Table of Contents

Blockchain-Based Internal Tamper-Proof Relational Database Management System Extension . 1

I. Synopsis 1

II. Research Questions 2

A. Environment Setup, Tools, and Programming 2

B. Design Methodology 2

C. Evaluation Methodology and Metrics 3

D. Results and Findings 5

III. Deliverables 6

IV. REUSE 6

V. Self-Evaluation 6

VI. Problems Encountered & Planned but Didn’t Do 7

VII. Threats to Validity 8

VIII. References 9

IX. Appendix 10

Blockchain-Based Internal Tamper-Proof Relational Database Management System Extension

George DiNicola (gd2581)

May 2, 2022

I. Synopsis

Internal tampering is an issue companies often deal with but sometimes don't have the means to detect or measure. A great solution to this problem is a middleware application that can store checksums of records on a private blockchain via smart contracts, so that internal tampering can be detected in real time without changing the underlying relational database. Although a middleware application typically results in relatively high query response time, minimizing access to the blockchain can boost the performance. The authors of TDRB [1] proposed a Redis cache database to fix this problem, however I propose a different method for reducing the number of interactions with the blockchain. The method I propose queries *ranges* of records in a single blockchain query rather than sending a query to the blockchain or cache database for each record in the relational database. The goal of this extension is to discover methods for improving the middleware's latency without having to maintain and incur the overhead storage of a cache database, ultimately attempting to extend this novel and unique research.

For my final project, I chose to extend the research I did for my midterm paper by implementing the middleware application design presented in TDRB [1], which proposed a blockchain-based middleware design that detects relational database tampering in real time. The proposed application successfully adds tamper-proof functionality to a centralized or distributed relational database by storing hashed records onto a distributed blockchain where each node holds a copy of a ledger representing database transactions. Since the TDRB middleware does not have any source code published, I implemented the design based on the guidelines provided in the TDRB paper [1]. The design specifications are vague; therefore, it is unlikely that the application I've created is exactly like the one created for the study.

To extend the current design proposed by the authors, I've made two major modifications. The first is functionality that adds a "tamper_flag" column to the results so that database users can continue querying the database when tampering occurs and be made aware of which records have been tampered with (i.e., records that have a value of "1" in their "tamper_flag" column). The second modification is to the algorithm for the row encryption record detection process (used to determine illegal modifications to the database tables) presented in [2]. Rather than repeatedly querying the blockchain for each database record found, I propose using a "queryByRange" blockchain API call on a world state database to minimize the number of blockchain queries needed by the application to detect tampering. The original design proposed in [1] queries the blockchain for each of the relational database records, thereby introducing a performance bottleneck into the application. The authors of [1] mitigated the bottleneck by implementing and evaluating a cache database into their design, whereas the modification I propose queries up to 1,000 blockchain records at a time, thereby only requiring interaction with the blockchain once every 1,000 records. The goal of my design is similar to [1]: decrease the query response time of the middleware by minimizing its interaction with the blockchain.

The user communities for which my project is targeted are database administrators, database developers, security professionals, software engineers, and system administrators. The value my project I will provide to my targeted user communities is a middleware application, like TDRB, that minimizes the middleware's interaction with the blockchain to boost the performance of query response time. The method I used to implement TDRB (queryByRange on the blockchain) has potential for organizations with a slower blockchain or situations where the connection to the blockchain is slow or unreliable (few queries to it rather than repeated to scale the size of the entire table or tables). In addition, the only blockchain-based tamper-proof middleware available as open-source code is ChainSQL [2], which does not have real-time tamper detection and can only audit the history of the blockchain. The other existing design proposed in [3] does not have any open-source code available. My documentation of the application is very in-depth and will be helpful for other engineers to not only understand what I built, but to also understand how the middleware conceptually works. Finally, the middleware I've implemented successfully detects tampering in real-time and was designed in a way that is easy for a user to configure. Additionally, I provided scripts that seed the blockchain for the user using their specified database tables.

My scope has changed since my progress report. When writing the progress report, I proposed testing different node configurations for the Hyperledger Fabric blockchain network; however, this became unrealistic because of how much computational resources would be required on the Amazon Web Services virtual machine I used. While developing the application, I found the queryByRange functionality and CouchDB (world state database) configuration for the blockchain ledger databases, which led me to pursue improving the tamper detection (specifically for row encryption records) process proposed in [1] to minimize the interaction between the middleware and blockchain network without using a cache database that would require storage overhead and maintenance (e.g., swapping out the specific cached records when the cache database gets full).

II. Research Questions

The purpose of implementing the middleware proposed in [1] is to answer the following research questions:

Research Question 1: What is the improvement in query response time of the middleware I've implemented using the tamper detection algorithms presented in TDRB [1] vs. the queryByRange blockchain functionality I am proposing?

Research Question 2: How does the middleware I've implemented based on the TDRB [1] study compare to other existing middleware implementations like MOON [3] and ChainSQL [2]?

A. Environment Setup, Tools, and Programming Languages

I used the same tools as the authors of [1] to implement their design, except for the middleware itself. I used Python 3.8 instead of Node.js v12 (the programming language the authors of [1] used) since I am more familiar with Python and wanted to minimize the development time. The relational database used in the system implementation was MySQL, the blockchain network was Hyperledger Fabric v1.4, the chaincode implementation was done using Node.js v12, and the containerization technology used for the blockchain network was Docker. I developed and evaluated the application on a t2.xlarge (4 vCPUs, 16GB RAM) Amazon Web Services virtual machine running Ubuntu 18.01 LTS with 30 GB of SSD space. The blockchain network is configured with four peer nodes (used for holding the ledger of transactions committed to the blockchain), two certificate authority nodes (used for verifying the identity of the user committing data to the blockchain and the peer nodes), and one orderer node (determines the global ordering of transactions and can be thought of as a global audit log to ensure the distributed system agrees on the state of events).

B. Design Methodology

[Appendix – Figure 1]

[Appendix – Figure 2]

Figure 1 illustrates the original design for the system proposed by [1] and Figure 2 shows the modifications I've made to the design. As mentioned in Section 1, blockchain-based middleware applications increase query response latency, therefore minimizing access to the blockchain, which can boost the performance. The authors of [1] proposed a cache database to fix this problem; I've proposed a different method of interacting with the blockchain which queries ranges of records in a single blockchain query, rather than sending a query to the blockchain or cache for *each* record in the relational database.

The reason for the high latency incurred by the original design presented in [1] is because of the repeated calls it makes to the blockchain, shown on line 14 within the for loop (the repeated call to query the cache, the resolution proposed by the authors, is shown on line 7 of the algorithm shown in Figure 3). The queryByRange method queries a bulk set of records first (rather than sending a query for each individual row in the query results), then iterates over the query results from the relational database, converting each row to the item hash representation, just as proposed in [1], and ensures that is matches the corresponding record in the blockchain. This design choice successfully reduces the number of interactions/queries sent to the blockchain, thereby matching the goal of the authors of [1] without the required storage overhead and maintenance of a cache database.

[Appendix – Figure 3]

I also chose to implement a world state database, CouchDB, into the blockchain to make it easier to query the *current* state of specific key-value pairs in the blockchain ledger. CouchDB provides a helpful abstraction for the API calls to the blockchain, so the programmer does not have to be concerned with the sequential nature of storing records on a blockchain (i.e., hash-chaining records to preserve the integrity of their ordering). In addition, CouchDB supports the "queryByRange" functionality that allows the modification I've made to the original TDBR design possible. The authors of [1] and [2] used Hyperledger Fabric's default database, LevelDB [7], which supports blockchain-rich queries. Although both databases offer the "queryByRange" functionality, the authors of [1] chose not to utilize it. An advantage of CouchDB over LevelDB is that CouchDB supports indexes, which should enable faster retrieval for key-value pairs. Figure 4 shows a depiction provided by Hyperledger showing the blockchain network's peer nodes and orderer node interacting with the application via transactions [9].

[Appendix – Figure 4]

C. Evaluation Methodology and Metrics

[1] and [3] measure the query performance of DDL, DML, and DQL statements over the proposed middleware to analyze the query response latency introduced by their designs. [1] analyzed the average performance of three MySQL databases with and without the middleware for the following operations: insert, select, update, delete, right join, left join, inner join, and natural join. Each operation was executed 40 times for each of the three databases and the specific query statements that were executed were not shown. [3] took a different approach by conducting a case study for six specified queries that use select, update, and insert operations. The queries were executed 100 times each on a traditional RDBMS, the middleware proposed, as well as a standard blockchain (BigchainDB). The average query response times were compared for each of the six queries, partitioned by each of the three database types.

RQ1: I evaluate the query response time of the middleware as done in [1] and [3]. In addition, I show the specific SQL statements for the database operations (query, create, update, delete) to generate the results with the goal of measuring the speedup increase of using queryByRange from the baseline design I implemented using the original design. More specifically, how does the percentage decrease in query response time (speedup) from the baseline compared to the percentage decrease in query response time (speedup) in the TDRB study? I will measure the average query response time over 40 iterations for each of the database operations, as done in [1], as well as measure the speedup of the solution I am using to address the issue of the latency created by the blockchain, as done in [1]. Unlike [1], which measures the speedup for different percentages of query repetitions (since the authors' solution utilizes a cache database), I measure the speedup for the same SQL statements, holding all else equal, query response time of my original implementation vs. my second implementation. Below are the exact query statements sent to the middleware application for the evaluation:

[Appendix – Set of Query Statements 1]

RQ2: For the second research question, I compare the modifications to the TDRB [1] design that I am presenting to an existing, MOON [3]. The overall goal is to understand how the average query response time queryByRange method compares to existing blockchain-based middleware designs similar to TDRB. MOON [3] does not perform real-time tampering, so to compare the tools I've adjusted the middleware to skip the tamper detection step (by commenting it out in the Python code) to mirror the process proposed in [3], but instead the hashed checksums will be stored on the blockchain rather than the plaintext records, accomplishing the internal tamper-proof goal both studies set out to accomplish. This modification to the TDRB design makes it comparable to MOON in that it performs database and blockchain operations with the ability to audit the blockchain ledger later to find tampering. To compare my modified implementation to [3], I recreated the proposed relational database tables and keys in MySQL since the authors published their relational database schema with their methodology. Fortunately, the number of existing records/rows in the database tables is not important for measuring performance since tamper checking is not being done in real-time (and performance differences would only begin to deviate for large-scale tests, which is not the goal of this evaluation). The authors of [3] used a local network, just as I am, to avoid any significant network delay that could be incurred over the internet. [3] stores the "doctor", "laboratory worker", and "patient" tables in the relational database, and the "exam" table on the blockchain because its novel approach gives the database designer the choice to partition the relational tables into on-chain (stored on the blockchain) records and off-chain (stored in the relational database only) records. To ensure a proper comparison, I only test my implementation using the tests from [3] that involve storing any records on the blockchain. The MOON study used real data but did not publish the dataset. For my evaluation, I generated the 100 test data records for the *exam* table proposed in [3] using an online test data generation tool called Mockaroo. Mockaroo enables the creation of test data based on parameters and ranges set for values of column/data types. For the update test, I updated the table using the primary key field since the middleware application I've implemented does not yet allow update statements with multiple attribute updates in a single statement (although this would be interesting future work). I've used the primary key in the where clause and am updating the "glucose" column.

ChainSQL reported an average middleware latency of 1.7 seconds for 5 nodes [2]; however, the design proposed in ChainSQL does not detect tampering in real time. Although the authors of [2] measure the latency of the ChainSQL application, they did not publish any specifics regarding their database schemas, number of records, query operation types, etc. that could be used for comparing the two tools. Unfortunately, [1] also did not publish any of the specific SQL operations they evaluated nor the tables or schemas those operations were used on. [1] provided the *entire* databases used to evaluate the middleware, which does not provide enough information for a comprehensive comparison of my modified implementation to their original.

For the tests to compare the middleware application to the one presented in [3], I scripted each operation to repeat 100 times with five seconds of sleep between executions, since the goal is to evaluate the query response time, not how the system handles concurrent operations. [3] also obtains the query response time for the application using 100 test iterations. The test I conduct with the middleware I implemented for the extension of [2] uses hashing and encryption to ensure data leaks via the blockchain aren't possible. It should be acknowledged that record encryption and hashing will increase latency to the application I am using.

[Appendix – Set of Query Statements 2]

D. Results and Findings

<u>RQ1</u>: Below are the results, in seconds, of executing "query", "insert", "delete", and "update" operations through the middleware using the algorithm proposed in [1] and the queryByRange function I added to the design in place of the Redis cache database. The operations were executed 40 times each with five seconds of sleep between each execution.

[Appendix – Table 1]

Although the query response time results in Table 1 for the algorithm proposed in [1] are significantly higher than the results reported by [1], the results might have been affected by the AWS instance or the Hyperledger sandbox development kit I use, which is mentioned in section VII of this paper. The results show that minimizing the interaction between the application significantly speeds up the middleware. These results could boost the performance of the middleware application when the blockchain network is slow or unreliable. In addition, the middleware application exhibited significant decrease in latency, 92.3%, 92.09%, 88.74%, 94.69% for the query, insert, update, and delete operations respectively, without incurring the storage overhead and maintenance of a cache database as proposed in [1]. It is worth noting the variance of the "delete" statement in the evaluation for the original method (see the Deliverables section for the location of the log for the executions). The variation of the results indicates that the Hyperledger Fabric SDK I am using is likely unreliable since sometimes the time to send the delete operation to the blockchain is as large as two times other iterations.

<u>RQ2</u>: The goal of the comparison to MOON is to evaluate the difference in query response latency for storing the hashed checksums on the blockchain [1] vs. storing unencrypted records on the blockchain to prevent tampering [3]. While the design approach used in [3] minimizes storage overhead by allowing database designers to choose which data should be stored on the blockchain, it is not secure from data leaks (an attacker has the ability to read the data) since the records are stored in plaintext (must be able to be received to be read), it does not detect tampering in real time, it does not protect records in the relational database that are not stored (if more need to be stored the latency could become an issue), and it may introduce higher latency than a design storing checksums. The results in Table 2 show that the middleware I implemented was slower than the design evaluated in [3] for the insert and update operations. The reason for this is likely the latency of the Hyperledger Fabric SDK I used, which I elaborate on in the "Threats to Validity" section. The results also show that the middleware I implemented was much faster than [3] for the query operation, 0.91177 seconds vs. 2.57 seconds. The faster query response time for the query operation can be explained by the design approach in [1] storing checksums of the database records on the blockchain rather than the entire records. The results indicate that [3] could benefit from using the storage of checksums (at least for the query operation) since [1]'s design allows for query records directly from the relational database rather than the blockchain, and simply using the checksums to ensure the results are correct rather than converting the key-value blockchain data into relational data.

[Appendix – Table 2]

III. Deliverables

The application I've implemented based on the design proposed in [1] can be found in the following GitHub repository on the "main" branch: https://github.com/GeorgeDiNicola/TDRB-Middleware-Extension. The repository contains a README.md file that explains the design of the application, the purpose of each function, the purpose of each file, the dependencies of the application, how to set the middleware up, how to start and configure the blockchain network, and how to execute/run the application. The "milestone assignments" for the project are located in the "milestone_assignments" folder. In addition, the Kanban board I used for managing the development of this project is located at the following link: https://github.com/GeorgeDiNicola/TDRB-Middleware-Extension/projects/1.

IV. Reuse

- **How to reproduce the work and evaluations:** https://github.com/GeorgeDiNicola/TDRB-Middleware-Extension/blob/main/README.md
- Hyperledger Fabric sample Javascript code/tutorial (chaincode piece of the application) and Docker network configuration scripts (blockchain network portion of the application)
    - Link to the tutorial: https://hyperledger-fabric.readthedocs.io/en/release-1.4/write_first_app.html
    - Hyperledger Fabric Github Repository of sample usage (I used and modified these scripts so they would work with the middleware application specifications): https://github.com/hyperledger/fabric-samples/tree/release-1.4
        - Chaincode: https://github.com/hyperledger/fabric-samples/tree/release-1.4/chaincode/fabcar/javascript/lib
        - Sample Node application for committing operations to the Hyperledger Fabric blockchain: https://github.com/hyperledger/fabric-samples/tree/release-1.4/fabcar/javascript
        - Scripts for configuring and bringing up everything necessary for the Hyperledger Fabric blockchain network: https://github.com/hyperledger/fabric-samples/tree/release-1.4/first-network
- Hyperledger Fabric: https://www.hyperledger.org/use/fabric
- MySQL open-source relational database: https://www.mysql.com/
- Docker open-source containerization and network tools: https://www.docker.com/
- CouchDB: https://couchdb.apache.org/
- Python libraries/modules: argparse, base64 , Cryptodome, hashlib, os, pandas, subprocess
    - More information about the usage of these libraries in my README (https://github.com/GeorgeDiNicola/TDRB-Middleware-Extension/blob/main/README.md)

V. Self-Evaluation

What I accomplished and learned

I successfully implemented the middleware application proposed in the TDRB study, modified the algorithm proposed in the research to improve it for slow blockchains (in my situation it was needed), and evaluated the modified design proposed in the study. In addition, I learned how to build blockchain functionality into applications as well as gaining a much deeper understanding of blockchain concepts through practice. I learned how to use Hyperledger Fabric, a popular open source blockchain tool, which is a very powerful functionality to be able to integrate into applications or databases in general. Resolving errors and developing more functionality while using Hyperledger Fabric solidified the blockchain concepts I learned when completing the literature review/midterm project for this topic. Finally, I gained experience developing scripts and applications with Node.js, which is a serverless JavaScript-like language. Although I implemented the application using Python, the chaincode that executes on the Peer nodes and API calls to the blockchain were only available in the language Node.js. Luckily using Node.js was very similar to programming using Javascript, however the framework for how the chaincode logic is implemented was challenging and useful.

What I found challenging

Coding with blockchain was the most challenging of the development work required, but was very rewarding. Hyperledger fabric and programming using blockchain had a very steep learning curve and it took about a week of usage before having enough control/understanding of it to get it to work for my own purposes. Developing using a relatively new tool (Hyperledger Fabric) was difficult due to the sparse documentation and missing references to syntax for specific languages. In addition, online forums like StackOverflow, StackExchange, etc. lack community questions surrounding Hyperledger. The only way to find the syntax for some operations was by trial and error, which was tough but worked out (with persistence). It was difficult to configure the dependencies for Hyperledger Fabric. Despite the documentation and examples stating that it could work on MacOS, I could not get them to. After a lot of configuration attempts, I got it to work on my AWS instance using Ubuntu 18.01 LTS. It was challenging to only be able to test the applications use of the blockchain only on the Linux virtual machine, rather than locally. It was also very difficult to implement the middleware based on the specifications proposed in [1]. Although the authors were specific about the tools, which was very helpful, they were vague in their description of the configuration of the execution environment and blockchain network.

Suggestion for the course

I think it would be helpful to have a minimum number of discussion topics/questions for each person in the course to bring up for a paper that someone else presented. In my opinion, the most valuable benefit of a topics course is hearing what others have to say since every class contains students with diverse backgrounds and experiences. Prior to the first class, we were assigned to read two articles to prepare us for getting better at reading research papers. I think it would be useful to have specific students be responsible for identifying the “big question”, the “specific” questions, determine whether the results answer the specific question(s), and find out what other researchers say about the paper, as mentioned in [4]. I think if five students were chosen each class to identify each of those questions (the student to question mapping could rotate), meaningful discussion could arise since multiple students are more invested in the research paper (since answering any of the above requires them to read it or at least understand it). In addition, [5] informs the reader to identify the motivations for the work, as well as asks what questions the reader is left with, their “take-away message” from the paper, and what they think the future directions are for the research. I think these are points that each presenter should be required to address during their presentation.

VI. Problems Encountered & Planned but Didn’t Do

I would have liked to implement the cache database and compare the performance of it to my own baseline (to see how the speedup compares to queryByRange for my own configuration), however I am already using a lot of computational resources on Amazon Web Services and a Redis cache database will put it over the limit and likely cause the performance of the virtual machine to suffer. I originally proposed evaluating how the query response time of the TDRB [1] design is affected by the number of blockchain nodes configured on the network (i.e., number of peer nodes and number of certificate authority nodes), as well as how the application scales when more Peer nodes (nodes that will hold copies of the ledger) are added to the network. But as previously mentioned, I am already using a lot of computational resources on Amazon Web Services and the constraint on resources will skew the evaluation since more nodes cannot be supported or will begin to fail. In addition, I would have liked to implement the JOIN operation in the middleware, but the design requires a separate blockchain for each database table, which resulted in the use of too many resources for the test configuration I've deployed.

I chose to compare a benchmark of my own implementation and the speedup from using queryByRange since speedup is relative and can properly be compared. I did the best I could to compare my work to the existing implementations; however, the other implementations were not transparent about either the data, the SQL statements, or both used to generate the results. In addition, TDRB differs from the other existing implementations in that it performs tamper detection in real-time and makes use of checksums to store records on the blockchain rather than their plaintext formats. Because of this, it only made sense to remove the tampering detection checks, only for the tests comparing it to MOON [3], to gain an understanding of how the middleware applications compare in their latency for operating on both the relational database and blockchain.

VII. Threats to Validity

The first threat to validity for the evaluation I conducted is the use of the sandbox development kit for Hyperledger Fabric. The blockchain network running through the SDK appears to be much slower than the network used in [1]. Unfortunately, the authors did not disclose the exact Hyperledger Fabric product they used, nor whether they used a paid solution such as the one offered on Amazon Web Services [8]. Another potential threat to the validity of the query response time is the use of a single AWS virtual machine, rather than multiple instances networked together in [1]. I chose to implement the system in a large, virtual environment rather than across multiple nodes to avoid capturing the unpredictable round-trip response time (RTT) that would be introduced by communicating over the internet. This would not reflect a practical environment for the middleware application since the individual components would be networked over the internet in practice and therefore would have latency at least as high as the test results (as well as unpredictable from potential outages or delays over the network). Another threat to validity is the speed of the benchmark queries sent to the MySQL database. The benchmarks were much faster in [1] for the relational database than the database I used, despite using the same product and same version. The final threat to validity is that [1]'s proposed design was very general, and therefore I cannot determine if the benchmarks I tested for the "original" version of the design (the design without the cache database) would be the same if tested using the application developed in [1].

VIII. References

[1] Lian J, Wang S, Xie Y. Tdrb: An efficient tamper-proof detection middleware for relational database based on blockchain technology. IEEE Access. 2021 Apr 28;9:66707-22. https://ieeexplore.ieee.org/document/9417201

[2] Muzammal M, Qu Q, Nasrulin B. Renovating blockchain with distributed databases: An open source system. Future generation computer systems. 2019 Jan 1;90:105-17. https://doi.org/10.1016/j.future.2018.07.042.

[3] Marinho SC, Costa Filho JS, Moreira LO, Machado JC. Using a hybrid approach to data management in relational database and blockchain: A case study on the E-health domain. In2020 IEEE International Conference on Software Architecture Companion (ICSA-C) 2020 Mar 16 (pp. 114-121). IEEE. https://ieeexplore.ieee.org/abstract/document/9095697

[4] Raff J. How to read and understand a scientific paper: A step-by-step guide for Non-Scientists [Internet]. HuffPost. HuffPost; 2017 [cited 2022Apr30]. Available from: https://www.huffpost.com/entry/how-to-read-and-understand-a-scientific-paper_b_5501628

[5] Griswold WG. How to Read an Engineering Research Paper [Internet]. CSE210. University of California San Diego; Available from: https://cseweb.ucsd.edu/~wgg/CSE210/howtoread.html

[6] Li D. Learn hyperledger fabric via questions and answers sets [Internet]. Medium. Medium; 2019. Available from: https://medium.com/@lichunshen84/learn-hyperledger-fabric-via-questions-and-answers-sets-8a19e2d40bd0#:~:text=(8)%20Level%20DB%20is%20the,comprise%20what%20type%20of%20data%3F&amp;text=(9)%20Blockchain%20services%20consist%20of%20three%20major%20components

[7] Using couchdb [Internet]. hyperledger. Available from: https://hyperledger-fabric.readthedocs.io/en/release-2.2/couchdb_tutorial.html#:~:text=A%20design%20document%20is%20a,one%20index%20per%20design%20document.&text=When%20defining%20an%20index%20it,along%20with%20the%20index%20name

[8] What is Hyperledger Fabric? [Internet]. Amazon. For Dummies, a Wiley brand; 2019. Available from: https://aws.amazon.com/blockchain/what-is-hyperledger-fabric/

[9] Writing your first application [Internet]. Hyperledger. Available from: https://hyperledger-fabric.readthedocs.io/en/release-1.4/write_first_app.html

IX. Appendix

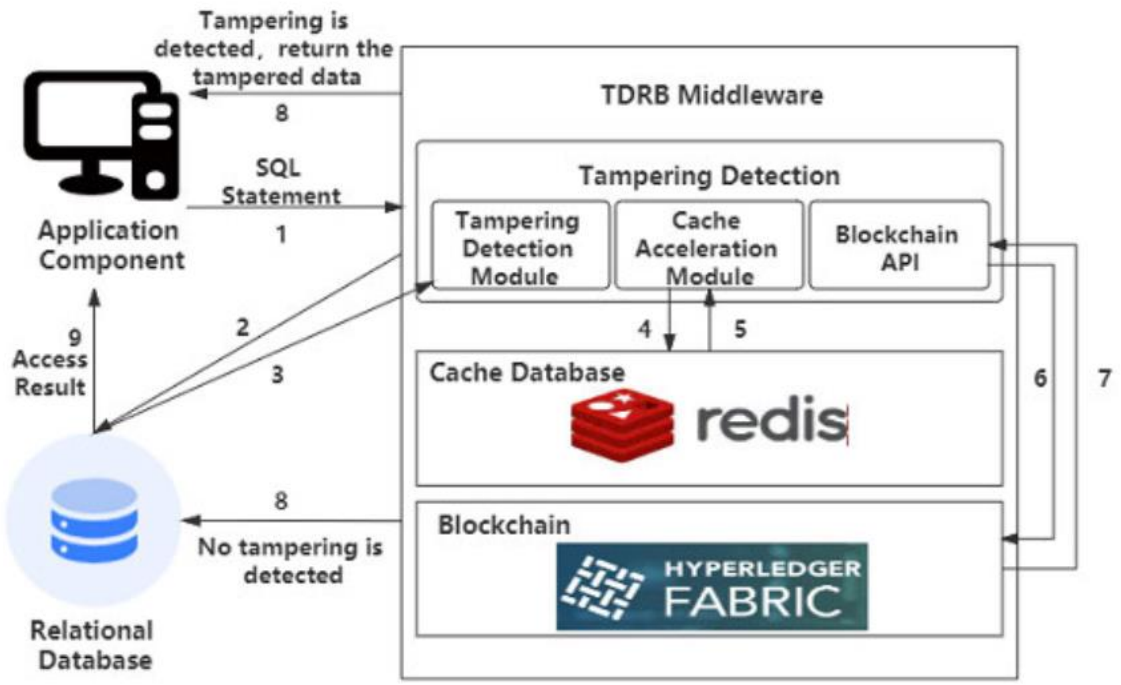


Figure 1: The TDRB middleware design proposed in [1].

Tampering is detected, return the target data back with a tamper flag set

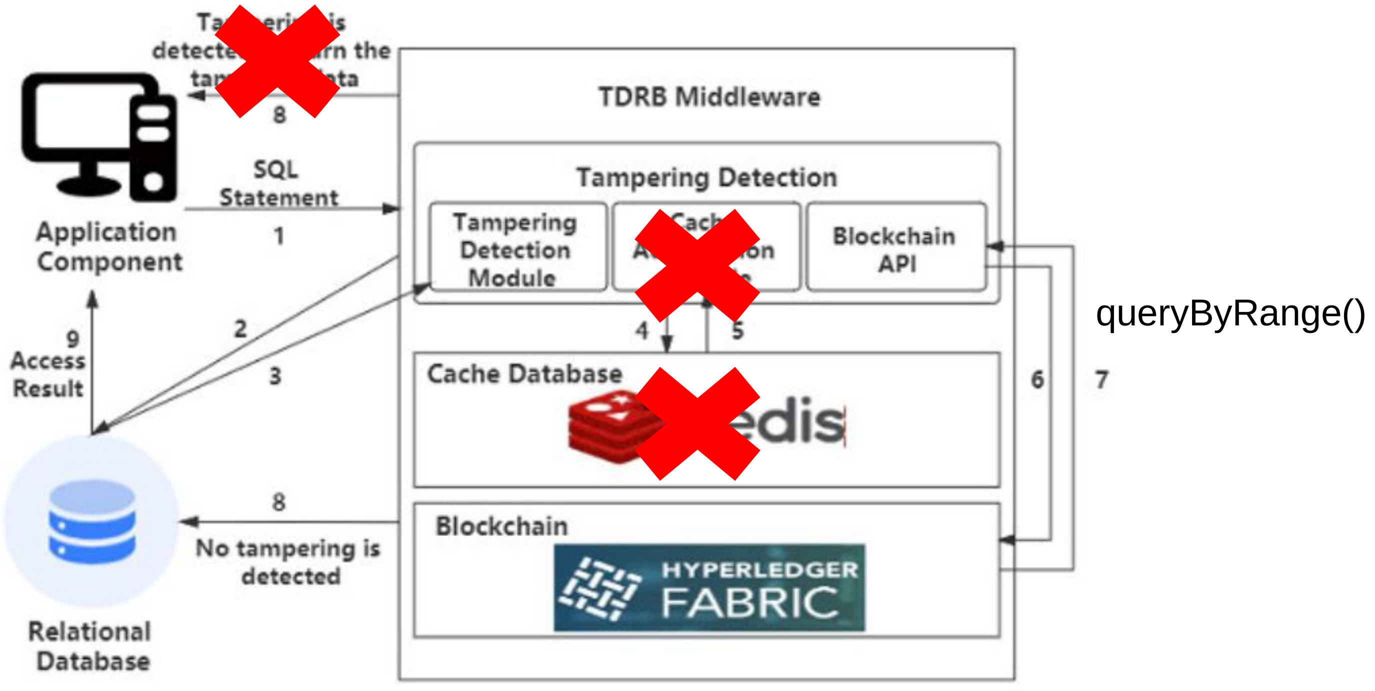
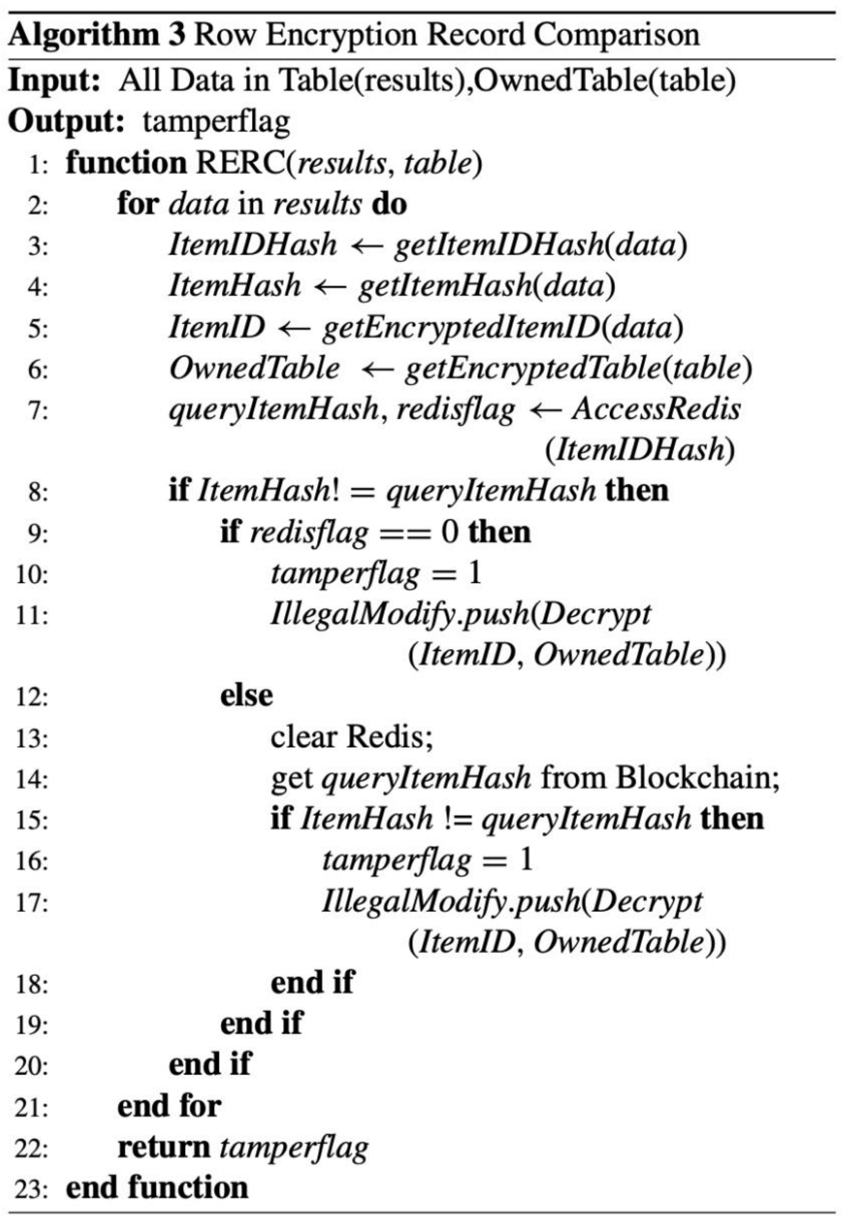


Figure 2: The modification I've made to the TDRB middleware design proposed in [1]. The application has been modified to return the database results with a "tamper_flag" column so the user can ensure they are working with untampered data. In addition, the cache database has been swapped out in favor of the queryByRange blockchain API call to evaluate the performance boost of the middleware, as the authors did in [1] when evaluating their original design against their design with the Redis cache database.



Algorithm 3 Row Encryption Record Comparison

Input: All Data in Table(results),OwnedTable(table)

Output: tamperflag

```
 1: function RERC($results$, $table$)
 2:     for $data$ in $results$ do
 3:         $ItemIDHash \leftarrow getItemIDHash(data)$
 4:         $ItemHash \leftarrow getItemHash(data)$
 5:         $ItemID \leftarrow getEncryptedItemID(data)$
 6:         $OwnedTable \leftarrow getEncryptedTable(table)$
 7:         $queryItemHash, redisflag \leftarrow AccessRedis$
                                              $(ItemIDHash)$
 8:         if $ItemHash! = queryItemHash$ then
 9:             if $redisflag == 0$ then
10:                 $tamperflag = 1$
11:                 $IllegalModify.push(Decrypt$
                            $(ItemID, OwnedTable))$
12:             else
13:                 clear Redis;
14:                 get $queryItemHash$ from Blockchain;
15:                 if $ItemHash$ != $queryItemHash$ then
16:                     $tamperflag = 1$
17:                     $IllegalModify.push(Decrypt$
                            $(ItemID, OwnedTable))$
18:                 end if
19:             end if
20:         end if
21:     end for
22:     return $tamperflag$
23: end function
```

Figure 3: The Row Encryption Record Comparison algorithm proposed in [1] for identifying illegal modification operations to the relational database. Line 14 shows the query sent to the blockchain *within* the for loop, slowing down query performance due to requiring more queries be sent back and forth for the same amount of data between the application and blockchain. Line 7 shows the improvement the authors of [1] made while pursuing their research, a query sent to the cache database within the for loop (indicating repeated queries sent to the cache database for each record).

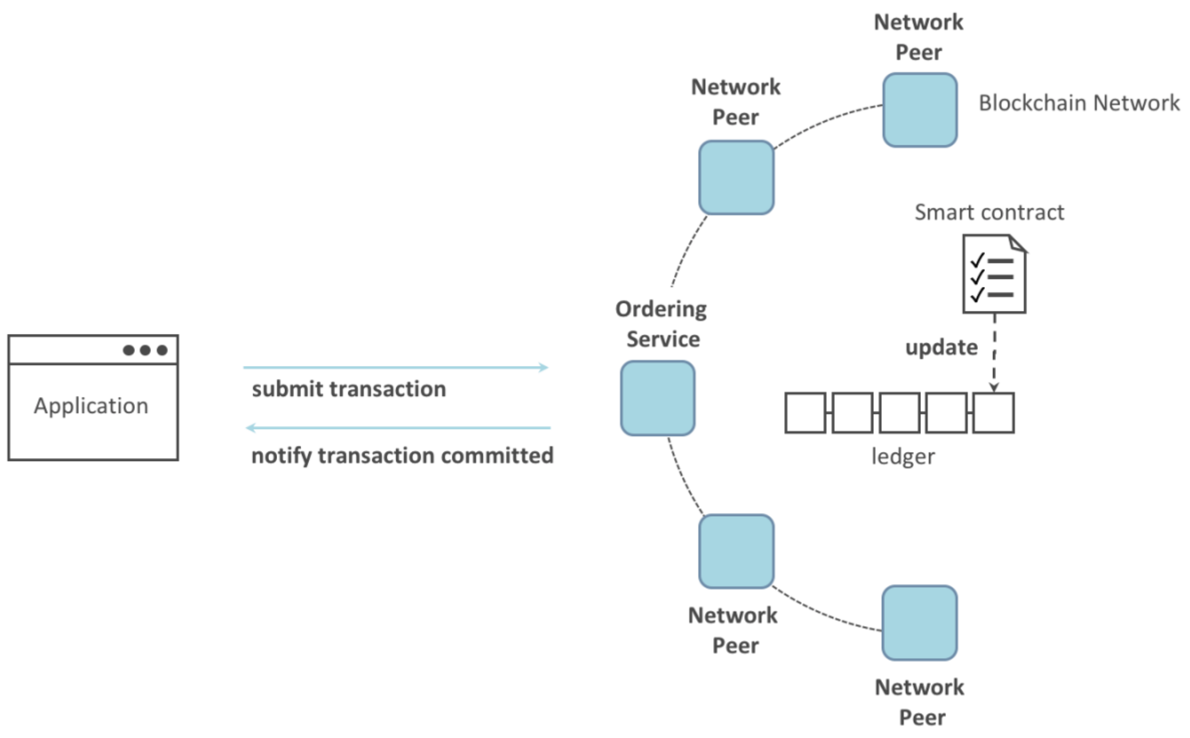


Figure 4: Depiction of the submission of transactions to the Hyperledger Fabric blockchain network from an application [9]. The database records for the TDRB middleware [1] are stored as smart contracts on the ledgers held by the four peer nodes and the global ordering of transactions is maintained by the ordering serviced (the "orderer" node).

	Query [sec]	**Insert** [sec]	**Update** [sec]	**Delete** [sec]
Original Method from [1]	117.657	125.15545	86.184	120.820
queryByRange method	3.182	9.903	9.707	6.418

Table 1: Results (in seconds) showing the query response time performance of the algorithm proposed in [1] (on the network configured in my evaluation) vs. the queryByRange method.

	SQL Statement 1 (Query) [sec]	**SQL Statement 2 (Insert)** [sec]	**SQL Statement 3 (Update)** [sec]
TDRB queryByRange modification	0.91177	7.43819	4.14156
MOON Results [3]	2.61	0.055	2.894

Table 2: Results (in seconds) showing the query response time performance of the queryByRange method on the test data for SQL statements evaluated in [3] and the results reported by [3] for the same statements.

Set of Query Statements (Research Question 1)*

SQL statement 1 (Query): SELECT * FROM student;

SQL statement 2 (Insert): INSERT INTO student (id, name, sex, age) values (${i}, 'test${i}', 'Male', ${i});"

SQL statement 3 (Delete): DELETE FROM student WHERE id={i};

SQL statement 4 (Update): UPDATE student SET name='updated' WHERE id=${i};

Set of Query Statements (Research Question 2)*

SQL statement 1: SELECT * FROM exam;

SQL statement 2: UPDATE exam SET glucose=87 WHERE id=${i};

SQL statement 3: INSERT INTO exam VALUES (1, 2, 86, ‘2020-01-08’, 70, 2.7, 8.8, 9.7, 7.9, 417.1);

*The “${i}” syntax represents the iteration number for the dynamic query being tested in the middleware application.