloc de date este valid (se află înregistrat în structură) într-un timp cât mai scurt.

Pentru a demonstra însă că un bloc nu se află îtr-o astfel de structură, este necesar să parcurgem toate elementele și să observăm fiecare bloc de date în parte. Acestă verificare poate fi realizată în timp logaritmic dacă arborele Merkle creat de la început are frunzele sortate după o caracteristică ce ne-ar facilita căutarea. În acest caz, este nevoie doar să parcurgem drumul până la două frunze învecinate între care ar fi trebui să se găsească blocul de date căutat.

Pointerii hash pot fi folosiți pentru a construi orice strucură de date obișnuită. Excepție fac structurile în care apar cicluri pentru că, în acest caz, funcția hash se va calcula la infinit pentru blocurile de date legate.

## 1.5. Semnături digitale

O semnătură digitală se dorește a fi analogul unei semnături scrise: trebuie să poată fi realizată doar de persoana căreia îi aparține, oricine o vede trebuie să poată recunoaște că aparține acestei persoane și nimeni nu o poate extrage dintr-un document semnat pentru a o aplica altui document. Semnăturile digitale se construiec folosind perechi de chei publice și private. Cheia publică este disponibilă oricui dorește să verifice autorul documentului semnat, iar cheia privată este cunoscută doar de posesorul ei pentru a o folosi în realizarea semnăturii.

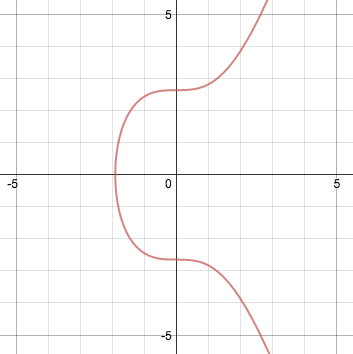
Cheile sunt generate aleator, dar procesul de semnare trebuie să asigure câteva condiții. În primul rând, cheia publică trebuie să permită identificarea doar documentelor semnate cu perechea ei privată. Acest lucru se realizează prin asigurarea unicității cheilor și a injectivității funcției de semnare. Două chei private diferite vor genera întotdeauna două semnături diferite. În al doilea rând, trebuie ca cineva care deține un număr mare de documente semnate dde către aceeași persoană să nu poată calcula cheia privată a acesteia.

Un test cunoscut pentru asigurarea acestei condiții este de a permite atacatorului să trimită un număr rezonabil de documente alese de el spre a fi semnate. Odată ce atacatorul a obținut semnătura victimei pe suficiente documente, acesta va încerca să creeze un mesaj pe care să îl semneze cu o cheie privată pe care încearcă să o ghicească. Acesta va câștiga dacă reușește să trimită către victimă mesajul semnat cu cheia ghicită, iar aceasta să verifice semnatura mesajului creat cu cheia sa publică.

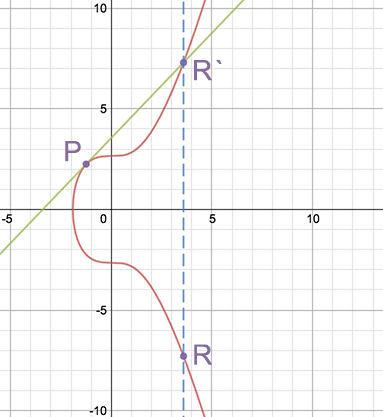
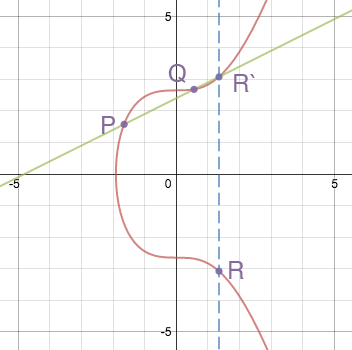
În practică, este mult mai eficient să semnăm rezumatul hash al unul bloc de date în locul datelor în sine, deoarece funcția de semnătură digitală poate presupune ca intări șiruri de lungime fixă. De fapt, pentru a indica faptul că atât datele cât și rezumatul lor hash au fost originate de noi, se poate aplica semnătura digitală direct pe pointerul hash al acestora.

Schema de semnătură digitală folosită în modelul Bitcoin se numește Algoritmul de Semnătură Digitală pe Curbe Eliptice (Elliptic Curve Digital Sgnature Algorithm – ECDSA). Algoritmul este derivat din schema de criptare Digital Signature Algrthm – DSA, dar aplicat folosind curbe eliptice. Același algoritm este folosit și în schimbul de chei în navigarea securizată pe internet, singura diferență constă în parametrii aleși de standard pentru a descrie ecuația curbelor eliptice.

ECDSA include doi algoritmi separați pentru procedeul de semnare și cel de cirificare a semnăturii. În semnarea unor date se folosește cheia privată iar pentru verificarea semnăturii se face apel la perechea acesteia (cheia publică). În primul rând se alege o curbă eliptică reprezentată printr-o ecuație de forma: . De exemplu, în cazul Bitcoin, și iar curba eliptică este de forma: . Reprezentarea ei grafică este de forma:



Proprietățile specifice acestor curbe constă în faptul că orice linie cu excepția celor verticale (cu panta infinit) care intersectează două puncte non-tangente de pe curbă va intersecta întotdeauna al treilea punct de pe aceasta. De asemenea, orice tangentă non-verticală a curbei va intersecta mereu un alt punct de pe curbă. Aceste proprietăți sunt folosite pentu a defini două operații aplicate punctelor de pe curbă. Prima operație este adunarea a două puncte, fie P și Q de pe crubă. Operația folosește prima proprietate pentru a determina simetricul celui de al treilea punct de intersecție față de axa Ox. În exemplul de mai jos, dată fiind dreapta care intersectează punctele P și Q, se determină punctul R și rezultatul adunării este simetricul acestuia, R'.



A doua operație, dublarea unui punct (P+P) se calculează prin determinarea tangentei care trece prin acel punct și apoi a celui de al doilea punct de intersecție R'. Rezultatul dublării este simetricul acestui punct față de axa Ox. Prin aceste două operații pot fi realizate calcule mai complexe pe curba eliptică, de exemplu multiplicarea unui punct de un număr de ori, realizat prin dublări repetate.

În ECDSA, curbele elipice sunt utilizate pe domenii finite (numere întregi modulo un număr dat) care nu schimbă propritetățile acestora doar felul cum sunt reprezentate. În cazul monedelor electronice, sistemul de semnătură precizează un set de parametrii care sunt folosiți de toți cei care apelează la acea monedă. Acești parametrii includ domeniul finit, de obicei definite prin câmpuri Galois date printr-un număr prim mare, domenii în care vor fi realizate calculele. Un alt parametru important este ordinul cheii, un număr suficient de mare pentru a lăsa un inerval întins de alegere a cheii private. Aceasta este aleasă în intervalul 1 și ordinul dat. Cheia publică este calculată prin multiplicarea pe cruba eliptică definită a cheii private cu un punct de bază ales ca parametru.

Principala problemă a plăților electronice este problema cheltuielilor duble. În sistemul de plăți tradițional prin internet, banca acționează ca autoritate centrală pentru a ține evidența banilor deținuți de clienți și pentru a verifica dacă o tranzacție este validă. Acest lucru asigură implicit și faptul că o persoană poate cheltui o sumă de bani doar o singură dată. Pentru a păstra conceptul de portofel electronic, chema propusă de Nakamoto în 2008 definește identitatea unui client și a portofelului electronic asociat prin două chei criptografice: o cheie publică și o cheie privată.

Cheia publică poate fi văzută ca număr de cont bancar al clientului și este necesară pentru alți membri care doresc să îi transfere bani. Cheia privată poate fi văzută ca parola clientului folosită pentru semnarea tranzacțiilor. În modelul lui Nakamoto, dacă o persoană dorește să efectueze o tranzacție, semnează tranzacția cu cheia sa privată și o trimite spre aprobare serverului de timp. La nivelul serverului, un bloc de date este compus din mai multe tranzacții. Pentru toate blocurile sunt calculate amprentele hash. Fiecare bloc nou conține amprenta has a tuturor blocurilor anterioare. Acest lucru menține un instoric al tuturor blocurilor și, totodată asigură că toate trazacțiile precedente sunt valide. Datorită caracteristicilor funcțiilor criptografice, acest istoric nu poate fi modificat. (MergedFile.pdf)

# CAPITOLUL 2. Contracte Smart

Un contract smart este un protocol digital care facilitează procesul de înțelegere asupra condițiilor unui contract între diferitele părți prin aplicarea anumitor reguli. Conceptul de contract smart a fost introdus inițial de Nick Szabo în 1997 [??]. În articolul său propune includerea unor clauze contractuale precum drepturile de proprietate direct în hard și software pentru a crește dificultatea ca părțile între care s-a încheiat contractul să încalce regulile stabilite.

El prezintă un sistem digital de securitate ipotetic pentru transferul mașnilor, în cazul în care o cheie criptografică reprezintă proprietatea asupra unei mașini și este transferată numai dacă termenii implementați în protocol sunt respectați. În exemplul său, proprietarul unei mașini poate retrage cheia de la un utilizator al mașinii care nu plătește chiria lunară. În acel moment, contractele inteligente nu erau practic realizabile, pentru că nu exista nici o infrastructură digitală care să permită executarea sigură a protocoalelor fără nevoia unui TTP la un moment dat. Odată cu introducerea monedei virtuale Bitcoin și a infrastructurii acesteia, s-au putut implementa și contractele smart pe același suport.

Deși funcționalitatea Bitcoin poate fi extinsă și dincolo de simplul transfer de monede, limbajul de scripting folosit pentru Bitcoin nu dispune de toate mijloacele necesare pentru a exprima logica complexă a contractelor smart ca în exemplul imaginat de Nick Szabo. Ethereum [??], o monedă virtuală care a fost descrisă inițial la sfârșitul anului 2013 de Vitalik Buterin [??27] și a fost utilizată public în anul 2015, este prima monedă a cărei infrasctructră poate permite implementarea tuturor caracteristicilor contractelor smart.

În rețeaua Ethereum, contract smart reprezintă o instanță a unui program care rulează prin intermediul aplicației blockchain, adică este executat de către toate nodurile care au format un consens. Un contract smart este format din codul programului, un fișier de stocare, și datele unui cont. Orice utilizator poate crea un contract prin publicarea unei tranzacții în blockchain. Codul dintr-un contract este interpretat atunci când contractul este creat și nu poate fi schimbat.

Logica programului care a fost generat dintr-un contract este executată de rețeaua de mineri care ajunge la un consens cu privire la rezultatul execuției acestuia și la actualizarea rețelei blockchain. Codul contractului este executat de către un nod din rețea de fiecare dată când acesta primește un mesaj, fie de la un utilizator, fie de la un alt contract. În timpul executării codului său, programul contractului poate include comenzi de citire sau scriere din fișierul său de stocare. Un contract poate să includă execuția unor tranzacții pentru a primi bani în contul ale cărui date sunt stocate sau trimite bani din acesta către conturile altor contracte sau utilizatori. Un contract smart poate fi văzut ca o parte terță de încredere care spre deosebire de părțile terțe din modelul tradițional,nu va păstra și confidențialitatea conturilor prelucrate și a operațiilor efectuate pe acestea, deoarece contractele sunt accesibile public pentru citire.

***Apelarea contractelor.***Codul unui contract va fi invocat ori de câte ori se primește o tranzacție de la un utilizator. Un contract poate avea mai multe puncte de execuție din care se poate începe rularea unor bucăți de cod diferite. Conținutul unei tranzacții va specifica punctul de intrare pe care îl va invoca în codul contractului. Prin urmare, tranzacțiile se comportă ca apelurile unor funcții în mod obișnuit în limbajele de programare. După încheierea execuției unui contract, acesta poate returna o valoare către utilizatorul care a generat apelul contractului.(2015-460.pdf)

## 2.1. Vulnerabilități ale contractelor smart în Ethereum.

*2.1.1. Apeluri reperate ale aceleiași funcții.* Această categorie de vulnerabilități se adresează situațiilor în care funcțiile sunt scrise plecând de la presupunerea că sunt executate secvențial fără a permite suprapunerea altor execuții în timpul rulării. Prima metodă observată exploatare a acestei vulnerabilități a fost prin simpla reapelare a unei astfel de funcții înainte de a se fi terminat. Acest lucru duce la execuția simultană a două instanțe ale aceleiași funcții care poate genera probleme destul de grave. Un exemplu ilustrativ [??] este funcția de mai jos prin care se dorește lichidarea contului prin retragerea tuturor monedelor din acesta și setarea balanței la 0.

*mapping(address=>uint)privateconturiUtilizatori;*

*functionretragere()public{*

*uintsumaDeRetras=conturiUtilizatori[msg.sender];*

*require(msg.sender.call.value(sumaDeRetras)());//Reapelarea pentru retragerea repetată înainte de setarea balanței la 0*

*conturiUtilizatori[msg.sender]=0;*

*}*

Setarea contului după apelul funcției de retragere permite unui atacator să apeleze de mai multe ori aceeași funcție și să facă retrageri succesive dintr-un cont a cărui balanță nu a fost încă actualizată. Acest tip de atac poate fi ușor evitat prin simpla reorganizare a liniilor de cod și setarea balanței înainte de apelul funcției externe call.value()():

*mapping(address=>uint)privateconturiUtilizatori;*

*functionretragere()public{*

*uintsumaDeRetras=conturiUtilizatori[msg.sender];*

*conturiUtilizatori[msg.sender]=0;*

*require(msg.sender.call.value(sumaDeRetras)());*

*}*

Există situații în care este necesar ca funcția de mai sus să fie apelată înaintea altor operații sau funcții, caz în care suntem nevoiți să apelăm la alte mijloace de a ne asigura că fluxul de lucru nu este întrerupt prin alte apeluri. Strict pentru exemplul de mai sus se poate înlocui funcția externă ***call.value()()*** cu funcția ***send()*** care nu permite execuția unui cod din afara programului să intervină peste funcția curentă. Problema apare însă în cazul în care dorim să apelăm funcția ***retragere()*** dintr-o altă funcție, de exemplu ***operațiune(),*** și trebuie să avem în vedere că modificările realizare de funcția ***retragere()*** să nu poată fi afectată de reapelarea fucnției ***operațiune()***.

( <https://consensys.github.io/smart-contract-best-practices/known_attacks/)>

Un astfel de atac a fost realizat în 2016 asupra unui contract inclus în structura unei Organizații Descentralizate (DAO). În contractul de bază există o funcție apelată în cazul în care se dorește cumpărarea unei părți din organizație. Ca în orice tranzacție obișnuită, cumpărătorul primește o parte din aceasta. Partea din contract care avea rolul de transfer al sumei de la Organizație către cumpărător realiza întâi retragerea și apoi actualizarea balanței cumpărătorului, fapt care a permis unui atacator să retragă prin apeluri repetate 150 de milioane de dolari din acea organizație. (http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/ )

*2.1.2. Apeluri intersectate între funcții*. Un atac asemănător apelului repetat se poate realiza prin rularea simultană a două funcții care modifică aceeași stare a sisemului. De exemplu, dacă pe lângă funcția de lichidare a contului, mai avem o funcție de transfer care este apelată în timp ce se face lichidarea contului, dar înainte de a fi setată balanța la 0, ajungem să putem obține în contul *destinatar* o sumă transferată dintr-un cont nul.

*mapping(address=>uint)privateconturiUtilizatori;*

*functiontransfer(addressdestinatar,uintsuma){*

*if(userBalances[msg.sender]>=suma){*

*userBalances[destinatar]+=suma;*

*userBalances[msg.sender]-=suma;}*

*}*

*functionretragere()public{*

*uintsumaDeRetras=conturiUtilizatori[msg.sender];*

*require(msg.sender.call.value(sumaDeRetras)());//Apelul funcției transfer*

*conturiUtilizatori[msg.sender]=0;}*

Atacurile de acest tip se pot realiza chiar dacă funcțiile nu sunt în acleași contract, deci mijloacele de protecție pentru organizarea apelurilor din interiorul unei funcții nu sunt suficiente să asigure protecția în cazul aplicațiilor mari sau ale contractelor interdependente. În aceste cazuri singura metodă sigură de păstrare a ordinii de execuție a programului este utilizarea variabilelor mutex. În Solidity, acest lucru se poate realiza cu o variabilă booleană a cărei valoare este schimbată înainte și după intrarea în orice fel de zonă critică.

*mapping(address=>uint)privateconturiUtilizatori;*

*boolprivatelockProcedura;*

*functionretragere()public{*

*require(!lockProcedura);*

*lockProcedura = true;*

*uintsumaDeRetras=conturiUtilizatori[msg.sender];*

*require(msg.sender.call.value(sumaDeRetras)());//Apelul funcției transfer*

*conturiUtilizatori[msg.sender]=0;*

*lockProcedura=false;*

*}*

Și această soluție are dezavantajele sale deoarece programatorii trebuie să aibă în vedere că pentru fiecare blocare a variabilei, programul trebuie să ajungă la linia de deblocare. De asemenea, în cazul funcțiilor și contractelor interpedendente, utillizarea variabilelor mutex poate duce la deadlockuri și livelockuri. [Todo apelul contractelor in Prolog ]

*2.1.3. Dependența tranzacțiilor de ordinea de execuție.* Înainte de a fi inclusă într-un bloc, o tranzacție este stocată în coada de aștepare pentru o perioadă. În acest interval, cineva poate ști ce operații au fost incluse în tranzacție și să intervină cu alte acțiuni care ar modifica starea tranzacției. Acest tip de atac este specific licitațiilor în care cineva s-ar putea folosi de intervalul în care o tranzacție de cumpărare nu este încă validată ca să realizeze acea tranzacție în numele altui beneficiar. Pentru protecție în fața acestui tip de atacuri s-au realizat licitații în bloc astfel încât să scadă numărul trazacțiilor mici și frecvente care s-au putea intersecta.

*2.1.4. Dependența tranzacțiilor de un server de timp valid.* Un alt tip de vulnerabilitate inerent acestui sistem este faptul că minerii pot modifica ștampila de timp a blocului de tranzacții ce îl vor adăuga în rețea. În cazul aplicațiilor de tip loterie sau licitație, o varianție de căteva secunde poate schima ordinea a două tranzacții. Sunt cunoscute de asemenea, cazuri de folosiea a ștampilei de timp pentru generarea numerelor aleatoare, iar manipularea secvenței de timp poate duce la aflarea rezultatului aleator. ([https://consensys.github.io/smart-contract-best-practices/ recommendations/#timestamp-dependence](https://consensys.github.io/smart-contract-best-practices/%20recommendations/#timestamp-dependence))

În referința Ethereum (https://ethereum.github.io/yellowpaper/paper.pdf ) se specifică faptul că secvența de timp a fiecărei tranzacții trebuie să fie mai mare decât cea precedentă. Cu alte cuvinte, tranzacțiile trebuie să se succeadă cronologic. Mai mult decât atât, în câteva implementări ale Ethereum, (Geth și Parity) momentul unei tranzacții trebuie să fie după cel precedent, dar nu cu mai mult de 15 secunde în viitor.

*2.1.5. Depășirea intervalului pentru numere întregi.* Această problemă poate fi tratată diferit în funcție de limbajul de programare. Cel mai comun comportament este că dacă un tip de date trece peste limita de adresare să fie trunchiat la aceasta. În Solidity există chiar și o listă a acestor situații care pot duce la rezultate eronate. (https://github.com/ethereum/solidity/issues/796#issuecomment-253578925)

*mapping(address=>uint256)publicconturi;*

*functiontransfer(addressdestinatar,uint256suma){*

*require(conturi[msg.sender]>=suma);*

*conturi[msg.sender]-=suma;*

*conturi[destinatar]+=suma; }*

Funcția de transfer verifică dacă sursa transferului are în cont suma ce va fi transferată, o scade din contul acesteia și o adună în contul destinatarului. Deși funcția folosește cel mai mare tip de date întreg adresabil în Solidity, este totuși limitat la 2^256 și riscă, dacă în contul destinatar exista deja o sumă suficient de mare, să depășească această valoare și tranzacția să se finalizeze având contul mai mic după adunarea cu suma. O variantă corectă ar fi verificarea preliminară a acestor cazuri adăugând o simplă condiție înainte de efectuarea tranzacției.

*functiontransfer(addressdestinatar,uint256suma){*

*require(conturi[msg.sender]>=suma&&conturi[destinatar]+suma>=conturi[destinatar]);*

*conturi[msg.sender]-=suma;*

*conturi[destinatar]+=suma;}*

Un exemplu care poate avea efecte mult mai ample asupra întregului contract este publicat de Doug Hoyte în 2017. Acesta arată faptul că implementarea unei liste de elemente și a metodelor de extragere și adăugare trebuie să aibă în vedere că membrul ***length*** în Solidity este implementat ca tip de date întreg fără semn. Înainte de extragere dacă se verifică lungimea listei astfel: require (lista.length >= 0); condiția va fi mereu adevărată. Acest lucru permite accesarea datelor nu doar din listă, ci și din alte zone de memorie.

Hoyte a demonstrat cum poate modifica o variabilă privată accesând prin extrageri repetate zona de memorie în care era stocată acea variabilă și apoi modificând-o ca și cum ar fi un element din listă. Deși au fost aduse modificări în Solidity care să indice cănd astfel de situații pot exista și să atragă atenția programatorilor, variabilele statice pot fi totuși adresate direct permițând modificări nedorite asupra acestora.

(https://github.com/Arachnid/uscc/tree/master/submissions-2017/ doughoyte)

*2.1.6. DoS cu retragerea tranzacției*.În cazul unei licitații, dacă un atacator folosește un contract care execută anumite operațiuni în caz de eșec. Blocarea acestor acțiuni face ca sistemul să fie blocat în starea curentă. De exemplu, dacă folosim funcția de ofertare de mai jos și suntem în rolul de atacatori este suficient să obținem la un moment de timp rolul de lider și să ne asigurăm că suma ofertată de noi nu poate fi returnată. Dacă cineva face o ofertă mai mare, valoarea va depăși maxOferta, dar rambursarea sumei oferite de liderul curent nu se poate realiza, deci nici schimbarea liderului și a sumei maxime nu vor mai putea fi realizate prin asignările ulterioare.

*contractAuction{*

*addresslider;*

*uinthighestBid;*

*functionoferta()payable{*

*require(msg.value>maxOferta);*

*require(lider.send(maxOferta));*

*lider =msg.sender;*

*maxOferta =msg.value;}}*

*2.1.7. Atacuri DoS folosind limita de combustibil*. Fiecare block de tranzacții are o limită de combustibil ce poate fi utilizată pentru realizarea operațiilor. Acest lucru a fost realizat pentru menținerea unor tranzacții atomice și limitarea volumului de calcul ce poate fi efectuat într-un contract. Dacă operațiile solicitate depășesc limita de combusitbil a blocului, tranzacțiile vor eșua. De exemplu, în cazul în care un contract presupune rambursarea a mai mult de o entitate în caz de eșec, numărul de tranzacții efectuate în acel contract poate depăși această limită. Dacă un atacator cunoaște limita unui contract și volumul aproximativ de operații efectuate, poate interpune alte trazacții (de exemplu, poate adăuga destinatari pentru a fi rambursați) astfel încât limita să fie depășită de fiecare dată și contractul să nu se mai finalizeze cu succes niciodată.

O metodă destul de directă de rezolvare a acestei situații este ca în cazul în care se iterează printr-un șir de tranzacții să se verifice de fiecare dată dacă mai există suficient combustibil pentru a continua.

Dar există situații în care atacatorul nu încearcă să adauge un număr mare de tranzacții, ci doar să crească complexitatea computațională a acestora astfel încât combusitbilul să nu fie suficient. Dacă atacatorul reușește să genereze tranzacții destul de dificile, vor fi destul de mari astfel încât să nu mai permită niciunei alte tranzacții să fie adăugate în același bloc. Acest lucru poate permite atacatorului să antedateze anumite acțiuni blocând tranzacțiile legitime până după ce acțiunile sale au fost adăugate în blockchain.

Un astfel de atac a fost realizat asupra unei aplicații de jocuri de noroc Fomo3D. Ultimul cumpărător al unei ,,chei” ar fi primit premiul, dar un atacator a cumpărat o astfel de cheie după care a adăugat în rețea tranzacții numeroase care consumau mult combustibil și nu au mai permis niciunei tranzacții de cumpărare de cheie să mai fie executată. Acest tip de atacuri este totuși destul de costisitor deoarece trebuie ca recompensa să fie sigură și să fie mai mare decât costul tranzacțiilor false adăugate în rețea pentru a bloca celelalte tranzacții.

*2.1.8. Trimiterea forțată de monede către un contract.* De exemplu, în cazul unui contract către care dorim să interzicem plățile putem considera că suprascrierea funcției ***payable()*** care să inverseze tranzacțiile către acesta ar fi suficient. În exemplul de mai jos, apelarea funcției ***revert()*** va refuza orice tranzacție făcută către contract. Funcția ***fallback payable()*** este unică unui contract și definește comportamentul acestuia atunci când sunt transferate monede fără date asociate direct către acel contract. Totuși, această funcție poate fi ignorată dacă se apelează o altă metodă a contractului și anume, ***selfdesctruct()***. În această metodă se pot face plăți către contract fără a apela ulterior funcția fallback.

*contractVulnerable{*

*function()payable{*

*revert();}*

*functionsomethingBad(){*

*require(this.balance>0);*

*// Do something bad}}*

Diverse alte atacuri asupra contractelor smart au fost descoperite și publicate, existând intenția de a realiza un registru central al vulnerabilităților cunoscute asociate cu exemple de cazuri de testare pentru a permite dezvoltatorilor să verifice automat rezistența codului lor la aceste tipuri de atacuri. (https://smartcontractsecurity.github.io/SWC-registry/) Printre cele publicate și încă nerezolvate în 2019 sunt:

* *DoS prin epuizarea limitei de combustibil,*
* *implementarea în ordine greșită a moștenirilor în Solidity care poate duce la comportament diferit de cel așteptat,*
* *scrierea într-o locație de memorie,*
* *lipsa protecției împotriva atacurilor de tip replay asupra semnăturii digitale,*
* *lipsa verificării corecte a acestei semnături.*

Ultimele două exemple de vulnerabilități sunt și singurele din listă în acest moment pentru care nu există teste de validare.

# Bigchaindb

BigchainDB este un software open-source care îmbină proprietăți de blockchain (descentralizare, imutabilitate, definirea unui proprietar pentru fiecare element stocat) cu proprietăți specifice unei baze de date (rata de tranzacțtii ridicată, latentă redusă, indexarea și interogarea datelor structurate). A fost lansat pentru prima dată în februarie 2016 și a fost îmbunătățit încontinuu de atunci. Este considerată o bază de date blockchain deoarece este o bază de date distribuită, compusă din mai multe noduri globale, dar care pot fi implementate și ca o rețea internă de noduri. Baza de date pe care construiește backend-ul aplicației poate fi o instanță sau mai multe noduri distribuite ale unor baze de date NoSQL MongoDB sau RethinkDB. În cadrul acestei aplicații nodurile rulează instanțe ale bazei de date MongoDB.

Pentru a obține proprietățile de toleranță la eșecuri de tip bizantin, BigchainDB foloseste engine-ul Tendermint pentru stabilirea unui conses între nodurile din rețea. Fiecare nodul are propria sa bază de date locală MongoDB iar întreaga comunicare între noduri se face folosind protocoalele Tendermint. O consecință a utilizării Tendermint este că sistemul rezultat este BFT, deoarece Tendermint este BFT. Un alt rezultat este că, dacă cineva rău intenționat reușește să obțină privilegii de administrare la una din bazele de date locale MongoDB, atunci cel mai rău lucru pe care îl pot face este coruperea sau ștergerea datelor din acea bază de date locală; bazele de date MongoDB din celelalte noduri nu vor fi afectate. Tendermint are o formă de sincronicitate prin menținerea propunerii actuale a blocului de către validatori, dar aceștia se schimbă cu fiecare rundă (folosind robin rotund) și se încearcă în continuare creșterea non-determinismului în alegerea următoarelor blocuri pentru a fi mai puțin previzibil.

## Concepte generale ale bazei de date

Bazele de date SQL structurează datele sub formă de tabele. Bazele de date NoSQL utilizează alte formate pentru a structura date precum JSON și perechi cheie-valoare, dar și tabele. În BigchainDB datele sunt strucurate ca bunuri (assets). Un bun poate caracteriza orice obiect fizic sau digital de exemplu poate fi un utilizator, un set de date sau un drept de proprietate intelectuală. Aceste bunuri pot fi înregistrate pe BigchainDB în două moduri:

1. de căre utilizatori în tranzacțiile CREATE.

2. Transferate (sau actualizate) către alți utilizatori în tranzacțiile TRANSFER.

Majoritatea utilizărilor BigchainDB se concentreaza asupra proceselor de afaceri (de ex. aplicații pentru rezervarea și procesarea comenzilor clienților, aplicații pentru urmărirea livrării produselor etc.). Ideea care a stat la baza BigchainDB s-a orientat mai degrabă pe procese decât pe bunuri (de exemplu, o comandă de client poate fi un bun care este urmărit pe parcursul întregului său ciclu de viață). Această schimbare în perspectivă spre un proces orientat pe bunuri influențează mult modul în care sunt definite procesele din aplicații.

Odată ce datele sunt stocate într-o rețea BigchainDB, acestea nu pot fi modificate sau șterse, sau cel puțin nu fără mari dificultăți. Dacă unele date reușesc să ajungă modificate sau șterse direct din baza de date, atunci acest lucru este detectabil. BigchainDB folosește mai multe strategii pentru a obține imutabilitate datelor. În primul rând, nu există API-uri furnizate de BigchainDB pentru a schimba sau șterge datele (singurele tranzacții acceptate fiind cele de CREATE și TRANSFER). O altă strategie este că fiecare nod are o copie completă a tuturor datelor din baza de date MongoDB independentă (adică nu există o bază de date globală MongoDB).

Chiar dacă un nod devine corupt sau distrus, celelalte noduri nu vor fi afectate și va avea în continuare o copie a tuturor datelor. Cel de-al treilea mecanism și cel mai important în acest sens, este că toate tranzacțiile sunt semnate criptografic. După ce o tranzacție este stocată, schimbarea conținutului acesteia va schimba semnătura, modificare ce poate fi detectată (cu excepția cazului în care cheia publică este schimbată la rândul ei, dar și în acst caz schimbarea va fi vizibilă deoarece fiecare bloc de tranzacții este semnat de către un nod și cheile publice ale tuturor nodurilor sunt cunoscute).

### Componentele unui Asset

Un bun(asset) este obiectul cu care pot fi realizate tranzacțiile în baza de date. Acesta poate modela orice element din realitate în funcție de câmpurile de date definite. Un bun trebuie să aibă cel puțin un proprietar, dar poate fi definit ca fiind propriul lui proprietar. Când este creat printr-o tranzacție CREATE, un bun poate fi ctnull (ex. None în Python), sau un vector de perechi cheie-valoare. Cheia trebuie sa fie *data* iar valoarea poate fi orice vector de perechi. (https://docs.bigchaindb.com/en/latest/permissions.html) De exemplu, o bicicleta poate fi reprezentată ca un bun în modul de mai jos:

{

"data "

{

"culoare " : " rosu " ,

"marca" : "Pegas "

}

}

### Componentele unei tranzacții

O tranzacție poate fi implementată ca un vector de perechi cheie-valoare (de exemplu ca un dicționar în Python). O tranzacție are următoarea structură de bază:

{

" id " : id ,

" version " : version ,

" inputs " : inputs ,

" outputs " : outputs ,

" operation" : operation ,

" asset " : asset ,

"metadata " : metadata

}

1. ID-ul tranzacției - este valoarea hash SHA3-256 prin care se poate identifica tranzacția.

2. Version - Versiunea indică regulile de validare ale tranzacției. Aceste reguli sunt asociate cu versiunea de specificații ale BigchainDB și sunt de forma "2.0".

3. Inputs - Fiecare input al tranzacțiilor folosește output-ul tranzacției precedente. O tranzacție de tipul CREATE trebuie să aibă un singur input, iar o tranzacție TRANSFER trebuie să aibă cel puțin un input. Un exemplu de input poate fi de forma:

{[TODO exemplu tranzactie de la mine]

4. Outputs - Fiecare output indică condițiile pe care trebuie să le indeplinească cineva care dorește să transfere tranzacția mai departe.

5. Operation - Se referă la tipul operației efectuate, este de tip string și poate avea două valori: CREATE sau TRANSFER.

6. Asset - Bunul descris prin vectori cheie-valoare care va fi tranzacționat sau creat.

7. Metadata - Sunt date definite de utilizator pentru a descrie tranzacția. Trebuie să aibă forma unui vector cheie-valoare sau săefi de tipul ctnull.

## Caracteristicile BigchainDB

* Descentralizare - Datele nu sunt concentrate într-un singur nod pentru a evita pierderea datelor în cazul în care nodul principal este inaccesibil. Descentralizarea se stabilește printr-un sistem de vot între nodurile din rețea.
* Câmpuri de tip immutable - Suport pentru definirea unor proprietari ale bunurilor care nu pot fi modificate.
* Suport pentru multiple tipuri de bunuri - Deoarece nu există o monedă activă care sa fie transferată în blockchain, orice tip de bun poate fi creat și transferat.
* Toleranța la eșecuri arbitrare - Până la o treime din nodurile din rețea pot avea eșecuri arbitrare, iar restul rețelei vor putea negocia blocul de date corect.
* Definirea permisiunilor - Permisiunile sunt similare privilegiilor într-o bază de date SQL. Aceste permisiuni se pot defini la nivelul bunurilor în câmpul output al unei tranzactii. De exemplu, putem deifni o permisiune ca pentru transferul unui anumit bun[1], ca numai persoana a cărei cheie publică apare în campul output să poată face mai departe tranzacții cu acel bun.

Driver-ul BigchainDB este disponibil pentru programarea în JavaScript, Python sau Java. Acesta pune la dispozițiile funcțiile și modulele necesare pentru conectarea și apelul de tranzacții către o baza de date BigchianDB la distanță sau locală. Fiecare nod dintr-o rețea BigchainDB 2.0 are propria bază de date locală MongoDB. Aceasta înseamnă că fiecare operator de noduri are acces la baza de date MongoDB completă pentru indexarea și interogarea datelor stocate (tranzacții, bunuri, metadate și blocuri, toate fiind șiruri de caractere JSON).

### Elemente criptografice ale BigchainDB

**SHA3-256.**

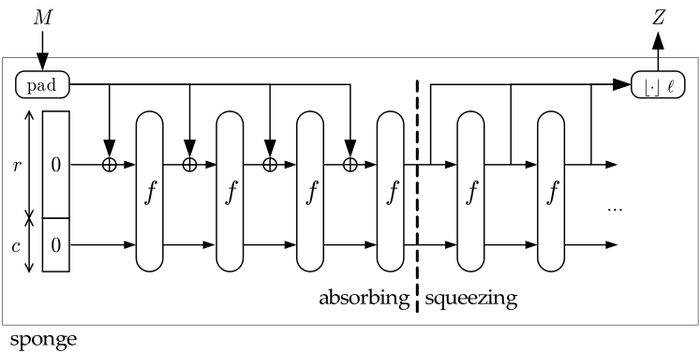
BigchainDB calculează amprenta hash a tranzacțiilor și blocurilor folosind o implementare a algoritmului SHA3-256 furnizat de pachetul pysha3, care este un wrapper peste implementarea optimizată a echiper Keccak: G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche și R. Van Keer (http://keccak.noekeon.org.). De la selectarea algoritmului de hashing propus de Keccak pentru SHA-3 în 2012, NIST a lansat o nouă versiune a hash-ului folosind același algoritm, dar cu parametri ușor diferiți. De la versiunea 0.9, BigchainDB utilizează versiunea mai recentă a algoritmului.

Autorii lui Keccak au propus o schemă simplă de hashing prin așa-numita funcție burete (cryptographic sponge). În interiorul acestui "burete" există o stare (având dimensiunea de 1600 de biți pentru SHA-3), la care se aplică aceeași funcție care aplică permutări pseudo-aleatoare la fiecare rundă. Algoritmul este în esență un cifru bloc de dimensiune 1600 implementat fără cheie. De exemplu dacă plecăm din starea 0 în care tot blocul este nul, executăm 10 runde (aplicând funcția f într-o direcție), și apoi de același număr de ori în direcția opusă (cu funcția inversă f), vom obține starea inițială.

Pentru a face fiecare porțiune de cod dependentă de restul blocului, nu sunt amprentate blocuri de 1600, ci de o dimensiune mai mică *r* restul blocului fiind completat cu șiruri generate prin generatorul pseudo-aleator. Procesul se repetă la fiecare rundă cu porțiuni *r* din șirulul original asemănându-se cu absorbția datelor de intrara ca printr-un burete.

După ce întreg șirul de intrare a fost procesat se continuă aplicarea funcției de permutare aleatoare pe rezultatele precedente și se selectează doar prima porțiune de lungime *r.* Funcția se reapelează pentru extragerea numărului de secvențe necesare pentru construirea hash-ulu de lungime 256 în acest caz. În versiunile de 384 și 512 se execută funcțe de permutare de un număr mai mare de ori, de aceea sunt și mai lente funcțiile de hash de lungime mai mare.

În imaginea de mai jos se pot observa cele două secvențe de ,,absorbție” și ,,stoarcere” care au loc în calcularea amprentei hash pentru un mesaj M împărțit în secvențe de lungime *r* . La fiecare rundă, secvența de text de intrare sau secvența obținută la runda anterioară este completată cu *c* elemente generate pseudo-aleator până la lungimea lenecară ca intrare în funcția de permutare *f*.



**Ed25519.**

BigchainDB folosește sistemul de semnătură publică Ed25519 pentru generarea perechilor de chei publice/private. Ed25519 este o formă a algoritmului de semnătura digitală pe curbe eliptice Edwards (EdDSA). În decembrie 2016, EdDSA a fost publicat ca un "Internet-Draft" al IETF(rfc8032.pdf), dar este deja utilizat pe scară largă.

(ed25519-20110926.pdf)

EdDSA are 11 parametri:

1. O putere primă *p*. EdDSA utilizează o curbă eliptică peste câmpul finit GF (p).

2. Un număr întreg b cu . Cheile publice EdDSA au exact *b* biți și semnăturile EdDSA au lungime 2\*b. *b* este recomandat să fie un multiplu de 8, astfel încât lungimile cheii publice și semnăturii sunt un număr întreg de octeți.

3. Codificarea în (b-1) biți a elementelor câmpului finit GF (p).

4. O funcție hash criptografică H care produce 2\* b-biți de ieșire.

5. Un număr întreg c care este 2 sau 3.

6. Un număr întreg n cu c <= n <b.

7. Un element nepătratic d din câmpul GF (p). Recomandarea obișnuită este ca acesta să fie valoarea cea mai apropiată de zero, care determină o curbă acceptabilă.

8. Un element pătratic a din GF (p), a = -1 dacă p mod 4 = 1 și a = 1 dacă p mod 4 = 3.

9. Un element B! = (0,1) din mulțimea E = {(x, y) este un membru al GF (p) x GF (p) astfel încât }.

10. Un număr prim L impar, astfel încât [L] B = 0 și. Numarul #E (numărul de puncte de pe curbă) face parte din datele specifice unei curbe eliptice E.

11. O funcție "prehash" PH. PureEdDSA înseamnă EdDSA unde PH este funcția de identitate, adică PH (M) = M.

Cheia privată este de lungime de 32 de octeți (256 de biți). Cheia publică este generată folosind cheia privată în modul următor:

a) Se calculeaza hash-ul cheii private folosind SHA-512 și stocând rezultatul de lungime 64 de octeți dintre care doar 32 vor fi folosiți pentru generarea cheii publice.

b) Cei mai nesemnificativi 3 biți ai primului octet, cel mai semnificativ bit și al doilea cel mai semnificativ bit ultimului octet sunt setați la 0.

c) Rezultatul este interpretat ca un întreg little-endian *s* și se realizează înmulțirea acestuia cu baza fixă dată ca parametru ().

d) Cheia publică este codificarea punctului de pe curba eliptică. Codificarea se realizează selectând întâi prima coordonată *y* ca un șir little-endian de 32 de octeți. Cel mai semnificativ bit al ultimului octet este întotdeauna zero. Pentru a forma codificarea punctului , se copiază cel mai puțin semnificativ bit al coordonatei x la cel mai semnificativ bit al octetului final. Rezultatul obținut este cheia publică.

Informațiile necesare pentru realizarea procedurii de semnare sunt: cheia privată, un șir de 32 de octeță și un mesaj M de mărime arbitrară. Pentru Ed25519ctx și Ed25519ph, mai sunt necesare și un context C de cel mult 255 octeți, un steag F care este setat 0 pentru Ed25519ctx și 1 pentru Ed25519ph.

a) Calculul amprentei hash SHA512 a cheii private. Fie h amprenta rezultată. Se construiește un scalar s pornind de la prima jumătate a amprentei rezultat ca în procesul de generare a cheii publice. Ne vom referi în continuare ca prefix la a doua jumătate a amprentei hash a cheii private (h[32],...,h[63]).

b) Se calculează SHA512(dom2(F,C)||prefix||PH(M)), unde M este mesajul care va fi semnat. Rezultatul va fi interpretat ca un întreg în reprezentare little-endian.

c) Se calulează punctul iar R va fi condificarea punctului rezultat.

d) Se calculează SHA512(dom2(F,C) | | R | | A | | PH(M)) și se interpretează rezultatul ca un întreg pe 64 de octeți.

e) Se calculează

f) Se formează semnătura prin concatenarea rezultatului R (de 32 de octeți) și codificarea little-endian a lui S (32 de octeți).

Pentru a verifica o semnătură pe un mesaj M folosind cheia publică A, cu flagul F fiind 0 pentru Ed25519ctx, 1 pentru Ed25519ph, C fiind contextul, se împarte mai întâi semnătura în două jumătăți de 32 octeți. Se decodifică prima jumătate ca fiind un punct R, și a doua jumătate ca un întreg S. Cheia publică A va fi interpretată ca un punct A'. În cazul în care oricare dintre decodificări eșuează (inclusiv S fiind în afara intervalului), semnătura este invalidă. Se calculează SHA512 (dom2 (F, C) || R || A || PH (M)) și se interpretează rezultatul ca întreg little-endian k. LA final se verifică ecuația grupului [S] \*B = R + [k] \*A'. Dacă ecuația se verfică, semnătura este validă.

**Base-58.**

Toate cheile sunt reprezentate în codul Base-58. Principalele motive pentru folosirea codificării base-58 în loc de base-64 în aplicațiile de monede electronice sunt au fost descrise Satoshi Nakamoto pentru utilizarea codificării în Bitcoin:

* eliminarea caracterelor 0(zero), O(litera o mare), I(litera i mare), l(litera l) care arată la fel în unele fonturi și ar putea fi utilizate pentru a crea numere de cont care ar putea fi vizualizate identic.
* un șir cu caractere non-alfanumerice nu va fi acceptat la fel de ușor ca un număr de cont.
* dacă sunt trimise prin email, acesta nu va adăuga automat întreruperi de linie dacă nu există semne de punctuație după care sa se facă întreruperea.

Un șir Base58Check este creat dintr-o șir de date *d* în modul următor.

1. Datele care vor fi codificate sunt precedate de un antet prin care se identifica aplicația/versiunea. De exemplu, în cazul Bitcoin se adaugă un octet 0 (0x00 hexazecimal).
2. Se calculează amprenta hash *h =* *SHA256 (SHA256 ( 0x00| d ));*
3. Se selectează primii 4 octeți a amprentei *h* și se concatenează la șirul de date *m = d | h* pentru detectarea erorilor.
4. Considerăm rezultatul anterior ca un singur număr big-endian. Conversia este realizată urmând pașii matematici. Secvența de numere obținute în urma conversiei sunt înlocuite cu cifrele corespunzătoare tabelului de conversie base-58.
5. Caracterul de inceput "1", care are o valoare zero în baza58, este rezervat pentru a reprezenta un octet de 0 la începutul numărului. Pot exista unul sau mai multe caractere 1 când este necesar să reprezinte unul sau mai mulți octeți zero.

## Tendermint

Tendermint este un software pentru asigurarea consistenței replicării unei aplicații pe mai multe mașini. Datele sunt menținute în condițile în care până la 1/3 din mașini pot avea un comportament arbitrar. Fiecare mașină care nu este defectă vede același jurnal de tranzacții și calculează aceeași stare. Replicarea sigură și consistentă este o problemă fundamentală în sistemele distribuite și are un rol esențial în toleranța la defecte aleatoare pentru o gamă largă de aplicații.

Abilitatea de a menține o stare consistentă a datelor chiar daca există în sistem mașini care au un comportament arbitrat sau malițios, este cunoscută sub numele de toleranță la eșecuri bizantine (BFT). Teoria BFT este destul de veche, însă implementările de software au devenit populare recent, datorită în mare parte succesului tehnologiei "blockchain" precum Bitcoin și Ethereum. Tehnologia Blockchain este doar o reformulare a BFT într-un cadru mai modern, cu accent pe rețelele peer-to-peer și autentificarea criptografică.

Tendermint este alcătuit din două componente principale: un motor care aplică algoritmul de consens și o interfață pentru interconectare cu diverse aplicații. Motorul de consens, numit Tendermint Core, asigură că aceleași tranzacții sunt înregistrate pe fiecare mașină în aceeași ordine. Interfața aplicației, numită Interface Application BlockChain (ABCI), permite procesarea tranzacțiilor în orice limbaj de programare. Motorul de conses Tendermin Core este implementat în principal în Go. (<https://github.com/tendermint/tendermint>)

Zookeeper, etcd și consul sunt exemple de implementări simiare prin care se pot stoca date de tipul cheie-valoare și menține consisența acestora printr-un un algoritm clasic, non-BFT de consens. Zookeeper utilizează o versiune a Paxos numită Zookeeper Atomic Broadcast, în timp ce etcd(<https://github.com/etcd-io/etcd>) și consul folosesc algoritmul Raft de obținere a consensului, care este mult mai mic și mai simplu. Un grup tipic conține 3-5 mașini și poate tolera căderi de avarie în până la 1/2 din mașini, dar chiar și o singură eroare bizantină poate distruge sistemul.

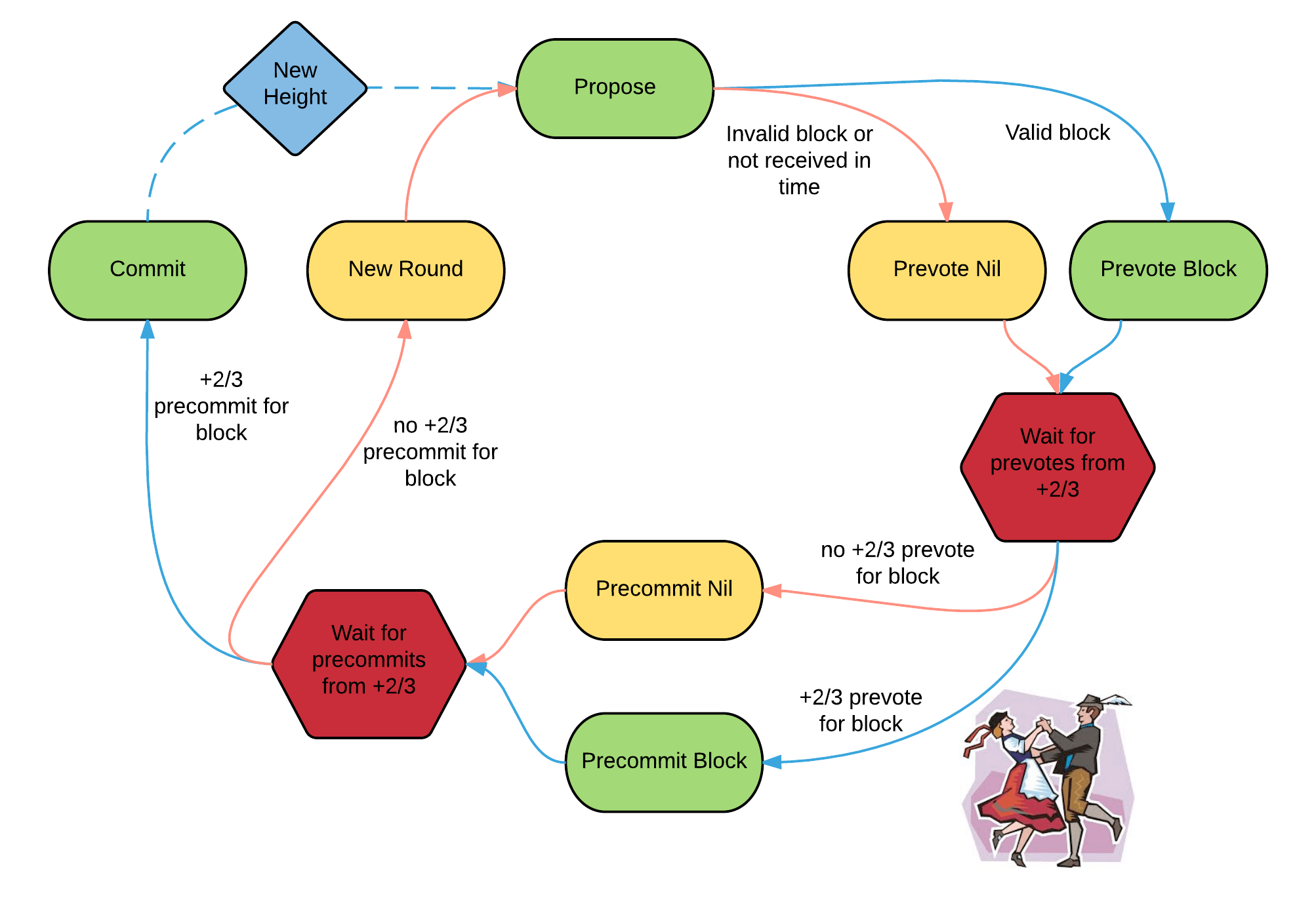
Fiecare dintre aceste variante oferă o implementare puțin diferită de stocare a datelor cheie-valoare, dar toate se concentrează, în general, pe furnizarea de servicii de bază pentru sistemele distribuite, cum ar fi configurația dinamică, descoperirea serviciilor, blocarea și alegerea liderilor.

Tendermint este un software similar, dar cu două diferențe cheie:

* Mecanismul de conses asigură toleranță la eșecuri bizantine, adică poate tolera până la o treime din noduri compromise, aceste eșecuri pot include comportament arbitrar, inclusiv atacuri hacking și atacuri rău-intenționate.
* Nu restrânge modurile de stocare ale datelor la formatul cheie-valoare. Se concentrează asupra replicării informaței din noduri și rezistența la comportamente arbitrare.

Sistemul pentru replicarea nodurilor este interfațat către aplicațiile care doresc să se integreze cu Tendermint prin interfața Interfața Application BlockChain (ABCI). ABCI este o interfață care definește limita dintre motorul de replicare (blocul de blocuri) și aplicația cu care se integrează. Folosind un protocol bazat pe comunicare prin socket, un motor de consens care rulează într-un singur proces poate gestiona o stare a aplicației care rulează într-un alt proces. Pot exista mai multe conexiuni ABCI socket la o aplicație. Tendermint Core creează trei conexiuni ABCI la aplicație; una pentru validarea tranzacțiilor în timpul publicării în mempool, una pentru motorul consens pentru a rula propunerile de bloc și una pentru interogarea stării aplicației.

Comportamentul protocolul poate fi reprezentate printr-o mașină finită de stări care arată astfel:



Participanții la protocol sunt numiți *validatori*; ei se schimbă, propunând blocuri de tranzacții și votând asupra lor. Blocurile sunt adăugate într-un lanț, cu câte un bloc la fiecare nivel. Este posibil ca un bloc să nu fie adăugat, caz în care protocolul se deplasează la următoarea rundă și un nou validator ajunge să propună un bloc pentru acest slot. Sunt necesare două etape ale votului pentru a comite cu succes un bloc: pre-vot și pre-comite. Se înregistrează un bloc atunci când mai mult de două treimi din validatori votează pentru același bloc în aceeași rundă.

Atunci când mai mult de două treimi din validatori pre-votează pentru același bloc, se consideră că se creează o polca. Fiecare pre-comitere trebuie să fie justificată de o polca în aceeași rundă.

Este posibil ca validatorii să nu reușească să comită un bloc din mai multe motive; propunerea curentă poate fi offline sau rețeaua poate fi lentă. Tendermint le permite să stabilească faptul că un validator ar trebui să fie omis. Validatorii așteaptă un timp redus pentru a primi un bloc complet de propuneri de la propunător înainte de a vota pentru a trece la runda următoare. Această dependență de un timeout este ceea ce face ca Tendermint să fie un protocol slab sincron, mai degrabă decât unul asincron. Cu toate acestea, restul protocolului este asincron, iar validatorii fac progrese numai după ce au auzit de la mai mult de două treimi din setul validator. Un element simplificator al Tendermint este că folosește același mecanism pentru a comite un bloc ca și pentru a trece la următoarea rundă.

Presupunând că mai puțin de o treime din validatori sunt bizantine, Tendermint garantează că siguranța nu va fi niciodată încălcată - adică validatorii nu vor comite blocuri conflictuale la aceeași nivel. Pentru a face acest lucru, se introduc câteva reguli de blocare care modulează care trasee pot fi urmate în diagrama fluxului. Odată ce un validator pre-comite un bloc, acesta este blocat pe acel bloc. În acest caz, validatorul trebuie să voteze pentru blocul pe care este blocat și se poate debloca pentru a vota pentru alt bloc numai dacă există o polcă pentru acel bloc într-o rundă ulterioară.

**CONCLUZII**