Final Report

Securing NoSQL databases using blockchains

MongoDB + ARK Blockchain

Gurpreet, Paul and Fernando

April 12, 2018

CS4411

Securing NoSQL databases using blockchains

Introduction

The contents of a modern production database should be secure and interactions should be easily verifiable. We would like to propose a solution that strengthens the security of modern NoSQL databases using bleeding edge blockchain solutions.

In current database management software, the only way to verify the interactions that occur within a database is by checking the log created by the system. The problem is that logging can easily be disabled or reduced to a level where malicious actions could occur without leaving a trace.

The solution we are proposing is to use hashing to capture the active state of a Mongo database and use an Ark blockchain to verify its integrity. We would like to be able to specify a query resulting in a set of documents. The result will then be hashed and stored on the blockchain to a corresponding address. This functionality could then be automated to check the database's secured queries periodically to effectively secure it.

Motivation

The motivation for combining these two technologies into a security solution comes from pre-existing database logging functionality. The common way to trace back the interactions that occur within a database management system is to look at its logs. Database systems have the ability to disable logs or only record logs up to a specific error level. When relying on these logs to make sure no malicious activity has taken place, the investigator has to assume that the logging is setup correctly and no one disabled or tampered with it during the attack. This is where the problem arises; there is no completely secure way to verify the state of a database and ensure no one has tampered with the documents.

Databases provide fast and reliable data storage systems and blockchains provide secure, distributed and immutable data storage. The problem is interesting because it has to potential to leverage the best qualities of both these technologies and improve data storage all together.

The idea of using a blockchain to secure the data stored in other types of storage systems has been attempted by a web service called Tierion using their token TNT [2]. This system provides end points where you can send a hash and it sends back a timestamp ensuring it has been pushed to

a blockchain. A downside to this system is that it is a Software as a Service (SAAS) and therefore costs money to use on a large scale deployment. Private companies may not wish to have to send hashes of their data to an external service and Tierion does not allow for you to deploy your own instance of their system. The system itself is not very impressive either as it just provides a simple interface to the Bitcoin and Ethereum blockchains, neither of which are set up to handle such a task. Both BTC and ETH have long transaction times and were developed with a different purpose in mind. Tierion can take more than 10 minutes to push a hash to one of the chains which is too slow for an automated system. The idea is much better suited to data oriented blockchains such as the own we will be using called Ark. Ark has 7 second block times which will allow fast automated processing.

Ark is a Delegated Proof of Stake (DPOS) blockchain which allows transactions to carry a small amount of data [1]. Each transaction has more than enough space to carry the hash of a single document. Using this free and open source blockchain also allows us to run a local instance, which means we don't have to pay to secure our data and no information is exposed to the outside world.

Architecture and Environment

We used Ruby to create the command line utility that executes the ark mongo program. The reasoning behind Ruby is that it is simple to develop on as well as Ruby already contains and Ark Ruby library that makes it easy to establish and communicate with ARK for our blockchain. For our database we used MongoDB because we did not need any the added complexity that MySQL brings with relationships. We needed a simple scheme less, NoSQL database and from class and our research, MongoDB executed the job perfectly. The blockchain backend that we used was Ark blockchain. The reasoning behind this is that for starters, it is open source which enables us to implement it the way we need it without any restrictions but we have previously worked and developed for ARK therefore the technology is familiar and much simpler to integrate with.

A couple tools that we ended up using for the hosting and deployment of our tool are: Postgres, Docker, and Vagrant. We used Postgres to store the data within the blockchain. Docker was used for the deployment of the MongoDB. The benefit of using docker is that we can simulate the connection and storage for the database. Lastly, we used Vagrant for the deployment of the ARK blockchain. Vagrant allowed us to host the blockchain in virtual machine which mitigated the trouble in creating

all the connection and hosting it in our own servers. Although complex for setup, it made it easier to deploy and test on different machines. Vagrant and Docker guarenteed that our program would be able to work on any machine. Other tools that we used for visualization and development were NeoVim and Robo3T.

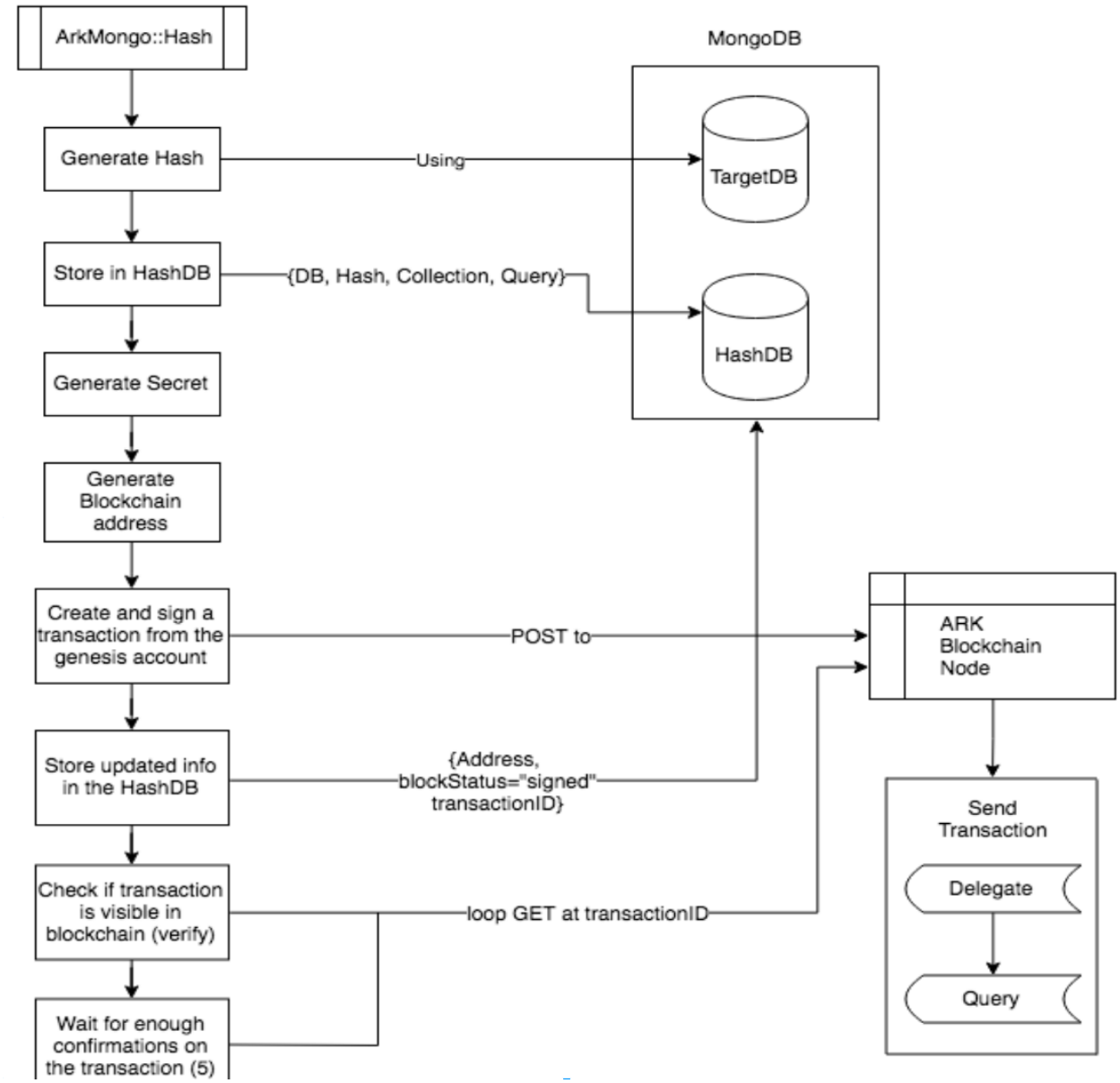

Figure 1: Flowchart of Architecture

Implementation

Developing the ArkMongo project came with many difficulties and challenges because it was quite an innovative idea using new technologies. We didn't have many resources to base the implementation off and we were all fairly inexperienced with the use of databases combined with a blockchain.

In order to gain enough experience with the technology to develop such a solution we looked at it in major sections and tackled each individually. The first section and challenge was creating a hash of a document set within MongoDB. Since we were going to write the command line utility in Ruby, we also did all of our prototyping in Ruby as well. The solution we came to for hashing document sets was to run a query and add each document returned to a sha256 hash. To do this we used the Digest built-in library in Ruby. One of the bugs we encountered in this stage was that the hash wasn't being stored to the database correctly. We later discovered that we had to encode the hash in hex form so we could view it correctly.

The next challenge was saving to the blockchain and database. The database was fairly simple because we learned how to do this in class, but setting up a local blockchain was not an easy task. It took us many iterations of configurations to deploy a blockchain that worked correctly for our task. The major challenge we faced in this step was discovering a protocol that would allow us to discover data we sent to the blockchain. We decided to allocate one address per individual query and then store that address inside our caching database so we could discover it again and add to it later.

The program was verified by running it and monitoring the changes within the caching database and the blockchain addresses. The result we were looking for include a new entry in the cache database and a new hash pushed to the address allocated for that query on the blockchain.

Results

The end product we have developed is a command line utility that runs singular operations of hashing a query or validating a query. The first main feature of the product is hashing. Hashing requires the following parameters: MongoDB's URI, a secret key of an address to send transactions from, a nethash of the blockchain, and a collection name and a query.

An example of an execution of the hashing function is as follows:

```
./arkmongo hash mongodb://127.0.0.1:27017/test 'planet wrap clever dirt silk
dance prefer view try swap enact island' '6a0eab08d4b8c3c818f93bc45fb51f7317e50c4c764fbedbf967
films -q Rating:G
```

This function begins by initializing a hashing database if one doesn't already exist. The database will be used as a cache for previously ran hash requests and it will allow the user to later verify their hashes quickly. Having this cache means the user doesn't have to remember anything but their collection and query to return their hashes. Next this function runs the query in the specified MongoDB and runs each document through a sha256 hash. At the end of the results the query itself is also added to the hash. Then the hash is encoded into hex and stored on to the query_hashes collection in the cache database we initialized earlier.

Now we generate a new secret key for the blockchain address by making use of the target database name, collection, and query. This way if we have the same query, we end up with the same address. Keeping the secret private is not required in this case because we care more about the history of a specific address than the actual state of the account. Using the secret and the public key that we derive from it, we can then create and sign a transaction with our hash in the data-field. The next step involves pushing this transaction onto a block in the blockchain. We can do this by sending it to the API endpoint on an Ark Node. The node's address was defined in the input parameters.

Once the transaction has been pushed to a node, we need to begin querying the blockchain to see if we can find that transaction in the wild. This process usually takes around 8 seconds but can take a bit longer sometimes. We send GET requests to the node periodically until the transaction is found. Once it is, we indicate this to the user and continue querying until we have enough confirmations on the block to be 100% sure it can't be tampered with. This process usually takes around 40 seconds because we wait for 5 confirmations and each new block takes around 8 seconds.

The screenshot of the final state of MongoDB after a hash has completed is pictured below and shows all fields that will be persistently stored.

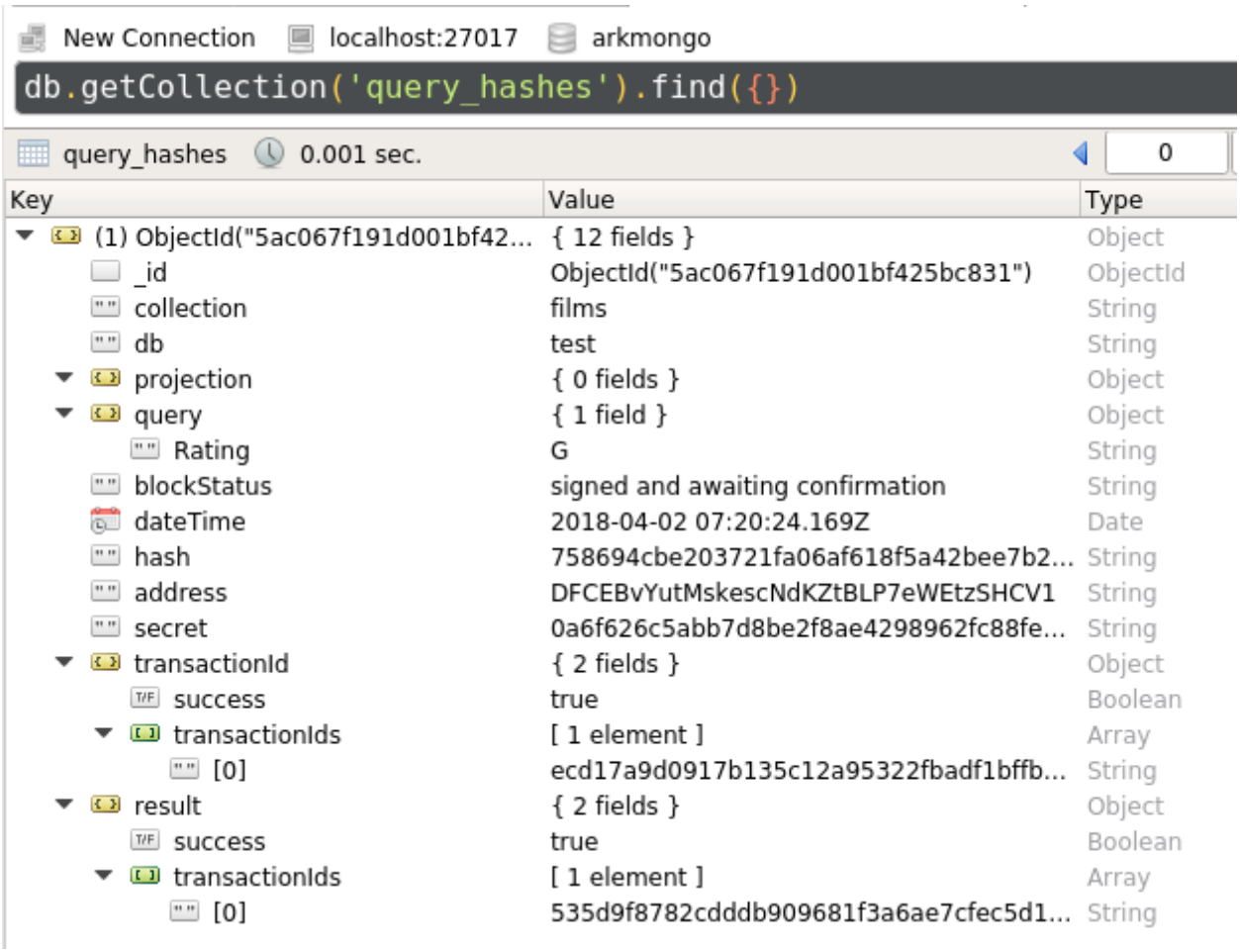

Figure 2: MongoDB document structure and final state

Conclusion and Future Work

Currently, database logging isn't a completely reliable way to verify if someone has tampered with the documents in a database. By combining databases fast and reliable data storage and blockchains secure, distributed and immutable data storage, we are able to secure a locally stored traditional database without compromising much of the speed of the database system. By choosing Ark as our blockchain technology, we were able to deliver a feasible solution for securing databases. This project could be improved upon in the future by integrating it into a typical workflow so the user doesnt even realize that our software is running in the background when they are accessing the database.

References

[1] The ARK Crew. *A Platform for Consumer Adoption*. February 25, 2018. URL: https://ark.io/Whitepaper.pdf.

[2] Jason Bukowski Wayne Vaughan and Glenn Rempe. *A global platform for verifiable data*. February 25, 2018. URL: https://tokensale.tierion.com/TierionTokenSaleWhitePaper.pdf.