ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი

გიორგი როხაძე

ტექნიკის მოხმარების რეესტრში ინფორმაციის გაყალბებისგან დაცვის ერთი კრიპტოგრაფიული მიდგომის შესახებ

კომპიუტერული მეცნიერება

ნაშრომი შესრულებულია კომპიუტერული მეცნიერების მაგისტრის აკადემიური ხარისხის მოსაპოვებლად

ხელმძღვანელი: მიხეილ თუთბერიძე, ფიზ.-მათ. მეცნ. კანდიდატი

თბილისი

2018

სარჩევი

შესავალი4

ამოცანის დასმა6

გამოყენებული ტექნოლოგიები7

მონაცემთა ბაზის სტრუქტურა8

პროგრამული კოდის საკვანძო ფრაგმენტები და აღწერა11

დასკვნა19

გამოყენებული ლიტერატურა20

ანოტაცია

წინამდებარე ნაშრომის ფარგლებში შემუშავებულია ვებ აპლიკაცია, რომელიც განკუთვნილია ტექნიკის შემოწმებების და შეკეთებების შესახებ რეესტრის საწარმოებლად. რეესტრის სანდოობის მისაღწევად გამოყენებულია ბლოკჩეინის მექანიზმი, რაც გამორიცხავს ინფორმაციის გაყალბების შესაძლებლობას რეესტრის მომსახურე პირების ან ბოროტმოქმედების მიერ.

Abstract

Giorgi Rokhadze

On One Cryptographic Approach of Preventing Falsification of Information in Registry of Technics Usage

In the scope of the present work the Software intended for registering information about the checking and repair of technics is developed. To assure the confidence of the information the blockchain mechanism is used which excludes possibility of falsification of information by registry serving persons or perpetrators.

შესავალი

თანამედროვე ტექნიკის ექსპლოატაციისას სულ უფრო და უფრო აქტუალური ხდება ტექნიკის შეკეთების რეესტრის წარმოება საგარანტიო მომსახურებისთვის, პროდუქტის განმეორებით რეალიზაციისთვის და ა. შ. ასეთი რეესტრი შეიძლება წარმოებდეს ტექნიკის მწარმოებლის მიერ, თუმცა მწარმოებელი ხშირ შემთხვევაში წარმოადგენს დაინტერესებულ მხარეს და ამგვარად, ასეთი რეესტრი ნაკლებად სანდოა მისი მომხმარებლებისთვის.

მომხმარებლისთვის უფრო სანდო იქნება ისეთი რეესტრი, რომელსაც აწარმოებს ნეიტრალური მხარე. ასეთი მხარის მოძებნა საკმაოდ რთულია. თუმცა მეორეს მხრივ საკმაოდ მოსახერხებელი იქნება ბლოკჩეინის იდეოლოგიის გამოყენება. ამ შემთხვევაში რეესტრი რამდენიმე ეგზემპლარად ერთდროულად ინახება რამდენიმე კომპიუტერზე მუდმივი სინქრონიზაციის რეჟიმში და ერთ კომპიუტერზე ინფორმაციის გაყალბების შემთხვევაში სხვა კომპიუტერებზე არსებული კორექტული ვერსიები გამოავლენენ გაყალბების ფაქტს სინქრონიზაციისას. მეორეს მხრივ, რეესტრის მთლიანობა დაცულია კრიპტოგრაფიული ალგორითმით, რომლის განხორციელებაც საკმაოდ რთულია და დიდ დროის მოითხოვს. ამიტომაც რეესტრის გაყალბება საკმაოდ დიდ დანახარჯებთან იქნება დაკავშირებული.

გამოქვეყნებულია მრავალი ნაშრომი, რომელშიც ბლოკჩეინის მექანიზმი არის განხილული.

[1] ნაშრომში აღწერილია ბლოკჩეინის მექანიზმის გამოყენება სამედიცინო ჩანაწერების წარმოებისას.

[2] ნაშრომში განხილულია ბლოკჩეინის მექანიზმის გამოყენების შესაძლებლობა სამეცნიერო ექსპერიმენტის შედეგების გამჭვირვალედ და დაცულად შესანახად.

[3] ნაშრომში განხილულია ბლოკჩეინის გამოყენების შესაძლებლობა საბიბლიოთეკო და სამედიცინო საქმეში.

გარდა ამისა, აღნიშნულ თემას ეძღვნება ნაშრომები: [4], [5] და სხვა მრავალი.

წინამდებარე ნაშრომზე მუშაობის ფარგლებში შემუშავებულია რეესტრის წარმოების ფუნქციონალის მქონე აპლიკაცია, რომელშიც ინფორმაციის სანდოობა მიიღწევა ბლოკჩეინის მექანიზმით.

ამოცანის დასმა

როგორც უკვე შესავალში იქნა აღნიშნული, წინამდებარე ნაშრომის მიზანს წარმოადგენს ისეთი აპლიკაციის შექმნა, რომელსაც ექნება ტექნიკის მოხმარების და შეკეთების რეესტრის წარმოების ფუნქციონალი. თუმცა, აპლიკაციის მიმართ არსებობს შემდეგი მოთხოვნა:

1. რეესტრი არ უნდა ინახებოდეს ცენტრალიზებულად, რადგან არსებობს მისი გაყალბების საფრთხე. რეესტრის მონაცემები უნდა ინახებოდეს პარალელურად რამდენიმე კომპიუტერზე, რომელთა შორის მოხდება სინქრონიზება. ამით მიღწეული იქნება ის, რომ თუ რომელიმე ასეთი კომპიუტერის მფლობელი გააყალბებს მონაცემებს, დანარჩენი კომპიუტერების მფლობელები სინქრონიზაციის დროს გამოავლენენ აღნიშნულ ფაქტს.

2. რეესტრის მთლიანობისთვის გამოყენებულ უნდა იქნას რაიმე ისეთი ალგორითმი, რომ თუ რეესტრის რომელიმე ჩანაწერი გაყალბდება, რეესტრის ალგორითმმა ეს გამოავლინოს. ამგვარად, რეესტრის გასაყალბებლად აუცილებელი უნდა იყოს აღნიშნული ალგორითმის თავიდან გამოყენება რეესტრის თითოეული ჩანაწერისათვის, რაც დიდ დროს და რესურსს უნდა მოითხოვდეს.

აღნიშნულ მოთხოვნებს სრულებით აკმაყოფილებს ბლოკჩეინის მექანიზმი, ამიტომ სწორედ ბლოკჩეინის მექანიზმი უნდა იქნას გამოყენებული რეესტრის შესანახად.

გამოყენებული ტექნოლოგიები

სერვერის მხარეს ბილდის ავტომატიზაციისთვის გამოყენებულია Gradle. რაც შეეხება აპლიკაციის გამართული მუშაობისთვის ჩონჩხად გამოყენებულია Spring Boot. ეს ბიბლიოთეკა საშუალებას გვაძლევს, შევქმნათ ცალკე მომუშავე აპლიკაციები, რომლებიც ზედმეტი ძალისხმევის გარეშე გაეშვება და ასევე საშუალებას გვაძლევს დამატებით გამოვიყენოთ დამხმარე ბიბლიოთეკები, როგორებიც ამ შემთხვევაში არის Spring Boot Web, Spring Boot Actuator, Spring Boot JPA. ეს ტექნოლოგიები საშუალებას გვაძლევენ, მარტივად ავაწყოთ მზა სისტემა, რომელიც იმუშავებს ზედმეტი ძალისხმევის გარეშე. მონაცემთა ბაზებთან სამუშაოდ გამოყენებულია H2. ეს გახლავთ რელაციურ მონაცემთა ბაზების მენეჯმენტის სისტემა. სადემონსტრაციო პროგრამის ასაწყობად გამოყენებულია პროგრამირების ენები Kotlin და Java. ასევე გამოყენებულია სხვა მცირე დამხმარე ბიბლიოთეკები, როგორიცაა Jackson Kotlin Module, რომელიც გამოიყენება JSON-ის სერიალიზაცია/დესერიალიზაციისთვის Kotlin-ში.

რაც შეეხება ვებ გვერდს, მის ასაწყობად გამოყენებულია Angular, TypeScript, როგორც ძირითადი ტექნოლოგიები. ბილდის ავტომატიზაციისთვის გამოყენებულია NPM.

მონაცემთა ბაზის სტრუქტურა

სადემონსტრაციო მაგალითისთვის შექმნილ მონაცემთა ბაზაში გვაქვს შემდეგი მთავარი ცხრილები – Record და Owner, რომლებიც შემდეგნაირად გამოიყურება:

```
data class Owner(
        @Id
        @GeneratedValue(strategy = AUTO)
        var id: Long?,
        @Column(unique = true)
        var personalId: String,
        @OneToMany
        var records: MutableList<Record>,
        var purchaseDate: LocalDate,
        var ownedUntilDate: LocalDate?,
        var purchaseLocation: String,
        var estimatedKilometersPerYear: Int
) {


        fun toBean() = OwnerBean(
                personalId = personalId,
                records = records.map(Record::toBean).toMutableList(),
                purchaseDate = purchaseDate,
                ownedUntilDate = ownedUntilDate,
                purchaseLocation = purchaseLocation,
                estimatedKilometersPerYear = estimatedKilometersPerYear
        )


}
```

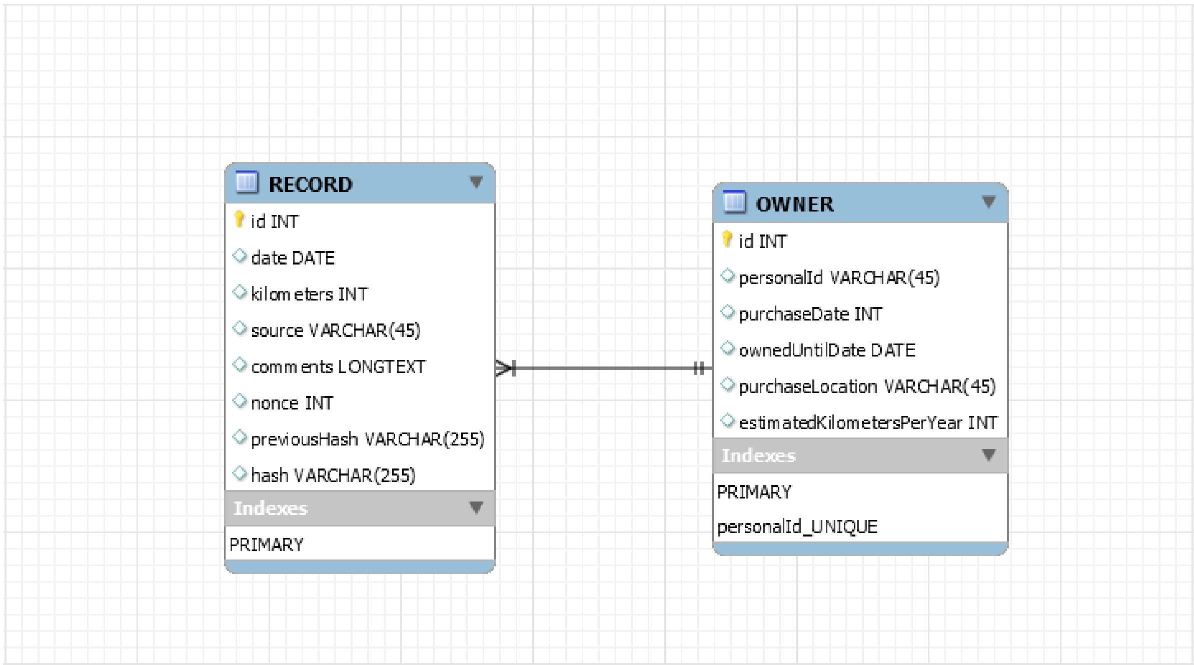
ეს წარმოადგენს ცხრილს, რომელშიც ინახება – ჩვენს შემთხვევაში – ავტომობილის მფლობელები.

```
data class Record(
        @Id
        @GeneratedValue(strategy = AUTO)
        var id: Long?,
        var date: LocalDate,
        var kilometres: Int,
        var source: String,
        var comments: String,
        var nonce: Long,
        var previousHash: String,
        var hash: String?
) {


        fun toBean() = RecordBean(
                date = date,
                kilometres = kilometres,
                source = source,
                comments = comments,
                nonce = nonce,
                previousHash = previousHash,
                hash = hash
        )


}
```

ამ ცხრილში კი შენახულია მონაცემები ავტომობილის შემოწმებაზე, რომლის გაყალბებისგან დაცვასაც ვცდილობთ.


RECORD
id INT
date DATE
kilometers INT
source VARCHAR(45)
comments LONGTEXT
nonce INT
previousHash VARCHAR(255)
hash VARCHAR(255)
Indexes
PRIMARY
OWNER
id INT
personalId VARCHAR(45)
purchaseDate INT
ownedUntilDate DATE
purchaseLocation VARCHAR(45)
estimatedKilometersPerYear INT
Indexes
PRIMARY
personalId_UNIQUE

პროგრამული კოდის საკვანძო ფრაგმენტები და აღწერა

მართალია, ბევრი ფიქრობს, რომ ბლოკჩეინი არის გადაწყვეტილება, რომლის პრობლემებიც ჯერ არ გამოვლენილა, მაგრამ ფაქტია, რომ ეს ტექნოლოგია ძალიან დიდი ნაბიჯია კომპიუტინგის ისტორიაში.

ბლოკჩეინი გახლავთ ტექნოლოგია, რომელშიც ჩანაწერები განხორციელებულია თანმიმდევრულად და ღიად, შესაბამისად ეს ყოველივე წარმოდგენს ღია მონაცემთა ბაზას, რომელშიც მონაცემებს ვინახავთ. ამიტომ დავიწყოთ ჩანაწერის განხილვით, თუ რას შეიცავს თითოეული ნოუდი (ჩვენს შემთხვევაში ავტომობილის ინფორმაცია) რომელიც შემდეგი სახითაა წარმოდგენილი:

```
@Entity
 data class Record(
        @Id
        @GeneratedValue(strategy = AUTO)
        var id: Long?,
        var date: LocalDate,
        var kilometres: Int,
        var source: String,
        var comments: String,
        var nonce: Long,
        var previousHash: String,
        var hash: String?
 ) {


        fun toBean() = RecordBean(
               date = date,
               kilometres = kilometres,
               source = source,
               comments = comments,
               nonce = nonce,
               previousHash = previousHash,
               hash = hash
        )


 }
```

```
data class RecordBean(
        var date: LocalDate?,
        var kilometres: Int?,
        var source: String?,
        var comments: String?,
        var nonce: Long?,
        var previousHash: String?,
        var hash: String?
)
```

თითოეულ ავტომობილზე ჩვენ ვინახავთ შემოწმების თარიღს, კილომეტრებს – რომელიც ნაჩვენები იყო ოდომეტრზე შემოწმების ამ თარიღისთვის, ფირმას – რომელმაც შეამოწმა ავტომობილი დამატებით კომენტარებს დაზიანებების შესაცებ, სპეციალურად შერჩეულ nonce მნიშვნელობას, რომლის პოვნაც წარმოადგენს სირთულეს ბლოკის ჰეშის დასათვლელად, წინა და ამჟამინდელი ბლოკების ჰეშს.

ამ ტექნოლოგიაში თითოეული ბლოკი საჭიროებს ინფორმაციას წინა ბლოკიდან. შესაბამისად, პირველი ბლოკის შესაქმელად საჭიროა უკვე არსებული ინფორმაცია და ამიტომაც ვქმნით საწყის *genesis* ბლოკს, რომელსაც ვინახავთ მონაცემთა ბაზაში აპლიკაციის ჩართვისთანავე:

```
fun createGenesisRecord(recordRepository: RecordRepository): CommandLineRunner {
                return CommandLineRunner {
                        recordRepository.save(
                                Record(
                                        id = null,
                                        date = LocalDate.MIN,
                                        kilometres = 0,
                                        source = "Genesis",
                                        comments = "Genesis",
                                        nonce = 0,
                                        previousHash = "0",
                                        hash = "0"
                                )
                        )
                }
        }
```

იმის შემდეგ, რაც დავამატებთ საწყის ბლოკს, მონაცემთა ბაზაში უკვე შეგვიძლია ახალი ბლოკების დამატებაზე ვიფიქროთ. ამის განხორციელებაში ჩვენ გვეხმარება შემდეგი ლოგიკა:

```
fun hash(id: Long, date: LocalDate, kilometres: Int, source: String, comments: String, nonce: Long, previousHash: String) = MessageDigest
              .getInstance(SHA_256)
              .digest(
                     StringBuilder()
                            .append(id)
                            .append(date)
                            .append(kilometres)
                            .append(source)
                            .append(comments)
                            .append(nonce)
                            .append(previousHash)
                            .toString()
                            .toByteArray()
              )
              .fold("") { str, it -> str + "%02x".format(it) }
```

ინფორმაციის შესანახად საჭიროა წინა ბლოკის ინფორმაცია ჯაჭვის მთლიანობის შესანარჩუნებლად. თითოეული ჩანაწერის დამატებასთან ერთად სირთულე იზრდება. თუ ასე არ მოვიქცევით, ამ შემთხვევაში გარე მხარეს მარტივად შეეძლება შეცვალოს ჩვენს ჯაჭვში შენახული ინფორმაცია.

```
@RequestMapping("/create")
fun createRecord(@RequestBody createRecordBean: CreateRecordBean) {
       val owner = ownerRepository.findByPersonalId(createRecordBean.ownerPersonalId)
       var record = createRecordBean.recordBean.toEntity()
       record = recordRepository.save(record)
       owner.records.add(record)
       ownerRepository.save(owner)
       recordRepository.save(populateData(record))
}
```

ჩანაწერის შესაქმნელად ჩვენ ვიყენებთ შემდეგ მიდგომას. რადგანაც ჩვენ გვჭირდება, რომ თითოეულ ჩანაწერის გაკეთება წარმოადგენდეს გარკვეულ სირთულეს

და სირთულე იზრდებოდეს ჩანაწერების ზრდასთან ერთად, გამოყენებულია შემდეგი ალგორითმი: როდესაც გვჭირდება ახალი ჩანაწერის გაკეტება, ვეძებთ ისეთ ჰეშს რომელიც იწყება გარკვეული ნოლების რაოდენობით, რომელსაც სირთულეს ვუწოდებთ. როდესაც სირთულე 1-ის ან 2-ის ტოლია, ამ ამოცანის გადაჭრა თანამედროვე კომპიუტერებს თითქმის მყისიერად შეუძლია ამიტომაც სადემონსტრაციოდ გამოყენებულია 5 სირთულე, რასაც გარკვეული დრო სჭირდება. იმისათვის რომ ვიპოვნოთ ისეთი ჰეში რომელიც 5 ნოლიანით იწყება, ვზრდით nonce-ის მნიშვნელობას და ვითვლით ხელახლა მანამ სანამ არ ვიპოვნით შესაბამის მნიშვნელობას.

ეს ყოველივე შესანიშნავია მაგრამ ჩვენ ასევე გვჭირდება, რომ შევამოწმოთ ჩვენი ჯაჭვის მთლიანობა. ამისათვის გვჭირდება შემდეგი ლოგიკა:

```
fun isValidChain(records: Iterable<Record>): Boolean {
        var previous: Record? = null
        for (record in records) {
                if (previous == null) {
                        previous = record; continue
                }
                if (record.hash != recordHashingService.hash(record)) return false

                if (previous.hash != record.previousHash) return false
        }
        return true
}
```

აქ ჩვენ თავიდან ვითვლით თითოეული ბლოკის ჰეშს, მაგრამ ეს არ წაიღებს დიდ დროს, რადგან მთავარ სირთულეს რაც წარმოდგენდა (nonce - ის პოვნა), უკვე ამოხსნილია და ეს ინფორმაცია ჩვენს მონაცემთა ბაზაში ინახება. ნებისმიერი ცვლილება, რომელიც განხორციელდება ბლოკებზე, გამოიწვევს ჩვენი ჯაჭვის მთლიანობის დარღვევას და გავიგებთ ამის შესახებ.

იმისათვის რომ მონაცემთა ბაზებთან მემუშავა, დამჭირდა გარკვეული რეპოზიტორიების დამატება:

```
interface OwnerRepository : CrudRepository<Owner, Long> {

        fun findByPersonalId(personalId: String): Owner

}
```

ეს კონკრეტულად კითხულობს მფლობელთა ცხრილიდან პირადი ნომრის მიხედვით მფლობელს. რადგანაც ვიყენებ Spring JPA-ს, ამიტომაც ამ ინტერფეისის იმპლემენტაცია არ დამჭირდა. ამას ჩემს მიერ გამოყენებული ფრეიმვორკი თავად გააკეთებს.

```
interface RecordRepository : CrudRepository<Record, Long> {

        fun findFirstByOrderByIdDesc(): Record

}
```

ასევე ამ შემთხვევაში ჩანაწერების ცხრილში უახლესი ჩანაწერის საპოვნელად მხოლოდ მეთოდის აღწერის დამატება დამჭირდა.

გარდა ამ ყველაფრისა, ასევე საჭირო იყო API ს შექმნა, რისთვისაც შემდეგი კლასები იქნა გამოყენებული:

```
private const val DIFFICULTY = 5

@RestController
@RequestMapping("/record")
```

```
class RecordController(
        private val ownerRepository: OwnerRepository,
        private val recordRepository: RecordRepository,
        private val recordHashingService: RecordHashingService
) {


        private lateinit var prefix: String


        init {
                val chars = CharArray(DIFFICULTY)
                Arrays.fill(chars, '0')
                prefix = String(chars)
        }


        @RequestMapping("/create")
        fun createRecord(@RequestBody createRecordBean: CreateRecordBean) {
                val owner = ownerRepository.findByPersonalId(createRecordBean.ownerPersonalId)
                var record = createRecordBean.recordBean.toEntity()
                record = recordRepository.save(record)
                owner.records.add(record)
                ownerRepository.save(owner)
                recordRepository.save(populateData(record))
        }


        private fun RecordBean.toEntity(): Record {
                val previousHash =
checkNotNull(recordRepository.findFirstByOrderByIdDesc().hash) { "Previous hash is empty!" }
                return Record(
                        id = null,
                        date = checkNotNull(date) { "Field date can not be null!" },
                        kilometres = checkNotNull(kilometres) { "Field kilometres can not be
null!" },
                        source = checkNotNull(source) { "Field source can not be null!" },
                        comments = checkNotNull(comments) { "Field comments can not be null!"
},
                        nonce = 0,
                        previousHash = previousHash,
                        hash = null
                )
        }
```

```
    private fun populateData(record: Record): Record {
        record.nonce = 0L
        var hash = recordHashingService.hash(record)
        while (!hash.startsWith(prefix)) hash = recordHashingService.hash(record.apply
{ nonce++ })
        record.hash = hash
        return record
    }


}


data class CreateRecordBean(val ownerPersonalId: String, val recordBean: RecordBean)
```

ჩანაწერებთან მუშაობაში დამხმარე API.

```
@RestController
@RequestMapping("/owner")
class OwnerController(
    private val ownerRepository: OwnerRepository
) {


    @RequestMapping(value = ["/create"], method = [POST])
    fun createOwner(@RequestBody owner: OwnerBean) =
        ownerRepository
            .save(owner.toEntity())
            .toBean()


    @RequestMapping("/list")
    fun listOwners() =
        ownerRepository.findAll().map { it.toBean() }.toList()


    private fun OwnerBean.toEntity() = Owner(
```

```
                id = null,
                personalId = checkNotNull(personalId) { "Field personalId can not be empty!"
},
                records = mutableListOf(),
                purchaseDate = checkNotNull(purchaseDate) { "Field purchaseDate can not be
empty!" },
                ownedUntilDate = ownedUntilDate,
                purchaseLocation = checkNotNull(purchaseLocation) { "Field purchaseLocation
can not be empty!" },
                estimatedKilometersPerYear = checkNotNull(estimatedKilometersPerYear) { "Field
estimatedKilometersPerYear can not be empty!" }
        )


}
```

მფლობელების მონაცემებთან სამუშაო API.

```
@RestController
@RequestMapping("/chain")
class ChainController(
        private val recordRepository: RecordRepository,
        private val validationService: ChainValidationService
) {


        @RequestMapping("/list")
        fun listChain() = recordRepository.findAll().toCollection(mutableListOf())


        @RequestMapping("/validate")
        fun validateChain() = validationService.isValidChain(recordRepository.findAll())


}
```

ჯაჭვთან მუშობაში დამხმარე API.

დასკვნა

ამრიგად, წინამდებარე ნაშრომის ფარგლებში შექმნილია აპლიკაცია, რომელიც აწარმოებს ტექნიკის შეკეთების და შემოწმების რეესტრს. რეესტრის ჩანაწერები ინახება ბლოკჩეინის მექანიზმის გამოყენებით, რაც საშუალებას იძლევა, რომ მოხდეს რეესტრის დეცენტრალიზაცია და გაიზარდოს მისი სანდოობა საბოლოო მომხმარებლის მიერ. აპლიკაცია არის მუშა მდგომარეობაში და შესაძლებელია მისი ექსპლოატაცია.

გამოყენებული ლიტერატურა

1. Nugent, T., Upton, D., & Cimpoesu, M. (2016). Improving data transparency in clinical trials using blockchain smart contracts. F1000Research, 5, 2541.
2. Jonathan Bell, Thomas D. LaToza, Foteini Baldmitsi, and Angelos Stavrou. (2017). Advancing open science with version control and blockchains. In Proceedings of the 12th International Workshop on Software Engineering for Science (SE4Science '17). IEEE Press, Piscataway, NJ, USA, 13-14.
3. Matthew B. Hoy. (2017). An Introduction to the Blockchain and Its Implications for Libraries and Medicine. Medical Reference Services Quarterly, 36(3), 273-279.
4. Guy Zyskind ; Oz Nathan ; Alex 'Sandy' Pentland. (2015). Decentralizing Privacy: Using Blockchain to Protect Personal Data. 2015 IEEE Security and Privacy Workshops.
5. Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, Charalampos Papamanthou. (2016). Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. 2016 IEEE Symposium on Security and Privacy.