Storing Credit Card Information Securely using Shamir Secret Sharing in a Multi-Provider Cloud Architecture

Levent Ertaul, William Marques Baptista, Rishi Maram
CSU East Bay, Hayward, CA, USA.
levent.ertaul@csueastbay.edu, william.marquesbaptista@gmail.com,
rmaram2@horizon.csueastbay.edu

***Abstract*— Cloud storage and Online database services allow information to be accessed from virtually any location around the world with Internet access. While this makes information readily accessible and shareable, it also exposes data to the hostile environment that is the Internet, rendering it vulnerable to malicious attackers. Credit card information is one type of data that can be targeted by malicious attackers. This paper proposes an implementation that could store credit card information securely on a multi-provider cloud architecture by using Shamir secret sharing as the foundation and alternative to traditional encryption. The primary objective of this work is to implement the Shamir secret sharing algorithm in multi-provider cloud architecture. The Java programming language is used to implement a proof-of-concept application that connects to Amazon RDS and Google SQL databases. Performance analysis of the implementation is discussed, demonstrating its efficiency and revealing where bottlenecks may be encountered when processing credit card numbers. Finally, some remarks are made regarding the evolution of the implementation throughout the undertaking of the project, and additional improvements to the mechanisms are proposed.**

I. Introduction

Cloud storage brings a competitive advantage to smaller businesses that are constrained by a limited budget and employees as a cost-efficient means of competing [1]. Advances in cloud storage services such as Amazon RDS [2] or Google Cloud SQL [3] among others, along with the ever-growing market of cloud-storage solutions [4], gives smaller businesses the flexibility to choose between different service providers.

Customer credit card information is an example of sensitive data which is often transmitted over the Internet, whether while shopping Online, transferring funds to and from virtual wallets [5], or paying with a smart phone at a checkout counter [6]. Not only are there third parties which provide businesses with a service to handle credit card transactions [7], a security standards council [8] also exists that develops various standards related to data security and payment applications. However, there have been doubts as to the effectiveness of this council itself [9], which would have a significant impact on not only how users shop, but also how businesses and banks handle monetary transactions.

Instead of trusting a single entity with credit card information, said information could be split into multiple pieces and stored across multiple database service providers. In this scenario, small businesses not only use cloud services as a cost-effective competitive advantage [1], but also decide who to trust with storing their customer's credit card information.

Traditional encryption is computationally expensive [10], so to counter that cost this implementation suggests an alternative solution. It is both based on work in [10][11], which serve as a foundation to store credit card information securely by means of the Shamir secret sharing scheme [11] in a multi-provider cloud architecture. This implementation takes a credit card number as input, generates multiple pieces called shares, and stores those shares across multiple databases from multiple cloud providers. In doing so, it is capable of eliminating the informational value of any given share should there be a security breach at any of the database service providers' location.

Section II of this paper addresses obstacles that may be encountered regarding the storage of sensitive data. Section III will discuss the method of creating shares as a form of encryption, presenting the general idea behind secret sharing. Section IV will highlight the details of how the methods were implemented, and provide the reasoning behind the decisions. Section V will present performance analysis comparing different numbers of shares. Section VI highlights changes made to the implementation during the undertaking of the project, and includes remarks as to what aspects of the implementation may be improved. Section VII concludes with final remarks on the project.

II. Traditional Encryption vs. Secret Sharing

Businesses may sometimes opt for some form of traditional encryption when storing data on local databases, especially those containing customer information that includes credit card numbers. However, doing so may sometimes require special hardware such as Hardware Security Modules, and traditional methods of encryption can be computationally expensive [10].

Attackers often need to breach just a single entity to obtain customer data, but any one of these breaches can prove to be disastrous: Heartland Payment Systems-134 million cards exposed[9][12]; TJX Companies Inc.-94 million card numbers exposed[13]; Card Systems Solutions-40 million card accounts exposed[12]. While these statistics only mention large entities, smaller entities are just as vulnerable to attacks, especially if all the required information is stored in a single provider location, let alone locally in a single database.

An alternative solution would be to store data remotely across multiple providers vs. locally. This would increase the number of targets an attacker would have to breach to obtain the desired data. A secret sharing scheme [11] could be used to break credit card information into multiple pieces and store those pieces at different locations. Ideally, different providers would be used to further reinforce the concept of mutually suspicious entities with conflicting interests [11].

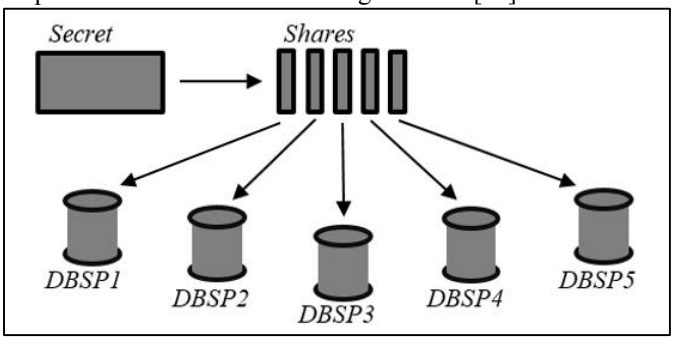

Fig. 1. A secret is split into shares, and distributed to different database service providers (DBSP).

Figure 1 depicts an example of what our implementation accomplishes. This implementation takes a credit card number and splits it into multiple shares by means of a secret sharing scheme [11]. The shares are then stored across different cloud databases obtained from different service providers. The greatest strength of our implementation is that an attacker must know a) how many shares a particular secret has, b) where these shares have been stored and have access to them, c) the corresponding pairs of each share and how they are calculated, d) the secret prime used during the initial computation, and e) parameters specific to the polynomial used to produce the shares. It becomes clear when presented with these obstacles that a secret sharing scheme can be an effective solution in securing credit card information.

III. Shamir Secret Sharing

In [11] Shamir proposes a mechanism for robust key management schemes, where a numerical value is split into pieces, and can be easily reconstructed with a subset of those pieces. This is known as a threshold scheme [11] where a subset *k* of *n* total pieces is required to reconstruct the original secret. The implementation in this paper requires that all *n* pieces be available to reconstruct the original secret, in other words, as opposed to a *(k, n)* threshold scheme, this implementation is a *(n, n)* scheme.

A. Creating the Shares

To create the shares, let *k* represent the number of pieces, or shares, to be produced where *k=n total pieces*. Then, a prime *P* must be chosen where *P>M* and where *M* is the numerical value or secret being split into shares. A polynomial of *k-1* degree is then constructed with *k-1* random coefficients *C*, where *P>C>0*.

$$f(x) = M + C_{(1)}x^{(1)} + C_{(2)}x^{(2)} + \ldots + C_{(k-1)}x^{(k-1)} \quad (1)$$

Polynomial (1) is the result of this construction, where the secret *M* is the term C_0x^0, each *C* is a randomly generated coefficient, and substituting *0* for *x* in the polynomial returns this secret value *M*. Pairs are created as an *x* input into (1) resulting in the corresponding output share *f(x)*, thus forming the pair *(x, f(x))*. Any number of pairs can be produced or even replace existing pairs. Once the shares have been created, we no longer have use for the random coefficients as they are randomly chosen for every secret. The resulting pairs and the prime used to select random coefficients are the only information needed to be recorded at this time.

B. Reconstructing the Secret

All *k* pairs *(x, f(x))* are required to reconstruct the secret using Lagrange polynomial interpolation [11][14].

$$M = \sum_{k=1}^{n} y_{(k)} \prod_{i=1, i\neq k}^{n} \frac{-x_{(i)}}{x_{(k)} - x_{(i)}} mod P \quad (2)$$

We then expand (2) using the pairs that were originally created to recover the secret.

C. Unrecoverable Secret:

We should note that, as mentioned in the beginning of this section, we need all pieces to reconstruct the secret. If, for instance, we were missing one or more pairs, our result would be quite different. Some alternatives to this limitation are suggested in section VII.

IV. Implementation Methods and Reasoning

This project was implemented on a notebook computer with a 1.6GHz quad core processor, 8 GB of RAM, and running a GNU/Linux OS [15] with kernel version 3.13.0-46-generic. NetBeans 8.0.2 [16] integrated development environment (IDE) was used to implement the project using the Java [17] programming language. The IDE was used to create a proof-of-concept Swing GUI application, taking advantage of an integrated profiler for performance analysis. The profiler allows us to monitor the performance characteristics of different methods in the implementation without making changes to the original source code.

Eleven databases were created, 5 on Amazon RDS [2] using MySQL 5.6.22 [18], and six on Google Cloud SQL [3] using MySQL 5.5 [18]. A MySQL JDBC Driver library [19] was used to interface between the proof-of-concept application and the remote cloud databases. Each database contains one table, and a user was specifically created to access these databases with ***SELECT***, ***INSERT*** and ***DELETE*** privileges.

As opposed to [10], there is no need to evaluate max/min values, or perform calculations on ranges of values stored in the databases. Apart from the GUI-related code, there are four prominent code portions categorized as 1) global variables and structures, 2) generators and equations, 3) database operations, and 4) additional methods.

A. Global Variables and Structures

The global variables in the implementation are the username and password information to access the different databases, a prime *P*, the number of shares *K*, and the database *IP* addresses. There are also three array list structures: a

database address array list, a coefficient array list, and a share array list.

The database username and password information was hard-coded to facilitate the interaction between the proof-of-concept application and the databases. As a security precaution, these credentials should not be stored within a production version of this proof of concept, but instead an alternative method of authenticating a user or application with the different databases should be implemented. That, however, is outside the scope of this project.

The prime number was implemented as a *BigInteger*, which allows the use of integers larger than the 64-bit limitations of the *long* data type [20]. It was hard-coded as a 2048-bit value for the purpose of this work, but should be kept secret at all times. This large prime ensures that the coefficients generated are also very large, making it difficult for an attacker to recover the secret due to the discrete logarithm problem [21]. How the prime number should be securely stored, encrypted, or recovered in a production implementation is outside the scope of this paper, as that may vary from one application to another.

The number of shares *K* denotes how many shares a secret will be divided into. This variable is also used throughout the application as an iterative loop parameter for database connections and generation of the various lists.

The coefficient array list stores randomly generated numbers and is accessed during share generation. After generating all shares, this list is no longer needed, so the contents can be overwritten if another secret is being processed.

The share array list is populated during the generation of shares and used to commit records on the different databases. It is cleared and re-populated during the recovery process of a secret as shares are collected from the different databases.

The database address list contains addresses to the different databases, and they are also used as one of the parameters to create record name fields for each database's respective share.

B. Generators and Equations

The three generators in this implementation are the database address generator, the secure random coefficient generator [22], and the share generator. The database address generator runs only once, whereas the remaining two run once for every secret that is processed.

1) Database Address Generator

```
private static void enumDBList() {
        int i, n;
        if (K % 2 == 0) {
            n = K/2;
        } else {
            n = K/2+1;
        }
        for (i = 0; i < K; i++) {
            if (i<n) {
                dbList.add(dbGoogle + (i+1));
            } else {
                dbList.add(dbAmazon + (i+1-n));
            }
        }
    }
```

The database address generator is a simple algorithm that enumerates the database address list, and is directly dependent on the number of shares. The database names used at both service provider locations are named from secsharedb1 to secsharedb5 for Amazon RDS [2], and secsharedb1 to secsharedb6 for Google SQL [3], for a total of eleven databases. The IP addresses are hardcoded global variables with the first portion of the database name. They are then referenced using the variables *dbGoogle* and *dbAmazon* for short while the database numbers are concatenated at the end of the string within the algorithm's iterative loop.

We wanted to make sure that both providers were always used, so this algorithm distributes the number of databases based on the value of shares *K*: for an even number of shares, the same number of databases is used at each provider location; for an odd number of shares, there is one more database used at Google's location. There is no particular reason for choosing a Google database over an Amazon database other than to fill the gap. Although a seemingly simple and innocuous algorithm, it is safe to say that this is what controls which databases are being accessed in the different provider locations. More on this algorithm is discussed in Section VI.

2) Random Coefficient Generator

```
private void genCoefs(int shares) {
        BigInteger r;
        coefs.clear();
        int i;
        for (i = 0; i < shares - 1; i++) {
            r = new BigInteger(2047, new SecureRandom());
            if (r == BigInteger.ZERO) {
                r = r.add(BigInteger.ONE);
            }
            coefs.add(r);
        }
    }
```

The coefficient generator above is used to generate *K-1* 2047-bit cryptographic-strength pseudo random numbers [22], and this process is repeated for every secret being split into shares. More on the decision for its size is discussed in section VII. During the test, smaller coefficients were used, and we noticed that there were times where a zero was returned as a secure random number. This would have changed an entire term to zero during multiplication, so a check was added to add the value one to the secure random number when this was the case. The coefficients are stored in the coefficient array list and only accessed when generating shares, after which they are no longer useful since a different set of coefficients will be generated when processing another secret.

3) Secret Share Generator

```
private void genShares(int shares) {
        long secret = Long.parseLong(jTextField2.getText());
        BigInteger coef, term, result;
        secShares.clear();
        int i, j;
        for (i = 0; i < shares; i++) {
            result = BigInteger.ZERO;
            for (j = 0; j < shares-1; j++) {
                coef = (BigInteger) coefs.get(j);
                term = (BigInteger.valueOf(i+1)).pow(j+1);
                result = result.add(coef.multiply(term));
            }
```

```
            result = result.add(BigInteger.valueOf(secret));
            secShares.add(result);
        }
    }
```

The share generator above produces shares corresponding to each *x* input. It uses a polynomial such as (1) using the generated coefficients mentioned earlier, and produces as many shares as indicated by the global variable *k*. In this implementation, the *x* values are generated incrementally with a loop, and more remarks regarding this process will be mentioned in section VI. The resulting shares vary in size, depending on the number of shares being produced, but their sizes increase as the number of shares increase. The shares were also implemented as a *BigInteger*, again due to the 64-bit limitations of the *long* data type [20].

4) Secret Recovery Algorithm – summation portion

```
private void secRecover() {
        BigInteger pair, term;
        lagSum = BigInteger.ZERO;
        int i;
        for (i = 0; i<K; i++) {
            pair = (BigInteger) secShares.get(i);
            term = pair.multiply(lagrange(i+1));
            lagSum = lagSum.add(term.mod(P));
        }
        if ((K % 2) == 0) {
            jTextArea1.append("M: " +
P.subtract(lagSum.mod(P)) + "\n");
        } else {
            jTextArea1.append("M: " + lagSum.mod(P )+ "\n");
        }
    }
```

The secret recovery algorithm above was implemented iteratively as it was the simplest implementation method at the time. A recursive method was not tested. This first algorithm implements the summation portion of the terms in (2). During the test trials, we noticed that for an even number of shares, the result resembled the prime rather than the secret. After closer inspection, when using an even number of shares the summation result has to be subtracted from the prime to obtain the secret back. For this reason a test is performed where if *k mod 2* is *0*, then the summation result is subtracted from the prime to obtain the secret.

5) Secret Recovery Algorithm – cross product portion

```
private BigInteger lagrange(int curShare) {
 BigInteger numer = BigInteger.ONE;
 BigInteger denom = BigInteger.ONE;
 BigInteger result;
 int i;
 for (i = 1; i < K+1; i++) {
  if (i != curShare) {
   numer = numer.multiply(BigInteger.valueOf(i));
   denom = denom.multiply(BigInteger.valueOf(curShare-i));
  }
 }
 result = numer.multiply(denom.modInverse(P));
 return result;
}
```

An important component of the secret recovery algorithm is the cross product portion of (2) which is implemented as a sub-method and called within an iterative loop. This decision made it easier to read the code in terms of the summation and cross product portions of (2). Both the summation and cross product portions take advantage of the iterative loop used to generate the corresponding *f(x)* share for a given *x* value. This simplifies the code implementation. However, doing so raises issues which will be covered in section VI.

C. Database Operations

This portion of the code consists of 5 methods, of which 4 are basic database operations that could be implemented in a production environment. The operations are responsible for committing a record, fetching a record, deleting a record, and deleting all records.

Figure 2 below depicts the "add" button, which ensures that neither the name field nor the credit card number field is empty before proceeding. It then triggers the random coefficient generator, the share generator, and finally calls the record commitment method which is one of the 4 database operations. As mentioned previously in this section, *x* values for (1) is generated incrementally within a loop. For this reason, the databases are accessed in the same order every time. More remarks on this mechanism are found in section VI.

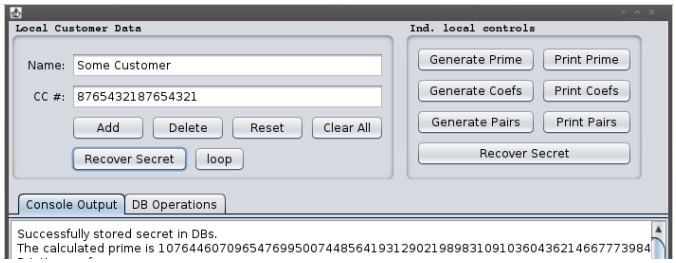

Fig. 2. Proof of concept Application depicting some of the control buttons. Add, Delete, Recover Secret, and Loop all operate on the databases, while the independent local controls do not interact with any database, and are used for debugging.

The record commitment method stores the various shares in the different databases. First, an SQL command is built, and a hash along with the corresponding share are concatenated to the SQL command. This command is then executed within a loop, where a different database is accessed, on each iteration. If a duplicate name field is found in the database, then the error is caught and signaled as an output in the DB Operations output area whose tab is shown in figure 2. Additional remarks on how *x* value generation may be improved are discussed in section VI.

Fetching a record works similarly as the record commitment function, where each database is accessed incrementally by means of a loop. Because every table only has 2 fields, name and share, the name field is used to search a corresponding share. For this work, the name field is built as a hash of the customer name times the hash of a password times the hash of the database where the corresponding share is to be fetched. More remarks on how the record name field is created are made in section VI. As the shares are fetched, they are placed in the share list to be used during the secret recovery step. If a record is not found, the error is caught and fetching the record fails.

Deleting a record works similarly as the record commitment method in that each database is accessed incrementally within a loop. The record name field is searched by calculating the

hash of the customer name times the hash of a password times the hash of the database name where the corresponding share is being searched, and once found, the record entry is deleted.

Deleting all records in all databases is trivial, where every database is accessed and data in their respective tables is dropped. While this wouldn't seem to be a necessary or routine part of the implementation, this procedure may be used as part of a kill-switch in case of emergencies.

D. *Additional Methods*

The remaining methods in the code consist of the detection of a duplicate entry when committing a record, a function that checks for valid input, a function that creates the name fields for each share using hashes of different data, and a test method that was used during the performance analysis. Details on the performance of the implementation will be reviewed in section V.

V. Performance Analysis and Results

The code was implemented in Java [17] using NetBeans 8.0.2 [16] as the integrated development environment. The reasoning behind this decision was that the profiler available in NetBeans was the least intrusive means to measure the performance of the individual methods, capable of collecting CPU timing information at a fine granularity with no changes to the original code. The average CPU times in establishing connections are times to connect to the set of Amazon RDS [2] and Google SQL [3] databases as dictated by the number of shares *k*.

A test button was implemented which loops through 3 principle methods: ***addRecord()***; ***getRecord()***; ***secRecover()***;. Of these three, we will focus on two, namely ***addRecord()*** and ***secRecover()***, since these include the sub-methods and results we are more interested in.

As mentioned earlier in section IV, the ***addRecord()*** method is responsible for generating secure random coefficients[10], generating shares using a corresponding *x* value for each database, and committing the changes to the databases. There were 8 runs of the test loop to gather data on the generation of coefficients and pairs, each run executing the three methods stated earlier once. The times were then recorded and averaged on a spreadsheet.

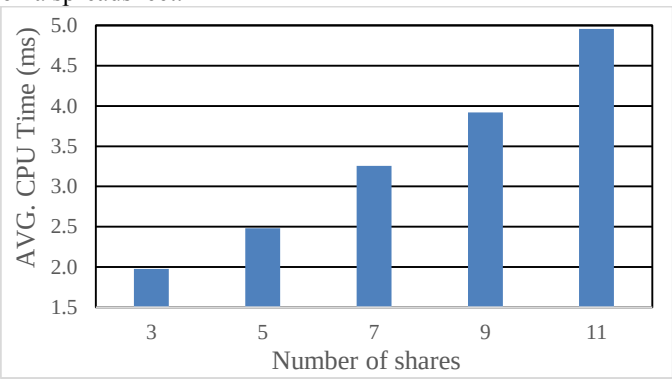

Fig. 3 Average CPU time to generate secure random coefficients. The number of coefficients generated per group of shares is (number of shares-1).

In figure 3, we can observe the performance results for the generation of secure random coefficients. On average, it took a mere 4.96ms to generate eleven 2047-bit secure random numbers. The fastest time was recorded at 3.93ms, whereas the slowest was 5.91ms. The results for any given number of shares will tend to vary slightly depending on the availability of resources on the test notebook computer; however, we can observe a clear trend that as the number of shares increase, the time it takes to generate secure random coefficients increases almost linearly.

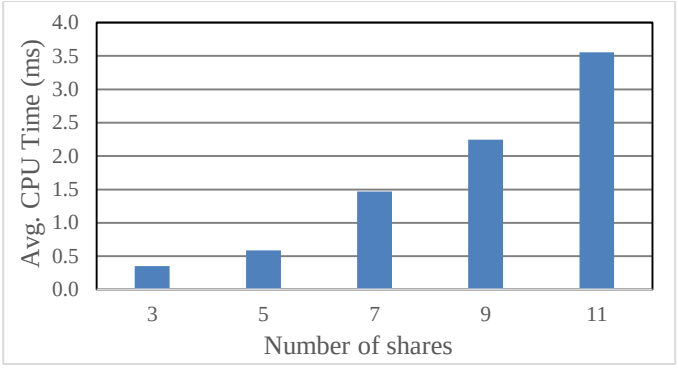

Fig. 4 Average CPU time to generate shares.

In figure 4 above, we can clearly observe that as the number of shares generated increase, the time to generate said shares increases exponentially. This is due to the fact that, because we are implementing [10][11] as a (*n, n)* scheme where *k=n*, the polynomial used to generate a corresponding share will increase in degree for every additional share *k*. No tests were performed with methods using a fixed-sized polynomial, as that would counter our implementation of a (*n, n)* scheme. During the tests, eleven shares were produced at a mere 2.11ms at its fastest, compared to 6.10ms at its slowest which is more than double the fastest time. These results also vary due to the availability of resources on the notebook computer.

During these tests, the results that stood out the most were the database connection times. The ***addRecord()*** method contains three sub-methods: ***genCoefs()*** which generates the secure random coefficients; ***genPairs()*** which generates the corresponding shares to an *x* input; and ***commitRecord()*** to establish the database connections and store the information.

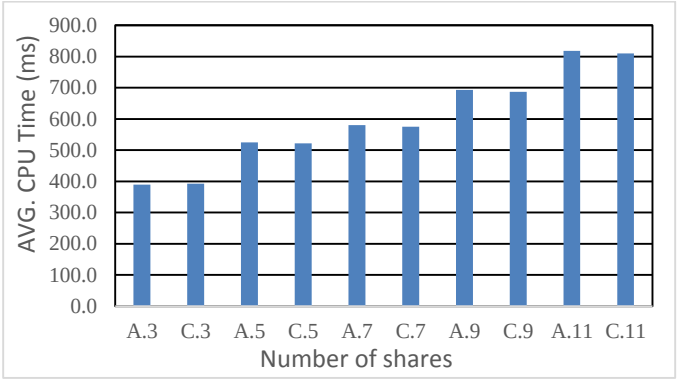

Fig. 5 Average CPU time to add a record 'A.' vs. portion of time spent establishing database connections 'C.' per number of shares.

In figure 5, we can observe how much time the ***commitRecord()*** method takes in the process of adding records to the databases. For each set of shares, the left column denoted by 'A.*x*' shows the average CPU time spent executing the three sub-methods, while the right column 'C.*x*' shows what portion of that time is spent establishing the

connection and sending the SQL ***INSERT*** command. From figure 5 we can conclude that the numbers of shares, and consequently the number of database connections being established, tend to have a more significant impact with respect to time than the methods involved in generating the secure random coefficients, and generating shares.

The objective in the following test was to analyze the average CPU time it took to recover a secret based on the number of shares used. The secret recovery method produces the summation of the *f(x)* terms multiplied by the cross product performed in the sub-method. This was done by means of an iterative loop, which called the sub-method at each iteration.

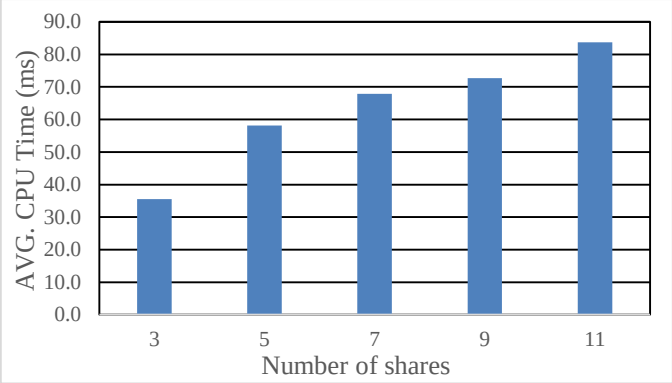

Fig. 6 Average CPU time to recover a secret per number of shares.

As we can observe in figure 6 above, the average CPU times to compute the secret increases as the number of shares increase. During the tests, it took 78.50ms at best to recover the secret for eleven shares, compared 90.60ms at its slowest. While these results are for almost four times the number of shares as the test for three shares, we can also observe that there is not a significant impact on the average CPU time to recover a secret based on the number of shares. For three shares, it took an average of 35.48ms to recover the secret, whereas for eleven shares it took an average CPU time of 83.68ms, which is a ratio of ~1:2.3.

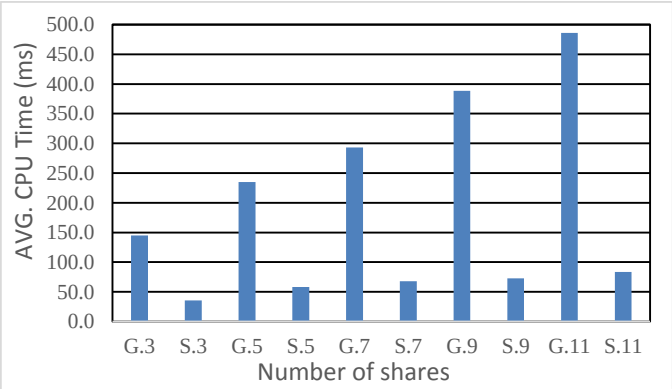

Fig. 7 Average CPU time to fetch shares denoted by 'G.' vs. average CPU time to recover secret denoted by 'S.'.

In figure 7 above, we can observe that the average CPU time to recover a secret denoted by 'G.*x*' does not produce as much of an impact as does the average CPU time spent establishing a connection denoted 'S.*x*' to recover the shares from the different databases. For example, using three shares as the data with the smallest gap, it took an average CPU time of 144.72ms to fetch all shares from the different databases, compared to an average CPU time of 35.48ms to recover the secret. In other words, it took roughly 25% of the average CPU time to recover the secret than it did to retrieve all shares from the different databases.

While the average CPU times during these tests were calculated by making connections to eleven databases across two providers as stated in the implementation details in section IV, these times may be expected to be different if the connections were to be made to more databases and providers around the world. In that case, we would mostly be concerned with the database connection times to recover the different shares, whereas the process to create the different shares or to recover the original secret would remain relatively insignificant in comparison.

VI. Suggested Improvements

A. *Generating x Inputs*

Generating *x* values for (1) were fairly simple as they were generated by means of a loop, generating incremental values of *x* from 1 to *k*. A security improvement to this method would be to use larger values of *x* which are either database or service provider-specific. Using this method would not only allow databases to be accessed randomly, it would also reduce the correlation between different databases in terms of the order in which they are accessed.

B. *Generating Secure Random Coefficients*

Initially, 2048-bit coefficients were being generated. This quickly proved to be problematic since there may be times at which the coefficients were larger than the prime being used, which would violate the rule in [11] of $P>C>0$. A check was then implemented to compare the size of the prime and the coefficient, where coefficients would be regenerated if they were larger than the prime. This additional step was deemed unnecessary, resulting in the decision of removing it altogether and simply generating 2047-bit coefficients. This large coefficient still enforces the discrete logarithm problem [21].

C. *Generating Record Name Fields*

For the purpose of this paper, the record name field is built as a hash of the customer name times a hash of the customer password times a hash of the database address where the particular share is being stored. This was done in an effort to suggest that the correlation between record name fields for a particular secret across all databases be reduced, or different for every database. However, this particular method is not sufficient since customers may have several credit cards whose shares may be stored in a particular database. A solution would be to also include a hash of some combination of numbers from a specific card. For the purpose of this paper it was enough to show that shares for a particular secret need not have the same identifying fields across all databases, however, a better mechanism should be used to enable a customer to have multiple credit card shares stored in the same database without record name field conflicts.

D. *Shuffling Records*

An additional security measure would be to shuffle the

records in any one of the databases to reduce the correlation among records across different databases. If records corresponding to a same secret are located in the same region or have the same order in any particular database, and the number of shares needed to reconstruct the secret is known, and the respective databases are known, then it would be easier for an attacker to assemble the required shares in attempt to reconstruct a secret. The task would still be very difficult, but this additional layer of security in just one of the databases would be enough to render the task even more difficult.

E. Diversity in Providers

This project advocates the use of multiple providers in effort to not only reduce the risks associated with a breach, but also to ensure availability. Although only two providers were used in this implementation, more could be used if desired, which would also afford new availability features. More on availability will be explained later in this section. Additionally, as suggested by [11], we would like to view providers as mutually suspicious with conflicting interests, which is why we do not want to store all shares for a particular secret within one same provider.

F. Planning For Redundancy

Although this implementation isn't a traditional *(k, n)* threshold scheme, that concept may be extended to the number of providers. For instance, for a secret that is split into fifteen shares and stored across three different databases or providers, it only takes one of those databases or providers to be unavailable for the secret to be unrecoverable. Instead, a threshold implementation would maybe create thirty shares to be spread over five providers for instance, and randomly accessing any combination of three providers would be enough to recover a secret. Together with the suggestion in 'F.', whenever the number of shares needed to reconstruct the secret is reached, all other connection attempts can be aborted. Additionally, having shares across different providers ensures that not all shares required to recover a secret are stored within one same provider.

VII. CONCLUSION

We have shown that implementing Shamir's secret sharing scheme to store credit card information on a multi-provider cloud architecture can be a viable solution. The performance tests show that the process of generating shares and recovering the secret is relatively fast and efficient when compared to the time spent establishing database connections, and this is valid for any application needing to connect to remote databases. The security feature of not being able to recover credit card information should any of the databases be breached brings an advantage that single-provider or singe local databases do not. This paper has shown that the Shamir secret sharing scheme is fast, reliable and secure, but most importantly that it is applicable. The suggested improvements in section VI along with additional contributions could produce a more secure and production-ready implementation for multiple environments.

VIII. REFERENCES

[1] K. Bessai, S. Yousef, A. Oulamara, C. Godart, S. Nurcan, "Scheduling Strategies for Business Process Applications in Cloud Environments," International Journal of Grid and High Performance Computing, Volume 5 Issue 4, pp. 65-78 October 2013.

[2] Amazon RDS: Relational Database Service. https://aws.amazon.com/rds/

[3] Google Cloud SQL. https://cloud.google.com/sql/docs

[4] Sage, Cloud Storage Market to Grow by 2019. http://na.sage.com/us/articles/technology/cloud-storage-market

[5] The PNC Financial Services Group, Inc. Virtual wallets. https://www.pnc.com/en/personal-banking/banking/checking/virtual-wallet.html

[6] Anonymous, "Young People in Particular are Annoyed by Queues at the Cash Register - Germans are Open to Paying by Smartphone," PR Newswire Association LLC, 01 Jul 2014, ProQuest Newsstand, 01 Jul 2014.

[7] Anonymous, "USA ePay Joins the Secure Vault Payments Network," Business Wire, 16 Nov 2010, ProQuest Newsstand, 16 Nov 2010.

[8] The PCI Security Standards Council, "Verify PCI Compliance, Download Data Security and Credit Card Security Standards," Accessed 15 Mar 2015, https://www.pcisecuritystandards.org/organization_info/index.php

[9] D. Wolfe, "Breach Revives Doubts About Card Industry Security Standard," American Banker, 03 Apr 2012, ProQuest Newsstand, 17 Apr 2012

[10] D. Agrawal, A. El Abbadi, F. Emekci, A. Metwally, "Database Management as a Service: Challenges and Opportunities," IEEE 25th Int'l Conf. Data Engineering (ICDE 09), IEEE CS Press, 2009, pp. 1709–1716.

[11] A. Shamir, "How to share a secret," Commun. ACM, vol. 22, no. 11, pp. 612–613, 1979.

[12] B. Krebs, "Payment Processor Breach May Be Largest Ever," The Washington Post, 20 Jan 2009, Accessed 15 Mar 2015. http://voices.washingtonpost.com/securityfix/2009/01/payment_processor_breach_may_b.html

[13] T. Armerding, "The 15 worst data security breaches of the 21st Century," CSOOnline, IDG Enterprise, 15 Feb 2012, Accessed 15 March 2015. http://www.csoonline.com/article/2130877/data-protection/the-15-worst-data-security-breaches-of-the-21st-century.html

[14] Lagrange polynomial interpolation. http://www2.lawrence.edu/fast/GREGGJ/Math420/Section_3_1.pdf

[15] Free Software Foundation. What's in a Name? https://www.gnu.org/gnu/why-gnu-linux.en.html

[16] Oracle Corporation, NetBeans. https://netbeans.org/about/index.html

[17] Oracle Corporation, Java. https://www.oracle.com/java/index.html

[18] Oracle Corporation, MySQL. http://www.mysql.com/

[19] Oracle Corporation, MySQL JDBC Driver. http://www.mysql.com/products/connector/

[20] D. Flanigan, Java in a nutshell: a desktop quick reference. BigInteger subclass, Sebastopol, CA: O'Reily Media, Inc. Mar 2005, pp. 546.

[21] R. Barbulescu et al., in A Heuristic Quasi-Polynomial Algorithm for Discrete Logarithm in Finite Fields of Small Characteristic, 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, 2014 © International Association for Cryptologic Research. Doi: 10.1007/978-3-642-55220-5_1

[22] D. Flanigan, Java in a nutshell: a desktop quick reference. SecureRandom subclass, Sebastopol, CA: O'Reily Media, Inc. Mar 2005, pp. 639

[23] B. M. Brosgol, "A Comparison of the Mutual Exclusion Features in Ada and the Real-Time Specification for Java," http://www.adacore.com/uploads/technical-papers/MutEx-Ada-RTSJ-paper.pdf