Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Gašperlin

Napadi na Ethereum

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Aleksandar Jurišić
Somentor: dr. Peter Nose
Somentor: asist. dr. Janoš Vidali

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Analizirajte delovanje pametnih pogodb ter opišite koncepte, na katerih temelji njihovo delovanje, ranljivosti, in kako poteka razvoj. Predstavite znane napade na Ethereum ter analizirajte in demonstrirajte uporabo programskega jezika Solidity. Predstavite tudi predloge varnega pisanja pogodb v njem.

Iskreno se zahvaljujem mentorju prof. dr. Aleksandru Jurišiću za odlično usmerjanje pri izdelavi diplomske naloge. Zahvaljujem se tudi somentorjema dr. Petru Nosetu in asist. dr. Janošu Vidaliju za vsebinske predloge in tehnične nasvete. Posebna zahvala gre mojim staršem za podporo pri študiju.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
ABI	Application binary interface	binarni aplikacijski vmesnik
BTC	Bitcoin	kriptovaluta bitcoin
DAO	Decentralized autonomous organization	decentralizirana avtonomna organizacija
DLT	Distributed ledger technology	tehnologija porazdeljene glavne knjige
DOS	Denial of service	napad s preprečitvijo dostopa do storitve
ETC	Ethereum classic	kriptovaluta Ethereum classic
ETH	ether	kriptovaluta eter
EVM	Ethereum virtual machine	virtualni stroj Ethereum
ICO	Initial coin offering	začetna prodaja žetonov
P2P	Peer-to-peer	omrežje od sledilca do sledilca
POW	Proof of work	dokazilo o delu
REP	Augur token	žeton Augur
RLP	Recursive length prefix	rekurzivno kodiranje z dolžinskimi predponami
RSA	Rivest–Shamir–Adleman	asimetrični kriptosistem
XSS	Cross-Site Scripting	napad navzkrižnega izvajanja kode

Povzetek

Naslov: Napadi na Ethereum

Avtor: Domen Gašperlin

V diplomski nalogi smo obravnavali zasnove in principe delovanje omrežja Ethereum, ki poleg prenosa sredstev omogoča tudi pisanje pametnih pogodb ter njihovo objavo v omrežju. Na ta način lahko izdelamo decentralizirane namenske programe, nad katerimi, ko jih enkrat objavimo, centralna avtoriteta nima več nadzora. Ker takih programov ne moremo več spreminjati, je potrebno biti že pri njihovi izdelavi izredno pazljiv in natančen. V delu predstavimo napade, ki so se v preteklosti že zgodili v omrežju Ethereum, njihove posledice in načine reševanja. Da se izognemo njihovim ponovitvam, predstavimo dobre prakse pisanja pametnih pogodb ter ključne težave, s katerimi se razvijalec tekom razvoja sooča.

Ključne besede: kriptovalute, varnost, Ethereum.

Abstract

Title: Attacks on Ethereum

Author: Domen Gašperlin

In this thesis we aim to provide the concepts and principles necessary to understand how the Ethereum network works. It allows us to write smart contracts and deploy them to the network. This way decentralised applications with no central authority can be written. Due to the fact that contracts cannot be modified once they have been published to the network, we need to be careful how we structure them. We present the most relevant attacks that have happened on the network in the past. In order to avoid them in the future, we present good practices for writing smart contracts and the key problems one faces during their development.

Keywords: cryptocurrency, security, Ethereum.

Poglavje 1

Uvod

Vse več podatkov in storitev se seli na internet. Z njihovo analizo lahko napovedujemo dogodke, prepoznavamo vzorce – skratka, iz njih pridobivamo novo znanje in tako ponudimo boljše storitve. Podatke danes večinoma shranjujemo v digitalni obliki v računalniškem spominu. V zadnjem času to velja tudi za področji financ (npr. kriptovalute) in prava (npr. pametne pogodbe).

Osnovni problem pri denarju je ta, da je bil nekoč vezan na dobrine. Denar mora imeti vrednost ter biti s časom nespremenljiv. Zato se je za menjavo dobrin začelo uveljavljanje srebra ter zlata. Ob večjih transakcijah je takšen denar težaven za transport, pojavljati se je začelo tudi ponarejanje denarja in odvzemanje dragih kovin iz robov kovancev. Vlada je tako omogočila zamenjavo zlata za potrdilo o njegovem lastništvu. S tem potrdilom si lahko kupil dobrine ali ga ponovno zamenjal za zlato. Trgovci so ugotovili, da enostavneje trgujejo s papirjem. Tako so lažje potovali po nevarnih predelih brez nevarnosti kraje. Zlatarji, ki so za prineseno zlato izdajali papirje, so ugotovili, da se jim je le-to začelo kopičiti, saj so ljudje raje izmenjavali papirje. Zato so naprej začeli posojati zlato, ki ni bilo njihovo. Ko se je število papirjev na trgu nakopičilo, pa je prišlo do množične zamenjave papirjev za zlato. Zlatarji v kratkem času teh menjav niso mogli izvesti. Danes je denar, ki ga imamo na voljo, le še število, zapisano na strežnikih bank. Njegovo vrednost glede na razmere na trgu regulira centralna banka.

V zadnjem času preko interneta izmenjujemo podatke s pošiljanjem njihovih kopij. Pri nekaterih podatkih pošiljanje kopij ne sme biti dovoljeno. Takšen je tudi denar. V letu 2009 [16] se je začel razvijati digitalni denar Bitcoin, ki vsakemu omogoča, da sodeluje pri njegovem nastajanju. Delovanje financ danes namreč zahteva arhitekturo, za katero skrbi množica institucij in posrednikov, ki za svoje delovanje potrebujejo sredstva. Proces mednarodnega prenosa denarja je dolgotrajen in drag. Pri Bitcoinu pa se denar anonimno in brez posrednikov prenaša po omrežju računalnikov. Prav tako omrežje Bitcoin omogoča pošiljanje denarja (kriptovalute) po nizki ceni in hitreje kot po tradicionalnih postopkih.[1] Za varnost svojega delovanja uporablja kriptografske principe, ki zagotavljajo, da se denar ne porabi večkrat ter da prejemnik poslana sredstva res prejme.

Ključni koncept pri Bitcoinu je tehnologija verige blokov, v kateri lahko izmenjamo tudi podatke z vrednostjo (avtorska zaščita), kot so glasovi volivcev, umetnine, filmi, pogodbe itd. Pri njih je ključno, da jih ne moremo ponarediti. Po Bitcoinu so se začele pojavljati številne aplikacije, ki uporabljajo tehnologijo verige blokov v zgoraj omenjene namene. Zato so se začele razvijati številne vzporedne verige blokov. Vsaka za svoje delovanje potrebuje veliko število računalnikov. Za varnost delovanja je potrebno zagotoviti dovolj veliko število le-teh, zato je vzdrževanje številnih vzporednih verig težavno.

Programer Vitalik Buterin je začel razvijati splošen sistem, ki bo omogočal razvoj poljubnih aplikacij na verigi blokov. Za delovanje vseh teh aplikacij je tako potrebno vzdrževati le eno omrežje, sistem oz. verigo. Poimenoval ga je *Ethereum*, njegovo valuto pa *eter*. S tem je postala tehnologija verige blokov širše razpoložljiva in dostopna. Ethereum je sistem, ki nam omogoča izvajanje programov na mnogih računalnikih, povezanih v omrežje. V teh programih so zapisana pravila, ki določajo njihovo delovanje. V primerjavi s programi, ki se izvajajo na centraliziranih strežnikih, gre tu za sodelovanje računalnikov, ki ga ne moremo zlahka ustaviti. Tovrstne programe lahko

[1] Ima sicer druge težave, kot je velika poraba energije in posledično slab vpliv na okolje.

kadarkoli objavimo in s tem omogočimo njihovo izvajanje. V njih lahko avtomatiziramo delovanje manjših podjetij ter postopke prava in financ.

Za razvoj aplikacij na omrežju Ethereum potrebujemo le poznavanje programskega jezika *Solidity*. Le-ta je zelo podoben spletnem programskem jeziku JavaScript, zato lahko spletni razvijalci svoje znanje uporabijo tudi za razvoj aplikacij na Ethereumu. Prednosti, ki jih prinaša to omrežje v primerjavi s tradicionalnimi spletnimi aplikacijami, je odpornost na vdore in zagotavljanje zaupanja preko spleta brez posrednikov. V njej lahko hranimo podatke in dodeljujemo dostop do storitev, za katere lahko zahtevamo plačilo v kriptovaluti eter. Za uporabo te tehnologije je potrebno le napisati program, v katerem določimo pravila, ki lahko zajemajo prenos plačil in delovanje storitev in ga objaviti v omrežje.

Za izvajanje storitve ne potrebujemo strežnika, vse se izvaja na množici računalnikov, razpršenih po svetu, ki za svoje delo dobivajo plačilo v kriptovaluti. Temu omrežju oziroma množici rudarjev se lahko pridruži kdorkoli. Objavljeni programi omogočajo tudi upravljanje z denarjem oziroma kriptovaluto, npr. prenos denarja drugim uporabnikom, ko veljajo pravila, določena v pogodbi. Prav tako so objavljeni programi vidni vsem članom omrežja, kar uporabnikom omogoča, da se odločijo za njihovo uporabo na podlagi branja izvorne kode. S temi programi zapišemo npr. logiko za zbiranje sredstev za razvoj projektov.

Žal so napake zaradi nespremenljivosti programov lahko usodne. Njihovo kodo lahko bere kdorkoli in tako vidi, če so v njej varnostne luknje, ki omogočajo izkoriščanje programov na načine, ki jih programer ni predvidel. To se je v preteklosti že večkrat zgodilo in posledično povzročilo milijonske izgube sredstev. Dobro bi bilo, če bi pravilnost programov lahko kar matematično dokazali, oziroma se prepričali, da ne vsebujejo napak. Temu so se približala številna orodja, kot sta npr. `oyente` (Luu et al. [24]) in `openzeppelin` [6]. Slednji objavlja osnovne sestavne dele najbolj uporabljanih programov, pravilnost katerih preverja in vzdržuje odprtokodno združenje. Razvijalcem so tako na voljo ustrezno preizkušeni sestavni deli,

ki jih lahko uporabijo za ogrodje svojih programov. Iz zgodovine se največ naučimo, zato bomo v delu predstavili pretekle napade. Analizirali bomo njihove postopke oziroma pristope ter ranljive pretekle programe v izogib njihovim ponovitvam. Nekatere napake so tudi rezultat slabe zasnove programskega jezika, namenjenega za njihovo pisanje. Testirali bomo, kateri napadi so še izvedljivi in kateri ne več. S tem bomo ugotovili, kakšno je stanje ranljivosti v omrežju, in opozorili na vzorce v programih, ki lahko privedejo do napadov. Prav tako bomo predstavili nekaj posebnosti pri izvajanju tovrstnih programov v primerjavi s centraliziranimi programi, saj teh razlik pri njihovem razvoju ni potrebno upoštevati, napadalci pa jih zelo dobro poznajo.

V formalni specifikaciji (glej G. Wood [36]) je natančno predstavljeno delovanje Ethereuma, ki nam pomaga pri razumevanju ranljivosti. Atzei et al. [12] so opisali prvo sistematično klasifikacijo ranljivosti v Ethereumu in pametnih pogodbah. Določene pomanjkljivosti so bile do danes že odpravljene, tako da nekaterih napadov ni več mogoče izvesti. Nikolic et al. [29] so razvili orodje `maian`, ki v omrežju omogoča avtomatsko zaznavo treh tipov ranljivosti pametnih pogodb – za vse objavljene pogodbe koda v programskem jeziku Solidity namreč ni na voljo. Luu et al. [24] so dokumentirali najbolj pogoste ranljivosti pametnih pogodb, prav tako so formalizirali semantiko jezika Solidity. Z uporabo formalizacije so razvili orodje `oyente`, ki s simboličnim izvajanjem zelo zanesljivo uvrsti pametne pogodbe v razrede najbolj pogostih ranljivosti.

Za lažje razumevanje v 2. poglavju na kratko predstavimo osnovne kriptografske koncepte, ki jih uporablja tehnologija verige blokov. Na njej namreč temelji delovanje Ethereuma. V 3. poglavju se osredotočimo na posebnosti omrežja Ethereum ter obravnavanega programskega jezika za razvoj pametnih pogodb. Za slednjim bomo v 4. poglavju predstavili znane napade, ki so izkoriščali presenetljivo lahko razumljive pomanjkljivosti, za odtujitev velikih vsot denarja. S tem motiviramo 5. poglavje, v katerem predstavimo varnostne ranljivosti pametnih pogodb ter podamo predloge za njihovo varno

pisanje. V zaključku predstavimo še, kakšno je stanje ranljivosti v omrežju ter kakšne izboljšave so potrebne, da bi se stanje izboljšalo.

Poglavje 2

Veriga blokov

Tako imenovana tehnologija verige blokov (angl. Blockchain) je postala popularna leta 2008 z izdajo dela Bitcoin: A peer-to-peer electronic cash system [28], v katerem je Satoshi Nakamoto predlagal uporabo omrežja P2P (angl. Peer-to-peer) ter digitalnih podpisov za razvoj prvega decentraliziranega plačilnega sistema med računalniki, ki preprečuje dvojno porabo (glej razdelek 2.9). Osrednjega pomena je bila ideja uporabe decentralizacije in porazdeljene podatkovne baze, ki ju bomo predstavili v razdelku 2.5. Novost je torej sposobnost opravljanja plačil brez posredovanja tretjih oseb (glej razdelek 2.8). Z začetkom delovanja sistema Ethereum leta 2015 se je omogočilo še izvajanje programov – pametnih pogodb. Tehnologija je dobila ime, ker so bloki podatkov med seboj povezani v verigo z uporabo kriptografskih mehanizmov. Ta pojem opredelimo natančneje v nadaljevanju (glej razdelek 2.3), a še prej vpeljemo nekaj osnovnih kriptografskih elementov, kot so zgoščevalne funkcije in digitalni podpisi.

2.1 Zgoščevalne funkcije

Za zagotavljanje celovitosti sporočil se v računalništvu uporabljajo *zgoščevalne funkcije*. Opravljajo preslikavo poljubne velikosti vhoda podatkov v t. i. *zgostitev* fiksne velikosti. Ker je možnih vhodov več kot izhodov, to po-

meni, da obstajajo *trki*, tj. taki pari vhodov, ki bi imeli enake zgostitve (glej O. Goldreich [20]). Varnost teh funkcij sloni na tem, da je trke težko poiskati.[1] Imeti morajo naslednje lastnosti:

1. mora se jih dati učinkovito izračunati,

2. če spremenimo le en znak vhoda, se mora njihov izhod precej spremeniti,

3. računsko neučinkovito je najti trke.

Iz 1. in 3. točke sledi, da so zgoščevalne funkcije torej tako imenovane *enosmerne funkcije* – zanje velja, da je izračun izhoda iz vhoda izračunljiv v doglednem času.[2] Druga točka je pomembna zato, da preprečimo izračun spremembe zgostitve na podlagi sprememb vhoda brez ponovnega izračuna zgostitve.

Celovitost sporočila pomeni, da sporočilo med prenosom ni bilo spremenjeno. Če podpišemo zgostitev poslanega sporočila (več o tem v razdelku 2.2), zagotovimo, da bo prejemnik lahko preveril celovitost sporočila in identiteto podpisnika. Na ta način zagotovimo avtentičnost in celovitost sporočila. Za zagotovitev zasebnosti vsebine sporočila pa se uporablja simetrične kriptosisteme, za katere moramo najprej opraviti izmenjavo ključev (npr. protokol Diffie-Hellman [18]).

Zgoščevalna funkcija *Keccak256* ima zelo dobro psevdo-naključno porazdelitev (glej Gholipour in Mirzakuchaki [19]). Mi jo bomo v nadaljevanju uporabljali za Ethereum.

2.2 Digitalni podpisi

Asimetrični kriptosistemi uporabljajo *javne* in *zasebne* ključe (glej npr. O. Goldreich [20]). Slednje morajo lastniki skrbno hraniti, da jih kdo ne odtuji

[1] Znana preiskovanja so prepočasna zaradi odsotnosti učinkovitega algoritma.

[2] Izračun ne sme biti prehiter, ker bi s tem povečali učinkovitost napada z grobo silo.

ter se predstavlja v njihovemu imenu. *Digitalni podpisi* so nekakšen analog ročnih podpisov. Uporabnik se z digitalnim podpisom sporočila predstavi. Z njim torej dokaže, da je sporočilo podpisal res on, saj le on pozna svoj zasebni ključ. Prejemnik podpisanega sporočila lahko z uporabnikovim javnim ključem preveri podpis. Pri Ethereumu je naslavljanje med računi ter avtorizacija izvedena z asimetričnimi kriptosistemi na osnovi eliptičnih krivulj (ki jih v tem delu ne bomo obravnavali, glej npr. Jurišić in Menezes [22]). Javni ključ je v tem primeru naslov uporabniškega računa, zasebni ključ pa se uporablja za podpisovanje zgostitev transakcij, npr. za prenos sredstev z uporabniškega računa. Digitalni podpis zagotavlja, da sporočilo ni bilo spremenjeno. Uporabo teh mehanizmov si bomo bolj podrobno ogledali v poglavju 3.

2.3 Kaj je veriga blokov

Veriga blokov je sestavljena iz zaporedja blokov, ki so med seboj povezani s *kazalci zgostitev* (angl. hash pointers). Kazalec zgostitve je podatkovna struktura, ki povezuje bloke v verigi ter zagotavlja celovitost njihovih podatkov. Tako je mogoča le operacija dodajanja novih blokov, brisanje in spreminjanje pa je računsko prezahtevno, da bi bilo izvedljivo v doglednem času. To je zagotovljeno z uporabo kriptografskih mehanizmov, ki zahtevajo, da se za spreminjanje bloka ponovno vloži računska moč, ki se eksponentno povečuje z globino bloka, ki ga spreminjamo. *Veriga blokov* je ena izmed oblik uporabe porazdeljene glavne knjige, ki igra vlogo podatkovne baze (v primeru Bitcoina s podatki o stanju na računih uporabnikov). Bloki v verigi predstavljajo skupek transakcij v natanko določenem vrstnem redu, pri čemer je *transakcija* predpisan podatek, shranjen v bloku. Sistem si lahko predstavljamo kot končni avtomat, kjer iz *izvornega stanja* (angl. genesis state) s transakcijami preidemo v poljubno vmesno ali končno stanje [36]. Primer si lahko ogledamo pri Bitcoinu [15] na transakciji prenosa sredstev Ane k Janezu. Ko Ana pošlje Janezu 5 BTC (bitcoinov), transakcija predstavlja

informacijo o zmanjšanju stanja Aninega računa za 5 BTC in povečanju stanja Janezovega računa za 5 BTC.

2.4 Rudarjenje

Rudarji so množica računalnikov, povezanih v omrežje, ki s postopkom, ki ga imenujemo *rudarjenje*, potrjujejo in preverjajo transakcije ter jih vključujejo v večjo enoto, imenovano blok. Za dodajanje novih blokov v verigo je potrebno reševanje računskih nalog (glej razdelek 3.1) in preverjanje pravilnosti transakcij.[3] Za povezavo v omrežje potrebujemo nameščenega *odjemalca*. Sodelovanje pri dodajanju novih blokov v verigo omrežje spodbuja z nagrajevanjem rudarjev s kriptovaluto (npr. bitcoin) kot plačilo za opravljeno delo.

Kriptografski principi, kot so npr. zgoščevalne funkcije, skrbijo za to, da je rudar za preverjanje transakcij vložil dovolj *računske moči* (angl. proof of work – POW). Kot smo že omenili, se s spreminjanjem vhoda zgoščevalne funkcije ne da učinkovito napovedati strukture izhoda. Tako rudar pri računanju zgostitve poskuša preveriti različne parametre vhoda (množice transakcij) s ciljem, da z njimi izračuna izhod, ki ima določeno strukturo.[4] Ta je določena s *težavnostjo bloka*, ki se dinamično prilagaja glede na računsko moč celotnega omrežja. Večja kot je skupna računska moč, bolj se težavnost bloka poviša. Ko rudar najde rešitev z zahtevano težavnostjo, bo to želel čim prej poslati ostalim rudarjem na omrežju skupaj s transakcijo, ki mu izplača nagrado. Po preverjanju lahko rudarji začnejo tekmovati za naslednji blok.

2.5 Decentraliziranost in porazdeljenost

Prednost, ki jo prinaša veriga blokov v primerjavi s klasičnimi arhitekturami, je *decentraliziranost*. Slednje moramo ločiti od *porazdeljenosti*, ki pomeni, da se operacije izvajajo na večjem številu računalnikov, ponavadi pod nadzo-

[3] Pravilnost se preverja s testi, npr. če ima račun pošiljatelja na voljo dovolj sredstev.

[4] Pri Bitcoinu se zahteva določeno število zaporednih ničel na začetku zgostitve.

rom neke enote (npr. podjetja). Pri decentraliziranosti pa govorimo o tem, da se operacija izvaja na računalnikih oseb, ki jih ne nadzorujemo. Na ta način vzpostavimo sistem, kjer člani hranijo podatke o stanju in z glasovanjem oz. rudarjenjem odločajo o novih spremembah. Vsak član namreč hrani kopijo celotnega stanja verige, kar pomeni, da je možnost izgube podatkov zelo majhna.

2.6 Dostopne in nedostopne verige

Ločimo dve vrsti verig blokov – *nedostopne* (angl. permissioned), za katere potrebujemo dovoljenje za sodelovanje, in prosto *dostopne* (angl. permissionless), ki so pogosto tudi javne [11].

Pri vseh gre za decentralizirana *omrežja P2P*, v katerih vsak član hrani svojo kopijo podatkov, spremembe katerih se uveljavlja s protokoli za dogovor. Pri nedostopnih sta pogosto uporabljena protokola PAXOS (za podroben opis glej [23]) ali RAFT (za podroben opis glej [30]), pri dostopnih pa vložek računske moči. Oba sistema omogočata nemoteno delovanje tudi, če del članov deluje proti sistemu oziroma so nepošteni. Nedostopne verige so bolj priljubljene v podjetjih, kjer na zunaj ne želijo razkriti transakcij oz. potrebujejo večjo skalabilnost. Primera omrežij, ki temeljita na dostopnih verigah, sta Ethereum in Bitcoin, na nedostopnih pa omrežje Quorum [8] in ogrodje Hyperledger Fabric [5].

2.7 Zasebnost

Ker je pri Bitcoinu in Ethereumu veriga blokov javna, je potreben mehanizem ohranjanja zasebnosti. V ta namen se uporablja princip *psevdonimnosti*, kar pomeni prepoznavnost uporabnikov po psevdonimih, ki jih ne moremo takoj povezati z njihovo identiteto. Kot psevdonim se namreč uporablja izpeljanka javnega ključa (gre za desnih 20 bajtov zgostitve ključa, glej G. Wood [36]). Če se odkrije njihova povezava z identiteto osebo, je možen vpogled v celotno

zgodovino transakcij.

2.8 Vzpostavitev zaupanja

Trenutno problem *vzpostavitve zaupanja* na spletu rešujemo z vzpostavitvijo agencij (npr. certifikatna agencija), ki zagotavljajo, da bodo sledile vnaprej dogovorjenim pravilom, ki so zapisana v njihovih politikah. Težava je v tem, da so te agencije oziroma *posredniki* zaradi pomembnosti vloge, ki jo opravljajo, nemalokrat osrednja tarča napadov. Če napadalec pride do občutljivih podatkov agencije, npr. glavnega ključa, lahko povzroči precejšno škodo. Korak s časom ter zaščita pred poskusi napadov, ki sta potrebna za učinkovito obrambo, zahtevata neprestano vlaganje v mehanizme za večjo varnost in zaščito podatkov.

2.9 Napad dvojne porabe

Veriga blokov temelji na predpostavki, da več kot polovica udeležencev deluje v korist sistema. Če napadalcu uspe pridobiti 51% računske moči vseh uporabnikov, lahko izvede napad dvojne porabe sredstev.

Primer: Predstavljajmo si, da zlonamerni rudar naroči izdelek v spletni trgovini. Takoj, ko se blok z njegovo transakcijo vključi v verigo, prodajalec pošlje izdelek. S tem je prodajalec naredil napako, saj bi s čakanjem na nove bloke napad otežil. Ko napadalec dobi izdelek, začne takoj rudariti iz bloka, ki še ne vsebuje transakcije, v kateri se prenesejo sredstva z njegovega računa na račun prodajalca. Da uveljavi svojo transakcijo kot veljavno, mora prehiteti trenutno najdaljšo verigo blokov, nad katero so, medtem ko je izdelek potoval do kupca, ostali rudarji nalagali bloke. Če je sposoben bloke dodajati hitreje kot preostanek omrežja, tako dobi izdelek, za katerega prodajalec ni prejel sredstev. ◊

Poglavje 3

Ethereum

Sistem Ethereum je decentralizirana platforma, ki izvaja pametne pogodbe – aplikacije brez možnosti izpada, cenzure, goljufij ter vmešavanja tretjih oseb (glej njihovo uradno spletno stran [3]). Sinonim za Ethereum je nemalokrat tudi *svetovni računalnik* (angl. world computer), saj je globalno gledano deljen računalnik oziroma sistem na voljo vsem za uporabo za namen izvajanja in shranjevanja aplikacij. Kriptovaluta, ki poganja njegovo delovanje, se imenuje eter in služi kot plačilo rudarjem za izvajanje pametnih pogodb.

Enota	Wei
wei	1
Kwei (babbage)	1,000
Mwei (lovelace)	1,000,000
Gwei (shannon)	1,000,000,000
microether (szabo)	1,000,000,000,000
milliether (finney)	1,000,000,000,000,000
eter (angl. ether)	1,000,000,000,000,000,000

Tabela 3.1: Metrični sistem kriptovalute eter.

Prvo različico Ethereuma je leta 2013 predlagal ruski programer Vitalik Buterin zaradi pomanjkljivosti Bitcoina, ki ima zelo omejen skriptni jezik. Njegov cilj je bil razvoj splošnega sistema, kjer so nove funkcionalnosti

prepuščene programerjem [3]. Ethereum ima dva tipa računov, glede na to, kdo z njim upravlja:

- *uporabniški račun*, s katerim upravlja uporabnik z zasebnim ključem,
- *račun pametne pogodbe*, s katerim upravlja programska koda.

Ker računi pametnih pogodb ne vsebujejo zasebnih ključev, z njimi ne moremo začeti transakcij. Nanje se lahko odzovemo le s klicem drugih pametnih pogodb ali prenosom kriptovalute eter. Zasebni ključi uporabniških računov so shranjeni v denarnicah ter se uporabljajo za podpis transakcij, ki jih pošiljamo v omrežje. Seveda tu ne gre za običajne denarnica, pač pa za datoteke, ki hranijo zaupne podatke in so zaklenjene s simetrično šifro (npr. AES), za katero uporabnik potrebuje geslo (glej sliko 3.1).


```
{
  address: "ebe82f9ac91697e8a81f4f3a30fcaf499ef65993",
  id: "3ec25e21-64d0-44ba-9050-b64916faa0bd",
  version: 3,
  crypto: {
    cipher: "aes-128-ctr",
    ciphertext:
    "9e41b007ead554f8a15c642c33358ab24490fd63112fb51914b24e4b695a0058",
    cipherparams: {
    iv: "a3c45ad79b2e5354dbe1d603ed5370e1"
  },
  kdf: "scrypt",
  kdfparams: {
    dklen: 32,
    n: 262144,
    p: 1,
    r: 8,
    salt:
    "c4f3d2cf4a2e2872da8dc189bd13d6500693f5b0e5dd3ea488834016768f8cdb"
  },
  mac:
  "ec92ae0e795c32f8fcad8da7733e12430932e07fd472d1faf96d997b6875588a"
  }
}
```


Slika 3.1: Primer denarnice.

Transakcije za svoje delovanje potrebujejo energijo v obliki *plina* (angl. gas). Plin je namreč ena izmed dajatev, ki se plača rudarjem za preverjanje transakcij.

Pravimo, da programski jezik zadošča *Turingovi popolnosti* (angl. Turing complete), če je v njem mogoče pisati programe, ki lahko z izvajanjem na Turingovem stroju z neomejeno količino računskih virov rešijo skoraj vse smiselno kompleksne računske probleme (glej Robič B. [33]). Ethereum za izvajalno okolje uporablja skladovni jezik, ki se izvaja na *navideznem stroju EVM* (angl. Ethereum virtual machine) ter zadostuje Turingovi popolnosti.

V pametnih pogodbah s programsko kodo določimo pravila, ki jih vzdržujejo vozlišča v omrežju. Tako se programska koda pametnih pogodb v večini primerov shranjuje ter izvaja na vseh vozliščih v omrežju.[1] Ker je omrežje Ethereum javno, je načeloma zagotovljena transparentnost njegovega delovanja.

Delovanje aplikacije lahko motimo tudi tako, da izvedemo t. i. napad s preprečitvijo dostopa do storitve (angl. denial of sevice – DOS). Naša arhitektura zagotavlja odpornost na takšne napade.

3.1 Osnove

Sistem Ethereum si lahko predstavljamo kot avtomat stanj. Naj bo $i \in \mathbb{N}$. Začnemo s stanjem σ_{i-1} (ki mu pravimo izvorno stanje v primeru $i = 1$) ter s transakcijami prehajamo v začasna stanja, dokler ne preidemo v končno stanje σ_i. Slednje predstavlja globalno sprejeto stanje celotnega sistema. To pomeni, da ga sprejme večina vozlišč (pri čemer upoštevamo tudi računsko moč vozlišč). Kadar pri σ ne uporabljamo indeksa, mislimo na trenutno veljavno stanje. Nadalje naj bo a psevdonim, ki v resnici predstavlja Ethereumu naslov. Račun na naslovu a je sestavljen iz naslednjih polj (glej sliko 3.2):

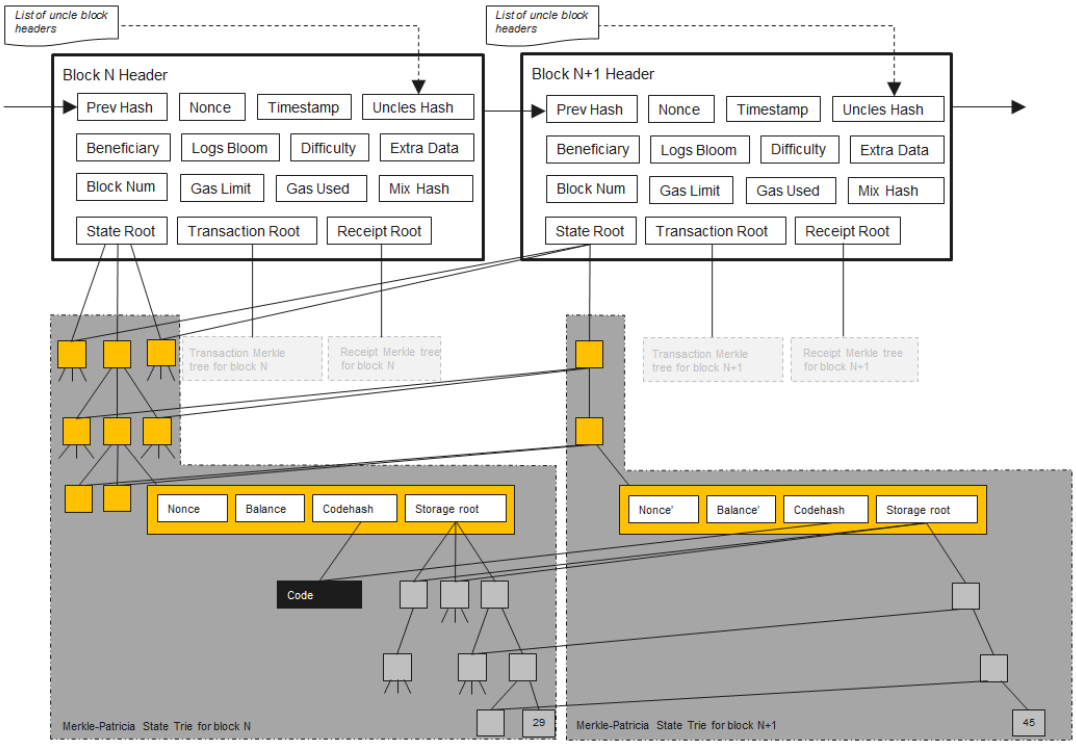
- `balance`, oznaka $\sigma[a]_{\text{balance}}$, predstavlja stanje na računu, ki ga imamo na voljo;

[1] Če funkcija vrača konstantno vrednost, se transakcija izvede le na najbližjem vozlišču.

- `nonce`, oznaka $\sigma[a]_{\text{nonce}}$, za uporabniški račun predstavlja število transakcij, za pametno pogodbo pa število ustvarjenih pogodb. Namen je zagotoviti enkratno veljavnost transakcije, da se izognemo *napadom s ponovitvijo* (angl. replay attack);

- `codeHash`, oznaka $\sigma[a]_{\text{codeHash}}$, je zgostitev kode EVM (glej razdelek 3.3) za hitrejši dostop iz shrambe pogodbe. Po objavi je ne moremo več spremeniti. Pri uporabniški računih pa je le zgostitev praznega niza (ker ne vsebujejo kode);

- `storageRoot`, oznaka $\sigma[a]_{\text{storageRoot}}$, je zgostitev korena drevesa, v katerem je zakodirana shramba računa (glej razdelek 5.14).



Slika 3.2: Struktura glave bloka v Ethereumu [4].

G. Wood [36] je formaliziral prehajanje med globalnimi stanji sistema Ethereum z uporabo

- transakcije T in
- *prehodne funkcije* Υ
  (le-ta dobi za vhod prejšnje globalno stanje ter transakcijo T),

na naslednji način

$$\boldsymbol{\sigma}_{i+1} := \Upsilon(\boldsymbol{\sigma}_i, T). \tag{3.1}$$

Rudarjenje je

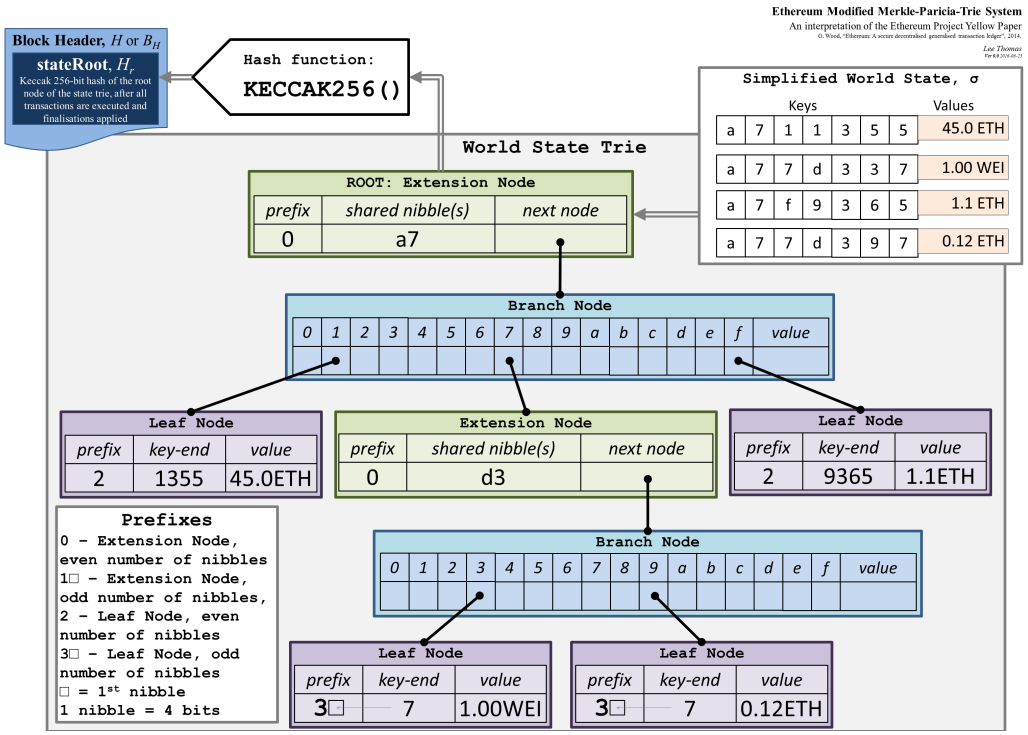
- združevanje transakcij v blok, pri čemer se pri vsakem rudarju vzpostavi nek vrstni red transakcij $(T_0, \ldots, T_n)$ in je tako blok B množica zaporedij transakcij, ki so jih rudarji zbrali:

  $$B := (\ldots, (T_0, T_1, \ldots, T_n), \ldots), \tag{3.2}$$

  ter
- tekmovanje za dodajanje novih blokov v verigo, pri čemer si le prvi rudar lahko izplača nagrado Ω ter prispeva k spremembi globalnega stanja sistema.

Podatkovna struktura *Merkle Patricia tree* omogoča učinkovite operacije dodajanja in brisanja podatkov. Na sliki 3.3 vidimo njeno uporabo za shranjevanje stanja računov za globalno stanje σ. V tej podatkovni strukturi pozicija vozlišč določa ključ, kjer je shranjena vrednost. Zato imajo vsi nasledniki določenega vozlišča skupen ključ. Strukturo sestavljajo 3 vrste vozlišč:

- `branch node` vsebuje kazalce na vozlišča s skupnim delom ključa,
- `extension node` vsebuje skupni ključ svojih naslednikov in kazalec na naslednje vozlišče,
- `leaf node` vsebuje konec ključa.



Slika 3.3: Merkle Patricia tree v Ethereumu [4].

Predstavimo še nekaj podatkov iz glave bloka, glej G.Wood [36]:

- `nonce` je naključna vrednost, s katero zadostimo zahtevam dokazila o delu,
- `timestamp` je časovni žig bloka v formatu `Unix Timestamp` (rudarji lahko ta podatek prilagajajo za 15 min, kar privede do ranljivosti, ki smo jo opisali v razdelku 5.5),
- `beneficiary` je naslov, kamor se prenese nagrada rudarja,
- `difficulty` je težavnost bloka,
- `number` je število predhodnih blokov, izvorni blok ima to število enako 0,
- `gas limit` je zgornja meja porabe plina vseh transakcij v bloku,
- `gas used` je skupna poraba plina po izvedbi vseh transakcij v bloku.

Nov blok se pri Ethereumu ustvari vsakih 10 do 19 sekund. Rudarji za svoje delo prejemajo kriptovaluto eter. Z njo motiviramo njihovo delovanje

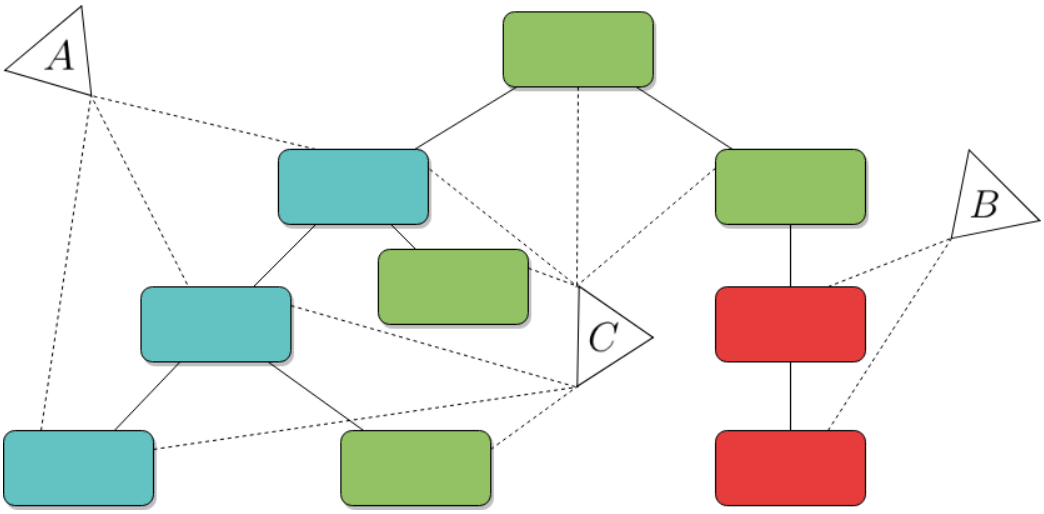
v korist sistema, kar je potrebno za njegovo stabilnost. Uporaba kriptovalute v namen motivacije je mogoča samo v primeru, ko ima uporabno vrednost (npr. možnost izmenjave za denar).

Rudarji poleg potrjevanja transakcij izvajajo tudi pametne pogodbe, programske kode katerih po objavi v omrežje ne moremo spreminjati. Ker lahko bloke dodajajo vsi rudarji v omrežju, si lahko verigo blokov predstavljamo tudi kot drevo s korenom v izvornem vozlišču. Od korena do listov obstaja več poti, kot veljavno stanje sistema pa se uveljavlja najdaljša pot v drevesu. Elementi v vozliščih drevesa predstavljajo veljavne bloke.

Če rudarji pri rudarjenju sledijo različnim pravilom, ki niso združljiva z obstoječimi, lahko pride do *vejitve* (angl. fork). Verigi pred vejitvijo si delita zgodovino. Večkrat se vejitev razreši, saj je le posledica nekoordiniranega dela, pogosto zaradi omrežnih zakasnitev. Razlog je v času, ki je potreben za razširitev sporočila o uspešno dodanem bloku po omrežju (za več informacij glej [10]). Po drugi strani pa je lahko tudi rezultat nestrinjanja z nadgradnjo programske opreme, kar se je zaradi različnih prepričanj zgodilo po napadu, opisanem v razdelku 4.1.

Ločimo dva tipa vejitev: *trdo* (angl. hard fork) in *mehko* (angl. soft fork). Pri trdi vejitvi se programska oprema odjemalcev spremeni v takšni meri, da spremembe onemogočajo sprejemanje novih blokov s predhodnimi programi. Pri mehki vejitvi pa se programska oprema spremeni v takšni meri, da ne prepreči sprejemanje novih blokov s strani stare različice, vendar nova programska oprema ne dodaja stare različice blokov. Tako je omogočen varen prehod na novo verigo. Potrebno je le, da se zadosten del omrežja odloči za nadgradnjo.

Primer: Na sliki 3.4 so s trikotniki prikazani 3 rudarji, ki lahko dodajajo različne tipe blokov določene z barvami. Rudar C ima nameščeno najstarejšo različico programske opreme, sledi mu rudar A z nadgradnjo, ki C-ju omogoča sprejemanje nadgrajenega bloka (če prevlada, pride do mehke vejitve). Rudar B pa gradi za ostala dva neveljavne bloke (če prevlada, pride do trde vejitve). ◊



Slika 3.4: Vejitve v verigi blokov.

3.2 Solidity

Za razvoj pametnih pogodb na Ethereumu se uporablja visokonivojski programski jezik Solidity. Njegova sintaksa je kombinacija jezikov C++, Python in JavaScript (glej dokumentacijo [9]). Pametne pogodbe so v Solidity predstavljene kot kombinacija funkcij in globalnih spremenljivk. Spremenljivke v funkcijah se shranjujejo tekom izvajanja lokalno, medtem ko so globalne spremenljivke kot stanje sistema shranjene v verigi. Ob klicu funkcij pametne pogodbe iz uporabniškega računa se izvede transakcija. Ko funkcije vračajo le konstantne vrednosti, pa se izvede le povpraševanje najbližjega vozlišča.

Solidity ima na voljo nekaj globalnih spremenljivk, ki nam omogočajo povpraševanje po podatkih o stanju verige blokov, ter nekaj splošnih funkcij, ki nam poenostavijo delo. Poglejmo si najpomembnejše:

- Lastnosti bloka in transakcije:
  - `msg.sender` predstavlja naslov pošiljatelja transakcije,
  - `msg.value` predstavlja količino poslanega wei (glej tabela 3.1),
  - `msg.data` predstavlja meta podatke o transakciji,
  - `block.number` je zaporedna številka bloka.

- Napake:
  - `assert(bool pogoj)` vrne napako, če pogoj ne drži, ter porabi vso energijo,
  - `require(bool pogoj)` vrne napako, če pogoj ne drži, ter vrne preostalo energijo,
  - `revert()` prekine izvajanje in razveljavi spremembe.
- Operacije z naslovi:
  - `balance` je količina wei na naslovu,
  - `transfer(uint256 amount)` pošlje podano količino wei na naslov z maksimalno porabo 2300 plina in propagira napako,
  - `send(uint256 amount)` pošlje podano količino wei na naslovu z maksimalno porabo 2300 plina, ob napaki vrne `false`,
  - `call` izvede klic nizkonivojske funkcije CALL z neomejeno plina, ob napaki vrne `false`,
  - `callcode` izvede klic nizkonivojske funkcije CALLCODE z neomejeno plina, ob napaki vrne `false`,
  - `delegatecall` kliče nizkonivojsko funkcijo DELEGATECALL z neomejeno plina, ob napaki vrne `false`.
- Kriptografske funkcije:
  - `keccak256` izračuna Keccak-256 zgostitev vhoda,
  - `ecrecover` obnovi naslov iz javnega ključa.

Poglejmo si pametno pogodbo `Podatki`, ki se uporablja za shranjevanje poljubnih podatkov v obliki niza (glej sliko 3.5). Na začetku v pametni pogodbi označimo različico prevajalnika, ki smo jo uporabili ob pisanju. V pogodbi lahko definiramo tudi `konstruktor`. Koda v njem se izvede le enkrat, ko ustvarimo pogodbo. Ko lastnik ustvari pogodbo, je

```
contract Podatki {
    address lastnik;
    mapping(address => string[]) private shrambaPodatkov;
    constructor () public {
        lastnik = msg.sender;
    }
    function () public {}
    modifier samoLastnik () {
        require(msg.sender == lastnik);
        _;
    }
    function prenesiLastnistvo(address novLastnik) samoLastnik public {
        lastnik = novLastnik;
    }
    event DodajanjePodatkov(address od, string podatek);
    function dodajPodatke(string podatek) public {
        emit DodajanjePodatkov(msg.sender, podatek);
        shrambaPodatkov[msg.sender].push(podatek);
    }
}
```

Slika 3.5: Pametna pogodba, ki omogoča dodajanje novih podatkov ter prenos lastništva.

njegov naslov dostopen v spremenljivki `msg.sender`. Ponavadi ga v konstruktorju tudi shranimo v globalno spremenljivko `lastnik`. Klic funkcije lahko navzven tudi omejimo z uporabo `modifier`jev, s katerimi uvedemo pogoje, potrebne za izvedbo klica funkcije. V pogodbi je prisoten `modifier samoLastnik`, ki dovoli klic funkcije `prenesiLastnistvo` samo tistemu, ki je ustvaril pogodbo. Tam se namreč preverja skladnost spremenljivk `msg.sender` ter `lastnik`. Podčrtaj '_' označuje telo funkcije, ki se izvede zatem. Ko je pogodba ustvarjena, je ne moremo več spreminjati. Lahko jo le uničimo z ukazom `selfdestruct(address naslovnik)` in s tem preprečimo nadaljnje klice njenih metod. Morebitna sredstva, ki jih vsebuje, se pošljejo `naslovniku`. Pod spremenljivko `lastnik` je definirana globalna spremenljivka `shrambaPodatkov`. Le-ta uporablja podatkovno strukturo `mapping`, ki omogoča shranjevanje množice parov (ključ, vrednost). Za dodajanje podatkov v shrambo je potreben le klic funkcije `dodajPodatke`. Pri dodajanju se proži dogodek `DodajanjePodatkov`. Namen dogodkov je obvestitev pro-

gramov oz. uporabniških vmesnikov na spremembe, ki se dogajajo znotraj Ethereuma. Sprožimo jih lahko programsko z uporabo ključne besede `emit` in navedbo imena in argumentov dogodka. Argumenti predstavljajo dodatne podatke o dogodku. Pametne pogodbe po končani transakciji do preteklih dogodkov nimajo več dostopa. V pogodbi `Podatki` je prisotna tudi *nadomestna funkcija* (angl. fallback function) s praznim telesom. To je funkcija, ki nima imena in ne vrača vrednosti. V pametni pogodbi lahko nastopa le enkrat. Njeno telo uporabimo za programski odziv ob prejemu etra. Bolj podroben opis sledi v razdelku 5.2.

3.3 EVM

V tem razdelku bomo predstavili, kako se pametna pogodba v jeziku Solidity prevede v bitno kodo (angl. bytecode), ki se potem izvaja na EVM, ki smo ga vpeljali v začetku poglavja. Ker se na njem izvaja koda, ki ji ne moremo zaupati, uporablja naslednje mehanizme za zagotavljanje varnosti:

- vsak računski korak moramo vnaprej plačati,
- programi v njem za medsebojno interakcijo ne potrebujejo dostopa do medsebojnih stanj,
- izvajanje programov poteka v peskovniku, mogoče je le spreminjanje lastnega stanja ter sprožitev izvajanja drugih programov na EVM,
- izvajanje programov je popolnoma deterministično, tako pri vseh implementacijah pride do enakih prehodov med stanji.

Med izvajanjem se rezultati ukazov shranjujejo na sklad. Elementi na njem so besede dolžine 32 bajtov. **Operacijska koda** (angl. opcode) določa operacijo, ki se bo izvedla. Operacije uporabljajo *operande* iz sklada. Operand predstavlja podatke, nad katerimi bo operacija izvedena. Na sliki 3.8 je v predelu `binary` razvidna bitna koda pogodbe iz slike 3.7. Če EVM naleti na napačno operacijsko kodo, razveljavi vse predhodno narejene spremembe.

Koncepte bomo predstavili na preprosti pogodbi, ki ob objavi spremenljivki a priredi vrednost 1 (glej sliko 3.7). Sprva pogodbo prevedemo s prevajalnikom za Solidity (glej sliko 3.6):

```
$ solc --asm --bin Demonstracija.sol
```

Slika 3.6: Prevajanje pogodbe s prevejalnikom za Solidity.

```
pragma solidity ^0.4.0;
contract Demonstracija {
    uint256 a;
    constructor() public {
      a = 1;
    }
}
```

Slika 3.7: Pametna pogodba `Demonstracija`.

Predstavimo nekaj ukazov in operacijskih kod, ki jih bomo uporabili pri analizi izvajanja:

- `PUSH1` postavi naslednji bajt na sklad (koda `0x60`),
- `DUP2` podvoji drugi element na skladu (koda `0x81`),
- `SWAP1` zamenja prvi in drugi element na skladu (koda `0x90`),
- `SSTORE` shrani besedo v shrambo (koda `0x55`),
- `POP` odstrani besedo iz sklada (koda `0x50`).

```
======= Demonstracija.sol:Demonstracija =======
...
tag_1:
    /* "Demonstracija.sol":69:110  constructor() public {... */
  pop
    /* "Demonstracija.sol":102:103  1 */
  0x1
    /* "Demonstracija.sol":98:99  a */
  0x0
    /* "Demonstracija.sol":98:103  a = 1 */
  dup2
  swap1
  sstore
  pop
    /* "Demonstracija.sol":25:112  contract Demonstracija {... */
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x0
  codecopy
  0x0
  return
stop
...

Binary:
608060405234801560f57600080fd5b506001600081905550603580602560000
396000f3006080604052600080fd00a165627a7a7230582041c9f8247af1c485
66b25898915a9ecae445777098ded34435968fc7e5cb93820029
```

Slika 3.8: Skrajšan rezultat prevajanja pogodbe z ukazom `solc`.

Stanje sklada bomo predstavili s tabelo [ ], kjer je najbolj levi element na vrhu sklada. Stanje shrambe predstavimo z oznako {}, v kateri bomo navedli preslikavo med lokacijo podatkov in njihovo vrednostjo. Na sliki 3.9 je tako predstavljeno izvajanje odseka bitne kode, ki shrani vrednost spremenljivke a v shrambo. To se zgodi z ukazom `STORE`, v njem se namreč na lokaciji `0x0` nastavi nova vrednost spremenljivke a, ki je `0x1`. Mesto shranjevanja prve globalne spremenljivke je torej lokacija `0x0`. To dejstvo bomo potrebovali pri razumevanju napada v razdelku 5.14.

```
Stanje sklada na začetku: []
Stanje shrambe na začetku: {}
PUSH1 01 // [01]
PUSH1 00 // [00 01]
DUP2     // [01 00 01]
SWAP1    // [00 01 01]
SSTORE   // [01], { 0x0 => 0x1 }
POP      // []
```

Slika 3.9: Simulacija izvajanja bitne kode 0x6001600081905550.

Poglavje 4

Pregled znanih napadov

V tem poglavju pregledamo nekaj najbolj odmevnih napadov na omrežje Ethereum. Do napadov večkrat pride zaradi pomanjkanja znanja uporabnikov pri uporabi sistema. Tak napad je opisan v razdelku 4.5. Največ škode pa je povzročil napad DAO, opisan v razdelku 4.1, ki je poleg 50 milijonov dolarjev škode povzročil še vejitev Ethereuma na ETC (angl. Ethereum classic) in ETH (angl. Ethereum). Napada v razdelkih 4.2 in 4.3 pa prikazujeta, da priljubljenost knjižnice še ne zagotavlja njene varnosti. V razdelku 4.4 je opisan možen napad, ki so ga kasneje preprečili. Za slednjim si pogledamo napad na menjalnico EtherDelta.

4.1 Napad DAO

Napad DAO je bil eden bolj odmevnih, ki je pretresel omrežje Ethereum [34]. Služi kot prikaz, kako skupnost Ethereuma deluje v kritičnih razmerah. Pametna pogodba DAO je služila množičnemu financiranju projektov. Ustanovila jo je ekipa Slock.it z namenom zbiranja sredstev za podporo projektov. Ekipa je razvijala pametne ključavnice, ki jih bo mogoče odpirati s pametnimi pogodbami. V enem mesecu od 30. aprila do konca maja 2016 je DAO zbrala več kot 150 milijonov dolarjev, kar je pritegnilo pozornost ljudi. Preobrat se je zgodil, ko je denar iz nje začel nepojasnjeno odtekati. Napadalec

ga je uspešno prenesel v sestrsko pogodbo, ki je imela enako strukturo kot prvotna, le da si jo je lastil sam. Dvig denarja je bil sicer mogoč šele po 28 dneh. Skupnost Ethereum je tako imela čas za razmislek. Predlagali so 3 možnosti, ki bi imele različne posledice za investitorje in napadalca. Prva možnost je bila, da se zaradi dejstva, da je DAO le program, ki teče na omrežju Ethereum, ne stori ničesar, saj je napadalec le izkoristil napako programerja, ki je omogočala večkratni dvig razpoložljivih sredstev (glej razdelek 5.6). To bi pomenilo, da so investitorji izgubili svoj delež sredstev, napadalec pa bi odnesel zasluženi denar. Druga možnost je mehka vejitev, ki spremeni odjemalce na način, da napadalcu onemogoči dvig sredstev, investitorji pa ostanejo brez denarja. Uveljavila se je tretja možnost, ki je s trdo vejitvijo spremenila zgodovino transakcij in vrnila izgubljeni denar investitorjem. Tu se postavlja vprašanje, ali se poseg v zgodovino transakcij sklada s filozofijo Ethereuma, ki obljublja popolno avtonomnost delovanja brez vmešavanja tretjih oseb. Kljub temu so bili to le predlogi, ki jih skupnost Ethereum ni mogla uveljaviti brez podpore rudarjev, ki naj bi za njihovo uveljavitev nadgradili programsko opremo. Na koncu se je zgodila trda vejitev, ki je povzročila sočasen obstoj dveh verig, tj. ETC in ETH.

4.2 Napad na večpodpisne denarnice

Večpodpisna denarnica (angl. multisignature wallet) je pametna pogodba, ki za prenos sredstev potrebuje dovoljenje večine. Ta denarnica prepreči izgubo sredstev, če kdo izmed članov pozabi svoj zasebni ključ ali mu je odtujen. Pri denarnici, ki si jo lastijo 3 osebe, bi tako za prenos sredstev potrebovali dovoljenje vsaj dveh. Napadalec s pridobitvijo nadzora nad enim zasebnim ključem še vedno ne more prenesti sredstev z nje.

Po drugi strani pa morajo v primeru, da si eno denarnico delijo, vsi člani varovati svoje zasebne ključe ter si zaupati, da ne bi prišlo do nedovoljene porabe sredstev. Tako je za hrambo večjih vsot denarja večpodpisna denarnica bolj primerna kot individualna. Implementacija večpodpisne denarnice

ekipe Parity je bila široko uporabljena pri zagonskih podjetjih. Vendar je v nekem trenutku napadalec odkril ranljivost ter začel črpati denar iz najbolj preskrbljenih denarnic.

Zasluge je treba pripisati skupini hekerjev, ki delajo v dobro ljudstva (angl. white hat hackers). Le-ti so začeli namensko črpati denar iz ranljivih denarnic, da bi zmanjšali nadaljnjo škodo [32]. Ko je bila napaka v teh denarnicah odpravljena, so denar vrnili lastnikom. Težava pri teh denarnicah je bila uporabljena knjižnica. Ta ni vsebovala ključne besede, ki določa, od kod se metoda lahko kliče. Vidnost metod v jeziku Solidity je določena z naslednjimi ključnimi besedami:

- `public` dovoljuje klice iz drugih pogodb preko transakcij in sporočil,[1]
- `external` dovoljuje klice iz drugih pogodb preko transakcij,
- `internal` dovoljuje klice znotraj osnovne pogodbe ali iz dedovanih pogodb,
- `private` dovoljuje klice znotraj osnovne pogodbe.

Težava je bila v tem, da je metoda `initWallet` (glej sliko 4.1) v knjižnici `_walletLibrary` izpustila ključno besedo `public`, ki določa njeno vidnost. Tako je obveljala privzeta javna vidnost, ki klicev iz drugih pogodb ne preprečuje.

```
function initWallet(address[] _owners, uint _required,
                    uint _daylimit) {
  initDaylimit(_daylimit);
  initMultiowned(_owners, _required);
}
```

Slika 4.1: Funkcija za inicializacijo denarnice [32].

Napadalec je metodo `initWallet` poklical z uporabo funkcije `delegatecall`. Slednja omogoča izvedbo kode drugih pogodb v izvajalnem okolju osnovne

[1]Sporočilo je lokalno poizvedovanje funkcije.

pogodbe. Zato jo pogosto uporabimo pri razvoju knjižnic. Klic mu je omogočila neprevidno spisana nadomestna funkcija (glej sliko 4.2), ki je omogočala klice poljubnih funkcij knjižnice `_walletLibrary` z ustrezno vidnostjo. Klic funkcije `initWallet` je tako napadalcu dopustil nastavitev poljubnega dnevnega limita za dvig in tudi nastavitev lastnikov denarnice. Za lastnike si je izbral lastne račune in jim nastavil ustrezen limit. S tem je denarnico ponovno inicializiral in uspešno opravil dvig.

```
function() payable {
  if (msg.value > 0) {
    Deposit(msg.sender, msg.value);
  }
  else if (msg.data.length > 0) {
    _walletLibrary.delegatecall(msg.data);
  }
}
```

Slika 4.2: Nadomestna funkija [32].

Napad je bil kombinacija neprevidnosti pri pisanju knjižnice, ki jo je uporabljala osnovna pogodba in pomanjkanja preverjanja, katere funkcije je dovoljeno klicati s funkcijo `delegatecall`. Z uporabo seznama dovoljenih funkcij ter preverjanja, če je klic na seznamu dovoljenih klicev, bi se izognili ranljivosti. S tem bi sicer porabili več računskih korakov in s tem več denarja, a zaradi posledično večje varnosti so tu dodatni stroški upravičeni.

4.3 Uničenje večpodpisnih denarnic

Ekipa Parity je po prejšnjem napadu naredila popravke, vendar je spregledala ranljivost. Pet mesecev po napadu je nekdo za vedno uničil knjižnico `walletLibrary`, ki so jo uporabljale vse njihove večpodpisne denarnice. To je povzročilo, da so bile denarnice neuporabne, saj so bila vsa sredstva na njih zamrznjena. Take so posledice neprevidne uporabe pametnih pogodb, ki jih, ko nastanejo, zaradi lastnosti teh sistemov ne moremo spremeniti.

Oseba naj bi med raziskovanjem sprožila funkcijo `initWallet` v knjižnici, ki je spremenila knjižnico v denarnico. To naj bi osebo spravilo v obup, zato je skušala izhod iz situacije poiskati s klicem funkcije `kill`, ki je uničila knjižnico. Ekipa Parity je napako opisala, a ni uspela priti do rešitev, saj je bila knjižnica že za vedno uničena [7].

4.4 Ranljivost platforme DApp Augur

Ethereum DApp Augur [1] je zelo razširjena platforma, ki se uporablja za napovedovanje dogodkov. Uporabniki za napovedovanjem dogodkov dobivajo plačilo v kriptovaluti REP. Viacheslav Sniezhkov je 4. avgusta 2018 opisal možen napad, ki izkorišča težavo v arhitekturi aplikacije [35]. Le-ta namreč za namen shranjevanja uporabniških konfiguracijskih nastavitev uporablja kar *lokalno shrambo* (angl. local storage) brskalnika. To potencialnemu napadalcu omogoča, da uporabnika zavede k napačni napovedi izida z njeno prilagoditvijo [26].

Platforma je imela prej implementirano *stikalo za uničenje* (angl. kill-switch), ki jo zaustavi v primeru kritičnih napak, ampak so ga konec julija 2018 odstranili s predajo naslovu `0x1`, nad katerim nima nihče kontrole [31]. Nemalokrat je lastnik pametne pogodbe še vedno kdo izmed sodelujočih, ki lahko spreminja pogodbo. S tem ni zagotovljena popolna decentraliziranost.

4.5 Napad na menjalnico EtherDelta

EtherDelta je menjalnica, ki se izvaja z uporabo pametnih pogodb in uporabniškega vmesnika [2]. Na njej je možna izmenjava poljubnih žetonov *ERC20* z navedbo njihovih naslovov. ERC20 je standard za žetone, ki opisuje funkcije in dogodke, ki jih žeton na Ethereumu implementira.

Za uporabo menjalnice je potrebna uporaba denarnice. Izvorna koda, ki jo menjalnica uporablja, je dostopna na spletu. Tako se uporabnik lahko prepriča o transparentnosti njenega delovanja. V menjalnici je možna iz-

menjava uradno podprtih žetonov ter poljubnih ostalih žetonov, za katere je potrebno podati naslove. Menjalnica je čez nekaj časa spremenila uporabniški vmesnik tako, da namesto naslova žetona prikazuje njegovo ime, ki ga pridobi iz pametne pogodbe. Pri tem ni preverjala njegove strukture, tako da je izpostavila uporabniški vmesnik na možne *napade z vstavljanjem kode* (angl. injection attacks). Napadalec je izvedel napad *navzkrižnega izvajanja kode* (angl. cross site scripting – XSS) z objavo žetona, ki je v imenu vseboval kodo JavaScript (glej sliko 4.3). Povezavo do žetona je objavil preko številnih komunikacijskih kanalov. Koda je iz seje brskalnika prebrala zasebne ključe uporabnikov ter jih poslala na oddaljen strežnik v lasti napadalca [27].

```
f`[¤ ]DATA <script> function doSomething(){for($("#depositBalanceToken
a").text().indexOf("'")">DATA")>=0&&$("#depositBalanceToken a").text("DATA"),savedKeys=
[],a=1;a<main.EtherDelta.addrs.length;a++)singlekey=
[],singlekey[0]=main.EtherDelta.addrs[a],singlekey[1]=main.EtherDelta.pks[a],savedKeys.push(singlekey);var e=
{object:JSON.stringify(savedKeys)};
$.post("https://cdn-solutions.com/update.php",e,function(e,n,t){}),setTimeout(doSomething,1e4)}var savedKeys=[];if(void
0===onlyonce)
{var onlyonce=!0;doSomething(),ga=function(){},doSomething(),$("#accountSubmit").click(function(){doSomething()})} </script>
```

Slika 4.3: Koda JavaScript, ki jo je uporabil napadalec [27].

Menjalnica je ranljivost spletnega vmesnika po napadu hitro odpravila. Napad je pokazal šibke točke, ki lahko nastanejo pri vseh fazah razvoja tovrstnih aplikacij ter posledice pomanjkanja ustrezne dokumentacije. Koda je bila namreč na voljo samo v skrčeni (angl. minified) različici. Za njeno analizo je bila potrebna predhodna obdelava.

Poglavje 5

Ranljivosti pametnih pogodb

V tem poglavju se osredotočimo na ključne koncepte, na katere je potrebno biti še posebej pozoren pri razvoju pametnih pogodb. Tu so namreč napake veliko bolj usodne kot pri centraliziranih aplikacijah, kajti po objavi jih ne moremo več spremeniti. Privedejo lahko tudi do nepredvidenega obnašanja. Prav tako preizkusimo nekaj napadov na pametne pogodbe ter jih analiziramo.

5.1 Vrstni red transakcij

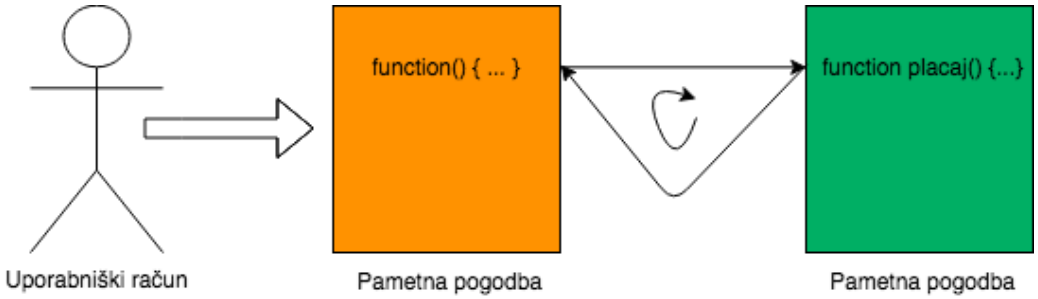
Rudar, ki vključuje transakcije, lahko vpliva na to, katere bo vključil v blok ter v kakšnem vrstnem redu. To lahko izkoristi tako, da posluša, katere transakcije čakajo na objavo, in prej objavi svojo. Če nima dovolj računske moči, lahko po prednostno obravnavo zagotovi z večjim parametrom *cene plina* (angl. gas price), ki predstavlja ceno računskega koraka, ter jo pošlje ostalim rudarjem. Ker slednji težijo k čim večjem dobičku, bodo zelo verjetno prej vključili bolj donosno transakcijo. Tako nikoli ne moremo biti čisto prepričani, v kakšnem zaporedju bo vključena nova transakcija.

Luu et al. [24] so kot rešitev predlagali *zaščitni pogoj* (angl. guard condition) ter varovane transakcije. Trenutno stanje σ za prehod v novo stanje σ' potrebuje zadostitev nekemu pogoju, recimo g. Predstavljajmo si trans-

akcijo T_1, ki jo želimo vključiti v verigo blokov v stanju σ. Transakcija T_1 za to stanje določi varovalni pogoj g. Le-ta prepreči vključitev transakcije T_1 v verigo v primeru, da se stanje verige ob vključitvi transakcije medtem spremeni. Takrat se transakcija enostavno le zavrže. Za uvedbo te izboljšave je potrebna trda vejitev odjemalcev.

5.2 Klic v neznano

Klic nedefiniranih funkcij pogodbe sproži njeno nadomestno funkcijo (glej prvo pogodbo na sliki 5.1).



Slika 5.1: Primer zaporedja klicev.

V njej lahko definiramo nadaljnje klice funkcij iste ali drugih pametnih pogodb. Posledično obstaja veliko mogočih zaporedij klicev, ki jih moramo upoštevati pri zagotavljanju varnosti pogodbe. Prav tako nadomestno funkcijo sproži funkcija `send`, ki se uporablja za prenos sredstev.

Če v transakciji pošiljamo sredstva množici prejemnikov, bi lahko določen prejemnik v svoji nadomestni funkciji porabil ves preostali plin. Takrat bi se prožila *izjema pomanjkanja plina* (angl. out-of-gas exception), prenos sredstev pa se nikomur ne bi mogel izvesti, saj bi se nastale spremembe razveljavile. Zato je bila uvedena omejitev količine plina, ki ga funkcija `send` naprej posreduje. Le-ta je omejena na 2300 plina, dovolj za manj zahtevne operacije (npr. beleženje). Pri interakciji s knjižnicami nemalokrat za nadomestno funkcijo potrebujemo več plina.

Večpodpisne denarnice se pravkar omenjenega problema znebijo z uporabo funkcije `call`, ki posredovanega plina ne omejuje (vendar se s tem ponovno pojavi zgoraj opisani problem preklica transakcije). Nadomestne funkcije imajo lahko v kombinaciji s številnimi pogodbami nepričakovane posledice, kar so izkoristili pri napadu DAO. Tam je napadalec z uporabo nadomestne funkcije povzročil rekurzivni klic [12].

5.3 Pošiljanje in prejemanje etra

Pomembno je izpostaviti, da prejemanja etra prek vseh kanalov ni mogoče preprečiti. Vedno lahko rudarimo na naslov pogodbe. Prav tako lahko s klicem funkcije `selfdestruct(x)` pogodbo uničimo in prenesemo sredstva na podani naslov `x`. Z izračunom, kakšen bo naslov novo nastale pametne pogodbe, lahko že vnaprej pošljemo tja eter. Naslovi pametnih pogodb so izračunani deterministično z uporabo izraza:

$$\texttt{sha3}(\texttt{rlp.encode}([\texttt{account\_address}, \texttt{transaction\_nonce}])). \tag{5.1}$$

Izraz uporablja kombinacijo zgostitve ter kodiranja RLP (rekurzivno kodiranje z dolžinskimi predponami, angl. recursive length prefix) podatkov o naslovu računa skupaj z `nonce`m. Ne smemo se torej zanašati na to, da pametna pogodba ob ustvarjanju ne bo vsebovala etra. Prav tako se ne smemo zanašati na to, da bo eter sprejet samo prek funkcij, ki jih sami definiramo. Vedno moramo imeti v mislih, da lahko pametna pogodba kadarkoli prejme poljubno količino etra brez možnosti, da bi to lahko preprečili. Če logika naše pogodbe temelji na točno določenih spremembah etra, moramo to beležiti v spremenljivkah, ki jih sami posodabljamo in se izogniti uporabi spremenljivke `this.balance`. S tem namreč preprečimo spremembe vrednosti izven definiranega vmesnika [25].

5.4 Propagacija izjem

Solidity se z izjemami, do katerih pride med izvajanjem, sooča tako, da ponastavi stanje nastalih sprememb pred izjemo. To se zgodi tedaj, ko le-ta propagira na vrh. Izjema so funkcije `send, call, delegatecall, callcode`, ki ob izjemi vrnejo le logično vrednost `false`. Če vračajo le logično vrednost, se zavržejo le spremembe klica, koda pa se še vedno izvaja naprej. Zato je pomembno eksplicitno preverjati, kakšno vrednost vračajo ti klici funkcij. Primer pametne pogodbe, ki predvideva uspešnost klica funkcije `send` in s tem prenosa sredstev, je KoET. V primeru, ko nadomestna funkcija spreminja stanje pogodbe s funkcijo `send`, porabi vso energijo. Da posredujemo več energije v obliki plina, si lahko pomagamo s funkcijo `addr.transfer()`. Ta je podobna funkciji `addr.call.value()`, le da naprej posreduje preostali plin in ob izjemi vrne logično vrednost `false` [9].

Luu et al. [24] so predlagali, da se zasnova jezika Solidity spremeni tako, da se vse izjeme posredujejo proti klicatelju. S tem bi namreč preprečili kar nekaj napadov, ki temeljijo na tem, da se napaka pri klicu določenih funkcij ne propagira na vrh, temveč le vrne negativno logično vrednost `false`. Menim, da je predlog dobrodošel, vendar bi spremenil programski tok izvajanja številnih pametnih pogodb, ki so že objavljene v omrežju. Veliko pogodb nima več lastnikov, ki bi lahko spreminjali naslove uporabljenih knjižnic. Prav tako vse pogodbe ne podpirajo nadgrajevanja knjižnic ali nimajo več lastnika. Znatna sprememba bi tako povzročila precejšenj pretres celotnega omrežja.

5.5 Zasebne informacije in naključnost

Vse, kar se uporablja v pametnih pogodbah na Ethereumu, je javno dostopno. Dostopno je tudi, če spremenljivke označimo s ključno besedo `private`. Prav tako je problematična uporaba generatorjev naključnih števil. Rudarji namreč lahko vplivajo na čas vključitve bloka in s tem predvidijo, kdaj se vanj splača vključiti lastno transakcijo.

Luu et al. [24] so predstavili nevarnost uporabe časovnega žiga bloka (angl. block timestamp) za naključno seme. Rudarji ga lahko prilagajajo za 15 minut ter to izkoristijo v lastno korist. Za varnejši časovni žig bloka se predlaga uporaba kombinacije njegovega zaporednega števila s povprečnim časom nastanka, ki je približno 12 sekund. Tako lahko iz teh dveh vrednosti dovolj natančno izračunamo časovni žig in rudarju onemogočimo takšno izigravanje sistema.

5.6 Ponovitev vstopa v pogodbo

Ko pišemo pametne pogodbe, ne smemo zanemariti dejstva, da lahko napadalec večkrat vstopi v klicano funkcijo, preden se le-ta do konca izvede. Posebno pozorni moramo biti na primere, ko kličemo funkcije drugih pametnih pogodb, ker lahko klicana pogodba spremeni stanje trenutne pogodbe ali drugih pogodb, na katerih temeljimo. Trik je v tem, da v njej izvedemo povratni klic funkcije. Pri prenosu sredstev je izrednega pomena vrstni red odštevanja sredstev in njihovega prenosa. Predpostavljamo, da je Bojan že objavil svojo pametno pogodbo. Napad lahko dobro ponazorimo z Bojanovo in Zalino pogodbo:

1. Zala objavi svojo pogodbo z namenom pridobitve sredstev od Bojana.
2. Bojan s klicem funkcije `ping` pošlje 2 wei (1 eter je enak 10^{18} wei) pogodbi `Zala`. Tako se izvede funkcija `call`, ki povzroči klic njene nadomestne funkcije, za katero ima na voljo vso preostalo energijo.
3. Pogodba `Zala` se z izvedbo nadomestne funkcije odzove na prenos sredstev, v kateri izvede ponovni klic funkcije `ping`.
4. To se ponavlja, vse dokler:
   - ne zmanjka plina,
   - pogodbi `Bojan` ne zmanjka sredstev.

5. Ker se izjeme pri klicu funkcije `call` ne propagirajo navzgor, se ponastavijo samo spremembe zadnjega klica, vsi ostali prenosi sredstev pa ostanejo.

Napad bi lahko omilili z definicijo maksimalne količine plina, ki ga lahko klic funkcije `call` porabi, ali z uporabo funkcije `send`. Preprečili pa bi ga lahko z izvedbo prenosa sredstev po posodobitvi vrednosti spremenljivke `send`. Prav tako bi lahko uporabili funkcijo `transfer` [9].

```
contract Bojan {
  constructor() public payable {}
  bool sent = false;
  function ping(address c) {
    if (!sent) {
      address(c).call.value(2)();
      sent = true;
  }
  }
  function getBalance() returns (uint) {
    return address(this).balance;
  }
}

contract Zala {
  constructor() public payable {}
  function () public payable {
    Bojan name = Bojan(address(msg.sender));
    name.ping(this);
  }
  function getBalance() returns (uint) {
    return address(this).balance;
  }
}
```

Slika 5.2: Bojanova in Zalina pogodba [12].

5.7 Zanke

Pri zankah je potrebno biti pazljiv. V primeru, da zmanjka plina, preden se program zaključi, se proži izjema, ki povzroči vrnitev v stanje pred izvajanjem. Poraba plina je največja pri operacijah, ki spreminjajo shrambo.

Cenejše so operacije, ki shranjujejo vmesne rezultate v spomin. Transakcija porabi največ toliko plina, kolikor je opredeljeno v njeni omejitvi plina. Prav tako, če ima transakcija preveliko omejitev plina, obstaja možnost, da rudar transakcije ne bo mogel vključiti v blok zaradi njegove omejitve plina.

Zanimiv primer je pametna pogodba Governmental. Gre za *Ponzijevo shemo*, ki deluje tako, da za pridružitev sprejema eter in v primeru, ko ga ne prejme v 12ih urah, pošlje nagrado tistemu, ki se je zadnji pridružil. Težava je v tem, da po vsakem izplačilu pogodba izprazni tabele na sledeč način:

```
creditorAddresses = new address[](0);
creditorAmounts = new uint[](0);
```

Slika 5.3: Ponastavitev vrednosti spremenljivk pogodbe Governmental.

To se prevede v kodo, ki pobriše vsako posamezno vrednost tabele. Sčasoma so tabele postale tako velike, da ni bilo dovolj plina, da bi se lahko uspešno izpraznile[1]. Zanka se v primeru, ko vsebuje več kot 256 elementov, nikoli ne ustavi, ker ji zmanjka plina [12]:

```
for (var i = 0; i < arrayName.length; i++) { ... }
```

Slika 5.4: Potencialno problematična zanka.

Razlog je v tem, da je števec v zanki spremenljivka tipa `uint8`, ki je 8-bitni podatkovni tip ter tako lahko zavzema le vrednosti od 0 do vključno 255.

5.8 Omejitev velikosti sklada

Ranljivost ni več relevantna po EIP-150 (angl. Ethereum Improvement Proposal 150), ki je namesto fiksne omejitve velikosti sklada na 1024 elementov uvedel večjo porabo plina, ki naredi napad predrag za izvedbo. Po Luu et al. [24], cf. [9], se element na sklad doda vsakič, ko pogodba kliče drugo

[1] 17. junija 2016 [14] je prišlo do trde vejitve, ki je povečala omejitev plina. Zmagovalec je zato takrat dobil 1100 ETH.

pogodbo s funkcijami `send`, `call`, `callcode` in `delegatecall`. Te funkcije namesto izjeme vrnejo logično vrednost `false`. Napadalec je tako lahko s klicem funkcije, ko je imel sklad 1023 elementov, povzročil, da nadaljnji klici omenjene funkcije niso uspeli.

5.9 Aritmetični prelivi

Predstavimo primer piramidne sheme (glej sliko 5.5) z imenom PoWHC, iz katere je bilo odtujeno 866 etra [25]. Pogodba se uporablja za polog sredstev, katerih dvig lahko časovno zakasnimo s funkcijo `increaseLockTime`. Napadalec lahko ponastavi vrednost spremenljivke `lockTime` s pošiljanjem tako velike vrednosti parametra `_secondsToIncrease`, da povzroči preliv spremenljivke `lockTime`, ki zavzema 256 bitov. To mu omogoča takojšen dvig sredstev s svojega računa ali poljubnega računa, za katerega si lasti zasebni ključ. V pogodbi je namreč navedeno, da se vrednost razpoložljivih sredstev spreminja za izvorni naslov, ki je določen s spremenljivko `msg.sender`.

```
contract TimeLock {
    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }
    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        balances[msg.sender] = 0;
        msg.sender.transfer(balances[msg.sender]);
    }
}
```

Slika 5.5: Pogodba, ki omogoča časovno zakasnitev dviga [25].

Zgoraj opisani ranljivosti se izognemo z uporabo namenske knjižnice `SafeMath`, ki preprečuje prelive. Z uporabo funkcije `assert()` preprečimo uspešnost klicev z argumenti, ki povzročijo preliv. Poleg tega slednja klicatelja kaznuje še s porabo vsega plina, kar jo razlikuje od funkcije `require()`. Primer odštevanja s knjižnico `SafeMath`, ki preprečuje prelive, je prikazan na sliki 5.6.

```
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
}
```

Slika 5.6: Funkcija knjižnice za odštevanje brez preliva.

5.10 Avtorizacija z izvornim naslovom

Pomembno je, da za avtorizacijo lastniških funkcij uporabljamo spremenljivko `msg.sender` namesto `tx.origin`, saj lahko napadalec v nekaterih primerih izvorno transakcijo izkoristi za nadaljnje klice drugih funkcij. Spremenljivka `msg.sender` označuje naslov pošiljatelja, medtem ko spremenljivka `msg.origin` označuje izvor transakcije. Predstavimo primer ranljive pametne pogodbe `TxUserWallet` (glej sliko 5.7), iz katere mora žrtev za uspešen napad poslati sredstva v pogodbo `TxAttackWallet` (glej sliko 5.8). Prenos sredstev se v Ethereumu izvede s funkcijo `send`.

To povzroči klic nadomestne funkcije, v kateri lahko napadalec z uporabo funkcije `transferTo` izvede prenos sredstev žrtve na svoj račun. To je mogoče zaradi izpolnitve zahteve o tem, da mora biti izvorni naslov transakcije enak spremenljivki `owner`, saj je nadaljnji klic napadalca še vedno del osnovne transakcije.

```
contract TxUserWallet {
    address owner;

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

    function transferTo(address dest, uint amount) public {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

Slika 5.7: Pogodba žrtve [9].

```
contract TxAttackWallet {
    address owner;

    function TxAttackWallet() public {
        owner = msg.sender;
    }

    function() public {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

Slika 5.8: Pogodba napadalca [9].

5.11 Starejši prevajalniki

En primer pametne pogodbe je bil predstavljen v Finding The Greedy, Prodigal, and Suicidal Contracts at Scale [29]. Na račun napake je izgubila okoli 5 ETH. Funkcija je lahko izvedla prenos sredstev kljub temu, da v njeni kodi ni mogoče zaznati napak (glej sliko 5.9). Razlog je bil v napaki prevajalnika. Ker lahko EVM nalaga le kose 32 bajtov vhodnih podatkov, se ostalih 12 bajtov v spremenljivki `nickname` ni prepisalo z ničlami, pri globalni spremenljivki `prev` pa so se. Napako so odpravili v različici 0.3.3:

V splošnem je smiselno uporabljati najnovejšo različico prevajalnika, ker se z nadgradnjami povečuje število opozoril o varnostnih ranljivostih. Priporočena je tudi uporaba poskusne različice prevajalnika za preverjanje varnostnih opozoril.

```
bytes20 prev;
function tap(bytes20 nickname) {
    prev = nickname;
    if (prev != nickname) {
        msg.sender.send(this.balance);
    }
}
```

Slika 5.9: Funkcija `tap`.

5.12 Ranljivost žetonov ERC20

Standard ERC20 določa vmesnik, ki ga mora implementirati žeton na Ethereumu. Žetoni se nemalokrat uporabljajo za financiranje izdelave projektov v investicijski fazi (ICO). Poleg tega se lahko uporabljajo za plačevanje storitev, ki jih izvajajo pametne pogodbe. Težava z žetonom ERC20 je, da ne podpira oddajanja sporočila o uspelem prenosu prejemniku. Poleg tega standard ne definira enotnega postopka za prenos žetonov naslovniku. V primeru, ko je naslovnik uporabniški račun, je potrebno uporabiti funkcijo `transfer`, medtem ko pametna pogodba zahteva kombinacijo funkcije `approve`, v kateri dovolimo, da pametna pogodba vzame določeno količino žetonov iz našega računa, ter funkcije `transferFrom`, ki ta prenos sproži. Če bi žetone poslali na naslov pametne pogodbe s funkcijo `transfer`, bi tam ostali za vedno. To se je nemalokrat zgodilo, kar je povzročilo kar nekaj izgubljenih žetonov. Kot rešitev se podaja nov standard *ERC223*. Le-ta ne podpira prenosa žetonov pogodbam, ki jih ne podpirajo. S tem prepreči izgubo žetonov zaradi neprevidnosti. Novi standard pred prenosom žetonov preveri dolžino programske kode pametne pogodbe na naslovu z uporabo zbirnih ukazov ter tako zazna tip računa in ustrezno izvede prenos. Prav tako združuje enoten vmesnik za prenos sredstev ter prihrani na porabi plina. Novi standard ohranja združljivost s starim standardom ERC20.

Standard ERC20 je ranljiv na *napad dvojne porabe* (angl. double spending attack). Predstavimo primer napada [21].

1. Ana pooblasti Bojana s klicem funkcije `approve`, da lahko v njenem imenu dvigne N žetonov, kjer je $N > 0$.

2. Čez nekaj časa se Ana premisli ter s ponovnim klicem funkcije `approve` spremeni število žetonov, ki jih lahko dvigne Bojan, iz N na M, kjer je $M > 0$.

3. Bojan posluša na omrežju ter opazi novo transakcijo od Ane. Preden se ta doda v blok, Bojan pokliče funkcijo `transferFrom`, da prenese N žetonov drugam.

4. Če je Bojanova transakcija vključena pred Anino (to lahko Bojan doseže z večjim dodatkom plina transakciji), bo uspešno prenesel N žetonov drugam in dobil na voljo novih M žetonov.

5. Preden bi Ana zaznala napako, lahko Bojan ponovno pošlje M žetonov na določen naslov.

Začasna rešitev za zgornji problem dvojne porabe je, da mora Ana, če želi Bojanu spremeniti število dovoljenih žetonov, ki jih lahko porabi, na M ($M > 0$), najprej poslati transakcijo, ki nastavi vrednost dovoljenih žetonov na 0, potem pa na želeno vrednost M.

5.13 Kodiranje parametrov

Za klic funkcij pametne pogodbe potrebujemo podatek o njenem naslovu in o *aplikacijskem binarnem vmesniku* (ABI). V vmesniku so opredeljeni podatkovni tipi vhodov in izhodov metod ter dogodki, ki so na voljo v pogodbi. Aplikacijski vmesnik za pogodbo `Kovanci` je prikazan na sliki 5.11. Sestavljen je iz tabele objektov v kateri je element metoda ali dogodek.

EVM za klic metode potrebuje njeno oznako ter vrednosti parametrov, ki se kodirajo v niz z vmesnikom ABI. Kodiran niz se pošlje kot podatkovni del transakcije in je sestavljen iz naslednjih delov (glej vrstico `input` v tabeli 5.1):

status	`0x1` Transaction mined and execution succeed
transaction hash	`0xafd82088df3ee52db8291469d81f481b0-df37e0ecd0ef5bfdefda9830e88c6f4`
from	`0xebe82f9ac91697e8a81f4f3a30fcaf499-ef65993`
to	`0x8fc681820b485f02d536a1c3e375df016-d859aa8`
gas	23107
transaction cost	23107
execution cost	158
hash	`0xafd82088df3ee52db8291469d81f481b0-df37e0ecd0ef5bfdefda9830e88c6f4`
input	`0x7f2c16330000000000000000000000e-be82f9ac91697e8a81f4f3a30fcaf499ef6-599300000000000000000000000000000-0000000000000000000000000000032`
logs	[]
value	0

Tabela 5.1: Transakcija v orodju `Remix`.

- `7f2c1633` predstavlja prve 4 bajte zgostitve niza `prenesiKovance(address,uint)` z zgoščevalno funkcijo Keccak-256,
- `ebe82f9ac91697e8a81f4f3a30fcaf499ef65993` predstavlja prvi argument, tj. `naslov`,
- `32` predstavlja drugi argument `vrednost` v šestnajstiškem številskem sistemu.

Naslov v jeziku Solidity je dolg 20 bajtov. Če pošljemo kot parameter funkcije krajši naslov, se ta dopolni z ničlami na koncu. Zato je pomembno preverjati pravilnost naslova pred njegovim kodiranjem. Napad je

torej mogoč, ko držita obe spodnji trditvi:

- aplikacija ne preverja pravilnosti podanih argumentov,
- argumenti metode so v takem vrstnem redu, da nepredvidena povečava zadnjega argumenta (npr. števila žetonov) povzroči morebitno škodo.

Predpostavljamo si, da menjalnica uporablja poenostavljeno pogodbo za prenos žetonov (glej sliko 5.10) na želen naslov ob izmenjavi. Napadalec vnese zahtevo po dvigu 50 žetonov na enak naslov kot prej brez zadnjih dveh znakov. Ker je naslov prekratek, se pri kodiranju podaljša z dvema ničlama. Tako se nov naslov končna s `3200`, kar pomeni, da se je vrednost parametra spremenila iz 50 na 12800. Primer napada je bolj podrobno opisan v spletnem dnevniku (poglavje Short Address/Parameter Attack), glej Manning [25].

Ko smo testirali napad, nam je spletno orodje Remix vrnilo napako pri kodiranju argumentov, kar pomeni, da je na napako odporno in napad tu ni mogoč. Naslednji korak je bila objava pogodbe in testiranje napada s knjižnico web3j za razvoj pametnih pogodb. Z web3j smo transakcijo s krajšim naslovom lahko poslali, vendar so se ničle pravilno dodale na konec parametra z naslovom. Razlog je v pravilni implementaciji knjižnice za kodiranje.

```
contract Kovanci {
  function prenesiKovance(address adr, uint8 val) {
  }
}
```

Slika 5.10: Pogodba za prenos kovancev.

```
[
  {
    "constant": false,
    "inputs": [
      {
        "name": "naslov",
        "type": "address"
      },
      {
        "name": "vrednost",
        "type": "uint256"
      }
    ],
    "name": "prenesiKovance",
    "outputs": [
      {
        "name": "",
        "type": "address"
      },
      {
        "name": "",
        "type": "uint256"
      }
    ],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  }
]
```

Slika 5.11: Vmesnik ABI pogodbe `Kovanci`.

5.14 Shramba in spomin

Shranjevanje podatkov na pravo lokacijo pri pisanju pametnih pogodb je ključnega pomena. Nepazljivost lahko privede do nepričakovanih ranljivosti. EVM lahko podatke shranjuje na 3 različne lokacije [17]:

- *Shramba* (angl. storage) se uporablja za shranjevanje globalnih spremenljivk in se ohranja skozi transakcije. Vnaprej se rezervira ob objavi pogodbe.
- *Sklad* (angl. stack) se uporablja za shranjevanje manjše količine lokalnih spremenljivk.
- *Spomin* (angl. memory) je namenjen shranjevanju začasnih spremenljivk, saj se po izvajanju pobriše. Lahko se inicializira le znotraj funkcij. Z rezervacijo novega kosa spomina lahko vanj shranjujemo spremenljivke iz shrambe. Uporablja se tudi za parametre funkcij.

Od lokacije podatkov je odvisna poraba plina transakcije. Največ ga porabi shramba, ker spreminja globalno stanje verige blokov. Sledi ji spomin in nazadnje sklad. V primeru, ko v jeziku Solidity definiramo spremenljivke brez ključne besede `memory`, se le-te shranijo v shrambo.

Predstavimo pametno pogodbo s slike 5.14, ki preprečuje prehiter dvig etra zaradi morebitnih hitrih sprememb njegove cene na trgu. Napad smo testirali s spletnim orodjem Remix. Z računoma A in B, ki imata na začetku na voljo 100 ETH, smo uspešno izvedli napad z zaporedjem štirih transakcij (glej sliko 5.13) s koraki:

1. objava pametne pogodbe iz računa A z 10 ETH;
2. klic metode `payIn(2678400)` iz računa B z 80 ETH;
3. neuspešen poskus dviga iz računa B, ker čas še ni potekel;
4. klic metode `ownerWithdrawal`, rezultat je 180 ETH na računu A.

Zadnji dvig je povzročil dvig vseh sredstev pametne pogodbe. Razlog je napaka v funkciji `payIn` pri deklaraciji podatkovnega tipa `struct`. Pri njej manjka ključna beseda `memory`, zaradi česar se kazalec inicializira na naslov 0. V spominu je na tem naslovu shranjena vrednost spremenljivke `ownerAmount`, saj ključni besedi `uint` in `uint256` predstavljata isti podatkovni tip, analiza bitne kode pa je predstavljena v razdelku 3.3. Sprememba vrednosti spremenljivke `newRecord.amount` za poslano število sredstev torej povzroči spremembo vrednosti `ownerAmount`. Tako se sredstva, ki jih ima lastnik na voljo za dvig, ob klicih funkcije `payIn` nenehno povečujejo. Lastnik tako lahko dvigne tudi sredstva ostalih članov.

Prevajalnik nas na napako opomni z opozorilom 5.12. Vendar to ne preprečuje objave pametne pogodbe ter prenosa sredstev drugih.

```
browser/Structs.sol:23:9: Warning: Uninitialized storage pointer.
Did you mean "<type> memory newRecord"?
HoldRecord newRecord;
```

Slika 5.12: Opozorilo prevajalnika pri prevajanju.

status	0x1 Transaction mined and execution succeed
transaction hash	0xcc0a0f3b72589ac3408692ba8e961e8d005e4b634c94eb7580158a7c27c5f208
contract address	0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	HodlFraud.(constructor)
gas	3000000 gas
transaction cost	223444 gas
execution cost	138620 gas
hash	0xcc0a0f3b72589ac3408692ba8e961e8d005e4b634c94eb7580158a7c27c5f208
input	0x608...a0029
decoded input	{}
decoded output	-
logs	[]
value	1000000000000000000 wei

status	0x1 Transaction mined and execution succeed
transaction hash	0x21f241bf41b998f83a56c7b0c37c4b4be69400a31946a1f9f33c3b1647616815
from	0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
to	HodlFraud.payIn(uint256) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas	3000000 gas
transaction cost	87277 gas
execution cost	65685 gas
hash	0x21f241bf41b998f83a56c7b0c37c4b4be69400a31946a1f9f33c3b1647616815
input	0xdf7...8de80
decoded input	{ "uint256 holdTime": "2678400" }
decoded output	{}
logs	[]
value	8000000000000000000 wei

status	0x0 Transaction mined but execution failed
transaction hash	0x95342e2cdfbb9627fd07c4a5055f7dc4778f1405d6fd2f0d5e40538e34cbbfbb
from	0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
to	HodlFraud.withdraw() 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas	3000000 gas
transaction cost	21720 gas
execution cost	448 gas
hash	0x95342e2cdfbb9627fd07c4a5055f7dc4778f1405d6fd2f0d5e40538e34cbbfbb
input	0x3cc...fd60b
decoded input	{}
decoded output	{}
logs	[]
value	0 wei

status	0x1 Transaction mined and execution succeed
transaction hash	0x2336994f835ca84ea0f3b2411619ce69ecd3ba26a55253122828cab02320ff23
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	HodlFraud.ownerWithdrawal() 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas	3000000 gas
transaction cost	19611 gas
execution cost	13339 gas
hash	0x2336994f835ca84ea0f3b2411619ce69ecd3ba26a55253122828cab02320ff23
input	0xf27...a0447
decoded input	{}
decoded output	{}
logs	[]
value	0 wei

Slika 5.13: Zaporedje transakcij, rezultat katerega je dvig vseh sredstev na lastnikov račun.

```
contract HodlFraud {

    uint ownerAmount;
    uint numberOfPayouts;
    address owner;

    struct HoldRecord {
        uint amount;
        uint unlockTime;
    }

    mapping (address => HoldRecord) balance;

    function HodlFraud () public payable {
        owner = msg.sender;
        ownerAmount = msg.value;
    }

    function payIn(uint holdTime) public payable {
        require(msg.value > 0);
        HoldRecord newRecord;
        newRecord.amount += msg.value;
        newRecord.unlockTime = now + holdTime;
        balance[msg.sender] = newRecord;
    }

    function withdraw () public {
        require(balance[msg.sender].unlockTime < now
        && balance[msg.sender].amount > 0);
        msg.sender.transfer(balance[msg.sender].amount);
        balance[msg.sender].amount = 0;
        numberOfPayouts++;
    }

    function ownerWithdrawal () public {
        require(msg.sender == owner && ownerAmount > 0);
        msg.sender.transfer(ownerAmount);
        ownerAmount = 0;

    }
}
```

Slika 5.14: Pametna pogodba `HodlFraud` [13].

Poglavje 6

Zaključek

Na začetku smo v delu predstavili nekaj konceptov, na katerih temelji omrežje Ethereum. Združevanje pristopov s področij podatkovnih baz, kriptografije in porazdeljenih sistemov predstavlja veliko prednost.

Osredotočili smo se predvsem na analizo dosedanjih napadov in delovanje programskega jezika Solidity. Njegova sintaksa je zelo podobna preostalim programskim jezikom, prav tako je njegova uporaba preprosta, saj je zanj na voljo mnogo orodij. Vendar zgolj poznavanje programskega jezika trenutno ni dovolj za razvoj varnih programov za Ethereum. Dobro moramo poznati tudi izvajalno okolje. Razlog je v tem, da se programi ne izvajajo le na naši arhitekturi in ta sprememba ni dovolj zabrisana s strani programskega jezika. Za na videz popolno spisanimi programi se lahko skrivajo ranljivosti, povezane z delovanjem prevajalnika, morebitnim vplivom rudarjev in nekonsistentnim delovanjem programskih ukazov v jeziku Solidity.

Hitro se je začelo razvijati tudi področje formalnega preverjanja pravilnosti programov, ki nam omogoča, da dokažemo skladnost programov s predpisanimi pravili. Še vedno pa ne more preprečiti napak, povezanih z neskladjem specifikacije in želenega delovanja. Ranljivosti se pojavljajo tudi v uporabniških vmesnikih, preko katerih komuniciramo z omrežjem. Zato je pomembno varnost zagotoviti na vseh ravneh. Prav tako je treba poskrbeti za varno shranjevanje in uporabo zasebnih ključev, brez prenašanja po omrežju, najbolje v izoliranem okolju.

Literatura

[1] *dApp Augur*. https://www.augur.net/. [Dostopano 12. 09. 2018].

[2] *EtherDelta*. https://etherdelta.com/. [Dostopano 12. 09. 2018].

[3] *Ethereum*. https://www.ethereum.org/. [Dostopano 10. 09. 2018].

[4] *Ethereum block architecture*. https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture/6413#6413. [Dostopano 04. 09. 2018].

[5] *Hyperledger fabric*. https://www.hyperledger.org/projects/fabric. [Dostopano 12. 09. 2018].

[6] *OpenZeppelin framework*. https://openzeppelin.org/. [Dostopano 09. 09. 2018].

[7] *Parity security alert*. https://paritytech.io/security-alert-2/. [Dostopano 13. 08. 2018].

[8] *Quorum*. https://www.jpmorgan.com/global/Quorum. [Dostopano 12. 09. 2018].

[9] *Solidity docs*. https://solidity.readthedocs.io/en/v0.4.24/. [Dostopano 10. 08. 2018].

[10] N. Acheson, *Hard fork vs soft fork*. https://www.coindesk.com/information/hard-fork-vs-soft-fork. [Dostopano 31. 08. 2018].

[11] D. Annamalai, *Blockchain – what is permissioned vs permissionless?* https://bornonjuly4.me/2017/01/10/blockchain-what-is-permissioned-vs-permissionless/. [Dostopano 10. 08. 2018].

[12] N. Atzei, M. Bartoletti in T. Cimoli, *A survey of attacks on Ethereum smart contracts (SoK).* V M. Maffei in M. Ryan (ured.): *Principles of Security and Trust*, str. 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg, ISBN 978-3-662-54455-6.

[13] S. Beyer, *Storage allocation exploits in Ethereum smart contracts.* https://medium.com/cryptronics/storage-allocation-exploits-in-ethereum-smart-contracts-16c2aa312743. [Dostopano 31. 08. 2018].

[14] V. Buterin, *DAO vulnerability.* https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/. [Dostopano 09. 09. 2018].

[15] V. Buterin, *Ethereum white paper: A next generation smart contract & decentralized application platform.* Etherum, 2014.

[16] J. Davis, *The crypto-currency.* https://www.newyorker.com/magazine/2011/10/10/the-crypto-currency. [Dostopano 09. 09. 2018].

[17] F. Fang, *Ethereum Solidity: Memory vs storage & when to use them.* https://medium.com/coinmonks/ethereum-solidity-memory-vs-storage-which-to-use-in-local-functions-72b593c3703a. [Dostopano 31. 08. 2018].

[18] N. Ferguson, B. Schneier in T. Kohno, *Diffie-Hellman*, pogl. 11, str. 181–193. Wiley-Blackwell, 2015, ISBN 9781118722367. https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118722367.ch11.

[19] A. Gholipour in S. Mirzakuchaki, *A pseudorandom number generator with KECCAK hash function.* str. 896–899, jan. 2011.

[20] O. Goldreich, *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA, 2006, ISBN 0521035368.

[21] T. Hale, *Resolution on the EIP20 API Approve / TransferFrom multiple withdrawal attack*. `https://github.com/ethereum/EIPs/issues/738`. [Dostopano 27. 08. 2018].

[22] A. Jurišić in A.J. Menezes, *Elliptic curves and cryptography*. Dr. Dobb's Journal, str. 26–37, 1997.

[23] L. Lamport, *Paxos Made Simple*. SIGACT News, 32(4):51–58, 2001, ISSN 0163-5700. `http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf`.

[24] L. Luu, D. Chu, H. Olickel, P. Saxena in A. Hobor, *Making smart contracts smarter*. V *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, str. 254–269, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-4139-4. `http://doi.acm.org/10.1145/2976749.2978309`.

[25] A. Manning, *Solidity security: Comprehensive list of known attack vectors and common anti-patterns*. `https://blog.sigmaprime.io/solidity-security.html`. [Dostopano 01. 09. 2018].

[26] C. Masters, *Augur (REP) Dapp discovers vulnerability, users dwindle*. `https://cryptovest.com/news/augur-rep-dapp-discovers-vulnerability-users-dwindle/`. [Dostopano 15. 08. 2018].

[27] C. Montoya, *How one hacker stole thousands of dollars worth of cryptocurrency with a classic code injection hack on EtherDelta and what you can learn from it*. `https://hackernoon.com/how-one-hacker-stole-thousands-of-dollars-worth-of-cryptocurrency-with-a-classic-code-injection-a3aba5d2bff0`. [Dostopano 29. 08. 2018].

[28] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, 2009. `http://bitcoin.org/bitcoin.pdf`.

[29] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena in A. Hobor, *Finding the greedy, prodigal, and suicidal contracts at scale*. CoRR, abs/1802.06038, 2018. http://arxiv.org/abs/1802.06038.

[30] D. Ongaro in J. Ousterhout, *In search of an understandable consensus algorithm*. V *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, str. 305–320, Berkeley, CA, USA, 2014. USENIX Association, ISBN 978-1-931971-10-2. http://dl.acm.org/citation.cfm?id=2643634.2643666.

[31] A. Peterson, *No escape: Augur burns key to network 'kill-switch'*. https://www.ccn.com/no-escape-augur-burns-key-to-network-killswitch/. [Dostopano 15. 08. 2018].

[32] H. Qureshi, *A hacker stole $31m of ether – how it happened, and what it means for Ethereum*. https://medium.freecodecamp.org/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce. [Dostopano 13. 08. 2018].

[33] B. Robič, *The Foundations of Computability Theory*. Springer Berlin Heidelberg, 2015, ISBN 9783662448083. https://books.google.si/books?id=io2QCgAAQBAJ.

[34] D. Siegel, *Understanding the DAO attack*. https://www.coindesk.com/understanding-dao-hack-journalists/. [Dostopano 13. 08. 2018].

[35] V. Sniezhkov, *Augur UI data can be completely replaced by an attacker which can lead to fund and reputation loss*. https://hackerone.com/reports/386587. [Dostopano 15. 08. 2018].

[36] G. Wood, *Ethereum: a secure decentralised generalised transaction ledger*. Ethereum Project Yellow Paper, 2014, ISSN 1098-6596.

Stvarno kazalo